Universidade Federal de Pernambuco
Centro de Informática

Pós-Graduacão em Ciência da Computacão

A FRAMEWORK FOR SPECIFICATION AND VALIDATION OF REAL TIME SYSTEMS USING CIRCUS ACTION

Adnan Sherif

TESE DE DOUTORADO

P.O. Box 7851 Cidade Universitária
50740-540 Recife - PE
Brasil
20/01/2006

Universidade Federal de Pernambuco
Centro de Informática

Adnan Sherif

A FRAMEWORK FOR SPECIFICATION AND VALIDATION OF REAL TIME SYSTEMS USING CIRCUS ACTION

Trabalho apresentado ao Programa de Pós-Graduacão em Ciência da Computacão do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obencão do grau de Doutor em Ciência da Computacão.

Orientador: *Prof. Dr. Augusto Sampaio*
Co-orientadora: *Profa. Dra. Ana Lucia Cavalcanti*

P.O. Box 7851 Cidade Universitária
50740-540 Recife - PE
Brasil
20/01/2006

ACKNOWLEDGEMENTS

God all mighty, from where I obtained forces to conclude this work.

To my Father, Mahmoud, and my Mother, Irenita, that showed me how to fight for my dreams and make them come true, my eternal thanks.

To Afaf Ali El-Garhni, my dear wife, that was always present in the happy and sad moments, and my children Nidal and Leila for their understanding in the difficult moments. To all my family for backing me, and specially to my ant Ireni for her help and attention and to my father-in-law Haj Ali for his continuous prayers and support.

To my professors, masters and friends Augusto Sampaio and Ana Cavalcanti, for their dedication and supervision during the development of this work. Special acknowledgments to Professor He Jifeng for his supervision and guidance during my fellowship at the International Institute of Software Technology, United Nations University (IIST/UNU) in Macau, during his supervision the main milestones of this work have been established.

I would like to thank the examiners of this thesis, Jim Woodcock, Roberto Bigonha, Alexandre Mota, Paulo Maciel and Nelson Rosa, for their comments and recommendations. In particular, we would like to thank Prof. Jim Woodcock for pointing out a limitation in the framework, as shown in Section 5.7.

To all the people that form the CIn/UFPE, IIST/UNU, Centro XML Recife and Qualiti for their help and assistance.

To all my friends for their encouragement and words of comfort.

ABSTRACT

Circus is a specification and programming language that combines CSP, Z, and refinement calculus constructs. The semantics of *Circus* is defined using the Unifying Theories of Programming (UTP). In this work we extend a subset of *Circus* with time operators. The new language is denominated *Circus* Time Action. We propose a new time model that extends the Unifying Theories of Programming model by adding time observation variables. The new model is used to give a formal semantics to *Circus* Time Action. Further, the algebraic properties of the original *Circus* are validated in the new model, and new properties are added and validated in *Circus* Time Action.

The advantage of using the unification pattern proposed by UTP is the ability to compare and relate different models. We define an abstraction function, L, that maps the timed model observations to observations in the original model (without time); an inverse function, R, is also given. The main objective of this mapping is to establish a formal link between the new time model and the original UTP model. The function L and its inverse function R form a Galois connection. Using the abstraction function, we introduce the definition of time insensitive programs. The mapping function allows the exploration of some properties of the timed program in the untimed model. We present a simple example to illustrate the use of this mapping. The abstraction function can be used to validate properties that are time insensitive.

Real-time systems have time constraints that need to be validated as well. We propose a framework for specification and validation of real-time programs using *Circus* actions. The framework structure is based on a partitioning process. We start by specifying a real-time system using *Circus* Time Action. A syntactic mapping is used to split the program into two parts: the first is an untimed program with timer events, and the second is a collection of timers used by the program, such that the parallel composition of both parts is semantically equivalent to the original program. Using timer events we show that it is possible to reason about time properties in an untimed language. To illustrate the use of the framework, we apply it to an alarm system controller. Because, in the validation process, the programs are reduced to the untimed model, we use a CSP model-checking tool (FDR) to conduct mechanical proofs. This is another important contribution of this work.

RESUMO

Circus é uma linguagem de especificacão e programacão que combina CSP, Z, e construtores do Cálculo de Refinamento. A semântica de *Circus* está baseada na Unifying Theories of Programming (UTP). Neste trabalho estendemos um subconjunto de *Circus* com operadores de tempo. A nova linguagem é denominada de *Circus* Time Action. Propomos um modelo novo do tempo que estende o modelo da UTP, adicionando variáveis de observacão para registrar a passagem de tempo. O novo modelo é usado para dar a semântica formal de *Circus* Time Action. Propriedades algébricas do modelo original de *Circus* são validadas no novo modelo; propriedades novas são exploradas e validadas dentro do contexto de *Circus* Time Action.

A vantagem de utilizar o padrão de unificacão proposto pela UTP é poder comparar e relacionar diferentes modelos. Definimos uma funcão de abstracão, L, que faz o mapeamento das observacões registradas no novo modelo (com tempo) para observacões no modelo original (sem tempo); uma funcão inversa, R, é também definida. O objetivo é estabelecer uma ligacão formal entre o modelo novo com tempo e o modelo original da UTP. A funcão L e sua funcão inversa R formam uma conexão de Galois. Usando a fun cão de abstracão, nós introduzimos a definicão de programas insenséveis ao tempo. A funcão de abstracão permite a exploracão de algumas propriedades não temporais de um programa. Apresentamos um exemplo simples para ilustrar o uso da funcão de abstra cão na validacão de propriedades que não têm tempo associado.

Entretanto, sistemas de tempo real têm requisitos temporais que necessitam ser validados. Neste sentido, propomos um *framework* para a validacão de requisitos não temporais usando os dois modelos e a relacão entre eles. A estrutura do *framework* é baseada em um processo de particionamento. Tendo como ponto de partida o programa e sua especificacão escritos em *Circus* Time Action, aplicamos uma funcão sintática que gera uma forma normal do programa e sua especificacão. A forma normal consiste de duas partes: a primeira é um programa sem operadores de tempo, mas com eventos que, por convencão, representam acões temporais; a segunda é uma colecão de temporizadores (timers) usados pelo programa. A composicão paralela de ambas as partes é semanticamente equivalente ao programa original. Usando apenas o componente da forma normal que não envolve tempo explicitamente, mostramos que é possível raciocinar sobre propriedades de tempo no modelo não temporal; provamos formalmente a validade deste resultado.

Para ilustrar o uso do *framework*, utilizamos um sistema de alarme simplificado como estudo de caso. Como a validacão é reduzida ao modelo sem tempo, usamos a ferramenta de verficacão de modelos de CSP (FDR) para realizar as provas mecanicamente. Esta é uma outra contribuicão importante deste trabalho.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Real-time computer systems differ from general-purpose computer systems in that they introduce the notion of time to the computational requirements of the system. Real-time systems must not only provide logically correct and adequate results but these results also have to be provided within a certain period of time.

1.1 MOTIVATION

First generation real-time applications were relatively simple and did not involve sophisticated algorithms or extensive computational complexity. However, things have changed in the last decades towards more complex and more safety critical applications such as aerospace navigation and control, monitoring factories and nuclear power plants, among other applications. A real-time computer system can be thought of as a system that receives stimuli from the environment through its input. Stimuli may be time triggered if the computer system periodically observes the input, or event triggered if the system waits for data to be handled in the form of interrupts or events. The system will perform operations on these inputs. Operations can be timed to start and/or end in predefined times. It can also be determined that a process operation has a maximum duration time determined by the system requirements. The output of a real-time application can also be time based.

An important part of a real-time system specification is the implication of the system not meeting its timing requirements. One possibility is when failing to meet its timing requirements leads to a general unrecoverable system failure. Such timing restrictions are known as *hard real-time constraints*. The other possibility is that by failing to meet a timing constraint, the failure is tolerated and the system can continue its operation, even though the performance and quality may degrade. This is known as a *soft real-time constraint*. A real-time system that contains at least one hard real-time constraint is called a *hard real-time system*. If all the timing constraints are soft then the system is called a *soft real-time system*.

Another important aspect of real-time systems is the type of timing requirements of the tasks that compose the system operations. There are two types of timing requirements for a task. The first is periodic, independent of the external environment behaviour. A simple data acquisition task that reads a sensor at fixed periods is an example of a *periodic task*. On the other hand, there are tasks which depend on external events from the environment to be executed. For example, a button pressed or a robotic arm reaching the designated destination. These are normally implemented by means of interrupts. Tasks of this type are known as *aperiodic tasks*. Time is continuous by nature, but a discrete representation of time is also satisfactory in most cases. In specification languages, time is represented by real numbers of continuous nature (as well).

However, in programming languages, time is represented as integers with a constant increment of one.

In Software Engineering, it is recommended to use formal specification languages to express the behaviour and the effect of a software system. The difference between formal specification languages and natural languages lies in the fact that formal specification languages are based on a well-defined meaning for each symbol, whereas natural languages may have various meanings for a single symbol. This aspect tends to make a specification in a natural language ambiguous. A formal specification language has a well-defined mathematical representation for its semantics; this allows one to analyze and study the different aspects of the software written in such a language.

The choice of language to be used in the specification of a real-time system is an important factor concerning the success of the entire development. The language should cover several facets of the system requirements, and should have a suitable observation model. The model is used to study the behaviour of the system and to establish the validity of desired properties. Such a formal model may also be the basis for a formal refinement into executable code. Another important factor to be considered is the popularity of the language.

During the last years, software engineering researchers have developed a large number of formal specification languages. However, each of these languages is suitable for expressing a particular characteristic and lacks the power of expressing others. For example, Z [53] is a formal language used to define data types and to show the effect of operations on these types. Nevertheless, Z lacks tools to express the order in which the operations are executed [15]. On the other hand, CSP [3] is a language suitable for showing the order of the occurrence of events but lacks the ability to express abstract data types and to show the effect of the events on data. Formalisms like temporal logics [1] concentrate on time aspects.

The increasing complexity of real-time systems (in the last years) made the use of formal specification more frequent, but systems are normally modelled as a mixture of data types and operations, and, in the case of real-time systems, time constraints. Moreover, formal development processes based on refinement are needed and mechanisms to derive and show the program correctness from the formal specification are required. For this reason, the current devotion of researchers is towards new and more complete specification languages. Taking into consideration that the language should be well-known and accepted, many proposals to extend existing languages have been introduced. Others propose the merging of two or more languages and give a new semantics to the integrated language. For example, *Circus* [58] combines CSP, Z, specification statements and commands, and thus *Circus* is a language suitable for both specification and programming. The semantics of *Circus* [59, 57] is given in terms of the Unifying Theories of Programming (UTP) [26]. Refinement laws for *Circus* are explored in [47, 4]. The combination of different formalisms and paradigms in *Circus* was made possible and natural because of the adoption of the semantic model of the UTP. The UTP proposes a unification of different programming paradigms based on the theory of relations. The unification permits the exploration of different paradigms. The relation between the paradigms can result in mappings that relate programs in abstract models to programs in more concrete

models; in UTP the refinement relation is simply logical implication.

1.2 PROPOSAL

In this work we introduce *Circus* Time Action; the language uses a subset of *Circus* and adds time operators to the notion of actions in *Circus*. We propose an extension to the semantic model of the UTP. The new model captures time information and registers these observations along with the original ones of the program; the model proposed is a discrete time mode. The decision of using a discrete time model instead of a continuous one is because *Circus* is both a programming and a specification language; discrete models can be used in both specification and programming. The language can be used to specify both hard and soft real-time systems; it is also suitable for representing periodic and aperiodic tasks.

As mentioned before, a major advantage of using formal methods in system specifications is the ability of studying a specification with the aim of assuring the presence and/or absence of properties in the system behaviour. To achieve this objective, model checking can be used to compare different observed behaviours of a system. Model checking is an automatic tool that helps to compare different specifications based on an observation model. We also explore the relation between the new time model and the UTP model. We conclude that the original model is an abstraction of the time model, and that several mappings can be found between them. We also show that suitable mappings can form a Galois connection between these models. We explore the relation between these models to show that timed programs satisfy untimed specifications that do not make reference to time information.

As a further contribution, we propose a framework based on the partitioning of a timed program into a normal form composed of untimed programs with timer events and a set of timers. When the new untimed program with timer events and timers are composed in parallel, the resulting program is semantically equivalent to the original program. The untimed part uses two new operators, *normal form external choice* and *normal form parallel composition*. These operators have specialised behaviours concerning the timer events. We give the semantics of these new operators and show that the algebraic characterisation is complete by a reduction process that eliminates these operators.

The normal form program is an untimed program that contains timer events and preserves the semantics of the original timed program. By representing time with timer events, the framework permits the use of untimed tools for instance FDR [20], to analyze timed specifications. To illustrate the framework, we analyse some properties of a simple alarm system controller. The specification of the system is given using *Circus* Time Action, and the normal form program is obtained by applying a syntactic transformation. The implementation of the alarm controller is also expressed in *Circus* Time Action and also transformed into a normal form. We use the CSP model checking tool to show that the program meets its specification, dealing only with the untimed components of each normal form.

1.3 DOCUMENT ORGANIZATION

This document is organized as follows. In Chapter 2, we introduce some of the state-of-the-art formal specification methods for real-time systems; the aim is to show the different formalisms and how the timing constraints of a real-time system are captured by each of the considered formalisms. The different formalisms chosen are: Timed-CSP (related to process algebras), RTL (related to logics) and TB Nets (related to Petri Nets).

In Chapter 3, we present the proposed formal specification language: *Circus* Time Action. The syntax of the new language is a subset of *Circus*, plus two new constructs related to expressing time constraints. The time operators are *Wait* t, which halts the program execution for t time units, and *timeout*, which is used to determine a time limit for initial events of a process. An extension to the semantic model of the UTP is given and new healthiness conditions are presented. The semantics of *Circus* Time Action constructs are defined using the new discrete time model. The semantics of the operators that are common to both *Circus* and *Circus* Time Action are discussed in detail.

Chapter 4 explores the relation between the timed model of *Circus* Time Action and the untimed model of the UTP. Mapping functions are defined and used in relating the two models. An important result in this chapter is the identification of a class of timed programs classified as *time insensitive*; they have the same semantics in the time and in the untimed models. The mapping between the two models is used to show that timed programs satisfy untimed requirements. The process of validation is based on obtaining an untimed version of the timed program with the aid of a mapping function. The untimed version of the program can be used in the validation. Unfortunately, the mapping functions can be used only to validate properties that are originally time insensitive and cannot be used to check wether the system meets its time requirements.

In Chapter 5, a validation framework is introduced. The framework uses a partitioning strategy: a real-time program is split into an untimed program and a collection of interleaving timers, in such a way that the untimed program contains no time operators; instead, it contains special timer events, and the time information of the original program is concentrated in a collection of timers. We ensure the equivalence between the untimed program with timer events, when in parallel with the collection of timers, and the original timed program. The framework is used for the validation of time constraints that cannot be validated using the semantic mapping functions. We also show the support of a tool (FDR) in the validation process. An example of an alarm controller is used to illustrate the process of validating time restrictions using the framework.

Finally, Chapter 6 presents the conclusions, related work and possible topics for future research.

CHAPTER 2

REAL-TIME SYSTEMS AND FORMAL METHODS

The significant characteristic of a real-time system is the existence of requirements concerning timeliness as well as functional behaviour. The correctness of the system depends not only on the results it produces, but also on the time at which the results are available. Most real-time systems are also reactive: they react to events or changes in their environment.

Real-time systems are classified into ***hard real-time*** systems and ***soft real-time systems***. In a ***hard real-time*** system, timeliness requirements are strict, and a result delivered too late (or too early) is considered incorrect. The system must meet all temporal requirements in order to be considered correct. An example of a ***hard real-time*** application is a nuclear plant reactor controller, where missing timeliness can be catastrophic. In ***soft real-time*** systems, the value of the result decreases if the timeliness restrictions are violated, but a late result might be better than no result at all. An example of a ***soft real-time*** application is a video cassette controller; missing the timeliness for displaying a frame will degrade the quality of the output.

Real-time systems have become more complex with the passage of time. The nature of real-time systems has also become more critical (as well). This fact makes real-time systems more difficult to specify and thus serious candidates for the application of formal methods.

The primary goal of this chapter is to present a brief and critical analysis of some of the state-of-the-art formal specification methods used for specifying real-time systems.

2.1 FORMAL SPECIFICATION OF REAL-TIME SYSTEMS

Normally, the specification of the functions and requirements is the first step in the design of any system. The conventional way to do this is to write (a lengthy) natural language description of the design goals. This has the advantage of being easily understood by a large number of people, as it does not require special skills which are normally needed by more formal approaches. However, such a description can lead to ambiguities, misinterpretations and even contradictions due to the semantic nature of natural languages.

In Software Engineering, the task of system specification has gained more attention in the last years, as the final stages of the development of a computer system tending to become more and more automated. The attention has been devoted to the specification of the system, aiming at making a more formal specification, free of ambiguities and misinterpretation as well as to allow for reasoning about the properties of the system (validation) before the system implementation. This has stimulated the use of formal methods at the system specification level.

Formal methods are maturing, slowly but steadily [2]. In the past, formal methods

were used by institutions and research groups to solve small or toy problems. But today, formal methods are applied to develop large and complicated systems. Several examples can be found in the *International Survey of Industrial Application of Formal Methods* [9] and more recently in [2]. The survey presents various examples showing the method applied in each case, the way it was applied, in which stage of the development process the method was introduced, up to which stage it was used, and what tools were used.

During the next sections we study several formal specification methods used in the specification of real-time systems. These represent a small subset of those available today. The selection was based mainly on the fact that these languages are devoted to the specification of real-time systems, and each one represents a different alternative for time representation.

Most of these methods are derived from classic untimed approaches with adjustments to express the time notions of the system specification. This makes them more popular and easier to understand.

The methods chosen for our survey are:

- Process Algebra.
- Temporal logic.
- Petri Nets.

To introduce each of these methods in a comparative way, a simple example (a Watchdog Timer - WDT) is introduced in the next section. The WDT is part of a more general case study, *The SACI-1 On-board computer*, which was explored in detail in [50].

2.2 THE WATCHDOG TIMER (WDT)

A watchdog timer is a mechanism used in many systems. It represents a timer which is activated, and when it reaches a pre-defined time it triggers and requires immediate attention. The timer should be reset by the watched process before the WDT triggers, to show that the process is still alive. The action taken by the watchdog depends on the system requirements.

In the SACI-1 satellite architecture [14] (see Figure 2.1) there are three watchdog circuits, one for each processor. The WDT communicates directly with the corresponding processor through an interrupt line to and from the CPU. The WDT is initialized at the startup of the system. When it triggers it raises an interrupt on the monitored CPU. This interrupt has the highest priority among all interrupts processed by the CPU. When the interrupt is raised the CPU must respond within a predefined period of time. It does this by sending a reset signal to the WDT. If the CPU fails to do so, then the WDT considers that the monitored CPU has failed and it should be cutoff from the rest of the system.

The observed CPU is permitted to fail for seven consecutive times before the WDT considers it unrecoverable, in which case the WDT informs the other two CPUs that this CPU is to be considered failed and will be cut off from the system. The other CPUs should take over any activities the failed CPU was performing by reconfiguring and redistributing the application processes.

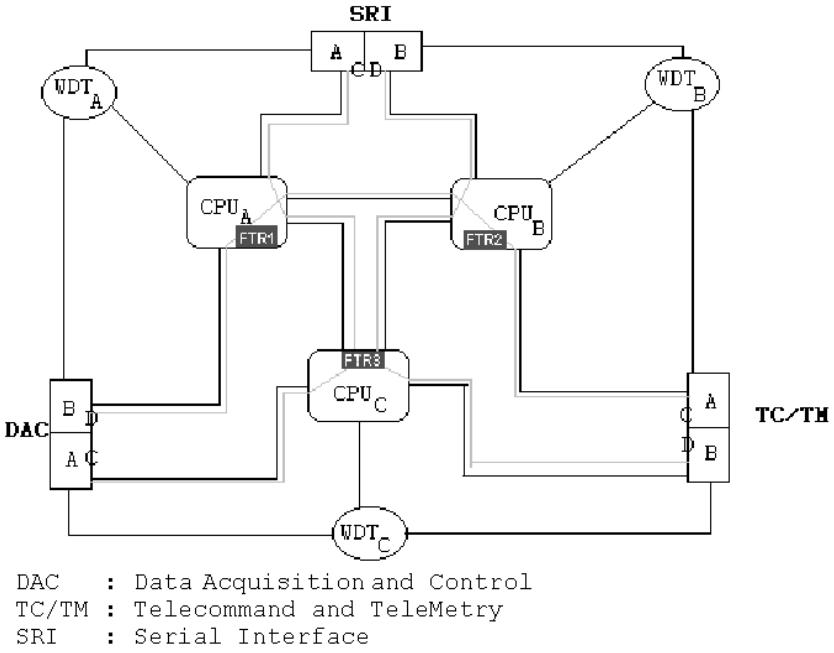

Figure 2.1. Architectural Diagram of the SACI-1 OBC.

Figure 2.1 shows the three CPUs and their related WDTs. There are three pairs of interfaces used by the On-board Computer (OBC) to communicate with the rest of the satellite. Each CPU has four I/O ports, of which two are connected to an interface circuit, while the other two are used to connect the CPUs with each other. The lighter color lines are alternative lines; they serve as alternative circuits to bypass a failed CPU and the other CPUs to take over the interface circuits of the defected processor.

Each processor has a Fault Tolerant Router (FTR) process. This process acts like a kernel of the OBC system. The Process is mainly responsible for controlling the traffic of messages between the application processes in the same CPU or in different CPUs. Another activity of the FTR is to monitor the input lines and handle the incoming interrupts. A complete description of the functions of the FTR can be found in [50].

During the next sections, we study several formal specification languages and the time model of each one of them. To further explain the use of the language and show the advantages and weak points of each, the WDT is used as a common example.

2.3 PROCESS ALGEBRA

The key concept behind process algebraic methods is the concept of processes and events. Processes are defined by an equation that gives a process name a behaviour defined based on other processes or events. Simple processes can be combined with the aid of process operators and produce more complex processes. Processes are combined sequentially, by selection or in parallel compositions. Process algebras are easy to use and understand. Along with each process algebra there is a set of algebraic rules that

can be used to explore properties such as equivalence. Three of the most used process algebras are Communicating Sequential Processes (CSP) [25],Calculus of Communicating Systems (CCS) [36] and π Calculus [37].

Process algebras have been extended with time to be used in the specification of real-time systems. An extension to CSP is Timed CSP presented by Reed and Roscoe [44, 43]. In CSP, as in other process algebras, the key concepts are events and Processes. For example a term is CSP $a \rightarrow SKIP$ is a process that starts by waiting to communicate on event a and then behaves as the process $SKIP$; where the $\rightarrow$ is a communication prefix operator and $SKIP$ is a predefined process that simply terminates. Other operators such as external choice $\Box$ and internal choice $\sqcap$; composition operators, $\|$ for parallel composition and ; for sequential composition and others can be found in CSP. Timed CSP adds some new operators to the language such as $Wait$ and $\rhd$ (timeout). Timed CSP introduces a new semantic model to represent dense time information. The model is derived from the untimed models of CSP. Davies and Schneider, in [10, 49], extended Reed and Roscoe's model to include a proof system. Further in this section, we present more details of Timed CSP as our case study for process algebras.

Calculus of Communicating Systems is very similar to CSP. An operational semantics of CCS is given using transition systems. The main difference between CSP and CCS is the communication of concurrent processes. In CSP a process $a \rightarrow SKIP \;||\; a \rightarrow SKIP$ is equivalent to the process $a \rightarrow SKIP$ as both parts in parallel synchronize on the event a. In CCS the synchronization only occurs between the event a and its complementary event $\overline{a}$. The process $a.NIL \mid \overline{a}.NIL$ represents two parallel processes that synchronize on the event a where NIL is the CCS construct equivalent to $SKIP$ and the period is equivalent to the communication prefix. The result of the synchronization in CSP is the single occurrence of the event a registered in the traces while in CCS the internal event τ is issued to represent the synchronization; therefore, in CCS synchronization of parallel processes is internal. Furthermore synchronization in CCS is optional. Therefore $a.NIL \mid \overline{a}.NIL$ is equivalent to the process $a.\overline{a}.NIL + \overline{a}.a.NIL + \tau.NIL$, where the operator $+$ is the external choice operator. Figure 2.2 shows the transition graph of the process $a.NIL \mid \overline{a}.NIL$.

A time extension for CCS is given by Chen in [8]. Chen uses Timed Synchronization Trees to give an operational semantics to Timed CCS. The only change to the syntax of the CCS operators is the prefix operator. In timed CCS, the term $a(t)_e^{e'}$ restricts the communication on a to happen at any moment in the closed time interval $[e, e']$ and only e' is allowed to be infinity ∞. After the communication on a occurs the variable t holds the time at which the communication occurred; t can then be used to calculate the relative time between events. For example, consider a process P in which a can occur at any time but b has to occur after passing 3 time units from the occurrence of a. The timed CCS process is defined as follows

$$P = a(t)_0^{\infty}.b_{t+3}^{\infty}.NIL$$

Proof rules for timed CCS were elaborated and shown to be independent of the time domain used.

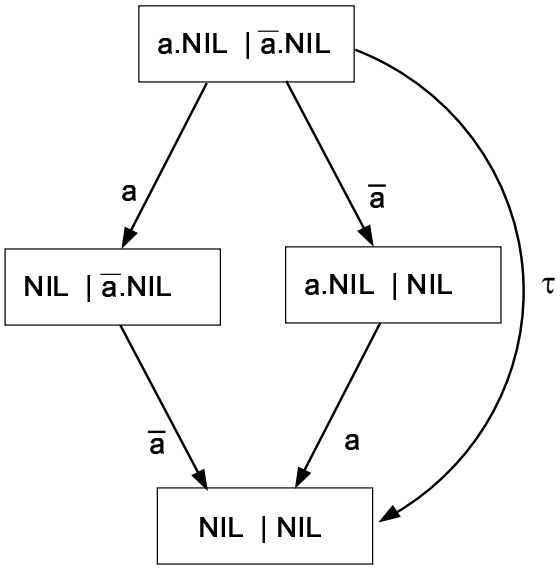

Figure 2.2. Transition Graph for the CCS process $a.NIL \mid \overline{a}.NIL$

The process algebra for mobile processes π Calculus is an extension of CCS. The π Calculus focuses on the definition of processes whose interconnections change during their life time. Consider the example of a printer server [37]. Figure 2.3 shows a server computer controlling a printer, or a pool of printers, and a client computer willing to send a document to the printer. The client communicates with the server first requesting a connection to an available printer, the server returns a connection to the printer and the client can then start printing, directing printing commands to the connection passed by the server. The printing server can be defined as the π calculus process $\bar{a}b.S$, where the server communicates over channel a the connection to channel b and then behaves as S. The client is defined as $a(c).\bar{c}d.C$; the client communicates with the server over channel a and obtains the connection that is assigned to the channel c. Therefore, after the synchronization with the server, the channel c connects the client to the printer. The client uses channel c to communicate to the printer the data it is willing to print, represented by the variable d in our example, and finally the client behaves as C.

In [16] the authors present a comparison between Timed CSP, Timed CCS, TE-LOTOS and Timed Z. The comparison involved applying the different methods in the specification of a case study: railway crossing. An interesting conclusion is that the choice of which process algebra to use is a matter of personal taste. In the following section we explore in some detail the Timed CSP process algebra as an example.

2.3.1 Timed CSP

The original (untimed) CSP was first presented by Hoare in 1978 [24], and has suffered several changes ever since. The result of these changes [25] is the version used in this work. We limit our presentation to the timed model of Timed-CSP and only cite

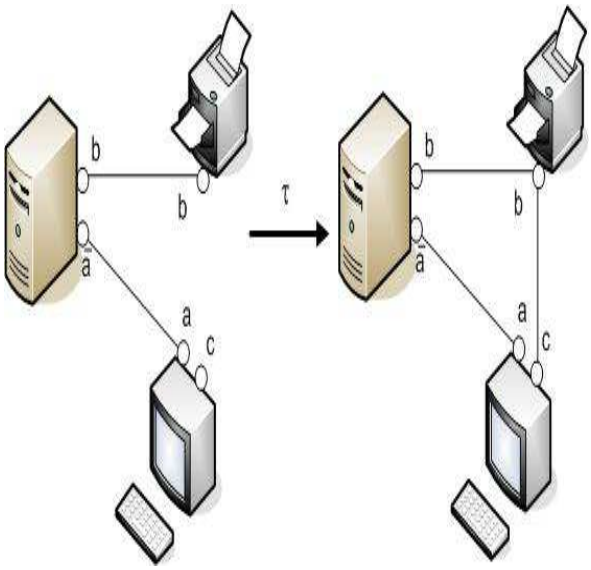

Figure 2.3. Printer server example

some similarities with the untimed model. For a complete reference to the untimed CSP see [25, 45].

Like CSP, Timed CSP uses processes and events to define the behaviour of programs, where a process is a set of observations which defines a pattern of behaviour [43], and an event is an observation mark indicating the occurrence of an associated action or stimulus.

The Timed CSP semantic model is influenced by the same properties of the untimed model of CSP [10]:

- *Maximum progress.* A program will execute until it terminates or requires some external synchronization.
- *Maximum parallelism.* Each component of a parallel composition has sufficient resources for its execution.
- *Synchronous communication.* Each communication event requires the simultaneous participation of every program involved.
- *Instantaneous events.* Events have zero duration.

2.3.1.1 The Time Model There are several time models for timed CSP. The simplest time model $\mathcal{M}_{TT}$ associates a program with a set of timed traces. The timed failure model $\mathcal{M}_{TF}$ records the events refused by a program during and after the observation of each trace. The timed failure-stability model $\mathcal{M}_{TFS}$ enriches the failures model with details of the program stability.

We limit ourselves to the $\mathcal{M}_{TF}$ model as presented by Davies and Schneider [10], where an observation is given by the pair:

$$(\textit{timed trace}, \textit{timed refusals})$$

in which the *timed trace* represents the observed sequence of timed events, and the *timed refusal* records the set of timed events refused. The behaviour of a process in the $\mathcal{M}_{TF}$ is captured by a set of such pairs. Our domain of time values is the non-negative real numbers.

$$TIME = [0, \infty)$$

The set of all timed events (TE) is given as the following cartesian product, where Σ is the set of all the events.

$$TE = TIME \times \Sigma$$

A timed trace is a finite sequence of timed events, such that events appear in a chronological order. The set of all time traces (TT) is given as follows

$$TT \in \{s \in seq\ TE \mid (\langle (t_1, a_1), (t_2, a_2) \rangle \leq s) \Rightarrow (t_1 \leq t_2)\}$$

where $s_1 \leq s_2$ stands for the order relation: s_1 is a subsequence of s_2.

If I is a finite half-open timed interval, and A is a set of events, then we say that the elements of the cartesian product $I \times A$ is a refusal token. The set of all the refusal tokens RT is given by

$$RT = \{[t_1, t_2) \times A \mid 0 \leq t_1 < t_2 < \infty \wedge A \subseteq \Sigma\}$$

Timed refusals are finite subsets of refusal tokens (subsets of RT). Therefore, the set of all timed refusals is given by

$$TR = \{\cup R \mid R \subseteq RT \wedge R \text{ is finite}\}$$

The set of all timed observations is expressed as

$$O_{TF} = TT \times TR$$

The behaviour of a process in the $\mathcal{M}_{TF}$ model is a subset of O_{TF}. Next, we introduce some auxiliary functions which will allow us further to give the operational semantics of the time operators of Timed CSP.

A function σ is defined upon each type of timed observation. It returns the set of events present in the observation trace. In thus definition, s is a timed trace and X is a timed refusal.

$$\begin{aligned} \sigma(s) &= \{a \mid \exists\, t \bullet \langle (t, a) \rangle \text{ in } s\} \\ \sigma(X) &= \{a \mid \exists\, t \bullet (t, a) \in X\} \\ \sigma(s, X) &= \sigma(s) \cup \sigma(X) \end{aligned}$$

Another function *times* is defined to return the set of all times present in the passed observation.

$$\begin{aligned} times(s) &= \{t \mid \exists\, a \bullet \langle (t,a) \rangle \leq s\} \\ times(X) &= \{t \mid \exists\, a \bullet (t,a) \in X\} \\ times(s,X) &= times(s) \cup times(X) \end{aligned}$$

To restrict our attention to events refused during a particular interval I, we use the operator $\uparrow$:

$$X \uparrow I = X \cap (I \times \Sigma)$$

We need to refer to the time of the first and last events in an observation. This is done with the functions *begin* and *end*.

$$\begin{array}{ll} begin(s) = inf(times(s)) & end(s) = sup(times(s)) \\ begin(X) = inf(times(X)) & end(X) = sup(times(X)) \\ & end(s,X) = max\{end(s), end(X)\} \end{array}$$

The infimum (*inf*) and supremum (*sup*) of the empty set of times are taken to be ∞ and 0 respectively, the function *max* returns the maximum of a set. The variations in the definition of *begin* and *end* is to contemplate obervation over timed traces or timed refusal sets. Notice that the function *begin* is not defined for the pair of timed traces and timed refusals (s, X). This is because the value of $begin((s,X))$ is equivalent to $begin(s)$ and $begin(X)$. We define a linear addition used to shift the recorded time by a given amount.

$$\begin{aligned} \langle\rangle + t_0 &= \langle\rangle \\ (\langle (t,a) \rangle \frown s) + t_0 &= \langle (t + t_0, a) \rangle \frown (s + t_0) \\ X + t_0 &= \{(t + t_0, a) \mid (t,a) \in X \wedge t + t_0 \geq 0\} \\ (s,X) + t_0 &= (s + t_0, X + t_0) \\ (s,X) - t_0 &= (s,X) + (-t_0) \end{aligned}$$

For the function to return valid traces and refusals it is required that

$$begin(s) + t_0 \geq 0$$

which means that time addition cannot be negative.

A specification is a predicate on observations. For example, a specification S in the timed failure model $\mathcal{M}_{TF}$ is a predicate of the form $S(s,X)$, where s is an arbitrary time trace and X is an arbitrary time refusal set. A program P satisfies a specification

if that specification holds for every observation of an execution of P. In the time failure model, we define a satisfaction relation as follows

$$P \ sat \ S(s, X) \Leftrightarrow \forall (s, X) \in \mathcal{F}_{TF}[\![P]\!] \bullet S(s, X)$$

where $\mathcal{F}_{TF}$ is the semantic function for a timed refusal model.

2.3.1.2 Time Operators The extension to the original CSP syntax was very small. The main alteration was to define a new semantics for all the CSP language in the new timed model. In this section we limit ourselves to show the new timed operators and a formal description of the operators behaviour within the time model presented in the previous section.

- **TimeOut**

$$P \overset{\mathsf{t}}{\triangleright} Q$$

The program will transfer control to Q unless P performs an external event before time t. This is expressed in the next logical expression; *sat* is used to show that a process satisfies the observation set.

$$\frac{\begin{array}{l} P \ sat \ S(s, X) \\ Q \ sat \ T(s, X) \end{array}}{\begin{array}{ll} P \overset{\mathsf{t}}{\triangleright} Q \ sat & (begin(s) \leq t \wedge S(s, X)) \\ & \vee \ (begin(s) \geq t \wedge S(\langle\rangle, X \uparrow [0, t)) \wedge T((s, X) - t)) \end{array}}$$

where $begin(s) \leq t \wedge S(s, X)$ states that any possible observation $S(s, X)$ satisfied by the process P, should begin before time t, while $(begin(s) \geq t \wedge S(\langle\rangle, X \uparrow [0, t)) \wedge T((s, X) - t))$ states that any possible observation $S(\langle\rangle, X \uparrow [0, t))$ satisfied by the process P is an empty trace observation and the refusals should be restricted to the interval $[0, t)$. The observation $T((s, X) - t)$, satisfied by the process Q, should not start earlier than time t.

- **Interrupt**

$$P_t \swarrow Q$$

The program behaves as P until time t, when control is transferred to Q.

$$\frac{\begin{array}{l} P \ sat \ S(s, X) \\ Q \ sat \ T(s, X) \end{array}}{\begin{array}{ll} P_t \swarrow Q \ sat & \exists \, sp, sq \bullet s = sp \frown sq \\ & \wedge end(sp) \leq t \leq begin(sq) \\ & \wedge S(sp, X \uparrow [0, t)) \\ & \wedge T((sq, X) - t) \end{array}}$$

- **Wait**

$$Wait\ \mathtt{t}$$

This program simply refuses all events for time t, and then performs a SKIP. It can also be interpreted as $STOP \swarrow SKIP$ where, $STOP$ is a process that waits forever.

$$\begin{array}{ll} Wait\ \mathtt{t}\ sat & s = \langle\rangle \wedge \surd \notin \sigma(X \uparrow [t, \infty)) \\ & \vee s = \langle (t', \surd) \rangle \wedge \surd \notin \sigma(X \uparrow [t, t')) \wedge t' \geq t \end{array}$$

2.3.1.3 Example To illustrate the use of Timed-CSP, we present the specification of the Watch Dog Timer described in Section 2.2. We first model the Fault Tolerant Router (FTR) responsible for resetting the Watch Dog Timer (WDT). The FTR is modelled as the CSP process *FTR*. It first performs a configuration event which permits it to identify the setting of local processes to be loaded onto the CPU under the responsibility of the FTR. This is done with the help of a configuration table (a sort of hash table) that returns different values for the configuration depending on the available CPUs. The specification of the FTR process is given as

$$FTR_i = configuration_i \rightarrow FTR1_i$$

The index i is used to express the fact that there is more than one FTR process in the OBC of the SACI satellite. Recall that there are three CPUs and, therefore, there are three FTR processes. The next process represents the body of the FTR. It first performs its main task by handling the traffic of messages between the application processes. For simplicity we will consider only *getinput* and *sendoutput* events without showing the details of the communication protocol for this example. A more detailed specification of the FTR can be found in [50]. The FTR I/O operations can be interrupted by an incoming interrupt or a reset signal generated by the WDT. To achieve this the CSP untimed interrupt operator $\hat{}$ is used.

$$\begin{array}{rcl} FTR1_i & = & ((HANDLE\_IO_i)\ \hat{}\ (INTHANDLER_i)) \\ HANDLE\_IO_i & = & getinput_i \rightarrow sendoutput_i \rightarrow HANDLE\_IO_i \\ INTHANDLER_i & = & (wdt\_int_i \rightarrow ((resetwdt_i \rightarrow FTR1_i) \,\Box\, (resetFTR_i \rightarrow FTR1_i))) \\ & & \Box\ (resetFTR_i \rightarrow FTR1_i) \\ & & \Box\ (failed_i \rightarrow STOP) \\ & & \Box\ (failed_{(i \oplus 3)+1} \rightarrow FTR_i) \\ & & \Box\ (failed_{((i+1) \oplus 3)+1} \rightarrow FTR_i) \\ & & \Box\ (otherint_i \rightarrow processint_i \rightarrow resetwdt_i \rightarrow FTR1_i) \end{array}$$

Observe that the $failed_i$ event is raised by the WDT_i indicating that the observed CPU has failed (behaving like STOP). The $failed_{(i \oplus 3)+1}$ and $failed_{((i+1) \oplus 3)+1}$ are issued

to indicate that both neighbor CPUs are considered by its corresponding WDT to have failed.

The WDT behaviour can be represented by a process that waits for a time period (wdt_period) in which it should be reset; otherwise it sends an interrupt to the monitored FTR and wait for another amount of time for a new reset. If it fails to detect a reset signal before the delay period elapses it sends a *resetFTR* event which restarts the FTR. The CPU is permitted to fail its time requirements for seven consecutive times before it is considered to be failed by the WDT, in which case it is cutoff from the system.

$$
\begin{aligned}
WDT_i(j) &= ((COUNTDOWN_i(j)) \;\hat{}\; (resetwdt_i \rightarrow WDT_i(0))) \\
WDT_i(7) &= failed_i \rightarrow STOP \\
COUNTDOWN_i(j) &= Wait \;\; \mathsf{wdt\_period};\; INTCPU_i(j) \\
INTCPU_i(j) &= wdt\_int_i \rightarrow (Wait \;\; \mathsf{int\_period};\; RESET_i(j)) \\
RESET_i(j) &= resetFTR_i \rightarrow WDT_i(j+1)
\end{aligned}
$$

Observe that pattern matching is used in the recursive process $WDT_i(j)$, where j is a parameter that is passed to all the sub-processes composing WDT_i. It simply represents a counter. This is one of the main disadvantages of CSP; the lack of data structures and data types makes it difficult to express some operations. On the other hand the power for expressing control behaviour, the formal semantics and the modularity of CSP are among its main advantages. Complicated issues, such as parallel composition and non deterministic choice, can be expressed and studied easily. Using this power of expression we go further and express the CPU_i process as the parallel composition of the processes FTR_i and WDT_i

$$CPU_i = FTR_i \underset{\{wdt\_int_i,\, resetwdt_i,\, failed_i,\, resetFTR_i\}}{||} WDT_i(0)$$

Where the set $\{wdt\_int_i, resetwdt_i, failed_i, resetFTR_i\}$ is an annotation of the parallel operator, which indicates the synchronization events. Finally, the OBC of the SACI-1 (see Figure 2.1) is expressed as a parallel composition of the three CPUs.

$$SACI1 = CPU_1 \underset{\{fialed_1,\, fialed_2,\, failed_3\}}{\|} (CPU_2 \underset{\{failed_2,\, failed_3\}}{\|} CPU_3)$$

As mentioned previously, the advantage of process algebras over other methods is the modularity of the processes and the facility and agility in the specification of complex systems behaviour. The main drawback in the use of process algebras, CSP specifically, is the lack of structured data types. This has lead researchers in the area of formal methods to propose different extensions to add state and data types to CSP and Timed CSP.

Fischer [19] proposes a combination of CSP and Z[18], namely CSP-Z; the new proposal is a language that uses CSP to describe the behaviour of the system while the specification language Z is used to describe the data types, state and state changes in

the process. In CSP-Z each CSP event is associated to a Z schema that represents the effect of such event occurrence on the state. The semantics of a CSP-Z process is given as the parallel composition of the Z specification and the CSP part of the process.

Mota and Sampaio [39] proposed a technique to use FDR [20] for model checking CSP-Z processes. The challenge in their work is the creation of an abstraction mechanism to reduce the number of states introduced by the data types. More recently, a tool for the automation of the process was presented in [17]. The tool applies the data abstraction technique to reduce the number of states introduced by the Z part; the result is a CSP process that can be analysed by FDR.

In [50], we proposed an extension to CSP-Z. Timed CSP-Z is a language that integrates Timed CSP and Z using the same approach used by CSP-Z. The language was used in the specification of an industrial case study, the SACI1 on-board computer, partially specified above. Rules have been introduced to convert a Timed CSP-Z specification into a special type of Timed Petri Net known as TBNets [21]. The validation of the satellite OBC was carried out using CABERNET [22, 46], a tool for the analysis of TBNets.

Dong and Mahony [33] introduce a language based on the combination of Timed CSP and Object-Z [52]. Timed Communicating Object-Z (TCOZ) models objects as processes and gives the semantics of Object-Z objects, predicates and operations based on the CSP syntax. The disadvantage is that there is no clear separation between the Timed CSP part of the specification and the Object-Z objects. In [42], Qin, Dong and Chin present a semantic model for TCOZ in the Unifying Theories of Programming [26] style. The model was inspired by the proposed discrete time model for the Unifying Theories of Programming presented in the next chapter. In [12], a set of transformation rules are presented to convert a TCOZ specification into a Timed Automata; then a tool, UPPAAL [31], is used to verify the time properties of TCOZ processes.

Circus is the most recent contribution to the family of CSP extensions. In [59, 57] a semantic model for *Circus* is given based on the Unifying Theories of Programming (UTP). Using UTP the semantic of the integration of Z schemas, CSP processes and specification statements were introduced in a natural manner by using the model for reactive and concurrent programs of the UTP. The objective of *Circus* is to be a specification language as well as a programming language, which motivated the inclusion of specification statements as in the refinement calculus [38]. The use of *Circus* was presented in a case study in [58]. Refinement laws for *Circus* are explored in [47, 4]. In the next chapters, we propose a time extension to the *Circus* actions.

2.4 LOGIC APPROACHES

Using logic in system specification allows for clearness and automation of property validation using existing theorem proving tools. Any logic to be used in the specification of a real-time system needs a suitable extension in its validation mechanism.

Modal logic [7] is a possible extension to add time reasoning in logical formulae. In modal logic the logical formulae are not evaluated to be true or false in a single world but use a whole set of worlds W for the evaluation of the formulae. An evaluation function V is used to evaluate a formulae in each element of the world set W following

an ordering relation R. The relation R is also called the reachability relation and $w_1 R w_2$ means that w_1 is directly reachable for w_2.

Temporal logic [34] is a modal logic in which the world W is the set of all possible time instances of a particular time domain and the precedence relation $<$ is the reachability relation. New operators are added to quantify over the new world. The operators are used for quantification either in the future or in the past. The operators to reason about the future are *always* and *eventually* denoted as $\Box$ and $\Diamond$, respectively. The corresponding operators to reason about the past are $\boxminus$ and $\blacklozenge$, respectively.

The evaluation function V evaluates a formula f at time t and is defined as follows

$$\begin{array}{lll} V(\Box f, t) & \textit{iff} & \forall\, t'(t < t' \Rightarrow V(f, t')) \\ V(\Diamond f, t) & \textit{iff} & \neg\, V(\Box(\neg f), t) \\ V(\boxminus f, t) & \textit{iff} & \forall\, t'(t' < t \Rightarrow V(f, t')) \\ V(\blacklozenge f, t) & \textit{iff} & \neg\, V(\boxminus(\neg f), t) \end{array}$$

The specification of real-time systems using temporal logic is based on the use of formulae to describe the events and the order in which they occur. The great disadvantaging of temporal logic is the lack of modularity. Usually temporal logic is used in combination with other formal techniques to extend them with time reasoning tools. An example of using Temporal logic with other formalisms is the work proposed by Duke and Smith [13] that uses temporal logic in the invariants of the Z specification language with the objective of adding liveness properties to the Z specification. Some extensions to temporal logic propose to add new notations to the logic; an example of such an approach is presented by Lamport in [30]. The extension proposed by Lamport adds the concept of actions to the classical temporal logic such that $[A]$ denotes an action defined using state predicate similar to Z. In such actions undecorated variables represent the state before the action and decorated variables represent the state after the action execution. The following is an example using Lamport's formalism

$$(x = 0 \wedge y = 0) \wedge \Box[x' = x + 1 \wedge y' = y + 2x + 1] \Rightarrow (y = x^2)$$

The above formula states that for an initial value of x and y equal to zero, accompanied by a steady increment by one in the value of x and the value of y' is given by $y + 2x + 1$, implies in $y = x^2$.

Duration calculus (DC) [6] extends temporal logic over intervals instead of time instances. It is based on finite continuous intervals and time instance formulae. In duration calculus, a system state is modelled as a boolean function on time instances. State functions can be combined to form more complex formulae. The duration of a state S over an interval is given by $\int S$ and is defined as the integral of S over the interval. The length of an interval is given by $\int 1$ also abbreviated as l. Formulae in DC can be combined with the *chop* operator $\frown$. Given two formulae F_1 and F_2 then $F_1 \frown F_2$ is a formula that is true in a given interval i if and only if the interval i can be split into two adjacent intervals i_1 and i_2 such that F_1 is true over the first interval i_1, and F_2 is

true over the second interval i_2. The classical operators $\Box$ and $\Diamond$ of temporal logic are redefined to operate over intervals such that

$$\Diamond F \mathrel{\widehat{=}} true \frown F \frown true \tag{2.4.1}$$

$$\Box F \mathrel{\widehat{=}} \neg\Diamond\neg F \tag{2.4.2}$$

Real-time logic (RTL) [27] takes a different approach to integrating time information to logic. The RTL extends the predicate logic by relating the events of the system with the time in which they occur. In the next section, we give a detailed description of RTL.

2.4.1 Real-time Logic

Introduced by Jahanian and Mok in 1986 [27], RTL is a first-order logic with predicates which relate the events of the system to their time of occurrence.

This formalism uses an event action model and then defines timing constraints on this model. In contrast to other forms of temporal logic, RTL allows specification of the absolute timing of events and not only their relative ordering. It also provides a uniform way of incorporating different scheduling disciplines [27].

In this model, events serve as temporal marks and do not require any time. On the other hand, actions have bounded non-zero duration. The occurrence of an event defines a time value named its *time of occurrence*. The execution of an action is denoted by two events, one representing the start of an action and the other the end of the same action.

2.4.1.1 Classes of Events There are basically two classes of events.

- Start/Stop. These events are used to denote the start and end of an action, and are referred to by <action>↑ (the start event), and <action>↓ (the end event), where <action> is the action name.
- Transition Events. A state variable describes a physical aspect of the system, e.g. switch ON or OFF. The execution of an action may cause one or more of these state variables to change. Each state variable has an attribute which is within the domain of values for the states represented by these variables. For instance, in the case of a switch represented by the state variable S, $S := T$ represents the state *Switch ON*, and S:=F represents the *Switch OFF* state. A transition event occurs when the attribute of a state variable changes.

Other types of events can be added as in [11], where a class of external events has been introduced. These events are preceded by Ω. They are not generated by the computer system but by an external stimuli, like a button being pressed by the system operator. All the events of a system constitute its event set D.

2.4.1.2 Occurrence Relation This relation, expressed by the letter Θ, holds between the event, the occurrence order of this event, and the time of the occurrence in the observed system. This is expressed as

$$\Theta(e, i, t)$$

where e is the event, i is a non-zero positive integer referring to the i^{th} occurrence of the event e, and t is a positive natural number denoting the time of the event occurrence. The relation can be expressed formally as a relation on the set

$$E \times Z^+ \times N$$

where E is a set of events, Z^+ is the set of positive integers, and N is the set of natural numbers such that the following axioms hold:

- Monotonic Axioms: For each event e in the set E:

$$\forall\, i \forall\, t \forall\, t'[\Theta(e,i,t) \wedge \Theta(e,i,t')] \rightarrow t = t'$$
$$\forall\, i \forall\, t[\Theta(e,i,t) \wedge i > 1] \rightarrow \exists\, t'\Theta(e,i-1,t') \wedge t' < t$$

- Start/Stop Event Axioms: For each pair of start/stop events in the set E:

$$\forall\, i \forall\, t\Theta(A\downarrow, i, t) \rightarrow \exists\, t'\Theta(A\uparrow, i, t') \wedge t' < t$$

- Transition Event Axioms: For the transition events in the set E corresponding to a state variable S:

$$\Theta((S := T), 1, 0) \rightarrow$$
$$(\forall\, i \forall\, t\Theta((S := F), i, t) \rightarrow \exists\, t'\Theta((S := T), i, t') \wedge t' < t \wedge$$
$$(\forall\, i \forall\, t\Theta((S := T), i+1, t) \rightarrow \exists\, t'\Theta((S := F), i, t') \wedge t' < t))$$

$$\Theta((S := F), 1, 0) \rightarrow$$
$$(\forall\, i \forall\, t\Theta((S := T), i, t) \rightarrow \exists\, t'\Theta((S := F), i, t') \wedge t' < t \wedge$$
$$(\forall\, i \forall\, t\Theta((S := F), i+1, t) \rightarrow \exists\, t'\Theta((S := T), i, t') \wedge t' < t))$$

From the above relation we can derive a function @, which, given an event and an occurrence, returns the time stamp of that occurrence.

$$@(e, i) = t$$

2.4.1.3 State Predicate A real-time system is usually specified referring to the properties of the system over time. RTL offers the notion of state predicates to define the time interval in which a state variable attribute is unchanged. Suppose S is a state variable whose truth value remains unchanged over an interval depending on the boundaries of the interval. RTL provides six forms of interval boundaries expression. Let x and y be time variables which represent the initial and the final time values for the interval. Let E_t be the transition event that makes S true and E_f the transition event that makes S false. Then the boundaries can be expressed informally as

[x denotes that E_t occurs at time x,
(x denotes that E_t occurs after or at time x,
$< x$ denotes that E_t occurs after time x,
y] denotes that E_f occurs at time y,
y) denotes that E_f occurs before or at time y,
$y >$ denotes that E_f occurs before time y.

For example, the predicate S[x,y] denotes that the state variable S is true within the interval between x and y.

2.4.1.4 Example The Watch Dog Timer example previously used to illustrate Timed CSP, is also used to show the application of RTL formalism.

Observe from the time model and the language itself that it contains no operators for parallel composition and no modularity either. This makes the language more adequate for requirement specifications. We will present the timing restrictions of our system based on the RTL for the WDT example.

The first equation shows that in case an *InterruptWDT* event occurs at a given time t then the event *resetWDT* should not occur in the period of length *WDTperiod* before t.

$$\text{Req1.}\ \ \forall\, i \forall\, t \Theta(\mathit{InterruptWDT}, i, t) \rightarrow \neg \exists\, t' \neg \exists\, j \Theta(\mathit{resetWDT}, j, t') \wedge t > t' + \mathit{WDTperiod}$$

The next statement affirms that if a *resetFTR* occurs, within the period before the occurrence of the event for the length of *Intperiod*, there should not exist any occurrence of the event *AcknowledgeInt*. Observe that the event *resetWDT* in this case is substituted by the new event *AcknowledgeInt* to distinguish between the reset signal that the WDT waits for the first time and the reset signal that it waits for after the interrupt.

$$\text{Req2.}\ \ \forall\, i \forall\, t \Theta(\mathit{resetFTR}, i, t) \rightarrow \neg \exists\, t' \neg \exists\, j \Theta(\mathit{AcknowledgeInt}, j, t') \wedge t < t' + \mathit{Intperiod}$$

The number of failures can be monitored by the occurrence variable of the model. The next statement affirms that if the seventh occurrence of the *resetFTR* event happens at time t then the event *failed* should occur at time $t + 1$ (i.e. immediately after).

$$\text{Req3.}\ \ \forall\, t \Theta(\mathit{resetFTR}, 7, t) \rightarrow \exists\, t' \Theta(\mathit{failed}, 1, t') \wedge t' = t + 1$$

The last statement shows that when the system fails, indicated by the *failed* event, the WDT should not respond to any events anymore.

$$\text{Req4.}\ \ \forall\, t \Theta(\mathit{failed}, 1, t) \rightarrow \neg \exists\, e \neg \exists\, i \neg \exists\, t' \Theta(e, i, t') \wedge t' > t$$

The statements above show clearly the timing requirements that a CPU must meet. However, no modularity that separates the WDT behaviour from the FTR behaviour can be observed. This is because the language lacks this type of expression capacity.

For this reason we will limit ourselves to show the requirements of the system for only one CPU. For further information regarding RTL see [27].

The following expressions describe the FTR behaviour.

$$\forall\, i \forall\, t \Theta(SendOutput, i, t) \rightarrow \exists\, t' \Theta(GetInput, i, t') \wedge t' < t$$
$$\forall\, i \forall\, t \Theta(ProcessInt, i, t) \rightarrow \exists\, t' \Theta(OtherInt, i, t') \wedge t' < t$$
$$\forall\, i \forall\, t \Theta(resetWDT, i, t) \rightarrow \exists\, t' \Theta(ProcessInt, i, t') \wedge t' < t$$

The WDT behaviour can be expressed by means of the following predicates.

$$\forall\, i \forall\, t \Theta(Acknowledge, i, t) \rightarrow \exists\, j \exists\, t' \Theta(Interrupt, j, t') \wedge t' < t + Intperiod$$
$$\forall\, i \forall\, t \Theta(resetFTR, i, t) \rightarrow \exists\, j \exists\, t' \Theta(Interrupt, j, t') \wedge (t' + Intperiod) < t$$
$$\forall\, i \forall\, t \Theta(resetWDT, i, t) \rightarrow \neg \exists\, t' \neg \exists\, j \Theta(Interrupt, j, t') \wedge t' < t + wdtperiod$$

The logical based approaches have a solid mathematical foundation and the advantage that theorem proving tools can be used to study the system properties expressed in the logic. All logics suffer from the absence of modular structures which makes them difficult to use in large and complex industrial systems.

2.5 PETRI NETS BASED APPROACH

Petri Nets allow mathematical modelling of discrete event systems in terms of conditions and events, and the relationship between them [40]. Petri Nets are among the oldest and most used techniques for the specification of computer systems. Since the introduction of Petri Nets a large variation of them can be found today.

Petri Nets are based on a weight directed graph, with two types of nodes called *places* and *transitions*. Graphically places are denoted as circles and transitions as rectangles. The edges of the graph are either from a place to a transition or vice versa. A marking assigns each place a non-negative integer value. The values assigned to places refer to tokens that appear as bullets in the graph. The weight on the arcs defines the number of tokens removed or inserted in a place upon the firing of a transition. A transition is enabled to fire when all places connected to the transition contain the number of tokens equivalent to the number on the edge connecting it to the transition. Once a transition is fired the places connected from the transition receive a number of tokens equivalent to the number marked on the edge of the transition. Consider the Petri Net in Figure 2.4. The places P1 and P2 are marked with a token each, while the places P3 and P4 are not marked. The arcs from P1 to T1 and from P2 to T2 have the weight of 1; this makes both the transitions T1 and T2 enabled to fire. Based on the firing semantic selected both transitions can fire simultaneously or alternatively. Both transitions generate a single token in the places P3 which is connected to transition T3 by an edge with weight 2, indicating that transition T3 is only enabled if two tokens are placed in P3.

Time information can be added to Petri Nets in a number of forms. The most common approach is to add time delays to transitions. Therefore, each transition is assigned a delay time that once it is enabled to fire the timer starts a count down until it reaches zero; only then the transition is fired. A similar approach assigns delays to

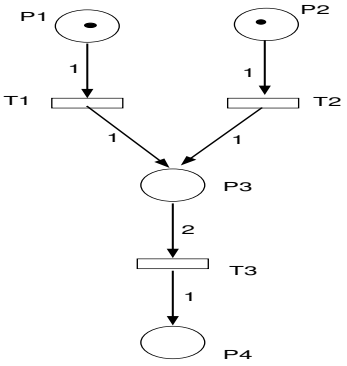

Figure 2.4. Example of a basic Petri Net.

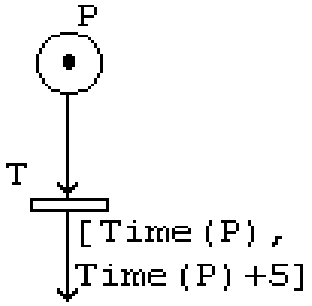

Figure 2.5. A Way of Representing Time Constraints in TBNet.

places instead of transitions, and creates a delay between the time the token arrives in a place and the time it enables a transition to fire.

A more flexible approach assigns intervals to the transitions and time stamps to the tokens. Such an approach is used by Time Basic Nets explored in details in the next section.

2.5.1 Time Basic Nets (TBNet)

Time Basic Nets (TBNet), introduced in [21], represent a different notation of time in Petri Nets.

The basic idea of TBNet is that each location (token) holds its creation time, i.e., the firing time of the transition that created it. Then a time condition is associated to each transition from that location. The time condition is given as a closed interval of some or all time stamps of tokens in the input places of the transition. In Figure 2.5 the term $Time(P)$ is used to denote the latest time stamp associated with a token in place P. An assumption is made in [21] to use closed intervals to represent the minimum and maximum time of a transition firing. The example in Figure 2.5 expresses that the transition T may fire in the interval starting from the moment P is armed up to 5 time units later; otherwise it will never fire.

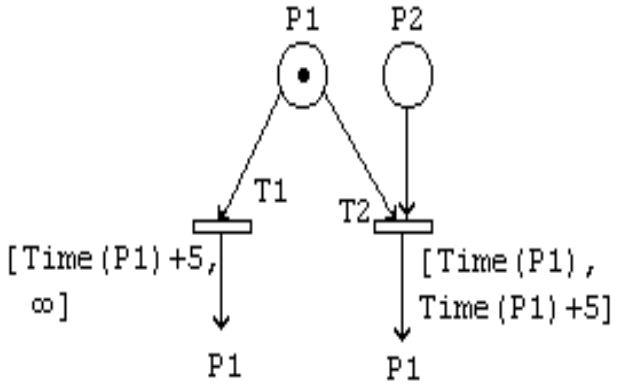

Figure 2.6. More Complex Constraints Using TBNet.

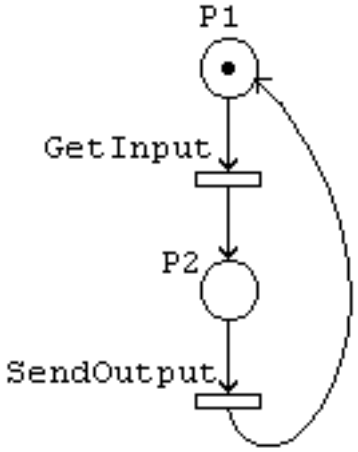

Figure 2.7. FTR Input/Output Operation in TBNets.

A transition is enabled to fire (but not necessarily will fire) if there is at least one token in each of its input places (a weight of 1 is assumed for each arc of the net), and the firing time is within the boundaries of the associated interval. This simple form of expressing time can be used to express more complex behaviours, such as the one represented in Figure 2.6. It shows a system that waits for an input (waits for place $P2$ to arm) for a determined time, and fires transition $T2$ if armed. If the input does not occur within this period then it should take another action (represented by transition $T1$). Observe that the two intervals in the previous example overlap at time (Time(P1)+5). In this case (at this time) and assuming $P2$ was armed, the choice is non-deterministic.

2.5.1.1 Example The next example is the same used to illustrate the two previous methods. As general Petri Nets (and TBNet in particular) offer no mechanisms for modularization, we break our example into several small nets. The first subnet is the FTR process, which has the main task of responding to input messages from source

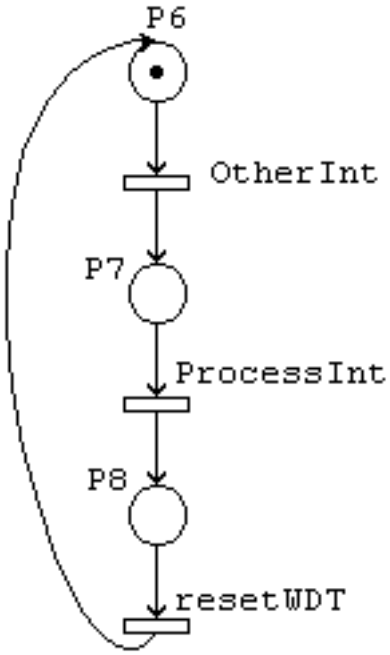

Figure 2.8. FTR Processing Other Interrupts in TBNets.

application processes and route these to the destination application process. This is expressed as shown in Figure 2.7. Observe that no time restrictions are needed as none of the timing requirements of the system are involved in the FTR operations. Transition *GetInput* represents the operation of receiving a message from an application program to be routed to another application on the same CPU or on another CPU. The routing process is represented as transition *SendOutput*. For simplicity we omit the routing process; details can be found in [50].

Figure 2.8 illustrates the operation of handling an interrupt by the FTR and the action associated to this interrupt. This interrupt does not include the WDT interrupt. Observe that after the interrupt is processed the FTR will fire a *resetWDT* transition which affects the WDT.

In Figure 2.9, the use of *WDTInterrupt*$'$ and *Acknowledge*$'$ as transition names to distinguish these transitions from the WDT transitions *WDTInterrupt* and *Acknowledge*. Next, we will illustrate the WDT operation. This is done in one single net (Figure 2.10). Observe that all timing requirements are managed by this process.

The timing restrictions for the network are the following:

$$\begin{aligned} WDTreset &= [time(P3), time(P3) + wdtperiod] \\ WDTInterrupt &= [time(P3) + wdtperiod, \infty] \\ Acknowlage &= [time(P4), time(P4) + intperiod] \\ ResetFTR &= [time(P4) + intperiod, \infty] \end{aligned}$$

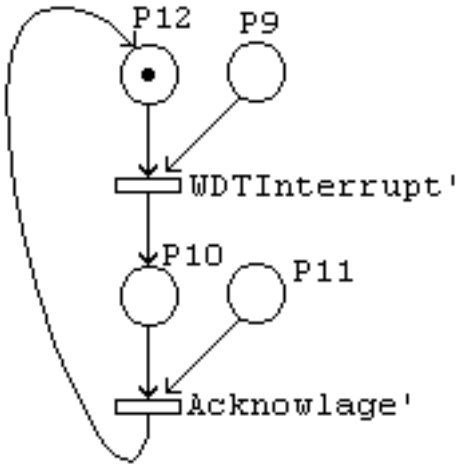

Figure 2.9. FTR Processing the WDT Interrupt.

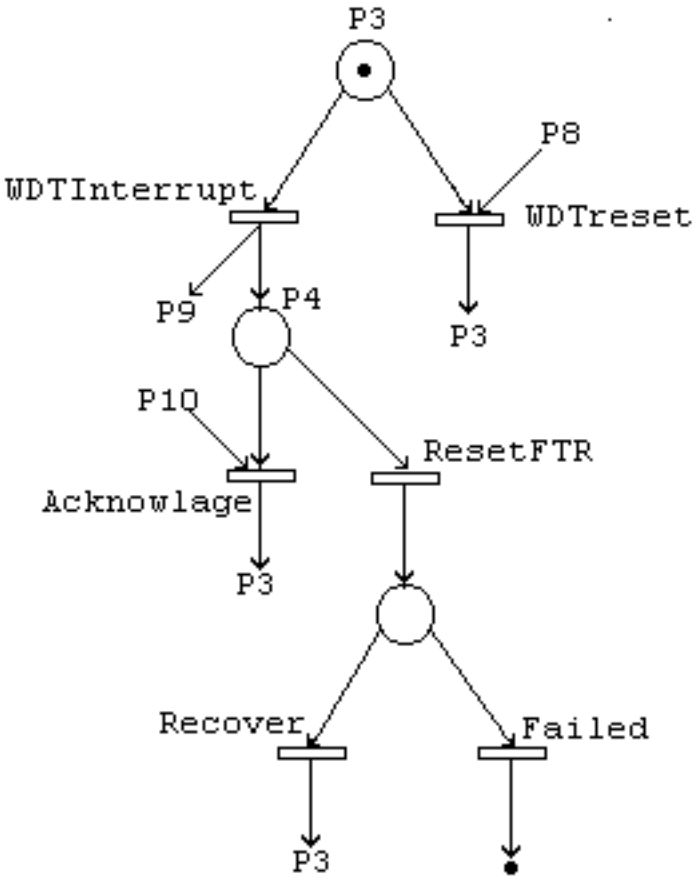

Figure 2.10. WDT Process Behaviour.

2.6 FINAL CONSIDERATIONS

In the previous sections, we studied three of the various methods for real-time system specification. Each one has advantages and drawbacks. The variety of methods gives some evidence of the need of formal methods in real-time systems, but also shows that none of the available formalisms seems to be suitable for all the types of real-time systems.

Some, such as Timed CSP, have shown to be suitable for the specification of complex systems and in advanced stages of the development, such as the design stage. This is due to the modularity and power of expression. Timed CSP, however, has the disadvantage of not having a form of expressing data types and variables, as shown in the previous example, in which to model a simple counter a recursive parameterized process was needed. Other methods such as RTL and TBNets have shown to be more adapted for requirement specification.

The language to be used in the specification of a system depends on the nature of the system and within which stages of the system development will the formal specification be used. For instance, if parallel composition and distributed processes communication via synchronized channels are relevant and absolute timing restrictions are not important, then Timed CSP is a good alternative. But if on the other hand the use of formal methods is simply to show the timing restrictions of the system at the requirements level, without much consideration of the details of the systems behaviour, then RTL or TBNet are a better alternative.

CHAPTER 3

CIRCUS TIME ACTION MODEL

This chapter introduces the language *Circus* Time Action, which is based on the *Circus* action notation. We add two time operators, give the resulting language a timed model, and explore its healthiness conditions. The semantic models of *Circus* and *Circus* Time Action are based on the Unifying Theories of Programming (UTP) [26], we compare these models when possible. The semantics of the language operators is given in the new model, and algebraic properties of the language are established.

The chapter is organized as follows. In the next section, we give a brief description of the Unifying Theories of Programming. Section 3.2 gives the syntax and an informal description of the *Circus* Time Action constructs. Section 3.3 shows the semantic variables used by the timed model and how they differ from the UTP semantic model. Section 3.4 gives the healthiness conditions of the new timed model and compares them with the original healthiness conditions of the UTP. Finally Section 3.5 provides the semantic of each of the language constructs, shows the difference between the timed language semantics and the original semantics of the UTP, and explores the algebraic properties of the new language.

3.1 UNIFYING THEORIES OF PROGRAMMING

The Unifying Theories of Programming by Hoare and He [26] uses the theory of relations as a unifying basis for representing programs across many different computational paradigms. Each computational paradigm has its own characteristics but shares some characteristics with other paradigms. For example, a sequential program is a special case of a parallel program, yet the parallel programming paradigm has its own characteristics which cannot be found in the sequential paradigm.

A consolidated basis for the specification of different paradigms is given in Unifying Theories of Programming. Specifications are all interpreted as relations between an initial observation and a single subsequent (intermediate or final) observation of the behaviour of a device executing a program. Different theories define composition of programs as relational composition, nondeterminism as disjunction, recursion is given as weakest fixed point, and both correctness and refinement are interpreted as inclusion of relations.

Actually, in the UTP, not only specifications, but all notations of a programming language are defined as relations and operations between relations. This denotational semantics can be related to an algebraic semantics, which presents the laws of the denotational semantics as axioms rather than theorems. The collection of laws are expected to be sufficiently powerful to derive a step relation for an operational semantics. In the UTP, the aim is that each style of semantic presentation can be derived from any of the others by mathematical construction and proof.

Within the UTP general theory, paradigms and programming languages are differentiated by their *alphabet*, *signature*, and a selection of laws known as *healthiness conditions*. The *alphabet* of a theory gives names for external observations of program behaviour. Similar to the Z notation, the UTP uses the convention that the name of an initial observation is undecorated, but the name of a similar observation taken subsequently is decorated with a dash. The signature of a theory provides a syntax for denoting the objects of the theory. The *healthiness conditions* select the valid objects of a subtheory from those of a more expressive theory. The healthiness conditions differentiate the theories in the unification base from the more general theories.

The *alphabet* of each theory is determined from the observations considered relevant to programs in a particular programming paradigm. For example, in the reactive paradigm, a boolean variable *wait* distinguishes an intermediate observation from one of a terminated program. In a theory of communicating processes, the variable *tr* is a sequence of events that records the interactions between a process and its environment. In all theories, the boolean variable *ok* records whether the system has been properly started in a stable state, and ok' records subsequent stabilization in an observable state. This permits a description of programs that fail due to nonterminating recursion.

A healthiness condition filters feasible descriptions from those that are not. There are healthiness conditions associated with each external variable, and with groups of related variables. For example, if a program P has not started, observation of its behaviour is impossible, and only a vacuous prediction can be made.

$$P = ok \Rightarrow P$$

In the example above, if $(ok = true)$ then the program P started and the result is the behaviour of the proper program P. Otherwise if $(ok = false)$ then its behaviour is unpredictable as its result the predicate *true*.

This gives a brief description of the UTP; we will further explore the model in later sections.

3.2 CIRCUS TIME ACTION: INFORMAL DESCRIPTION

Figure 3.1 presents the BNF description of the syntax of *Circus* Time Action. In this figure, b stands for a predicate, e stands for any expression, t stands for a positive integer expression, N for any valid name (identifier), s for a set of variable names, and CS for a set of channel names.

A *Circus* Time Action program is formed of one single action. An action can be basic or a combination of one or more actions. *Skip* is a basic action that terminates immediately. *Stop* represents deadlock, which simply puts a program in an ever waiting state. *Wait* t puts the program in a waiting state for a period of time determined by the positive integer t. *Chaos* is the worst action; nothing can be said about its behaviour.

An action can be prefixed with a communication (input ?N or output !e or .e) which takes place before the action starts. This action waits for the other actions that need to synchronize on the channel before the communication can take place. In b & A, b is a guard: a boolean expression that has to be *true* for the action A to take place; otherwise A cannot proceed.

$$
\begin{array}{ll}
\mathsf{Action} & ::= \mathit{Skip} \\
& \mid \mathit{Stop} \\
& \mid \mathit{Wait}\ \mathsf{t} \\
& \mid \mathit{Chaos} \\
& \mid \mathsf{Communication} \rightarrow \mathsf{Action} \\
& \mid \mathsf{b}\ \&\ \mathsf{Action} \\
& \mid \mathsf{Action} \sqcap \mathsf{Action} \\
& \mid \mathsf{Action} \Box \mathsf{Action} \\
& \mid \mathsf{Action};\ \mathsf{Action} \\
& \mid \mathsf{Action} \ [\![\mathsf{s}_1 \mid \{\!| \mathsf{CS} |\!\} \mid \mathsf{s}_2]\!]\ \mathsf{Action} \\
& \mid \mathsf{Action} \setminus \mathsf{CS} \\
& \mid \mathsf{Action} \overset{\mathsf{t}}{\rhd} \mathsf{Action} \\
& \mid \mathsf{N} := \mathsf{e} \\
& \mid \mathsf{Action} \lhd b \rhd \mathsf{Action} \\
& \mid \mu\, \mathsf{N} \bullet \mathsf{Action} \\
\mathsf{Communication} & ::= \mathsf{N}\ \mathsf{CParameter}^* \\
\mathsf{CParameter} & ::= ?\,\mathsf{N} \mid !\,\mathsf{e} \mid .\,\mathsf{e}
\end{array}
$$

Figure 3.1. Circus Time Action syntax

The internal choice $\mathsf{A} \sqcap \mathsf{B}$ selects A or B in a nondeterministic manner. The external choice $\mathsf{A} \Box \mathsf{B}$ waits for interaction with the environment; the first action that interacts with the environment (by either synchronizing on an event or terminating) is chosen. The sequential composition $\mathsf{A};\ \mathsf{B}$ behaves as A followed immediately by B.

The parallel composition of two actions ($\mathsf{A} \ [\![\mathsf{s}_1 \mid \{\!| \mathsf{CS} |\!\} \mid \mathsf{s}_2]\!]\ \mathsf{B}$) involves a set of channels (CS) containing the events on which they need to synchronize; the sets s_1 and s_2 are disjoint sets of variable names that each action can change. The program variables that do not appear in the set associated with the action cannot be changed by it; however, all variables can be accessed and the value they contain is the initial value before the parallel composition.

A hiding operation ($\mathsf{A} \setminus \mathsf{CS}$) also takes a set of channels (CS). The set is to be excluded from the resulting observation of the action communications; hidden channels can no longer be seen by other actions.

The timeout construct ($\mathsf{A} \overset{\mathsf{t}}{\rhd} \mathsf{B}$) takes a positive integer value as the length of the timeout; it acts as a time guarded choice. The behaviour of the resulting action is either A or B. If A performs an observable event or terminates before the specified time elapses, it is chosen. Otherwise, A is suspended and the only possible observations are those produced by B.

Assignment is a command; it simply assigns a corresponding list of values to a list of variables in the current state. If the variable already exists, its value will be overwritten; otherwise it will be added to the current state and assigned the respective value. The conditional command $\mathsf{A} \lhd b \rhd \mathsf{B}$ associates two actions with a predicate b. If the b evaluates to *true* then the action A is chosen, otherwise the action B is chosen. Finally,

$\mu \mathsf{N} \bullet \mathsf{A}$ defines a recursive action N; any reference to the action N within the body stands for a recursive call.

3.3 THE SEMANTIC MODEL

The first question that has to be answered is: what time model we would like to have, discrete time or continuous time? In principle, the continues time model seems to be more appropriate because it has the power to express time in both forms, and that time in the real world is continuous. However, it cannot be implemented by a software system. On the other hand, a discrete model can be implemented as a piece of software. As we follow the objective of *Circus* to be a programming language as well as a specification language, we adopt the discrete time in *Circus* Time Action. Using the discrete model in *Circus* Time Action, the original untimed refinement rules can be extended in a natural way. With the objective of making our time model a conservative extension of the existing model of *Circus*, we have made a choice for the discrete model.

Similar approaches, such as those in [32, 28], use Extended Duration Calculus (EDC) to add continuous time to a language semantics. Both works clearly show the elegance and powerful expression capacity of the EDC formulas. Nevertheless, both approaches make it clear that the new model cannot be easily related to the original untimed model. Proving properties in the new model is a tedious task.

In the *Circus* model, which is based on the Unifying Theories of Programming, a reactive system behaviour can be studied with a set of observations; each observation in the set is determined by the values of a set of observation variables: the alphabet of the model. The alphabet variables are divided into two types: initial observation variables (with undecorated names) record the state of the environment before the program starts, and the final observation variables (with decorated names) record the state of the environment at the moment the program reaches a stable state. A stable state is either a termination state or a non-termination state in which the program is waiting for an interaction with its environment [26]. If a program diverges, then it will not reach a stable state.

Final observation variables register the interaction of the program with its environment during and at the point of observation. This observation is registered in the form of a sequence of events that show the order in which the events occurred, and a set of refusals which indicate the events the program can refuse at the observation point.

In our work, we use a similar set of observation variables, which registers the initial and stable state of the actions; they are, however, enriched with time information. The interaction with the environment is recorded as a sequence of tuples, each element of the sequence denoting the interaction of a program with its environment over a single time unit, where discrete time units are represented by the sequence indices themselves. The first component of the tuple is a sequence of events which occurred during the time unit. The second component is the set of refused events at the end of the time unit.

The following is a formal description of the observation variables used by our model.

- ok and ok' are boolean variables. When ok is *true*, it states that the program started, and when ok' is *true* it indicates that the program is in a stable state: it

did not diverge.

$$ok, ok' : Boolean$$

- *wait* and *wait′* are also boolean variables. When *wait* is *true*, we are in an intermediate state of a preceding program. When *wait′* is *true*, the program has not reached a termination state; when it is false, it indicates a final (termination) observation.

$$wait, wait' : Boolean$$

- *state* and *state′* are mappings from variable names to values. This mapping associates each user variable in the program to a value. We use this notation instead of the arbitrary list of variables used by the UTP.

$$state, state' : N \rightarrow Value$$

  The dashed variable represents the state of the program variables at the final observation. We find this form easier to use and separates clearly the user variables from the alphabet used to give the semantics of the language.

- tr_t and tr'_t. The interaction of a program with its environment, in the UTP model, is captured with the aid of the observation variables *tr* and *tr′*: the sequence of events that occurred before the program started (*tr*) and the sequence of events that occurred before and during the program execution (*tr′*). In a similar manner in our time model, tr_t records the observations that occur before the program starts, and tr'_t records the final observations. Although the observation variables tr_t and tr'_t have the same meaning as in the UTP, they have a different structure and capture more information. Time trace is a sequence in which each element of the sequence represents an observation over one time unit. Each observation is a tuple, where the first element is the sequence of events that occurred at the end of the time unit, and the second is the associated set of refusals at the end of the same time unit.

$$tr_t, tr'_t : \operatorname{seq}^+(\operatorname{seq} Event \times \mathbb{P}\, Event)$$

  Event contains all possible events of a program. The sequences tr_t and tr'_t are defined to be nonempty (seq^+). We use a sequence with an index that starts with 0; the first element represents the initial observations of the program in the state in which it started and so it must include at least that element. We need to register this type of information because communication can take place without consuming any time. Some time models such as Timed CSP, shown in the previous chapter, impose the condition that communication should consume a minimum amount of time. This condition is to avoid the problem of registering infinite traces without time passing. Because *Circus* and, similarly, *Circus* Time Action use finite traces then this condition can be relaxed. Another reason for imposing the condition of

minimum communication time is to make the specification of the real time system realistic, since all actions in a real time system consume time. Because we use a discrete time model we can only make observations at fixed intervals and the exact time in which the event actually occurred is irrelevant in our model. However, the observations of our model can be more accurate and realistic if we choose a smaller step to capture time. Consider the following CSP time trace

$$\langle (a, 0.25), (b, 1.0) \rangle$$

Now if we take the step in the discrete model to be equal to 1 time unit, then the previous trace would be represented in our time traces as

$$\langle (\langle\rangle, r_1), (\langle a, b\rangle, r_2) \rangle$$

where r_1 and r_2 are the corresponding refusal set. The only thing we can observe is that the event a occurred before the event b. However, we lost the information that it occurred at time 0.25. On the other hand, consider again the same example but now using a step of 0.25 for each time unit. In such case, the following is the equivalent discrete time trace

$$\langle (\langle\rangle, r_1), (\langle a\rangle, r_2), (\langle\rangle, r_3), (\langle\rangle, r_4), (\langle b\rangle, r_5) \rangle$$

where each time unit represents 0.25 real time units. Therefore, the discrete time model depends on the size of the time step used.

- $trace'$ is a sequence of events that occurred since the last observation. In this observation we are interested in recording only the events without time.

$$\begin{array}{lcll} trace' & : & \operatorname{seq} Event & \\ trace' & = & Flat(tr'_t) - Flat(tr_t) & (3.3.1) \end{array}$$

where $Flat$ maps time traces to untimed traces, as defined in the following

$$Flat : \operatorname{seq}^+(\operatorname{seq} Event \times \mathbb{P}\, Event) \rightarrow \operatorname{seq} Event$$

$$\begin{array}{rcl} Flat(\langle (el, ref) \rangle) & = & \langle el \rangle \\ Flat(S \frown \langle (el, ref) \rangle) & = & Flat(S) \frown el \end{array}$$

From the definition of $Flat$, given two time traces t_a and t_b, it satisfies the following properties

Property 3.1

L1. $Flat(t_a) \frown Flat(t_b) = Flat(t_a \frown t_b)$

L2. $Flat(t_a) = fst(head(t_a)) \frown Flat(tail(t_a))$

L3. $Flat(t_a) = \langle\rangle \Leftrightarrow \forall\, i : 1..\#t_a \bullet fst(t_a)(i) = \langle\rangle$

L4. $t_a \leq t_b \Rightarrow Flat(t_a) \leq Flat(t_b)$

L5. $t_a = t_b \Rightarrow Flat(t_a) = Flat(t_b)$

A single observation is given by the combination of the above variables. We will define our programs as predicates over the observation variables. Before we define the semantics of our language, however, we define a set of conditions that need to be satisfied by all observations; such conditions are known as *healthiness conditions*.

3.4 HEALTHINESS CONDITIONS

In this section, we explore some additional algebraic properties that are satisfied by all *Circus* Time Action programs. A predicate that satisfies these properties is called *healthy*, and the properties are called *healthiness conditions*.

There are often intuitive reasons why programs satisfy a given healthiness condition. For example, it would be unusual for a program to make time go backwards or change the history of what happened before it started. Depending on the domain of the language, different healthiness conditions are relevant. *Circus* Time Action satisfies the same healthiness conditions defined in [26] for CSP processes, with some additional considerations regarding time.

Healthiness conditions are defined using idempotent functions on predicates. Given a healthiness condition H, we say that a program P is *H-Healthy* if

$$H(P) \equiv P$$

The first of these healthiness conditions is $R1$; it requires that a program can never change the traces that occurred before it started. The condition $R1$, in UTP, is defined as follows.

$$R1(X) \;\widehat{=}\; X \wedge tr \leq tr' \tag{3.4.1}$$

In the same way, the condition $R1_t$ states that execution of a program X can never undo any action performed previously; the time traces can only be expanded. This condition also imposes that the program cannot make time go backward, and new traces cannot be shorter than the traces registered prior to the program execution.

$$R1_t(X) \;\widehat{=}\; X \wedge Expands(tr_t, tr'_t) \tag{3.4.2}$$

We define a relation *Expands* between two timed traces.

$$Expands(tr_t, tr'_t) \;\widehat{=}\; (front(tr_t) \leq tr'_t) \wedge (fst(last(tr_t)) \leq fst(tr'_t(\#\ tr_t))) \tag{3.4.3}$$

Given two timed traces tr_t and tr'_t we state that tr'_t expands tr_t if, and only if, the initial part tr_t is a prefix of tr'_t, and the untimed trace registered at the last time unit of tr_t is a prefix of the trace registered at the same time in tr'_t. For any two time traces t_a and t_b such that $Expands(t_a, t_b)$, the following properties are satisfied

Property 3.2

L1. $Expands(t_a, t_b) \Rightarrow (Flat(t_a) \leq Flat(t_b))$

L2. $(t_a \leq t_b) \Rightarrow Expands(t_a, t_b)$

L3. $(t_a = t_b) \Rightarrow Expands(t_a, t_b)$

L4. $((Flat(t_a) = Flat(t_b)) \wedge Expands(t_a, t_b)) \Rightarrow (t_a = t_b)$

L5. $(t_a = t_b) \Rightarrow (last(t_a) = last(t_b))$

L6. $(t_a \leq t_b) \Rightarrow (last(t_a) = t_b(\#t_a))$

L7. $Expands(t_a, t_b) \wedge Expands(t_b, t_c) = Expands(t_a, t_c)$

L8. $Expands(\langle(\langle\rangle, ref)\rangle, t_b) = true$
for any arbitrary ref

The proofs of the above properties use structural induction over timed traces that satisfy the *Expand* relation. They can be found in Appendix D.

The healthiness condition $R1_t$ is idempotent and closed over conjunction, disjunction, conditional choice and sequential composition. For other properties of $R1_t$, and a detailed proof of these properties refer to Appendix B

The condition $R2$ states that the program execution is independent of the initial state of the environment. We can replace tr with an arbitrary trace s and the program behaviour will not change. In the UTP, two definitions for the healthiness condition $R2$ are given. The first is defined as

$$R2(X(tr, tr')) \;\widehat{=}\; \sqcap_s X(s, s \frown (tr' - tr)) \tag{3.4.4}$$

whereas the second is

$$R2(X(tr, tr')) \;\widehat{=}\; X(\langle\rangle, (tr' - tr)) \tag{3.4.5}$$

The equivalence of the two definitions in terms of their characterization of healthy predicates was explored by Cavalcanti and Woodcock in [5]. We define $R2_t$ in a similar form to the second definition, which we believe to be clearer.

The initial trace in our model is not $\langle\rangle$, but a trace that contains an empty sequence and an arbitrary refusal set in the first position. This imposes that the initial refusal set of a program is also irrelevant for its behaviour. The following is the definition of $R2$ in the time model.

$$R2_t(X) \;\widehat{=}\; \exists\, ref \bullet X[\langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t)/tr_t, tr'_t] \tag{3.4.6}$$

where *dif* is used to obtain the difference between two time traces and is defined as follows.

$$dif(tr'_t, tr_t) \;\widehat{=}\; \langle (fst(head(tr'_t - front(tr_t))) - fst(last(tr_t)), \\ snd(head(tr'_t - front(tr_t)))) \rangle \frown tail(tr'_t - front(tr_t)) \tag{3.4.7}$$

The standard operations on sequences *head*, *last*, *front*, *tail*, *fst* and *snd* are defined in Appendix A. The reason why we use the *dif* function instead of simply using trace subtraction as in the UTP is because the initial time trace tr_t is not necessarily a prefix of the final trace tr'_t (as imposed by $R1_t$); we use the *Expands* relation instead. The difference will be further discussed in this chapter, when defining communication.

Property 3.3

L1. $dif(tr, tr) = \langle (\langle\rangle, snd(last(tr))) \rangle$ *for any time trace* tr

L2. $dif(tr, \langle (\langle\rangle, ref) \rangle) = tr$
for any arbitrary ref

L3. $Flat(dif(tr'_t, tr_t)) = Flat(tr'_t) - Flat(tr_t)$

L4. $dif(t_a, t_b) = dif(t_c, t_b) \Leftrightarrow t_a = t_c$

L5. $(\exists\, t \bullet dif(tr'_t, tr_t) = t) \Leftrightarrow Expands(tr_t, tr'_t)$

L6. $Flat(dif(t_a, t_b)) = \langle\rangle \Leftrightarrow \begin{array}{l} \forall\, i : 1..\#dif(t_a, t_b) \bullet \\ fst(dif(t_a, t_b)(i)) = \langle\rangle \end{array}$

L7. $\begin{array}{l} \forall\, i : 1..\#dif(t_a, t_b) \bullet \\ snd(dif(t_a, t_b)(i)) = X \end{array} \Leftrightarrow \forall\, i : \#t_b..\#t_a \bullet snd(t_a(i)) = X$

L8. $(\#tr'_t = \#tr_t \wedge Flat(tr'_t) = Flat(tr_t) \wedge Expands(tr_t, tr'_t)) = tr_t = tr'_t$

L9. $fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \Rightarrow trace' \neq \langle\rangle$

L10. $dif(dif(t_a, t_c), dif(t_b, t_c)) = dif(t_a, t_b)$
Provided that $Expands(t_a, t_c) \wedge Expands(t_b, t_c) \wedge Expands(t_a, t_b)$ *is true.*

The proof of the properties above can be found in Appendix D.

In Appendix B we show that the healthiness condition $R2_t$ is idempotent, closed under conjuntion, disjunction, sequential composition and conditional choice. We also show that it is commutative with $R1_t$.

The healthiness condition $R3$, as defined in the UTP, assures that a program cannot start in a waiting state. If the variable *wait* is *true* (the previous program did not terminate), then the program behaves as Π; otherwise, its effect takes place. Π preserves the traces if the previous program diverges, otherwise it simply leaves the program variables unchanged.

$$R3(X) \;\widehat{=}\; \Pi \lhd wait \rhd X \tag{3.4.8}$$

where Π is defined as follows

$$\Pi \mathrel{\widehat{=}} \left(\begin{array}{c} (\neg ok \wedge tr \leq tr') \vee \\ (ok' \wedge (tr' = tr) \wedge (wait' = wait) \wedge (state' = state) \wedge (ref = ref')) \end{array} \right) \quad (3.4.9)$$

We define a healthiness condition $R3_t$ following the same principle; the definition is identical to that of $R3$ except for the use of Π_t instead of Π. Π_t acts on time traces and makes use of the relation *Expands*.

$$R3_t(X) \quad \widehat{=} \quad \Pi_t \lhd wait \rhd X \quad (3.4.10)$$

where Π_t is defined as

$$\Pi_t \quad = \quad \left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ (ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \end{array} \right) \quad (3.4.11)$$

The definition of Π_t does not mention explicitly $ref = ref'$ because, in the time model, the refusal information is encoded in the time traces, so the term $(tr_t = tr_t')$ includes the condition that the refusal sets need to be maintained.

In Appendix B we show that Π_t is $R1_t$, $R2_t$ and $R3_t$ healthy, we also find the following lemmas.

Lemma 3.1

$$\Pi_t \wedge ok = (ok \wedge ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state))$$

Proof:

$$\begin{array}{ll} & \Pi_t \wedge ok \\ = & \qquad [3.4.11] \\ & ok \wedge \left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ (ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \end{array} \right) \\ = & \qquad [Propositional\ calculus] \\ & \left(\begin{array}{c} (ok \wedge \neg ok \wedge Expands(tr_t, tr_t')) \vee \\ (ok \wedge ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \end{array} \right) \\ = & \qquad [Propositional\ calculus] \\ & (ok \wedge ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \qquad \Box \end{array}$$

Lemma 3.2

$$\Pi_t \wedge \neg ok = (\neg ok \wedge Expands(tr_t, tr_t'))$$

Proof:

$$\Pi_t \wedge \neg ok$$

$$
\begin{array}{ll}
= & \textit{[3.4.11]} \\
\neg ok \wedge \left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr'_t)) \vee \\ (ok' \wedge (tr'_t = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \end{array} \right) & \\
= & \textit{[Propositional calculus]} \\
(\neg ok \wedge Expands(tr_t, tr'_t)) & \Box
\end{array}
$$

The healthiness condition $R3_t$ has been shown to be idempotent and closed over conjunction, disjunction, sequential composition and conditional choices. Details of the proofs can be found in Appendix B.

A reactive program satisfies the condition R_t formed by the conditions $R1_t$,$R2_t$ and $R3_t$

$$R_t \;\widehat{=}\; R3_t \circ R2_t \circ R1_t \tag{3.4.12}$$

A similar condition R for reactive programs is given in the UTP. The order in which the healthiness conditions are applied in R_t is irrelevant as they are commutative; details of the proof can be found in Appendix B.

Because a *Circus* program is also a CSP process, then further healthiness conditions are needed. As described in the UTP, a reactive program is also a CSP process if it satisfies the following healthiness conditions. The condition $CSP1$ enforces that, if a program starts in an unstable state then, we can only guarantee that traces are extended.

$$CSP1(X) \;\widehat{=}\; (\neg ok \wedge tr \leq tr') \vee X \tag{3.4.13}$$

Again a similar condition is defined for the time model, except that we use the *Expands* relation instead of the trace prefix relation.

$$CSP1_t(X) \;\widehat{=}\; (\neg ok \wedge Expands(tr_t, tr'_t)) \vee X \tag{3.4.14}$$

The healthiness condition $CSP1_t$ has been shown to be idempotent, closed over conjunction, disjunction, sequential composition and conditional choice and commutative with $R1_t, R2_t$ and R_3. Detailed proof of these properties can be found in Appendix B.

The healthiness condition $CSP2_t$ imposes that a program P can not require nontermination. Therefore, if an observation of P is valid for $(ok' = false)$ it should also be valid for $(ok' = true)$. This is defined as follows

$$CSP2_t(X) \;\widehat{=}\; X;\; \left(\begin{array}{c} (ok \Rightarrow ok') \wedge \\ (tr'_t = tr_t) \wedge \\ (wait' = wait) \wedge \\ (state' = state) \end{array} \right) \tag{3.4.15}$$

The healthiness condition $CSP2_t$ was found to be idempotent and closed over disjunction, conditional choice and sequential composition. It is not closed under conjunction.

The conditions $CSP3_t$ and $CSP4_t$ enforce that *Skip* is the right and left unit of sequential composition.

$$CSP3_t(X) \;\widehat{=}\; Skip;\; X \tag{3.4.16}$$

$$CSP4_t(X) \;\widehat{=}\; X;\ Skip \tag{3.4.17}$$

The healthiness conditions $CSP3_t$ and $CSP4_t$ are shown in Appendix B to be idempodent and closed over disjunction, conditional choice and sequential composition. These conditional choices where found not to be closed over conjunction as well. For details of healthiness conditions properties and proofs see Appendix B.

Finally $CSP5_t$ requires that *Skip* is the unit of interleaving. Because *Circus* Time Action has no interleaving operator, we use the parallel composition operator with an empty synchronization set.

$$CSP5_t(X) \;\widehat{=}\; X \,[\![\, \Delta X \mid \{\!|\, \phi \,|\!\} \mid \phi \,]\!]\, Skip \tag{3.4.18}$$

Where ΔX is the set of variables that the action X can change. The conditions $CSP2_t$ to $CSP5_t$ are identical to their equivalent conditions in the UTP. A *Circus* Time Action action needs to satisfy the condition CSP_t.

$$CSP_t \mathrel{\widehat{=}} CSP5_t \circ CSP4_t \circ CSP3_t \circ CSP2_t \circ CSP1_t \circ R_t \tag{3.4.19}$$

The condition CSP_t includes the conditions for a timed reactive program R_t and the five healthiness conditions for a timed CSP process.

3.5 SEMANTICS OF CIRCUS TIME ACTION

In this section we give the semantics of *Circus* Time Action using the semantic model defined in the previous section. We use a denotational semantics, where each construct of the language is mapped to a predicate on the observation variables of the semantic model that respects all the healthiness conditions described in the previous section. Whenever possible, we always make reference to the equivalent semantic definition for the language constructs given in the UTP.

3.5.1 Basic Actions

The semantics of *Skip* is given as a program that can only terminate normally, without consuming time. It also has no interaction with the environment. The semantics of the action *Skip* in the UTP is given as

$$Skip \mathrel{\widehat{=}} R(\exists\, ref \bullet \Pi) \tag{3.5.1}$$

From the definition of Π (see 3.4.9), the definition of *Skip* can be expanded to

$$Skip \mathrel{\widehat{=}} R\left(\exists\, ref \bullet \begin{pmatrix} (\neg ok \wedge tr \leq tr') \vee \\ ok' \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge \\ (state' = state) \wedge \\ (ref' = ref) \end{pmatrix}\right)$$

This definition does not explicitly require $wait'$ to be *false* to record that the program terminates. The $R3$ healthiness condition forces the program to only start when *wait* is

false (see 3.4.8); the definition of Π assures that $(wait' = wait)$ and therefore $wait'$ is *false*. We take a similar approach to define *Skip* in *Circus* Time Action

$$Skip \mathrel{\widehat=} R_t(\exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t) \tag{3.5.2}$$

The action *Skip* is R_t healthy by definition. It is $CSP1_t$ and $CSP2_t$ healthy because Π_t has been shown to be healthy in Appendix B. It is also $CSP3_t$ and $CSP4_t$ healthy from Lemma 3.7, it is also $CSP5_t$ from property 3.12 L3.

The semantics of the action *Stop* is given as a program that waits forever. The UTP gives the following definition for the action *Stop*

$$Stop \mathrel{\widehat=} CSP1(ok' \land \delta) \tag{3.5.3}$$

where

$$\delta \mathrel{\widehat=} R3(tr' = tr \land wait') \tag{3.5.4}$$

From the definition of *Stop*, we see that the program waits forever ($wait'$) and does not interact with the environment: it does not communicate with its environment ($tr = tr'$). Internal events, such as change in the state, may occur. In *Circus* Time Action we adopt a similar definition, but we permit time to pass; still, the program can not interact with the environment. The following is the definition of *Stop* in *Circus* Time Action.

$$Stop \mathrel{\widehat=} CSP1_t(R3_t(ok' \land wait' \land trace' = \langle\rangle)) \tag{3.5.5}$$

As already mentioned, $trace'$ is derived from tr_t and tr'_t and the variable $trace'$ captures the communications of a program, independently of the time in which they occurred, yet maintaining the order in which the events occur. Enforcing that $trace' = \langle\rangle$, we allow time to pass, but the only possible element of the corresponding time trace is $(\langle\rangle, ref)$, where *ref* is an arbitrary refusal set.

From the definition of *Stop* we see clearly that *Stop* is $CSP1_t$ and $R3_t$ healthy. From properties 3.2 L3 and L7 we see that *Stop* is $R2_t$ and $R1_t$ healthy. *Stop* is also $CSP2_t$ healthy because it does not diverge. The action *Stop* is $CSP3_t$ healthy; the proof and justification follows from the fact that *Stop* is $CSP1_t$,$R1_t$ healthy, and Lemma 3.1. *Stop* is $CSP4_t$ due to property 3.6 L1. Another important issue here is that *ref* is arbitrary, that is, during the waiting period, any arbitrary set of events can be refused including the set of all the possible communications of an action.

The action *Chaos* is given as the predicate *true*. *Chaos* is the worst action and nothing can be said about it, except that it also needs to satisfy the condition R_t.

$$Chaos \mathrel{\widehat=} R_t(true) \tag{3.5.6}$$

This definition of *Chaos* is identical to its definition in the UTP.

The assignment operator assigns a value to a variable in the current state. If the variable does not exist in the state, it will be added, otherwise its value will be overwritten.

$$x := e \mathrel{\widehat=} CSP1\left(R_t\left(\begin{array}{c} ok = ok' \land \\ wait = wait' \land \\ tr'_t = tr_t \land \\ state' = state \oplus \{x \mapsto val(e, state)\} \end{array}\right)\right) \tag{3.5.7}$$

The above definition is similar to that in the UTP, except for the clause $(ref = ref')$, which is implicit in $(tr_t = tr'_t)$. We use an evaluation function $val(e, s)$ that returns the value of the expression e in state s. An important consideration that we make is that the assignment operation is instantaneous and does not consume time; this is also imposed by $(tr_t = tr'_t)$. This model assignment operator is similar to the one in RTL [27]. A time consuming assignment can be defined with the aid of $Wait\ \ \mathsf{t}$ as follows

$$x :=_t e \mathrel{\widehat{=}} Wait\ \ \mathsf{t};\ (x := e)$$

The clause $state' = state \oplus \{x \mapsto val(e, state)\}$ changes the variable x in $state'$. This is carried out by overriding $state$ with a mapping of the variable x to the value of the expression e; the variable x is added to the $state$, in case it is not there. For details of the $\oplus$ operator, and others used in the sequel refer to Appendix A.

In *Circus* variable declaration is obligatory and the assignment is not well defined if the variable is not previously declared. In our model, for simplicity, we do not consider variable declaration and no time information can affect variable declaration.

3.5.2 Wait

The only possible behaviour for the wait action is to wait for the specified number of time units to pass before terminating.

$$Wait\ \ \mathsf{t} \ \ \widehat{=} \ \ CSP1_t(R_t(ok' \land delay(\mathsf{t}) \land (trace' = \langle\rangle))) \tag{3.5.8}$$

where $delay$ is defined as

$$delay(\mathsf{t}) \mathrel{\widehat{=}} \left(\begin{array}{l} (wait' \land (\#tr'_t - \#tr_t) < \mathsf{t}) \lor \\ (\neg wait' \land (\#tr'_t - \#tr_t) = \mathsf{t} \land state = state') \end{array} \right) \tag{3.5.9}$$

The $delay$ predicate permits the program to wait for t time units, and terminate at time t. There is no equivalent action in the UTP. In the next chapter, we study the relation that exists between $Wait\ \ \mathsf{t}$ and the UTP $Skip$. The action $Wait\ \ \mathsf{t}$ satisfies the following properties

Property 3.4

L1. $$(Wait\ \ \mathsf{t} \land wait') = \begin{array}{l} CSP1_t \left(R_t \left(\begin{array}{l} ok' \land wait' \land \\ (\#tr'_t - \#tr_t) < t \land \\ trace' = \langle\rangle \end{array} \right) \right) \\ \land\ wait \end{array}$$

Proof: *From the definition of Wait* □

L2. $$(Wait\ \ \mathsf{t} \land \neg wait') = \begin{array}{l} CSP1_t \left(R_t \left(\begin{array}{l} ok' \land \neg wait' \land \\ (\#tr'_t - \#tr_t) = t \land \\ trace' = \langle\rangle \land \\ state' = state \end{array} \right) \right) \\ \land\ \neg wait' \end{array}$$

Proof: *From the definition of Wait* □

L3. Wait $0 = Skip$

Proof:

$$
\begin{array}{ll}
& Wait\ 0 \\
= & \textit{[3.5.8 and 3.5.9]} \\
& CSP1_t\left(R_t\left(\left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < 0 \wedge \\ trace' = \langle\rangle \end{array}\right) \vee \\ \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = 0 \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right) \end{array}\right)\right)\right) \\
= & \textit{[Time trace difference can not be negative (imposed by } R1_t\textit{)]} \\
& CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = 0 \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right)\right)\right) \\
= & \textit{[sequence properties]} \\
& CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ \#tr'_t = \#tr_t \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right)\right)\right) \\
= & \textit{[3.3.1]} \\
& CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ \#tr'_t = \#tr_t \wedge \\ Flat(tr'_t) - Flat(tr_t) = \langle\rangle \wedge \\ state' = state \end{array}\right)\right)\right) \\
= & \textit{[sequence properties]} \\
& CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ \#tr'_t = \#tr_t \wedge \\ Flat(tr'_t) = Flat(tr_t) \wedge \\ state' = state \end{array}\right)\right)\right) \\
= & \textit{[property 3.3 L9]} \\
& CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ tr'_t = tr_t \wedge \\ state' = state \end{array}\right)\right)\right) \\
= & \textit{[Commutativity of } CSP1_t\textit{]} \\
& R_t\left(CSP1_t\left(\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ tr'_t = tr_t \wedge \\ state' = state \end{array}\right)\right)\right)
\end{array}
$$

$$
\begin{array}{ll}
= & \textit{[3.4.14]} \\
R_t \left(\begin{array}{l} \left(\begin{array}{l} ok' \land \neg wait' \land \\ tr'_t = tr_t \land \\ state' = state \end{array} \right) \lor \\ (\neg ok \land Expands(tr, tr')) \end{array} \right) & \\
= & \textit{[}R3_t \textit{ is idempotent]} \\
R_t \left(R3_t \left(\begin{array}{l} \left(\begin{array}{l} ok' \land \neg wait' \land \\ tr'_t = tr_t \land \\ state' = state \end{array} \right) \lor \\ (\neg ok \land Expands(tr, tr')) \end{array} \right) \right) & \\
= & \textit{[3.4.10]} \\
R_t \left(\begin{array}{l} (\Pi_t \land wait) \lor \\ \left(\begin{array}{l} \neg wait \land ok' \land \\ \neg wait' \land tr'_t = tr_t \land \\ state' = state \end{array} \right) \lor \\ (\neg wait \land \neg ok \land Expands(tr, tr')) \end{array} \right) & \\
= & \textit{[3.4.9]} \\
R_t \left(\begin{array}{l} (wait \land \neg ok \land Expands(tr_t, tr'_t)) \lor \\ \left(\begin{array}{l} wait \land ok' \land \\ (tr'_t = tr_t) \land \\ (wait' = wait) \land \\ (state' = state) \end{array} \right) \lor \\ \left(\begin{array}{l} \neg wait \land ok' \land \\ \neg wait' \land tr'_t = tr_t \land \\ state' = state \end{array} \right) \lor \\ (\neg wait \land \neg ok \land Expands(tr, tr')) \end{array} \right) & \\
= & \textit{[Propositional calculus]} \\
R_t \left(\begin{array}{l} (\neg ok \land Expands(tr_t, tr'_t)) \lor \\ \left(\begin{array}{l} ok' \land (tr'_t = tr_t) \land \\ (wait' = wait) \land (state' = state) \end{array} \right) \end{array} \right) & \\
= & \textit{[3.4.9]} \\
R_t(\Pi_t) & \\
= & \textit{[Introduce ref]} \\
R_t(\exists\, ref \bullet \Pi_t \land ref = snd(last(tr_t))) & \\
= & \textit{[3.5.2]} \\
Skip & \Box
\end{array}
$$

L4. $Wait\ \mathsf{n} \land Wait\ \mathsf{n} + \mathsf{m} \land wait' = Wait\ \mathsf{n} \land wait'$ *provided that* $m \geq 0$

Proof:

$$Wait\ \ \mathsf{n} \wedge Wait\ \ \mathsf{n+m} \wedge wait'$$

$=$ [Propositional calculus]

$$(Wait\ \ \mathsf{n} \wedge wait') \wedge (Wait\ \ \mathsf{n+m} \wedge wait')$$

$=$ [Property 3.5 L1]

$$CSP1_t\left(R_t\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr_t' - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array}\right)\right) \wedge CSP1_t\left(R_t\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr_t' - \#tr_t) < n + m \wedge \\ trace' = \langle\rangle \end{array}\right)\right) \wedge wait'$$

$=$ [Closer of $CSP1_t$ and R_t over conjuntion]

$$CSP1_t\left(R_t\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr_t' - \#tr_t) < n \wedge \\ (\#tr_t' - \#tr_t) < n + m \wedge \\ trace' = \langle\rangle \end{array}\right)\right) \wedge wait'$$

$=$ [Sequence property]

$$CSP1_t\left(R_t\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr_t' - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array}\right)\right) \wedge wait'$$

$=$ [Property 3.5 L1]

$$Wait\ \ \mathsf{n} \wedge wait'$$ □

L5. $Wait\ \ \mathsf{n} \wedge Wait\ \ \mathsf{n+m} \wedge \neg wait' = \neg ok \wedge Expands(tr_t, tr_t') \wedge \neg wait'$
Proof:

$$Wait\ \ \mathsf{n} \wedge Wait\ \ \mathsf{n+m} \wedge \neg wait'$$

$=$ [Propositional calculus]

$$(Wait\ \ \mathsf{n} \wedge \neg wait') \wedge (Wait\ \ \mathsf{n+m} \wedge \neg wait')$$

$=$ [Property 3.5 L2]

$$CSP1_t\left(R_t\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr_t' - \#tr_t) = n \wedge \\ trace' = \langle\rangle \end{array}\right)\right) \wedge CSP1_t\left(R_t\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr_t' - \#tr_t) = n + m \wedge \\ trace' = \langle\rangle \end{array}\right)\right) \wedge \neg wait'$$

$=$ [Closer of $CSP1_t$ and R_t over conjuntion]

$$
\begin{array}{l}
CSP1_t \left(R_t \left(\begin{array}{l} ok' \land wait' \land \\ (\#tr'_t - \#tr_t) = n \land \\ (\#tr'_t - \#tr_t) = n + m \land \\ trace' = \langle\rangle \end{array} \right) \right) \land \neg wait' \\
= \qquad\qquad [\textit{Sequence property}] \\
CSP1_t(R_t(false)) \land \neg wait' \\
= \qquad\qquad [\textit{3.4.14,3.4.2,3.4.6 and 3.4.10}] \\
\neg ok \land Expand(tr_t, tr'_t) \land \neg wait' \qquad\qquad \Box
\end{array}
$$

3.5.3 Communication

The following is the UTP definition for communication.

$$a \rightarrow Skip \;\; \widehat{=} \;\; CSP1(ok' \land do_A(a)) \tag{3.5.10}$$

where $do_A(a)$ is defined as

$$do_A(a) \;\; \widehat{=} \;\; \Phi(a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) \tag{3.5.11}$$

and

$$\Phi(P) \;\; \widehat{=} \;\; R(P \land ((tr = tr') \land wait') \lor (tr < tr')) \tag{3.5.12}$$

The following is a expansion of the $do_A(a)$ definition.

$$
\begin{array}{l}
do_A(a) \\
= \qquad\qquad [3.5.11,3.5.12] \\
R \left(\begin{array}{l} (a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) \\ \land ((tr = tr') \land wait') \lor (tr < tr') \end{array} \right) \\
= \qquad\qquad [3.5.22] \\
R \left(\begin{array}{c} (a \notin ref' \land wait') \lor (tr' = tr \frown \langle a \rangle \land \neg wait') \\ \land ((tr = tr') \land wait') \lor (tr < tr') \end{array} \right) \\
= \qquad\qquad [\text{propositional calculs}] \\
R \left(\begin{array}{c} (a \notin ref' \land wait' \land (tr = tr')) \lor \\ (a \notin ref' \land wait' \land (tr < tr')) \lor \\ (tr' = tr \frown \langle a \rangle \land \neg wait' \land (tr < tr')) \end{array} \right) \\
= \qquad\qquad [\text{propositional calculs}] \\
R \left(\begin{array}{c} ((a \notin ref' \land wait') \land ((tr = tr') \lor (tr < tr'))) \lor \\ (tr' = tr \frown \langle a \rangle \land \neg wait' \land (tr < tr')) \end{array} \right) \\
= \qquad\qquad [\text{properties of traces and } R1]
\end{array}
$$

$$
\begin{array}{l}
R\left(\begin{array}{l}(a \notin ref' \land wait' \land tr \leq tr') \lor \\ \quad (tr' = tr \frown \langle a \rangle \land \neg wait')\end{array}\right) \\
= \qquad\qquad [3.5.22] \\
R((a \notin ref' \land tr \leq tr') \lhd wait' \rhd (tr' = tr \frown \langle a \rangle)) \quad \Box
\end{array}
$$

From the above derivation we see clearly that communication is defined as a disjunction of two distinct behaviours. The first $(a \notin ref' \land wait' \land tr \leq tr')$ represents the behaviour of a program that is waiting to engage in the communication; during the waiting period the program can not refuse to communicate on the channel a. The second disjunct $(tr' = tr \frown \langle a \rangle \land \neg wait')$ determines that the only possible terminating behaviour is that in which a communication over the channel a is registered in the traces.

In *Circus* Time Action we use the same approach except that, while the program is waiting to communicate on a channel, time can pass and this is registered in the trace as entries $(\langle\rangle, \{x : Event \bullet x \neq c\})$, where c is the channel through which communication is expected. Another difference is that, as in *Circus*, we use channels instead of alphabets. This difference is not apparent in the definition of communication, but has an effect over parallel composition to be explored in Section 3.5.9; this is motivated by the aim of making *Circus* a strongly-typed language like Z.

We divide the definition of communication in two parts. The first part is the predicate $wait\_com(c)$, which models the state of an action waiting to communicate on channel c. The only possible observation is that the communication channel cannot appear in the refusal set during the observation period.

$$wait\_com(c) \mathrel{\widehat=} \quad wait' \land possible(tr_t, tr'_t, c) \land trace' = \langle\rangle \qquad (3.5.13)$$

The difference in our definition is due to the fact that our time model registers refusal sets at the end of each time slot, as opposed to the UTP model that registers the refusals at the end of the observation regardless of time. During the waiting period, we may have more than one refusal set; the predicate $possible(tr, tr', c)$ is used to assure that the channel c is not refused in any of the refusal sets during the observation period.

$$possible(tr_t, tr'_t, c) \mathrel{\widehat=} \forall\, i : \#tr_t..\#tr'_t \bullet c \notin snd(tr'_t(i)) \qquad (3.5.14)$$

The final part of the predicate $wait\_com(c)$ indicates that, during the waiting period the program will not communicate on other channels, but would allow time to pass $(trace' = \langle\rangle)$.

The second part of the definition of communication models the terminating state, and is partially represented by the predicate $term\_com(c.e)$. It represents the act of communicating a value e over a channel c. The communication does not take any time $(\#tr' = \#tr)$, but the event appears in the traces of the observation.

$$term\_com(c.e) \mathrel{\widehat=} \quad \neg wait' \land trace' = \langle c \rangle \land \#tr'_t = \#tr_t \qquad (3.5.15)$$

It is important to observe that the above predicate *term_com* only represents the exact moment in which the communication takes place. Terminating observations in the time

model are either an initial waiting period, followed by termination or by immediate termination. This is expressed as follows:

$$terminating\_com(c.e) \mathrel{\widehat=} \begin{array}{l}(wait\_com(c);\ term\_com(c.e)) \lor \\ (term\_com(c.e))\end{array} \tag{3.5.16}$$

where ; stands for sequential composition (see Section 3.5.4). Finally, we define the communication $c.e \rightarrow Skip$ as

$$c.e \rightarrow Skip \mathrel{\widehat=} CSP1_t\left(ok' \land R_t\left(\begin{array}{c}wait\_com(c) \lor \\ terminating\_com(c.e)\end{array}\right)\right) \tag{3.5.17}$$

The following is a definition for an output of a value e through channel c, denoted by $c!e$; and an input through channel c of a value to be assigned to x, denoted by $c?x$.

$$c!e \rightarrow Skip = c.e \rightarrow Skip \tag{3.5.18}$$

$$c?x \rightarrow Skip = \exists\, e \bullet \begin{pmatrix} c.e \rightarrow Skip[state_o/state] \land \\ state' = state_o \oplus \{x \mapsto e\}\end{pmatrix} \tag{3.5.19}$$

The semantics of the communication prefix can be given in terms of communication and sequential composition.

$$comm \rightarrow Action \mathrel{\widehat=} (comm \rightarrow Skip);\ Action \tag{3.5.20}$$

Where *comm* stands for either an input $c?x$ or an output $c!e$. From the definition of communication we can prove the following property.

Property 3.5

L1. $(a \rightarrow Skip) \land wait' = CSP1_t(ok' \land R_t(wait\_com(a))) \land wait'$
Proof:
From definition □

L2. $(a \rightarrow Skip) \land \neg wait' = CSP1_t(ok' \land R_t(terminating\_com(a))) \land \neg wait'$
Proof:
From definition □

L3. $(a \rightarrow A) \land wait' \land trace' = \langle\rangle = CSP1_t(ok' \land R_t(wait\_com(a))) \land wait' \land trace' = \langle\rangle$
Proof:
From definition and A is R3 healthy □

3.5.4 Sequential composition

The sequential composition operator in *Circus* Time Action is the same as in the UTP.

$$A;\ B \ \widehat{=}\ \exists\, obs_o \bullet A[obs_o/obs'] \wedge B[obs_o/obs] \tag{3.5.21}$$

The sequential composition of two actions $A;\ B$ combines the actions in such a way that the final observations of A are the initial observations of B. We use the term obs to represent the set of observation variables ok,$wait$, tr_t and $state$; as is the case of obs' and obs_o.

We state and prove some lemmas of sequential composition. The first lemma states that the sequential composition preserves the *Expands* relation. This is expressed as follows

Lemma 3.3

$$Expands(tr_t, tr'_t);\ A = Expands(tr_t, tr'_t)$$

Provided A is R_t healthy
Proof:

$$
\begin{array}{ll}
& Expands(tr_t, tr'_t);\ A \\
= & \qquad [Propositional\ caclulus] \\
& Expands(tr_t, tr'_t);\ ((ok \vee \neg ok) \wedge (wait \vee \neg wait) \wedge A) \\
= & \qquad [Propositional\ calculus\ and\ Property\ 3.9\ L5,L6] \\
& Expands(tr_t, tr'_t);\ (\neg ok \wedge wait \wedge A) \vee \\
& Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge A) \vee \\
& Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge A) \vee \\
& Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge A) \\
= & \qquad [Assumption\ that\ A\ is\ R3_t\ healthy\ and\ Property\ B.3\ L1] \\
& Expands(tr_t, tr'_t);\ (\neg ok \wedge wait \wedge R3_t(A)) \vee \\
& Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge A) \vee \\
& Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge A) \vee \\
& Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge A) \\
= & \qquad [3.4.10] \\
& Expands(tr_t, tr'_t);\ (\neg ok \wedge wait \wedge (\Pi_t \lhd wait \rhd A)) \vee \\
& Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge A) \vee \\
& Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge A) \vee \\
& Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge A) \\
= & \qquad [wait = true\ and\ Property\ 3.7\ L5] \\
& Expands(tr_t, tr'_t);\ (\neg ok \wedge wait \wedge \Pi_t) \vee \\
& Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge A) \vee \\
& Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge A) \vee \\
& Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge A)
\end{array}
$$

$$
\begin{array}{ll}
= & \textit{[Lemma 3.2]} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (\neg ok \wedge wait \wedge Expands(tr_t, tr'_t)) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge A) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge A) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge A)} \\
= & \textit{[3.5.21]} \\
\multicolumn{2}{l}{\exists\, ok_o, wait_o, tr_o \bullet Expands(tr_t, tr_o) \wedge \neg ok_o \wedge wait_o \wedge Expands(tr_o, tr'_t) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge A) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge A) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge A)} \\
= & \textit{[Substitution and Predicate calculus]} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge A) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge A) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge A)} \\
= & \textit{[Assumption that } A \textit{ is } R1_t \textit{ healthy and property B.1 L1]} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge R1_t(A)) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge R1_t(A)) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge R1_t(A))} \\
= & \textit{[3.4.2]} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge A \wedge Expands(tr_t, tr'_t)) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge A \wedge Expands(tr_t, tr'_t)) \vee} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge A \wedge Expands(tr_t, tr'_t))} \\
= & \textit{[Relational calculus]} \\
\multicolumn{2}{l}{Expands(tr_t, tr'_t) \vee} \\
\multicolumn{2}{l}{((Expands(tr_t, tr'_t);\ (ok \wedge wait \wedge A \wedge Expands(tr_t, tr'_t))) \wedge Expands(tr_t, tr'_t)) \vee} \\
\multicolumn{2}{l}{((Expands(tr_t, tr'_t);\ (\neg ok \wedge \neg wait \wedge A \wedge Expands(tr_t, tr'_t))) \wedge Expands(tr_t, tr'_t)) \vee} \\
\multicolumn{2}{l}{((Expands(tr_t, tr'_t);\ (ok \wedge \neg wait \wedge A \wedge Expands(tr_t, tr'_t))) \wedge Expands(tr_t, tr'_t))} \\
= & \textit{[Propositional calculus]} \\
Expands(tr_t, tr'_t) & \Box
\end{array}
$$

The next lemma states that Π_t is the left unit of sequential composition, provided that the composed action is $R1_t$ and $CSP1_t$ healthy.

Lemma 3.4

$$\Pi_t;\ A = A$$

Provided that A *is* $R1_t$ *and* $CSP1_t$ *healthy*
Proof:

$$
\begin{array}{ll}
& \Pi_t;\ A \\
= & \textit{[Propositional calculus]} \\
& (ok \vee \neg ok) \wedge \Pi_t;\ A \\
= & \textit{[Propositional calculus and property 3.9 L5]} \\
& ((ok \wedge \Pi_t);\ A) \vee ((\neg ok \wedge \Pi_t);\ A) \\
= & \textit{[Lemma 3.1 and Lemma 3.2]} \\
& (((ok \wedge ok' \wedge (tr'_t = tr_t) \wedge (wait' = wait) \wedge (state' = state)));\ A) \vee \\
& ((\neg ok \wedge Expands(tr_t, tr'_t));\ A) \\
= & \textit{[Relational calculus]} \\
& (ok \wedge A) \vee (\neg ok \wedge (Expands(tr_t, tr'_t);\ A)) \\
= & \textit{[Lemma 3.4]} \\
& (ok \wedge A) \vee (\neg ok \wedge Expands(tr_t, tr'_t)) \\
= & \textit{[assumption that } A \textit{ is } R1_t \textit{ healthy and 3.4.2]} \\
& (ok \wedge A \wedge Expands(tr_t, tr'_t)) \vee (\neg ok \wedge Expands(tr_t, tr'_t)) \\
= & \textit{[Propositional calculus]} \\
& ((ok \wedge A) \vee \neg ok) \wedge Expands(tr_t, tr'_t) \\
= & \textit{[Propositional calculus]} \\
& (A \wedge Expands(tr_t, tr'_t)) \vee (\neg ok \wedge Expands(tr_t, tr'_t)) \\
= & \textit{[assumption, Property B.1 L1 and Property B.4 L1]} \\
& A \qquad \square
\end{array}
$$

The Lemma 3.5 states that Π_t is the right unit of sequential composition. We also impose that the composed action should be $R1_t$ healthy.

Lemma 3.5

$$A;\ \Pi_t = A$$

Provided that A is $R1_t$ healthy
Proof:

$$
\begin{array}{ll}
& A;\ \Pi_t \\
= & \textit{[Propositional calculus]} \\
& (A \wedge (ok' \vee \neg ok'));\ \Pi_t \\
= & \textit{[Relational calculus]} \\
& A;\ (ok \vee \neg ok) \wedge \Pi_t \\
= & \textit{[Propositional calculus]} \\
& A;\ ((ok \wedge \Pi_t) \vee (\neg ok \wedge \Pi_t))
\end{array}
$$

$$
\begin{array}{ll}
= & \textit{[Lemma 3.1 and Lemma 3.2]} \\
A;\ \left(\begin{array}{l}(ok \land ok' \land wait = wait' \land tr'_t = tr_t \land state' = state) \lor \\ (\neg ok \land Expands(tr_t, tr'_t))\end{array}\right) & \\
= & \textit{[Property 3.9 L6]} \\
\left(\begin{array}{l}(A;\ (ok \land ok' \land wait = wait' \land tr'_t = tr_t \land state' = state)) \lor \\ (A;\ (\neg ok \land Expands(tr_t, tr'_t)))\end{array}\right) & \\
= & \textit{[Relational calculus]} \\
\left(\begin{array}{l}(A \land ok') \lor \\ (A;\ (\neg ok \land Expands(tr_t, tr'_t)))\end{array}\right) & \\
= & \textit{[Assumption that } A \textit{ is } R1_t \textit{ healthy]} \\
\left(\begin{array}{l}(A \land ok') \lor \\ (A \land Expands(tr_t, tr'_t));\ (\neg ok \land Expands(tr_t, tr'_t))\end{array}\right) & \\
= & \textit{[Relational calculus]} \\
\left(\begin{array}{l}(A \land ok') \lor \\ (A \land Expands(tr_t, tr'_t) \land \neg ok')\end{array}\right) & \\
= & \textit{[Assumption and property B.1 L1]} \\
(A \land ok') \lor (A \land \neg ok') & \\
= & \textit{[Propositional calculus]} \\
A & \Box
\end{array}
$$

The next lemma states that the sequential composition of *Skip* with itself yields *Skip*. Notice also that *Skip* consumes no time and, therefore, the sequential composition of any number of *Skip* actions is the same as just one single *Skip* operation.

Lemma 3.6

$$Skip;\ Skip = Skip$$

Proof:

$$
\begin{array}{ll}
Skip;\ Skip & \\
= & \textit{[3.5.2]} \\
R_t(\exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t);\ R_t(\exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t) & \\
= & \textit{[}R_t \textit{ is closed over ; and } \Pi_t \textit{ is } R_t \textit{ healthy]} \\
R_t((\exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t);\ (\exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t)) & \\
= & \textit{[Relational calculus]} \\
R_t((\exists\, ref \bullet ref = snd(last(tr_t)) \land ((\Pi_t);\ (\exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t)))) & \\
= & \textit{[Lemma 3.2]} \\
R_t((\exists\, ref \bullet ref = snd(last(tr_t)) \land \exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t)) & \\
= & \textit{[Propositional calculus]}
\end{array}
$$

$R_t(\exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t)$
$=$ *[3.5.2]*
$Skip$ $\Box$

Lemma 3.7 states that the conjunction of two actions that agree on waiting for the first n time units is actually equivalent to waiting for the first n time units before behaving as the conjunction of the remaining behaviour. Observe that this is an important result of our time model. As expected, the lemma shows the existence of a universal global clock on which all actions in parallel are synchronized. The clock records the time passage; actions in conjunction share the same clock and, therefore, have to alow time to pass at the same speed.

Lemma 3.7

$$(Wait\ \ \mathsf{n};\ A) \land (Wait\ \ \mathsf{n};\ B) = Wait\ \ \mathsf{n};\ (A \land B)$$

Provided that A and B are healthy actions
Proof:

$(Wait\ \ \mathsf{n};\ A) \land (Wait\ \ \mathsf{n};\ B)$
$=$ *[3.5.8]*

$$CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle \end{array}\right) \lor \left(\begin{array}{l} ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \end{array}\right)\right)\right);\ A \land$$
$$CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle \end{array}\right) \lor \left(\begin{array}{l} ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \end{array}\right)\right)\right);\ B$$

$=$ *[Assumption and $CSP1_t$, R_t closed under ; and $\land$]*

$$CSP1_t\left(R_t\left(\left(\left(\begin{array}{l} ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle \end{array}\right) \lor \left(\begin{array}{l} ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state \end{array}\right)\right);\ A \land \right.\right.\right.$$
$$\left.\left.\left.\left(\left(\begin{array}{l} ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle \end{array}\right) \lor \left(\begin{array}{l} ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state \end{array}\right)\right);\ B\right)\right)\right)$$

$=$ *[Property 3.9 L5]*

$$
CSP1_t\left(R_t\left(\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle\end{array}\right);\ A \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state\end{array}\right);\ A\end{array}\right) \land \left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle\end{array}\right);\ B \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state\end{array}\right);\ B\end{array}\right)\right)\right)
$$

$=$ *[Relational calculus]*

$$
CSP1_t\left(R_t\left(\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle\end{array}\right);\ (wait \land A) \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state\end{array}\right);\ A\end{array}\right) \land \left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle\end{array}\right);\ (wait \land B) \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state\end{array}\right);\ B\end{array}\right)\right)\right)
$$

$=$ *[Assumption A and B are $R3_t$ healthy]*

$$
CSP1_t\left(R_t\left(\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle\end{array}\right);\ \Pi_t \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state\end{array}\right);\ A\end{array}\right) \land \left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle\end{array}\right);\ \Pi_t \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state\end{array}\right);\ B\end{array}\right)\right)\right)
$$

$=$ *[Lemma 3.6 and propositional calculus]*

$$
CSP1_t\left(R_t\left(\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ tr' = \langle\rangle\end{array}\right) \lor \left(\begin{array}{l}\left(\begin{array}{l}ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state\end{array}\right);\ A \land \\ \left(\begin{array}{l}ok' \land \neg wait' \land \#tr'_t - \#tr_t = n \land \\ tr' = \langle\rangle \land state' = state\end{array}\right);\ B\end{array}\right)\right)\right)
$$

$=$ *[3.5.21]*

$$
CSP1_t \left(R_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ trace' = \langle\rangle \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\
\left(\begin{array}{l} ok_o \land \neg wait_o \land \#tr_o - \#tr_t = n \land \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \land state_o = state \end{array} \right) \land \\
A[ok_o, wait_o, tr_o, state_o / ok, wait, tr_t, state] \end{array} \right) \land \\
\left(\begin{array}{l} \exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{l} ok_x \land \neg wait_x \land \#tr_x - \#tr_t = n \land \\ Flat(tr_x) - Flat(tr_t) = \langle\rangle \land state_x = state \end{array} \right) \land \\
B[ok_x, wait_x, tr_x, state_x / ok, wait, tr_t, state] \end{array} \right)
\end{array} \right)
\end{array} \right) \right)
$$

$=$ *[Susbtitution]*

$$
CSP1_t \left(R_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ trace' = \langle\rangle \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} \exists\, tr_o \bullet \\
\left(\begin{array}{l} \#tr_o - \#tr_t = n \land \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \end{array} \right) \land \\
A[true, false, tr_o / ok, wait, tr_t] \end{array} \right) \land \\
\left(\begin{array}{l} \exists\, tr_x \bullet \\
\left(\begin{array}{l} \#tr_x - \#tr_t = n \land \\ Flat(tr_x) - Flat(tr_t) = \langle\rangle \end{array} \right) \land \\
B[true, false, tr_x / ok, wait, tr_t] \end{array} \right)
\end{array} \right)
\end{array} \right) \right)
$$

$=$ *[Predicate calculus]*

$$
CSP1_t \left(R_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ trace' = \langle\rangle \end{array} \right) \lor \\
\left(\begin{array}{l} \exists\, tr_o, tr_x \bullet \\
\#tr_o - \#tr_t = n \land \\
Flat(tr_o) - Flat(tr_t) = \langle\rangle \land \\
A[true, false, tr_o / ok, wait, tr_t] \land \#tr_x - \#tr_t = n \land \\
Flat(tr_x) - Flat(tr_t) = \langle\rangle \land \\
B[true, false, tr_x / ok, wait, tr_t] \end{array} \right)
\end{array} \right) \right)
$$

$=$ *[Propositional calculus]*

$$
CSP1_t \left(R_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \#tr'_t - \#tr_t < n \land \\ trace' = \langle\rangle \end{array} \right) \lor \\
\left(\begin{array}{l} \exists\, tr_o, tr_x \bullet \\
\#tr_t = \#tr_o - n \land \\
\#tr_t = \#tr_x - n \land \\
Flat(tr_o) = Flat(tr_t) \land \\
Flat(tr_x) = Flat(tr_t) \land \\
A[true, false, tr_o / ok, wait, tr_t] \land \\
B[true, false, tr_x / ok, wait, tr_t] \end{array} \right)
\end{array} \right) \right)
$$

$=$ *[$tr_x = tr_o$]*

$$
CSP1_t\left(R_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land\\ trace' = \langle\rangle\end{array}\right) \lor\\ \left(\begin{array}{l}\exists\, tr_o \bullet\\ \#tr_t = \#tr_o - n \land\\ Flat(tr_o) = Flat(tr_t) \land\\ A[true, false, tr_o/ok, wait, tr_t] \land\\ B[true, false, tr_o/ok, wait, tr_t]\end{array}\right)\end{array}\right)\right)
$$

$=$ *[Introduce $ok_o, wait_o$ and $state_o$]*

$$
CSP1_t\left(R_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land\\ trace' = \langle\rangle\end{array}\right) \lor\\ \left(\begin{array}{l}\exists\, ok_o, wait_o, tr_o, state_o \bullet\\ ok_o \land \neg wait_o \land state_o = state \land\\ \#tr_o - \#tr_t = n \land\\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \land\\ A[ok_o, wait_o, tr_o, state_o/ok, wait, tr_t, state] \land\\ B[ok_o, wait_o, tr_o, state_o/ok, wait, tr_t, state]\end{array}\right)\end{array}\right)\right)
$$

$=$ *[Substitution distributes over conjunction]*

$$
CSP1_t\left(R_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land\\ trace' = \langle\rangle\end{array}\right) \lor\\ \left(\begin{array}{l}\exists\, ok_o, wait_o, tr_o, state_o \bullet\\ \left(\begin{array}{l}ok' \land \neg wait' \land state' = state \land\\ \#tr' - \#tr_t = n \land trace' = \langle\rangle\end{array}\right)\left[\begin{array}{l}ok_o, wait_o, tr_o, state_o/\\ ok', wait', tr'_t, state'\end{array}\right] \land\\ (A \land B)[ok_o, wait_o, tr_o, state_o/ok, wait, tr_t, state]\end{array}\right)\end{array}\right)\right)
$$

$=$ *[3.6]*

$$
CSP1_t\left(R_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land\\ trace' = \langle\rangle\end{array}\right) \lor\\ \left(\begin{array}{l}\left(\begin{array}{l}ok' \land \neg wait' \land state' = state \land\\ \#tr' - \#tr_t = n \land trace' = \langle\rangle\end{array}\right);\\ (A \land B)\end{array}\right)\end{array}\right)\right)
$$

$=$ *[A and B are $R3_t$ healthy and $R3_t$ closed under $\land$]*

$$
CSP1_t\left(R_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land\\ trace' = \langle\rangle\end{array}\right);\ (A \land B) \lor\\ \left(\begin{array}{l}ok' \land \neg wait' \land state' = state \land\\ \#tr' - \#tr_t = n \land trace' = \langle\rangle\end{array}\right);\ (A \land B)\end{array}\right)\right)
$$

$=$ *[Property 3.9 L5]*

$$
CSP1_t\left(R_t\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \#tr'_t - \#tr_t < n \land\\ trace' = \langle\rangle\end{array}\right) \lor\\ \left(\begin{array}{l}ok' \land \neg wait' \land state' = state \land\\ \#tr' - \#tr_t = n \land trace' = \langle\rangle\end{array}\right)\end{array}\right);\ (A \land B)\end{array}\right)\right)
$$

$=$ *[$CSP1_t$ and R_t are closed under sequential composition]*

$$
\begin{array}{l}
CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \#tr'_t - \#tr_t \leq n \wedge \\ trace' = \langle\rangle \end{array}\right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge state' = state \wedge \\ \#tr' - \#tr_t = n \wedge trace' = \langle\rangle \end{array}\right)\right)\right);\ CSP1_t(R_t(A \wedge B) \\
= \qquad\qquad [A \text{ and } B \text{ are } CSP1_t \text{ and } R_t \text{ healthy}] \\
CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \#tr'_t - \#tr_t < n \wedge \\ trace' = \langle\rangle \end{array}\right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge state' = state \wedge \\ \#tr' - \#tr_t = n \wedge trace' = \langle\rangle \end{array}\right)\right)\right);\ (A \wedge B) \\
= \qquad\qquad [3.5.8] \\
Wait\ \ \mathsf{n};\ (A \wedge B) \qquad\qquad \square
\end{array}
$$

The sequential composition $A;\ B$ is healthy if both actions A and B are healthy, the proof of this result follows from the closeness of ; with respect to all healthiness conditions as shown in Appendix B.

The sequential composition has the following properties.

Property 3.6

L1. $Stop;\ A = Stop$
provided that A is healthy

L2. $(x := e);\ (x := f(x)) = x := f(e)$
where f(x) is a function of x.

L3. $Wait\ \ \mathsf{n};\ Wait\ \ \mathsf{m} = Wait\ \ \mathsf{n} + \mathsf{m}$

L4. $A;\ (B;\ C) = (A;\ B);\ C$

L5. $(A \lhd b \rhd B);\ C = (A;\ C) \lhd b \rhd (B;\ C)$

For the detailed proof of the above properties see Appendix D.

3.5.5 Conditional Choice

The definition of the choice operator is exactly as in UTP [26], where in $A \lhd b \rhd B$, if b evaluates to *true* then A is executed, otherwise B executes. We also require that the condition b should not contain any dashed variables; its evaluation does not take time.

$$A \lhd b \rhd B \mathrel{\widehat{=}} \ (b \wedge A) \vee (\neg b \wedge B) \tag{3.5.22}$$

The conditional choice $A \lhd b \rhd B$ is healthy if both actions A and B are healthy; the proof follows from the closure with respect to healthiness conditions (see Appendix B for details).

The choice operator satisfies the same properties defined in [26].

Property 3.7

L1. $A \lhd b \rhd A = A$

L2. $A \lhd b \rhd B = B \lhd \neg b \rhd A$

L3. $(A \lhd b \rhd B) \lhd c \rhd C = A \lhd (b \wedge c) \rhd (B \lhd c \rhd C)$

L4. $A \lhd b \rhd (B \lhd c \rhd C) = (A \lhd b \rhd B) \lhd c \rhd (A \lhd b \rhd C)$

L5. $A \lhd \mathit{true} \rhd B = A = B \lhd \mathit{false} \rhd A$

L6. $A \lhd b \rhd (B \lhd b \rhd C) = A \lhd b \rhd C$

L7. $A \lhd b \rhd (A \lhd c \rhd B) = A \lhd (b \vee c) \rhd B$

L8. $((A \sqcap B) \lhd b \rhd C) = (A \lhd b \rhd C) \sqcap (B \lhd b \rhd C)$

The proof of the properties of conditional choice are detailed in Appendix D.

3.5.6 Guarded action

In a guarded action $\mathsf{b}\ \&A$, the predicate b needs to be satisfied before A can take place. If b is *false*, the only possible behaviour of the resulting action is to wait forever. On the other hand, if the predicate evaluates to *true*, then the result is any possible behaviour of A. The guarded action used by *Circus* Time Action is identical to one proposed in the UTP.

$$\mathsf{b}\ \&A \ \widehat{=}\ A \lhd b \rhd \mathit{Stop} \tag{3.5.23}$$

The guarded action satisfies the following properties.

Property 3.8

L1. $\mathit{false}\&A = \mathit{Stop}$

L2. $\mathit{true}\&A = A$

L3. $b\&\mathit{Stop} = \mathit{Stop}$

L4. $b\&(q\&A) = (b \wedge q)\&A$

L5. $b\&(A \sqcap B) = (b\&A) \sqcap (b\&B)$

L6. $b\&(A;\ B) = (b\&A);\ B$

L7. $b\&A = (b\&\mathit{Skip});\ A$

Detailed proof of the properties above can be found in Appendix D.

The guarded action $(b\&A)$ is healthy if the action A is healthy; the proof follows from the closure of conditional choice with respect to healthiness conditions (see Appendix B for details) and, *Stop* is healthy.

3.5.7 Internal Choice

The internal choice is not affected by the environment. The composition of two actions with the internal choice operator results in a new action that can exhibit the behaviour of either one of them. Our definition of internal choice is identical to the definition given in the UTP.

$$A \sqcap B \mathrel{\widehat{=}} A \vee B \tag{3.5.24}$$

The internal choice satisfies the following properties.

Property 3.9

L1. $Chaos \sqcap A = Chaos$

L2. $A \sqcap A = A$

L3. $A \sqcap B = B \sqcap A$

L4. $A \sqcap (B \sqcap C) = (A \sqcap B) \sqcap C$

L5. $(A \sqcap B);\ C = (A;\ C) \sqcap (B;\ C)$

L6. $A;\ (B \sqcap C) = (A;\ C) \sqcap (A;\ B)$

L7. $A \lhd b \rhd (B \sqcap C) = (A \lhd b \rhd B) \sqcap (A \lhd b \rhd C)$

L8. $A \sqcap (B \lhd b \rhd C) = (A \sqcap B) \lhd b \rhd (A \sqcap C)$

L9. $(c \to A) \sqcap (c \to B) = c \to (A \sqcap B)$

The proof of the above properties can be found in Appendix D.

The internal choice operation $A \sqcap B$ is healthy if both actions A and B involved in the choice are healthy; the proof follows from the closer of choice with respect to healthiness conditions (see Appendix B for details).

3.5.8 External Choice

The external choice between two actions is determined by the environment. The composed action behaves as either one of the two actions, whichever reacts first to the environment. This can be expressed as two possible behaviours: either the system is in a waiting state, and only internal behaviour that satisfies both actions involved in the choice can take place, or the system reacts to its environment after waiting for an external event that satisfies either of the component actions or both and, in this case, the choice is non-deterministic. The UTP gives the following definition for external choice

$$A \Box B \ \mathrel{\widehat{=}}\ CSP2((A \wedge B) \lhd Stop \rhd (A \vee B)) \tag{3.5.25}$$

From the definition of conditional (3.5.22), we obtain the following definition for external choice

$$A \,\Box\, B \;=\; CSP2(((A \wedge B) \wedge Stop) \vee ((A \vee B) \wedge \neg Stop))$$

The definition of external choice is divided into two distinct disjoint parts. The first part $((A \wedge B) \wedge Stop)$ represents the behaviour of the external choice where both processes agree on waiting. The question that arises is: what are they waiting for? The answer becomes clear with the following expansion of the definition of *Stop*.

$$
\begin{array}{ll}
Stop & [3.5.3] \\
= CSP1(ok' \wedge \delta) & [3.5.4] \\
= CSP1(ok' \wedge R3(tr = tr' \wedge wait')) & [3.4.8] \\
= CSP1(ok' \wedge (\Pi \lhd wait \rhd (tr = tr' \wedge wait'))) & [3.5.22] \\
= CSP1((ok' \wedge wait \wedge \Pi) \vee (ok' \wedge \neg wait \wedge tr = tr' \wedge wait')) & [3.4.13] \\
= \left(\begin{array}{c} (ok' \wedge wait \wedge \Pi) \vee \\ (ok' \wedge \neg wait \wedge tr = tr' \wedge wait') \vee \\ (\neg ok \wedge tr \leq tr') \end{array} \right) & [3.4.9] \\
= \begin{array}{c} ok' \wedge wait \wedge \left(\begin{array}{c} (\neg ok \wedge tr \leq tr') \vee \\ \left(\begin{array}{c} ok' \wedge (tr' = tr) \wedge (wait' = wait) \wedge \\ (state' = state) \wedge (ref = ref')) \end{array} \right) \end{array} \right) \vee \\ (ok' \wedge \neg wait \wedge tr = tr' \wedge wait') \vee \\ (\neg ok \wedge tr \leq tr') \end{array} & \\
= & [\text{Propositional calculus}] \\
\begin{array}{c} (ok' \wedge wait \wedge (\neg ok \wedge tr \leq tr')) \vee \\ (\neg ok \wedge tr \leq tr') \vee \\ \left(\begin{array}{c} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge (ref = ref') \end{array} \right) \vee \\ (ok' \wedge \neg wait \wedge tr = tr' \wedge wait') \end{array} & \\
= & [\text{Propositional Calculus}] \\
\begin{array}{c} (\neg ok \wedge tr \leq tr') \vee \\ \left(\begin{array}{c} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge (ref = ref') \end{array} \right) \vee \\ (ok' \wedge \neg wait \wedge tr = tr' \wedge wait') \end{array} & \Box
\end{array}
$$

We make the assumption that A and B are healthy; this is important because some predicates in the derivation allow unhealthy behaviour, and can be eliminated by this assumption. From the assumption we can further reduce the above expression when adding the term $A \wedge B$, as follows

$$
\begin{array}{ll}
A \wedge B \wedge Stop & \\
= & [\text{From the above derivation}]
\end{array}
$$

$$
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \\ (ok' \land \neg wait \land tr = tr' \land wait')\end{array}\right) \lor \ \land (A \land B)
$$

$=$ [Predicate Calculus]

$$
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \land (A \land B) \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \\ (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))\end{array}\right) \land (A \land B) \ \lor
$$

$=$ [A and B satisfy $CSP1$]

$$
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \land (CSP1(A) \land CSP1(B)) \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \\ (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))\end{array}\right) \land (A \land B) \ \lor
$$

$=$ [$CSP1$ closed over $\land$]

$$
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \land CSP1(A \land B) \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \\ (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))\end{array}\right) \land (A \land B) \ \lor
$$

$=$ [3.4.13]

$$
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \land ((\neg ok \land tr \leq tr') \lor (A \land B)) \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \\ (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))\end{array}\right) \land (A \land B) \ \lor
$$

$=$ [Predicate calculus]

$$
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \\ (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))\end{array}\right) \land (A \land B) \ \lor
$$

$=$ [Assumption]

$$
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \\ (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))\end{array}\right) \land (R3(A) \land R3(B)) \ \lor
$$

$=$ [$R3$ closed over $\land$]

$$
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \\ (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))\end{array}\right) \land R3(A \land B) \ \lor
$$

$=$ [$R3$ and $wait$]

$$
\begin{array}{l}
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref')\end{array}\right) \land \Pi_t \quad \lor \\
\qquad (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))
\end{array}
$$

$=$ [3.4.9 and predicate calculus]

$$
\begin{array}{l}
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref')\end{array}\right) \land (\neg ok \land tr \leq tr') \lor \\
\qquad \left(\begin{array}{c} ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \end{array}\right) \land \qquad \lor \\
\qquad \left(\begin{array}{c} ok' \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \end{array}\right) \\
\qquad\qquad (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))
\end{array}
$$

$=$ [Predicate calculus]

$$
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref')\end{array}\right) \lor \qquad \Box
$$

$$
(ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B))
$$

From the above derivation, we see clearly that A and B can start,when they agree on waiting and not interacting with the environment. The second part of the definition of external choice $(A \lor B) \land \neg Stop)$ captures the moment in which the choice is made, this is conditioned by $\neg Stop$; an expansion of $\neg Stop$ is presented below.

$\neg Stop$

$=$ [From the derivation of *Stop* above]

$$
\neg \left(\begin{array}{l}
\left(\begin{array}{c}(\neg ok \land tr \leq tr') \lor \\ ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref'))\end{array}\right) \lor \\
\qquad (ok' \land \neg wait \land tr = tr' \land wait')
\end{array}\right)
$$

$=$ [De Morgans laws]

$$
\left(\begin{array}{l}
\neg(\neg ok \land tr \leq tr') \land \\
\neg \left(\begin{array}{c} ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref')) \end{array}\right) \land \\
\qquad \neg(ok' \land \neg wait \land tr = tr' \land wait')
\end{array}\right)
$$

$=$ [De Morgans laws]

$$
\left(\begin{array}{l}
(ok \lor tr \not\leq tr') \land \\
\left(\begin{array}{c} \neg ok' \lor \neg wait \lor \\ (tr' \neq tr) \lor \\ (wait' \neq wait) \lor \\ (state' \neq state) \lor \\ (ref \neq ref') \end{array}\right) \land \\
(\neg ok' \lor wait \lor tr \neq tr' \lor \neg wait')
\end{array}\right)
$$

$$
\begin{array}{ll}
= & \text{[Propositional calculus]} \\
\left(\begin{array}{c}
\left(\begin{array}{c}
(ok \lor tr \not\leq tr') \land \\
\left(\begin{array}{c}
\neg ok' \lor \left(\begin{array}{c}
\left(\begin{array}{c}
\neg wait \lor (tr' \neq tr) \lor \\
(wait' \neq wait) \lor \\
(state' \neq state) \lor \\
(ref \neq ref')
\end{array}\right) \land \\
(wait \lor tr \neq tr' \lor \neg wait')
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
= & \text{[Predicate calculus]} \\
\left(\begin{array}{c}
\left(\begin{array}{c}
(ok \lor tr \not\leq tr') \land \\
\left(\begin{array}{c}
\neg ok' \lor (tr' \neq tr) \lor \left(\begin{array}{c}
\left(\begin{array}{c}
\neg wait \lor \\
(wait' \neq wait) \lor \\
(state' \neq state) \lor \\
(ref \neq ref')
\end{array}\right) \land \\
(wait \lor \neg wait')
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
= & \text{[Predicates Calculus]} \\
\left(\begin{array}{c}
\left(\begin{array}{c}
(ok \lor tr \not\leq tr') \land \\
\left(\begin{array}{c}
\neg ok' \lor (tr' \neq tr) \lor \\
\left(\begin{array}{c}
\left(\begin{array}{c}
\left(\begin{array}{c}
\neg wait \lor \\
(wait' \neq wait) \lor \\
(state' \neq state) \lor \\
(ref \neq ref')
\end{array}\right) \land wait
\end{array}\right) \lor \\
\left(\begin{array}{c}
\neg wait \lor \\
(wait' \neq wait) \lor \\
(state' \neq state) \lor \\
(ref \neq ref')
\end{array}\right) \land \neg wait'
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
= & \text{[Predicates Calculus]} \\
\left(\begin{array}{c}
\left(\begin{array}{c}
(ok \lor tr \not\leq tr') \land \\
\left(\begin{array}{c}
\neg ok' \lor (tr' \neq tr) \lor \\
\left(\begin{array}{c}
(\neg wait \land wait) \lor \\
(wait' \neq wait \land wait) \lor \\
(state' \neq state \land wait) \lor \\
(ref \neq ref' \land wait)
\end{array}\right) \lor \\
\left(\begin{array}{c}
(\neg wait \land \neg wait') \lor \\
(wait' \neq wait \land \neg wait') \lor \\
(state' \neq state \land \neg wait') \lor \\
(ref \neq ref' \land \neg wait')
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
= & \text{[Predicates Calculus]}
\end{array}
$$

$$
\left(\begin{array}{c}
(ok \lor tr \not\leq tr') \land \\
\left(\begin{array}{c}
\neg ok' \lor (tr' \neq tr) \lor \\
\left(\begin{array}{c}
(\neg wait' \land wait) \lor \\
(state' \neq state \land wait) \lor \\
(ref \neq ref' \land wait)
\end{array}\right) \lor \\
\left(\begin{array}{c}
(\neg wait \land \neg wait') \lor \\
(wait \land \neg wait') \lor \\
(state' \neq state \land \neg wait') \lor \\
(ref \neq ref' \land \neg wait')
\end{array}\right)
\end{array}\right)
\end{array}\right)
$$

$=$ [Predicates Calculus]

$$
\left(\begin{array}{c}
(ok \lor tr \not\leq tr') \land \\
\left(\begin{array}{c}
\neg ok' \lor (tr' \neq tr) \lor \\
\left(\begin{array}{c}
(\neg wait' \land wait) \lor \\
(state' \neq state \land wait) \lor \\
(ref \neq ref' \land wait)
\end{array}\right) \lor \\
\left(\begin{array}{c}
(\neg wait' \land (\neg wait \lor wait)) \lor \\
(state' \neq state \land \neg wait') \lor \\
(ref \neq ref' \land \neg wait')
\end{array}\right)
\end{array}\right)
\end{array}\right)
$$

$=$ [Predicates Calculus]

$$
\left(\begin{array}{c}
(ok \lor tr \not\leq tr') \land \\
\left(\begin{array}{c}
\neg ok' \lor (tr' \neq tr) \lor \\
\left(\begin{array}{c}
(\neg wait' \land wait) \lor \\
(state' \neq state \land wait) \lor \\
(ref \neq ref' \land wait)
\end{array}\right) \lor \\
\left(\begin{array}{c}
\neg wait' \lor \\
(state' \neq state \land \neg wait') \lor \\
(ref \neq ref' \land \neg wait')
\end{array}\right)
\end{array}\right)
\end{array}\right)
$$

$=$ [Predicates Calculus]

$$
\left(\begin{array}{c}
(ok \lor tr \not\leq tr') \land \\
\left(\begin{array}{c}
\neg ok' \lor \\
(tr' \neq tr) \lor \\
\neg wait' \lor \\
(state' \neq state \land wait) \lor \\
(ref \neq ref' \land wait)
\end{array}\right)
\end{array}\right)
$$

$=$ [Predicates Calculus]

$$\left(\begin{array}{l} ok \wedge \left(\begin{array}{c} \neg ok' \vee \\ (tr' \neq tr) \vee \\ \neg wait' \vee \\ (state' \neq state \wedge wait) \vee \\ (ref \neq ref' \wedge wait) \end{array}\right) \vee \\ tr \not\leq tr' \wedge \left(\begin{array}{c} \neg ok' \vee \\ (tr' \neq tr) \vee \\ \neg wait' \vee \\ (state' \neq state \wedge wait) \vee \\ (ref \neq ref' \wedge wait) \end{array}\right) \end{array}\right) \qquad \square$$

From the above derivation, we notice that some terms in the final formula are unhealthy (due to the term $tr \not\leq tr'$). These unhealthy terms are eliminated when we explore the above formula in the second part of the definition of external choice $((A \vee B) \wedge \neg Stop))$. This elimination is shown in the following derivation

$$\left(\begin{array}{l} \left(\begin{array}{l} ok \wedge \left(\begin{array}{c} \neg ok' \vee \\ (tr' \neq tr) \vee \\ \neg wait' \vee \\ (state' \neq state \wedge wait) \vee \\ (ref \neq ref' \wedge wait) \end{array}\right) \vee \\ tr \not\leq tr' \wedge \left(\begin{array}{c} \neg ok' \vee \\ (tr' \neq tr) \vee \\ \neg wait' \vee \\ (state' \neq state \wedge wait) \vee \\ (ref \neq ref' \wedge wait) \end{array}\right) \end{array}\right) \end{array}\right) \wedge (A \vee B)$$

$=$ [Predicate calculus]

$$\left(\begin{array}{l} \left(\begin{array}{l} ok \wedge (A \vee B) \wedge \left(\begin{array}{c} \neg ok' \vee \\ (tr' \neq tr) \vee \\ \neg wait' \vee \\ (state' \neq state \wedge wait) \vee \\ (ref \neq ref' \wedge wait) \end{array}\right) \vee \\ tr \not\leq tr' \wedge (A \vee B) \wedge \left(\begin{array}{c} \neg ok' \vee \\ (tr' \neq tr) \vee \\ \neg wait' \vee \\ (state' \neq state \wedge wait) \vee \\ (ref \neq ref' \wedge wait) \end{array}\right) \end{array}\right) \end{array}\right)$$

$=$ [Assumption A and B are $R1$ healthy]

$$
\left(\begin{array}{l}
\left(\begin{array}{l} ok \land (A \lor B) \land \left(\begin{array}{c} \neg ok' \lor \\ (tr' \neq tr) \lor \\ \neg wait' \lor \\ (state' \neq state \land wait) \lor \\ (ref \neq ref' \land wait) \end{array}\right) \end{array}\right) \lor \\
\left(\begin{array}{l} tr \not\leq tr' \land ((A \land tr \leq tr') \lor (B \land tr \leq tr')) \land \\ \left(\begin{array}{c} \neg ok' \lor \\ (tr' \neq tr) \lor \\ \neg wait' \lor \\ (state' \neq state \land wait) \lor \\ (ref \neq ref' \land wait) \end{array}\right) \end{array}\right)
\end{array}\right)
$$

$=$ [Predicate calculus]

$$
\left(\begin{array}{l}
\left(\begin{array}{l} ok \land (A \lor B) \land \left(\begin{array}{c} \neg ok' \lor \\ (tr' \neq tr) \lor \\ \neg wait' \lor \\ (state' \neq state \land wait) \lor \\ (ref \neq ref' \land wait) \end{array}\right) \end{array}\right) \lor \\
\left(\begin{array}{l} ((A \land tr \not\leq tr' \land tr \leq tr') \lor (B \land tr \not\leq tr' \land tr \leq tr')) \land \\ \left(\begin{array}{c} \neg ok' \lor \\ (tr' \neq tr) \lor \\ \neg wait' \lor \\ (state' \neq state \land wait) \lor \\ (ref \neq ref' \land wait) \end{array}\right) \end{array}\right)
\end{array}\right)
$$

$=$ [Predicate calculus]

$$
\left(\begin{array}{l} ok \land (A \lor B) \land \left(\begin{array}{c} \neg ok' \lor \\ (tr' \neq tr) \lor \\ \neg wait' \lor \\ (state' \neq state \land wait) \lor \\ (ref \neq ref' \land wait) \end{array}\right) \end{array}\right)
$$

$=$ [Predicate calculus]

$$
\left(\begin{array}{l}
\left(\begin{array}{c} (ok \land A) \land \\ \left(\begin{array}{c} \neg ok' \lor \\ (tr' \neq tr) \lor \\ \neg wait' \lor \\ (state' \neq state \land wait) \lor \\ (ref \neq ref' \land wait) \end{array}\right) \end{array}\right) \lor \\
\left(\begin{array}{c} (ok \land B) \land \\ \left(\begin{array}{c} \neg ok' \lor \\ (tr' \neq tr) \lor \\ \neg wait' \lor \\ (state' \neq state \land wait) \lor \\ (ref \neq ref' \land wait) \end{array}\right) \end{array}\right)
\end{array}\right)
$$

$$
\begin{array}{ll}
= & \text{[Predicate calculus]} \\
& \left(\begin{array}{c}
(ok \land A \land \neg ok') \lor \\
(ok \land A \land (tr' \neq tr)) \lor \\
(ok \land A \land \neg wait') \lor \\
(ok \land A \land state' \neq state \land wait) \lor \\
(ok \land A \land ref \neq ref' \land wait)
\end{array}\right) \lor \\
& \left(\begin{array}{c}
(ok \land B \land \neg ok') \lor \\
(ok \land B \land (tr' \neq tr)) \lor \\
(ok \land B \land \neg wait') \lor \\
(ok \land B \land state' \neq state \land wait) \lor \\
(ok \land B \land ref \neq ref' \land wait)
\end{array}\right) \\
= & \text{[From assumption, } A \text{ and } B \text{ are } R3 \text{ heathy]} \\
& \left(\begin{array}{c}
(ok \land A \land \neg ok') \lor \\
(ok \land A \land (tr' \neq tr)) \lor \\
(ok \land A \land \neg wait') \lor \\
(ok \land R3(A) \land state' \neq state \land wait) \lor \\
(ok \land R3(A) \land ref \neq ref' \land wait)
\end{array}\right) \lor \\
& \left(\begin{array}{c}
(ok \land B \land \neg ok') \lor \\
(ok \land B \land (tr' \neq tr)) \lor \\
(ok \land B \land \neg wait') \lor \\
(ok \land R3(B) \land state' \neq state \land wait) \lor \\
(ok \land R3(B) \land ref \neq ref' \land wait)
\end{array}\right) \\
= & \text{[}(R3(P) \land wait) = (\Pi_t \land wait)\text{]} \\
& \left(\begin{array}{c}
(ok \land A \land \neg ok') \lor \\
(ok \land A \land (tr' \neq tr)) \lor \\
(ok \land A \land \neg wait') \lor \\
(ok \land \Pi_t \land state' \neq state \land wait) \lor \\
(ok \land \Pi_t \land ref \neq ref' \land wait)
\end{array}\right) \lor \\
& \left(\begin{array}{c}
(ok \land B \land \neg ok') \lor \\
(ok \land B \land (tr' \neq tr)) \lor \\
(ok \land B \land \neg wait') \lor \\
(ok \land \Pi_t \land state' \neq state \land wait) \lor \\
(ok \land \Pi_t \land ref \neq ref' \land wait)
\end{array}\right) \\
= & \text{[Lemma 3.1]}
\end{array}
$$

$$
\left(\begin{array}{c}
(ok \land A \land \neg ok') \lor \\
(ok \land A \land (tr' \neq tr)) \lor \\
(ok \land A \land \neg wait') \lor \\
ok' \land \\
(tr' = tr) \land \\
(ok \land \ (wait' = wait) \land \ \land state' \neq state \land wait) \lor \\
(state' = state) \land \\
(ref = ref') \\
ok' \land \\
(tr' = tr) \land \\
(ok \land \ (wait' = wait) \land \ \land ref \neq ref' \land wait) \\
(state' = state) \land \\
(ref = ref')
\end{array}\right) \lor
$$

$$
\left(\begin{array}{c}
(ok \land B \land \neg ok') \lor \\
(ok \land B \land (tr' \neq tr)) \lor \\
(ok \land B \land \neg wait') \lor \\
ok' \land \\
(tr' = tr) \land \\
(ok \land \ (wait' = wait) \land \ \land state' \neq state \land wait) \lor \\
(state' = state) \land \\
(ref = ref') \\
ok' \land \\
(tr' = tr) \land \\
(ok \land \ (wait' = wait) \land \ \land ref \neq ref' \land wait) \\
(state' = state) \land \\
(ref = ref')
\end{array}\right)
$$

$=$ [Predicate calculus]

$$
\left(\begin{array}{c}
(ok \land A \land \neg ok') \lor \\
(ok \land A \land (tr' \neq tr)) \lor \\
(ok \land A \land \neg wait')
\end{array}\right) \lor
\left(\begin{array}{c}
(ok \land B \land \neg ok') \lor \\
(ok \land B \land (tr' \neq tr)) \lor \\
(ok \land B \land \neg wait')
\end{array}\right)
$$

$=$ [Predicate calculus]

$$
ok \land (A \lor B) \land \left(\begin{array}{c}
\neg ok' \lor \\
(tr' \neq tr) \lor \\
\neg wait'
\end{array}\right) \qquad \Box
$$

From the above derivation we can notice clearly that a choice is taken to behaviour as A or B based on the following:

- $\neg ok'$: the diverging behaviour of either of the two processes;
- $(tr \neq tr')$: a change to the traces recording a reaction of the program to the environment independent of all other variables. This is independent of changes to

other state variables and of whether the program terminates or not;

- $\neg wait'$: termination, independent of changes to other state variables.

Therefore, an alternative definition for external choice; based on the derivations above, is as follows

$$A \Box B \mathrel{\widehat{=}} CSP2 \left(\begin{array}{c} (\neg ok \land tr \leq tr') \lor \\ \left(\begin{array}{c} ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land (ref = ref') \end{array} \right) \lor \\ (ok' \land \neg wait \land tr = tr' \land wait' \land (A \land B)) \lor \\ (ok \land (A \lor B) \land \neg ok') \lor \\ (ok \land (A \lor B) \land (tr' \neq tr)) \lor \\ (ok \land (A \lor B) \land \neg wait') \end{array} \right) \tag{3.5.26}$$

In *Circus* Time Action, we take a similar approach, but we need to consider time. Time considerations for an external choice ($A \Box B$) appear in two forms: first, while both actions agree on waiting we need to track the time that passes while the two are waiting. Secondly we need to eliminate the observations of A and B that do not satisfy the initial waiting conditions. To illustrate this consider the following action P.

$$P = Wait\ \ 2 \Box Wait\ \ 3$$

The action P makes a choice of either terminating after 2 time units ($Wait\ \ 2$) or after 3 time units ($Wait\ \ 3$). We apply the definition of external choice from the UTP (see 3.5.25), to obtain the following

$$P = CSP2((Wait\ \ 2 \land Wait\ \ 3) \lhd Stop \rhd (Wait\ \ 2 \lor Wait\ \ 3))$$

The predicate $Wait\ \ 2 \land Wait\ \ 3$ models the behaviour where both actions agree on waiting, but this only occurs in the first time slot; the action $Wait\ \ 2$ will not agree on waiting further and has to terminate. Therefore, the behaviour of the action P is equivalent to the behaviour of $Wait\ \ 2$. The terminating behaviour is represented by $(Wait\ \ 2 \lor Wait\ \ 3) \land \neg Stop)$. According to the derivation of $\neg Stop$ presented previously, this predicate allows the terminating observation of $Wait\ \ 2$ after 2 time units and also the terminating observation of $Wait\ \ 3$ after 3 time units; this is inadmissible in our time model of the language.

A similar consideration needs to be made regarding external choice involving *Wait* and communication. To illustrate the problem, we study the behaviour of the following action Q

$$Q = Wait\ \ 2 \Box (a \rightarrow Skip)$$

The external choice behaviour shows that both parts in the choice agree on waiting for the first time units. Therefore, the communication a can only occur within the first two

time units. In this case the external choice acts as a timeout, offering the communication a for the first two time units and then forces it to terminate. The following is a list of the possible final values of the time observations of the above program according to the UTP definition when stating in the state $tr_t = \langle(\langle\rangle, \{\})\rangle \land ok \land \neg wait$.

$$(ok' \land wait' \land tr_t = \langle(\langle\rangle, \{\})\rangle \land tr_t' = \langle(\langle\rangle, ref')\rangle) \quad [1]$$
$$(ok' \land \neg wait' \land tr_t = \langle(\langle\rangle, \{\})\rangle \land tr_t' = \langle(\langle a\rangle, ref'')\rangle) \quad [2]$$
$$(ok' \land wait' \land tr_t = \langle(\langle\rangle, \{\})\rangle \land tr_t' = \langle(\langle\rangle, ref'), (\langle\rangle, ref')\rangle) \quad [3]$$
$$(ok' \land \neg wait' \land tr_t = \langle(\langle\rangle, \{\})\rangle \land tr_t' = \langle(\langle\rangle, ref'), (\langle a\rangle, ref'')\rangle) \quad [4]$$
$$(ok' \land \neg wait' \land tr_t = \langle(\langle\rangle, \{\})\rangle \land tr_t' = \langle(\langle\rangle, ref'), (\langle\rangle, ref'), (\langle\rangle, ref')\rangle) \quad [5]$$
$$(ok' \land \neg wait' \land tr_t = \langle(\langle\rangle, \{\})\rangle \land tr_t' = \langle(\langle\rangle, ref'), (\langle\rangle, ref'), (\langle a\rangle, ref'')\rangle) \quad [6]$$

where $a \notin ref'$ and, ref' and ref'' are arbitrary refusal set.

From the above list, we notice that the only possible waiting states are represented by (1) and (3). This shows that the program can only wait without refusing to communicate on a for the first two time units. The predicate (2) models the case in which the communication happens at the very beginning, not consuming any time. The observations (4) and (6) correspond to the case in which the communication occurs at the end of the first and second time slot respectively. The observation (5) covers the case in which the program terminates without communicating on a, forced by $Wait2$. In the observations (5) and (6), at the end of the second time unit, the choice of communicating on a or terminating without communication is nondeterministic, just as it is in the case of the expression $Skip \Box (a \rightarrow Skip)$ in the UTP model.

Similar to the definition in the UTP, our definition of external choice is composed of two predicates. The first predicate $ExtChoice1(A, B)$ expresses the behaviour of the external choice when both actions are in a waiting state, agree on internal behaviour, and do not interact with the environment.

$$ExtChoice1(A, B) \;\widehat{=}\; (A \land B \land Stop) \quad (3.5.27)$$

The second predicate $ExtChoice2(A, B)$ captures the behaviour of the external choice in the case A and B do not agree on internal choices or on waiting for the environment, by either reacting to an external event or terminating.

$$ExtChoice2(A, B) \;\widehat{=}\; DifDetected(A, B) \land (A \lor B) \quad (3.5.28)$$

where $DifDetected(A, B)$ is used to determine which action should be chosen. The definition of $DifDetected(A, B)$ is divided into three parts: the first part captures the behaviour in which either one of the two actions diverges and as a consequence the external choice diverges. The second part captures the observations in which A and B agree on waiting or do not agree on waiting at all and capture this behaviour by $Skip$. This is followed by the last predicate which determines that the subsequent behaviour should be a terminating behaviour or if it interacts with the environment then it should be immediately after the waiting period.

$$DifDetected(A, B) \mathrel{\widehat=} \left(\begin{array}{c} \neg ok' \vee \\ \left(\begin{array}{l} (ok \wedge \neg wait) \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array} \right) \vee \\ Skip \end{array} \right) \end{array} \right) ; \\ \left(\begin{array}{l} (ok' \wedge \neg wait' \wedge tr'_t = tr_t) \vee \\ \left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr_t, tr'_t))) \neq \langle\rangle \end{array} \right) \end{array} \right) \end{array} \right) \tag{3.5.29}$$

The behaviour in which both A and B agree on waiting ($A \wedge B \wedge ok' \wedge wait' \wedge trace' = \langle\rangle$) or they do not agree to wait at the very beginning; this captures the behaviours (1),(3), and (5) of the example above. This initial part is sequentially composed with a predicate that represents a terminating behaviour that does not change the traces ($ok' \wedge \neg wait' \wedge tr' = tr$). This predicate is used to capture a terminating behaviour without any reaction to the environment, such as behaviour (5) in the example above. The alternative behaviour is a change in the trace, at exactly the same instant in time that the programs agreed on waiting. Notice that the tr used in the second predicate is the tr' of the first predicate, by definition of sequential composition (see Equation 3.5.21). Therefore, the second predicate affirms that any observation that follows the waiting state should have at least one communication ($trace' \neq \langle\rangle$), register this communication in the first element immediately after the waiting period ($ok' \wedge fst(head(dif(tr, tr'))) \neq \langle\rangle$), and that the traces should still satisfy the *Expands* relation. Observations (2), (4) and (6) satisfy these conditions and are the result of composing the waiting predicates (1), (3) and (5), respectively, with the second predicate. This predicate eliminates other observations that may be valid for either A or B, but are not for the external choice. Consider again the example program Q above; now consider the following observation.

$$\begin{array}{l} ok' \wedge \neg wait' \wedge tr_t = \langle(\langle\rangle, \{\})\rangle \wedge \\ tr'_t = \langle(\langle\rangle, ref'), (\langle\rangle, ref'), (\langle a\rangle, ref')\rangle \end{array} \quad [7]$$

The observation (7) is valid for the communication $a \rightarrow Skip$, but is not a valid observation for the external choice, because we cannot find a waiting observation that permits the external choice to wait for more than 2 time units: $Wait\ 2$ does not allow it.

The following is the definition of the external choice

$$A \,\Box\, B \;\; \mathrel{\widehat=} \;\; CSP2_t(ExtChoice1(A, B) \vee ExtChoice2(A, B)) \tag{3.5.30}$$

The external choice $A \,\Box\, B$ is healthy if both A and B are healthy. A proof follows from the definition of external choice which uses conjunction, disjunction and sequential composition, which where shown to be closed with respect to all healthiness conditions except for $CSP2_t$. To make the external choice $CSP2_t$ healthy, we explicitly used $CSP2_t$ in the definition of the external choice.

Property 3.10

L1. $A \Box Stop = A$
Provided that A *is healthy*

L2. $A \Box A = A$

L3. $A \Box B = B \Box A$

L4. $A \Box (B \Box C) = (A \Box B) \Box C$

L5. $A \Box (B \sqcap C) = (A \Box B) \sqcap (A \Box C)$

L6. $(a \rightarrow A) \Box (b \rightarrow A) = ((a \rightarrow Skip) \Box (b \rightarrow Skip)); A$

L7. $Wait\ \mathsf{n} \Box Wait\ \mathsf{n}+\mathsf{m} = Wait\ \mathsf{n}$
Proof:

$$
\begin{array}{ll}
& Wait\ \mathsf{n} \Box Wait\ \mathsf{n}+\mathsf{m} \\
= & [3.5.30] \\
& CSP2_t(ExtChoice1(Wait\ \mathsf{n}, Wait\ \mathsf{n}+\mathsf{m}) \vee ExtChoice2(Wait\ \mathsf{n}, Wait\ \mathsf{n}+\mathsf{m}); Skip \\
= & [3.5.27 \text{ and } 3.5.28] \\
& CSP2_t\left(\begin{array}{l}(Wait\ \mathsf{n} \wedge Wait\ \mathsf{n}+\mathsf{m} \wedge wait' \wedge trace' = \langle\rangle) \vee \\ DifDetected(Wait\ \mathsf{n}, Wait\ \mathsf{n}+\mathsf{m}) \wedge (Wait\ \mathsf{n} \vee Wait\ \mathsf{n}+\mathsf{m})\end{array}\right) \\
= & [3.5.29] \\
& CSP2_t\left(\begin{array}{l}(Wait\ \mathsf{n} \wedge Wait\ \mathsf{n}+\mathsf{m} \wedge wait' \wedge trace' = \langle\rangle) \vee \\ \left(\begin{array}{l}\left(\begin{array}{l}((Wait\ \mathsf{n} \wedge Wait\ \mathsf{n}+\mathsf{m} \wedge wait' \wedge trace' = \langle\rangle) \vee Skip); \\ \left(\begin{array}{l}(\neg wait' \wedge tr_t' = tr_t) \vee \\ (fst(head(dif(tr_t, tr_t')))) \neq \langle\rangle)\end{array}\right)\end{array}\right) \wedge \\ (Wait\ \mathsf{n} \vee Wait\ \mathsf{n}+\mathsf{m})\end{array}\right)\end{array}\right) \\
= & [3.5.8 \text{ and propositional calculus}] \\
& CSP2_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \wedge wait' \wedge \\ trace' = \langle\rangle \\ (\#tr_t' - \#tr_t) < n \wedge \\ (\#tr_t' - \#tr_t) < n+m\end{array}\right) \vee \\ \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}ok' \wedge wait' \wedge \\ trace' = \langle\rangle \\ (\#tr_t' - \#tr_t) < n \wedge \\ (\#tr_t' - \#tr_t) < n+m\end{array}\right) \vee Skip\end{array}\right); \\ \left(\begin{array}{l}(\neg wait' \wedge tr_t' = tr_t) \vee \\ (fst(head(dif(tr_t, tr_t')))) \neq \langle\rangle)\end{array}\right)\end{array}\right) \wedge \\ (Wait\ \mathsf{n} \vee Wait\ \mathsf{n}+\mathsf{m})\end{array}\right)\end{array}\right) \\
= & [((\#tr_t' - \#tr_t) < n \wedge (\#tr_t' - \#tr_t) < n+m) = (\#tr_t' - \#tr_t) < n]
\end{array}
$$

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor Skip \end{array} \right) ; \\ \left(\begin{array}{l} (\neg wait' \land tr'_t = tr_t) \lor \\ (fst(head(dif(tr_t, tr'_t))) \neq \langle\rangle) \end{array} \right) \end{array} \right) \land \\
(Wait\ \ \mathsf{n} \lor Wait\ \ \mathsf{n} + \mathsf{m})
\end{array} \right)
$$

$=$ *[Properties 3.9 L5 and L6]*

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) ; \\ (\neg wait' \land tr'_t = tr_t) \end{array} \right) \lor \\ \left(\begin{array}{l} \left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) ; \\ (fst(head(dif(tr_t, tr'_t))) \neq \langle\rangle) \end{array} \right) \lor \\ Skip;\ (\neg wait' \land tr'_t = tr_t) \lor \\ Skip;\ (fst(head(dif(tr_t, tr'_t))) \neq \langle\rangle) \end{array} \right) \land \\
(Wait\ \ \mathsf{n} \lor Wait\ \ \mathsf{n} + \mathsf{m})
\end{array} \right)
$$

$=$ *[3.6 and relational calculus]*

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \land \\ \neg wait' \end{array} \right) \lor \\ \left(\begin{array}{l} trace' = \langle\rangle \\ (\#tr_o - \#tr_t) <\land \\ (fst(head(dif(tr'_t, tr_o))) \neq \langle\rangle) \end{array} \right) \lor \end{array} \right) \land \\ (ok' \land \neg wait' \land tr'_t = tr_t) \lor \\ (ok' \land fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \end{array} \right) \\
(Wait\ \ \mathsf{n} \lor Wait\ \ \mathsf{n} + \mathsf{m})
\end{array} \right)
$$

$=$ *[Propositional calculus]*

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \land \\ \neg wait' \end{array} \right) \land Wait \ \ \mathsf{n} \lor \\
\left(\begin{array}{l} trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \land \\ \neg wait' \end{array} \right) \land Wait \ \ \mathsf{n} + \mathsf{m} \lor \\
\left(\begin{array}{l} \exists\, tr_o \bullet \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \\ (\#tr_o - \#tr_t) < n \land \\ (fst(head(dif(tr'_t, tr_o))) \neq \langle\rangle) \end{array} \right) \land Wait \ \ \mathsf{n} \lor \\
\left(\begin{array}{l} \exists\, tr_o \bullet \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \\ (\#tr_o - \#tr_t) < n \land \\ (fst(head(dif(tr'_t, tr_o))) \neq \langle\rangle) \end{array} \right) \land Wait \ \ \mathsf{n} + \mathsf{m} \lor \\
((ok' \land \neg wait' \land tr'_t = tr_t) \land Wait \ \ \mathsf{n}) \lor \\
((ok' \land \neg wait' \land tr'_t = tr_t) \land Wait \ \ \mathsf{n} + \mathsf{m}) \lor \\
((ok' \land fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \land Wait \ \ \mathsf{n}) \lor \\
((ok' \land fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \land Wait \ \ \mathsf{n} + \mathsf{m})
\end{array} \right)
\end{array} \right)
$$

$$
= \qquad \textit{[Propositional calculus]}
$$

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} \left(\begin{array}{l} trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \land \\ (Wait \ \ \mathsf{n} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} \left(\begin{array}{l} trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \land \\ (Wait \ \ \mathsf{n} + \mathsf{m} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} \left(\begin{array}{l} \exists\, tr_o \bullet \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \\ (\#tr_o - \#tr_t) < n \land \\ (fst(head(dif(tr'_t, tr_o))) \neq \langle\rangle) \end{array} \right) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \ \mathsf{n}) \end{array} \right) \lor \\
\left(\begin{array}{l} \left(\begin{array}{l} \exists\, tr_o \bullet \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \\ (\#tr_o - \#tr_t) < n \land \\ (fst(head(dif(tr'_t, tr_o))) \neq \langle\rangle) \end{array} \right) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \ \mathsf{n} + \mathsf{m}) \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t) \land \\ (Wait \ \ \mathsf{n} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t) \land \\ (Wait \ \ \mathsf{n} + \mathsf{m} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \ \mathsf{n}) \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \ \mathsf{n} + \mathsf{m}) \end{array} \right)
\end{array} \right)
$$

$$
= \qquad \textit{[3.5.8 and propositional calculus]}
$$

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} \left(\begin{array}{l} trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) \leq n \end{array} \right) \land \\ \left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n \land \\ state' = state \end{array} \right) \end{array} \right) \lor \\
\left(\begin{array}{l} \left(\begin{array}{l} trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \land \\ \left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n + m \land \\ state' = state \end{array} \right) \end{array} \right) \lor \\
\left(\begin{array}{l} \left(\begin{array}{l} \exists\, tr_o \bullet \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \\ (\#tr_o - \#tr_t) < n \land \\ (fst(head(dif(tr'_t, tr_o))) \neq \langle\rangle) \end{array} \right) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \ \mathsf{n}) \end{array} \right) \lor \\
\left(\begin{array}{l} \left(\begin{array}{l} \exists\, tr_o \bullet \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \\ (\#tr_o - \#tr_t) < n \land \\ (fst(head(dif(tr'_t, tr_o))) \neq \langle\rangle) \end{array} \right) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \ \mathsf{n} + \mathsf{m}) \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t) \land \\ (Wait \ \ \mathsf{n} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t) \land \\ (Wait \ \ \mathsf{n} + \mathsf{m} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \ \mathsf{n}) \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \ \mathsf{n} + \mathsf{m}) \end{array} \right)
\end{array} \right)
$$

$$
= \qquad \textit{[Propositional calculus and sequence]}
$$

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) \leq n \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n \land \\ state' = state \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} \exists\, tr_o \bullet \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \\ (\#tr_o - \#tr_t) < n \land \\ (fst(head(dif(tr'_t, tr_o))) \neq \langle\rangle) \end{array} \right) \land \\
((trace' \neq \langle\rangle) \land Wait \ \mathsf{n}) \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} \exists\, tr_o \bullet \\ Flat(tr_o) - Flat(tr_t) = \langle\rangle \\ (\#tr_o - \#tr_t) < n \land \\ (fst(head(dif(tr'_t, tr_o))) \neq \langle\rangle) \end{array} \right) \land \\
((trace' \neq \langle\rangle) \land Wait \ \mathsf{n} + \mathsf{m}) \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t) \land \\ (Wait \ \mathsf{n} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t) \land \\ (Wait \ \mathsf{n} + \mathsf{m} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \mathsf{n}) \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \land \\ ((trace' \neq \langle\rangle) \land Wait \ \mathsf{n} + \mathsf{m}) \end{array} \right)
\end{array} \right)
$$

$=$ *[Wait* n *does not change traces]*

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n \land \\ state' = state \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t) \land \\ (Wait \ \mathsf{n} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t) \land \\ (Wait \ \mathsf{n} + \mathsf{m} \land \neg wait') \end{array} \right)
\end{array} \right)
\end{array} \right)
$$

$=$ *[*$(tr'_t = tr_t)$ *implies that* $(\#tr'_t = \#tr_t)$*]*

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n \land \\ state' = state \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t \land (\#tr'_t - \#tr_t) = 0) \land \\ (Wait \ \ \mathsf{n} \land \neg wait') \end{array} \right) \lor \\
\left(\begin{array}{l} (ok' \land tr'_t = tr_t \land (\#tr'_t - \#tr_t) = 0) \land \\ (Wait \ \ \mathsf{n} + \mathsf{m} \land \neg wait') \end{array} \right)
\end{array} \right)
\end{array} \right)
$$

$=$ *[3.5.8 and propositional calculus]*

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n \land \\ state' = state \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} ok' \land tr'_t = tr_t \land \\ (\#tr'_t - \#tr_t) = 0 \end{array} \right) \land \\
\left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n \land \\ state' = state \end{array} \right)
\end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} ok' \land tr'_t = tr_t \land \\ (\#tr'_t - \#tr_t) = 0 \end{array} \right) \land \\
\left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n + m \land \\ state' = state \end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array} \right)
$$

$=$ *[only possible value for n and $n + m$ is 0]*

$$
CSP2_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n \land \\ state' = state \end{array} \right) \lor \\
\left(\begin{array}{l} ok' \land \neg wait' \land \\ tr'_t = tr_t \land trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n \land \\ state' = state \end{array} \right)
\end{array} \right)
\end{array} \right)
$$

$$
\begin{array}{ll}
= & \textit{[n arbitrary including 0]} \\
CSP2_t \left(\begin{array}{l} \left(\begin{array}{l} ok' \land wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) < n \end{array} \right) \lor \\ \left(\begin{array}{l} ok' \land \neg wait' \land \\ trace' = \langle\rangle \land \\ (\#tr'_t - \#tr_t) = n \land \\ state' = state \end{array} \right) \end{array} \right) & \\
= & \textit{[Wait is CSP2 healthy and 3.5.8]} \\
Wait\ \mathsf{n} & \Box
\end{array}
$$

L8. $(Wait\ \mathsf{n};\ A) \Box (Wait\ \mathsf{n};\ B) = Wait\ \mathsf{n};\ (A \Box B))$

L9. $(A \sqcap B) \Box (A \sqcap C) = A \sqcap (B \Box C)$

L10. $(Skip \Box (Wait\ \mathsf{n};\ A)) = Skip$ *Provided* $n > 0$

L11. $(a \rightarrow A) \Box (Wait\ \mathsf{n};\ (a \rightarrow A)) = (a \rightarrow A)$

3.5.9 Parallel composition

The parallel composition of two programs contains all the possible behaviours of both actions synchronized on a given set of events. The parallel composition of two actions will terminate only when both actions terminate.

The UTP defines a parallel composition with the aid of a restricted parallel operator. The parallel composition of two programs in the UTP is given by the conjunction of their behaviour, if their alphabets are disjoint.

$$P \| Q \ \widehat{=}\ P \land Q \tag{3.5.31}$$

A more general parallel merge operator is given with the aid of the restricted parallel composition. The parallel merge operator is associated with a merge function to show how common variables of the parallel programs should be combined. The parallel merge operator is defined as follows

$$P \|_M Q \ \widehat{=}\ ((P;\ U0(m)_{+A}) \| (Q;\ U1(m)_{+B}))_{+m};\ M \tag{3.5.32}$$

where m is the set of variables shared by the parallel programs

$$m \mathrel{\widehat{=}} out\alpha P \cap out\alpha Q$$

The predicates $U0(m)$ and $U1(m)$ are labelling functions; they introduce new variables in the program, with the prefixes 0 and 1, respectively. Their definitions follows.

$$U0(m) \ \widehat{=}\ \textbf{var}\ 0.m;\ (0.m = m);\ \textbf{end}\ m \tag{3.5.33}$$

$$U1(m) \ \widehat{=}\ \textbf{var}\ 1.m;\ (1.m = m);\ \textbf{end}\ m \tag{3.5.34}$$

In the above definition of $U0$ and $U1$, new variables are introduced, and assigned the value of the unmarked input variables. The new variables are added using the **var** operator and the original variables are removed using the **end** operator; these operators are not treated in the time model presented in this chapter and we recall their original semantics as defined by UTP.

$$\mathbf{var}\ x \mathrel{\widehat{=}} \exists\, x \bullet \Pi_A \tag{3.5.35}$$

$$\mathbf{end}\ x \mathrel{\widehat{=}} \exists\, x' \bullet \Pi_A \tag{3.5.36}$$

Notice that in the above definition we use Π_A, instead of Π_t, where A is the set of all the alphabet, including x. Notice however, that the variable declaration **var** and the variable undeclaration **end** operators are used to introduce new variables and to remove variables from the alphabet of the program, but, in this stage, have no effect over the *state* variables. These operators are not part of *Circus* Time Action and are simple predicate operators. The sequential composition of $(P;\ U0(m))$ adds new output variables with the prefix $0.m$, and the value of the new variables is the output value of the program P; the dashed variables are hidden from the output. The definition of parallel merge makes use of two more sets A and B that are defined as follows.

$$A \mathrel{\widehat{=}} out\alpha P \setminus m$$

$$B \mathrel{\widehat{=}} out\alpha Q \setminus m$$

The constructs $(P;\ U0(m)_{+A})$ and $(Q;\ U1(m)_{+B})$ are used in the definition of parallel merge to denote that the variables not in the set m (represented by the sets A and B) are not changed, and they retain the output value of the programs P and Q respectively. Such that $P_{+\{x\}} \mathrel{\widehat{=}} P \wedge x' = x$.

The function M used in the definition of the parallel merge is called a merge function. It takes as input the outputs of the parallel programs ($0.m$ and $1.m$) and produces a single output variable representing the merge of the two programs. A valid merge function should satisfy the following conditions:

1) symmetric on its inputs $0.m$ and $1.m$

$$(0.m, 1.m := 1.m, 0.m);\ M = M$$

2) associative

$$(0.m, 1.m, 2.m := 1.m, 2.m, 0.m);\ M3 = M3$$

where $M3$ is a three-way merge relation generated by M

$$M3 \mathrel{\widehat{=}} \exists\, x, t \bullet M(ok, m, 0.m, 1.m, x, t) \wedge M(t, m, x, 2.m, m', ok')$$

$M3$ is defined as a conjunction of two M functions such that, the output of the first function, captured by x and t, is one of the input parameters to the second function.

3) Satisfies the following condition

$$(0.m, 1.m := m, m);\ M = \Pi$$

The UTP defines CSP parallel composition with the aid of a merge function, as follows

$$P \|_{CSP} Q \mathrel{\widehat{=}} P \|_N Q \tag{3.5.37}$$

Where N is a valid merge function defined as follows

$$N \mathrel{\widehat{=}} \left(\begin{array}{c} ok' = (0.ok \wedge 1.ok) \wedge \\ wait' = (0.wait \vee 1.wait) \wedge \\ ref' = (0.ref \cup 1.ref) \wedge \\ \exists\, u. \left(\begin{array}{c} (u \downarrow AP = 0.tr - tr) \wedge \\ (u \downarrow AQ = 1.tr - tr) \wedge \\ (u \downarrow A(P \| Q) = u) \wedge \\ (tr' = tr;\ u) \end{array} \right) \end{array} \right) ;\ Skip \tag{3.5.38}$$

where AP, AQ and $A(P \| Q)$ stand for the alphabet of processes P,Q and $(P \| Q)$ respectively; $A(P \| Q)$ is defined as $AP \cup AQ$. We call the attention to the distinction between AP and αP; AP represents the events in which the process P that can engage: channel names in the case of *Circus*; αP represents the alphabet of the UTP model of the process P these variables include, ok, $wait$ and others. In $s \downarrow E$, s is a trace of events, E a set of events and $s \downarrow E$ stands for a sequence derived from s by restricting it to only elements of E (see Appendix A). The parallel composition merge function N states that: the parallel composition of two processes diverges if either of the processes diverges $ok' = (0.ok \wedge 1.ok)$; the parallel composition terminates if both processes terminate $wait' = (0.wait \vee 1.wait)$; if an event is refused by one process then the parallel composition refuses to engage in the same event $ref' = (0.ref \cup 1.ref)$; and the parallel composition is based on synchronization of the events in the alphabet of the processes that compose the parallel composition. Therefore, the processes synchronize on the common events in their alphabet; this is shown in the last term of the function N. Traces produced by the parallel composition are given by u, such that $(tr' = tr \frown u)$, the trace u when restricted to events in the alphabet of the process P (Q) should result in a valid trace of the process P (Q), given by $(u \downarrow AP = 0.tr - tr)$ $(u \downarrow AQ = 1.tr - tr)$. Still the resulting trace u should be a valid trace of the generic parallel composition $(u \downarrow A(P \| Q) = u)$.

The purpose of the definition of validity is to prove the following parallel composition properties. A valid merge M satisfies the following properties

Property 3.11

L1. $A \|_M B = B \|_M A$

L2. $A \|_M (B \|_M C) = (A \|_M B) \|_M C$

L3. $Skip \|_M Skip = Skip$

L4. $A \| Chaos = Chaos$

L5. $(A \lhd b \rhd B) \|_M C = ((A \|_M C) \lhd b \rhd (B \|_M C))$

L6. $A \|_M (B \sqcap C) = (A \|_M B) \sqcap (A \|_M C)$

L7. $(x := e;\ A) \|_M B = (x := e);\ (A \|_M B)$
providing that e *has no reference to* m.

Parallel composition in *Circus* Time Action is defined in a similar manner. We also use the parallel merge function defined in UTP, but we define a new merge function. The parallel composition in *Circus* and *Circus* Time Action is different from the UTP parallel composition in the sense that it takes two more parameters. The first is a set of events on which the parallel actions synchronize and, the second is a pair of disjoint sets of variables one for each action. These sets define the variables each action can change; these sets need to be disjoint.

As explained above, the parallel composition of the UTP is based on the alphabet of the processes on the other hand, the parallel composition in *Circus* and *Circus* Time Action is based on a synchronization set. Events that are common to both actions in parallel, only synchronize if the event appears in the synchronization set. The parallel composition can only refuse events that are refused by both actions and that appear in the synchronization set: an event that is refused by one action can be accepted by the other action, if it does not appear in the synchronization set.

$$\mathsf{A} \,[\![\, \mathsf{s_A} \mid \{\!|\, \mathsf{cs} \,|\!\} \mid \mathsf{s_B} \,]\!]\, \mathsf{B} \ \widehat{=}\ A \,\|_{TM(cs, s_A, s_B)}\, B \tag{3.5.39}$$

We define the merge predicate TM as follows; it has the parameter cs that stands for the set of events the two actions should synchronize on and, the parameters $\mathsf{s_A}$ and $\mathsf{s_B}$ that determine the set of variables each action can change. We take the variables ok, $wait$, tr and $state$ to be the shared variables between the two parallel actions.

$$TM(cs, s_A, s_B) \mathrel{\widehat{=}} \left(\begin{array}{c} ok' = (0.ok \land 1.ok) \land \\ wait' = (0.wait \lor 1.wait) \land \\ dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array} \right) \land \\ state' = (0.state - s_B) \oplus (1.state - s_A) \end{array} \right) \tag{3.5.40}$$

The above merge function states that the parallel composition diverges if one of the engaged actions diverges. The parallel composition only terminates when both actions terminate. The merge function also states that the resulting trace is a member of the set of traces produced by the synchronization function $TSync$. The function $TSync$ takes two timed traces, and a set of events on which the actions should synchronize on, and returns the set containing all possible traces. The following is the definition of the

function *TSync*.

$$TSync(S_1, S_2, cs) = TSync(S_2, S_1, cs) \quad (3.5.41)$$

$$TSync(\langle\rangle, \langle\rangle, cs) = \{\} \quad (3.5.42)$$

$$TSync(\langle(t, r)\rangle, \langle\rangle, cs) = \{\langle(t', r)\rangle \mid t' \in Sync(t, \langle\rangle, cs)\} \quad (3.5.43)$$

$$TSync\left(\begin{array}{l}\langle(t_1, r_1)\rangle \frown S_1,\\ \langle(t_2, r_2)\rangle \frown S_2,\\ cs\end{array}\right) = \left\{\begin{array}{l}\langle(t', r')\rangle \mid t' \in Sync(t_1, t_2, cs) \wedge\\ r' = \left(\begin{array}{l}((r_1 \cup r_2) \cap cs) \cup\\ ((r_1 \cap r_2) \backslash cs)\end{array}\right)\end{array}\right\} \frown TSync(S_1, S_2, \mathsf{cs}) \quad (3.5.44)$$

The function *TSync* is defined in an algebraic manner. The first equation states that the function is symmetric, followed by the case in which the two input traces are empty, and then the result is the empty set. If one of the traces is empty, then the set is composed by the result of the synchronization of the trace element from the non-empty trace with the empty trace; the refusal set is not affected. If both traces are non-empty then first elements of the sequences are synchronized. The synchronization is defined by another function: *Sync*. The refusal set is calculated at each time instant, based on the refusals of both actions at the time instant. The refusal set is $((r_1 \cup r_2) \cap \mathsf{cs}) \cup ((r_1 \cap r_2) \backslash \mathsf{cs})$, which includes only events that are refused by both actions and do not appear in the synchronization set,and events that appear in the synchronization set and are refused by either action. The function *Sync* is defined in [59], and it gives the set of all possible combinations of untimed traces, given two untimed traces and a synchronization set. The following is a definition of the function *Sync*.

$$Sync(t_1, t_2, \mathsf{cs}) = Sync(t_2, t_1, \mathsf{cs}) \quad (3.5.45)$$

$$Sync(\langle\rangle, \langle\rangle, \mathsf{cs}) = \{\langle\rangle\} \quad (3.5.46)$$

$$Sync(\langle e_1\rangle \frown t, \langle\rangle, \mathsf{cs}) = \{\} \textit{ iff } e_1 \in \mathsf{cs} \quad (3.5.47)$$

$$Sync(\langle e_1\rangle \frown t, \langle\rangle, \mathsf{cs}) = \{\langle e_1\rangle\} \frown Sync(t_1, \langle\rangle, \mathsf{cs}) \textit{ iff } e_1 \notin \mathsf{cs} \quad (3.5.48)$$

$$Sync(\langle e_1\rangle \frown t_1, \langle e_2\rangle \frown t_2, \mathsf{cs}) = \left(\begin{array}{l}(\{\langle e_1\rangle\} \frown Sync(t_1, \langle e_2\rangle \frown t_2, \mathsf{cs})) \cup\\ (\{\langle e_2\rangle\} \frown Sync(\langle e_1\rangle \frown t_1, t_2, \mathsf{cs}))\end{array}\right) \textit{ iff } e_1 \notin \mathsf{cs} \wedge e_2 \notin \mathsf{cs} \quad (3.5.49)$$

$$Sync(\langle e_1\rangle \frown t_1, \langle e_2\rangle \frown t_2, \mathsf{cs}) = (\{\langle e_1\rangle\} \frown Sync(t_1, \langle e_2\rangle \frown t_2, \mathsf{cs})) \textit{ iff } e_1 \notin \mathsf{cs} \wedge e_2 \in \mathsf{cs} \quad (3.5.50)$$

$$Sync(\langle e_1\rangle \frown t_1, \langle e_1\rangle \frown t_2, \mathsf{cs}) = (\{\langle e_1\rangle\} \frown Sync(t_1, t_2, \mathsf{cs})) \textit{ iff } e_1 \in \mathsf{cs} \quad (3.5.51)$$

$$Sync(\langle e_1\rangle \frown t_1, \langle e_2\rangle \frown t_2, \mathsf{cs}) = \{\} \textit{ iff } e_1 \in \mathsf{cs} \wedge e_2 \in \mathsf{cs} \wedge e_1 \neq e_2 \quad (3.5.52)$$

The above definition of *Sync* shows that the function is symmetric, it synchronizes the untimed events that occur in both actions on the synchronization set. A similar function can be found in [45].

Finally, the parallel merge function TM adds two local indexed variables $0.s$ and $1.s$, they act as places holders for the input variables s_A and s_B. The variables are removed at the end of the merge function, and so are made local to the merge function. The need for this is to preserve the symmetry of our merge function needed as one of the requirements for the merge function validity. The proof of the validity of the parallel merge function introduced above is explored in details in Appendix D.

The parallel composition satisfies the following properties

Property 3.12

L1. $A \llbracket s_A \mid \{\!| cs |\!\} \mid s_B \rrbracket B = B \llbracket s_B \mid \{\!| cs |\!\} \mid s_A \rrbracket A$

Proof:

Implied by the symmetry of the merge function □

L2.
$$A \llbracket s_A \mid \{\!| cs |\!\} \mid (s_B \cup s_C) \rrbracket B \llbracket s_B \mid \{\!| cs |\!\} \mid s_C \rrbracket C) = (A \llbracket s_A \mid \{\!| cs |\!\} \mid s_B \rrbracket B) \llbracket s_A \cup s_B \mid \{\!| cs |\!\} \mid s_C \rrbracket C$$

Proof:

Implied by the associativity of the merge function □

L3. $Skip \llbracket \{\} \mid \{\!| \{\} |\!\} \mid \{\} \rrbracket Skip = Skip$

Proof:

Implied by the last property of the merge function □

L4. $A \llbracket s_A \mid \{\!| cs |\!\} \mid \{\} \rrbracket Chaos = Chaos$

Proof:

Implied from the validity of the merge function □

L5. $Stop \llbracket \{\} \mid \{\!| cs |\!\} \mid s_A \rrbracket c \rightarrow A = Stop$

Provided that $c \in cs$

L6.
$$(A \lhd b \rhd B) \llbracket s_A \cup s_B \mid \{\!| cs |\!\} \mid s_C \rrbracket C = (A \llbracket s_A \mid \{\!| cs |\!\} \mid s_C \rrbracket C) \lhd b \rhd (B \llbracket s_B \mid \{\!| cs |\!\} \mid s_C \rrbracket C)$$

L7.
$$A \llbracket s_A \mid \{\!| cs |\!\} \mid s_B \cup s_C \rrbracket (B \sqcap C) = (A \llbracket s_A \mid \{\!| cs |\!\} \mid s_B \rrbracket B) \sqcap (A \llbracket s_A \mid \{\!| cs |\!\} \mid s_C \rrbracket C)$$

L8. $(x := e;\ A) \llbracket s_A \mid \{\!| cs |\!\} \mid s_B \rrbracket B = (x := e);\ (A \llbracket s_A \mid \{\!| cs |\!\} \mid s_B \rrbracket B)$

providing that e *has no reference to* m *and* $x \in s_A$ *and* $x \notin s_B$.

L9.
$$(Wait\ \mathsf{n};\ A) \llbracket s_A \mid \{\!| cs |\!\} \mid s_B \rrbracket = Wait\ \mathsf{n};\ (A \llbracket s_A \mid \{\!| cs |\!\} \mid s_B \rrbracket B) \llbracket s_A \mid \{\!| cs |\!\} \mid s_B \rrbracket (Wait\ \mathsf{n};\ B)$$

Proof:

$$
\begin{array}{l}
\quad (Wait\ \ \mathsf{n};\ A) \\
[\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!] \\
\quad (Wait\ \ \mathsf{n};\ B) \\
= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [3.5.39] \\
((Wait\ \ \mathsf{n};\ A;\ U0(m)) \mid\mid (Wait\ \ \mathsf{n};\ B;\ U1(m)));\ TM(cs, s_A, s_B) \\
= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [3.5.32] \\
((Wait\ \ \mathsf{n};\ A;\ U0(m)) \wedge (Wait\ \ \mathsf{n};\ B;\ U1(m)));\ TM(cs, s_A, s_B) \\
= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [Lemma\ 3.7] \\
(Wait\ \ \mathsf{n};\ ((A;\ U0(m)) \wedge (B;\ U1(m))));\ TM(cs, s_A, s_B) \\
= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [Property\ 3.6\ L4] \\
Wait\ \ \mathsf{n};\ ((((A;\ U0(m)) \wedge (B;\ U1(m))));\ TM(cs, s_A, s_B)) \\
= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [3.5.32] \\
Wait\ \ \mathsf{n};\ ((((A;\ U0(m)) \mid\mid (B;\ U1(m))));\ TM(cs, s_A, s_B)) \\
= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [3.5.39] \\
Wait\ \ \mathsf{n};\ (A\ [\![\ s_A \mid \{\!|\ cs\ |\!\} \mid s_B\ ]\!]\ B) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Box
\end{array}
$$

L10. $$\left(\begin{array}{c} (a \to A) \,\Box\, (b \to B) \\ [\![s_A \cup s_B \mid \{\!|\ cs\ |\!\} \mid s_C]\!] \\ (c \to C) \end{array}\right) \setminus cs = \left(\begin{array}{c} \left(\begin{array}{c} (a \to A) \\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_C]\!] \\ (c \to C) \end{array}\right) \\ \Box \\ \left(\begin{array}{c} (b \to B) \\ [\![s_B \mid \{\!|\ cs\ |\!\} \mid s_C]\!] \\ (c \to C) \end{array}\right) \end{array}\right) \setminus cs$$

Provided that $(a \notin cs) \wedge (b \notin cs) \wedge (c \in cs)$

L11. $$\left(\begin{array}{c} (a \to A) \\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!] \\ (a \to A') \,\Box\, (b \to B) \end{array}\right) \setminus cs = (a \to (A\ [\![\ s_A \mid \{\!|\ cs\ |\!\} \mid s_B\ ]\!]\ A')) \setminus cs$$

Provided that $a \in cs$ *and* $b \in cs$

L12. $$\left(\begin{array}{c} (a \to A) \\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!] \\ \left(\begin{array}{c} (Wait\ \ \mathsf{d};\ (a \to A')) \\ \Box \\ (b \to B) \end{array}\right) \end{array}\right) \setminus cs = \begin{array}{l} Wait\ \ \mathsf{d}; \\ (a \to (A\ [\![\ s_A \mid \{\!|\ cs\ |\!\} \mid s_B\ ]\!]\ A')) \setminus cs \end{array}$$

Provided that $a \in cs$ *and* $b \in cs$

L13. $$\left(\begin{array}{c} (a \to A) \\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!] \\ (b \to B) \end{array}\right) = a \to \left(\begin{array}{c} A \\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!] \\ (b \to B) \end{array}\right)$$

Provided that $a \notin cs$

The parallel composition of two actions P and Q is healthy if both actions are healthy. The prove of such is due to the close of the healthiness conditions over conjunction. The sequential composition of the merge function result with *Skip* assures the healthiness of the resulting action for the cases in which the healthiness conditions are not closed over conjunction as is the case of $CSP2_t$. For details of the healthiness conditions and their properties refer to Appendix B. □

3.5.10 Hiding

The hiding operator in CSP hides events from the environment. The events occur internally and are not registered in the program observation. When events are hidden, they occur automatically and instantaneously, as soon as they can. The CSP hide is defined in UTP as follows.

$$P(tr', ref') \setminus E \mathrel{\widehat{=}} R(\exists\, s \bullet P(s, (E \cup ref')) \wedge L);\ Skip \tag{3.5.53}$$

where

$$L \mathrel{\widehat{=}} (tr' - tr) = (s - tr) \downarrow (AP - E) \tag{3.5.54}$$

The definition of hiding is given by taking an arbitrary valid trace s of the process P, and restricting it to the set $(AP - E)$, which contains all the events in the alphabet of the process P (AP) except for those that occur in E. The hidden events are also added to the refusal set $(E \cup ref')$ of the resulting process observations

The following is a definition for the hide operator in the time model.

$$A \setminus cs \quad \widehat{=} \quad \begin{array}{l} R_t \left(\begin{array}{l} \exists\, s_t \bullet A[s_t / tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array} \right); \\ Skip \end{array} \tag{3.5.55}$$

We use a time trace restriction operator $\downarrow_t$, which, similarly to the sequence operator, restricts the resulting traces at each time unit to the the given set of events cs. Because time traces register the refusals at the end of each time unit, we add the complement of the restricted events $(Events - cs)$ to the refusal of the resulting time traces. The time trace restriction is defined as follows: given any two time traces t_a and t_b

$$t_b = t_a \downarrow_t cs \Leftrightarrow \forall\, i : 1..\#t_a \bullet \left(\begin{array}{l} fst(t_b(i)) = fst(t_a(i)) \downarrow cs \wedge \\ snd(t_a(i)) = (snd(tr_b(i)) \cup (Events - cs)) \wedge \\ \#t_a = \#t_b \end{array} \right) \tag{3.5.56}$$

It is important to notice that the time trace restriction does not change the size of the time trace, this is imposed by the predicate $\#t_a = \#t_b$. The time trace restriction satisfies the following properties

Property 3.13

L1. $t_a \downarrow_t Events = t_a$

L2. $\langle (t, r) \rangle \downarrow_t cs = \langle ((t \downarrow cs), (r \cup (Events - cs))) \rangle$

L3. $t_a \frown \langle (t, r) \rangle \downarrow_t cs = (t_a \downarrow_t cs) \frown (\langle (t, r) \rangle \downarrow_t cs)$

L4. $\#(t_a \downarrow_t cs) = \# t_a$

L5. $t_a \downarrow_t cs_1 \downarrow_t cs_2 = t_a \downarrow_t (cs_1 \cap cs_2)$

The proof of the above properties are listed in Appendix D

The sequential composition with *Skip* is to guarantee that hiding is healthy; the proof of the healthiness of the hiding operator is left as future work. The following are some algebraic properties of hiding.

Property 3.14

L1. $A \setminus \{\} = A$
Provided that A is a healthy Circus Time Action

L2. $A \setminus cs_1 \setminus cs_2 = A \setminus (cs_1 \cup cs_1)$
Provided that A is a healthy Circus Time Action

L3. $(A \sqcap B) \setminus cs = (A \setminus cs) \sqcap (B \setminus cs)$

L4. $(c \rightarrow A) \setminus cs = c \rightarrow (A \setminus cs)$ *if* $c \notin cs$

L5. $(c \rightarrow A) \setminus cs = A \setminus cs$ *if* $c \in cs$

L6. $(Wait\ \ \mathsf{n}) \setminus cs = Wait\ \ \mathsf{n}$

L7. $Skip \setminus cs = Skip$

L8. $(A \lhd b \rhd B) \setminus cs = (A \setminus cs) \lhd b \rhd (B \setminus cs)$

L9. $((a \rightarrow Skip) \Box (b \rightarrow Skip)) \setminus \{a\} = (Skip \Box (b \rightarrow Skip))$

L10. $((a \rightarrow A) \Box (Wait\ \ \mathsf{n};\ B)) \setminus a = A \setminus \{a\}$

L11. $(A;\ B) \setminus cs = (A \setminus cs);\ (B \setminus cs)$

L12. $Chaos \setminus cs = Chaos$

L13. $(x := e) \setminus cs = x := e$

L14. $((a \rightarrow A) \Box (b \rightarrow B)) \setminus cs = ((a \rightarrow (A \setminus cs)) \Box (b \rightarrow (B \setminus cs)))$
Provided that $(a \notin cs) \land (b \notin cs)$

The proof of the above properties can be found in Appendix D.

3.5.11 Recursion

An action A can be made less deterministic by adding the possibility that it behaves like another action B. In general an increase in nondeterminism makes things worse, as it becomes harder to predict how the new action will behave. This can be expressed as follows

$$[A \Rightarrow (A \sqcap B)] \tag{3.5.57}$$

Where the square brackets stand for universal quantification over all observation variables. From the above law we can determine an ordering relation between actions. We use the $\sqsupseteq$ to stand for a refinement relation where $A \sqsupseteq B$ states that action A is more deterministic than an action B and, is defined as follows.

$$A \sqsupseteq B \mathrel{\widehat{=}} [A \Rightarrow B] \tag{3.5.58}$$

The action *Chaos* is the most nondeterministic action, as nothing can be assured by its behaviour: it is totally arbitrary; so that a nondeterministic choice between *Chaos* and any action has no affect over the nondeterminism of *Chaos*. Therefore, any action A is more deterministic than *Chaos*.

$$\begin{aligned} & A \sqsupseteq Chaos && [3.5.58] \\ & = [A \Rightarrow Chaos] && [\text{property 3.9 L1}] \\ & = [A \Rightarrow (Chaos \sqcap A)]\Box \end{aligned}$$

This makes *Chaos* the bottom (weakest) element of our ordering relation.

The operator $\sqcap$ may be used to describe nondeterministic behaviour in terms of any finite set of n more deterministic behaviours

$$P_1 \sqcap P_2 \sqcap P_3 \ldots P_n$$

Let S be a set of actions, we use the term $\sqcap S$ to describe a system that behaves in any of the ways described by any of the actions in S. Because we cannot define infinite nondeterminism in terms of finite nondeterminism, therefore it is not possible to find a general definition for $\sqcap S$. The UTP gives the definition of $\sqcap S$ as an axiom as follows.

Property 3.15

L1. $X \sqsupseteq \sqcap S$ *for all* X *in* S

L2. If $X \sqsupseteq P$ *for all* X *in* S*, then* $\sqcap S \sqsupseteq P$

The first law states that: in the ordering relation, the disjunction of a set is the lower bound of all its members, and the second states that it is the greatest such lower bound. Therefore, considering the ordering relation $\sqsupseteq$, $\sqcap$ is the *greatest lower bound*.

An empty disjunction yields a predicate *false* which can certainly never be implemented

$$\sqcap\{\} = false$$

If it existed it would satisfy every specification as

$$[false \Rightarrow P]$$

Such a program is known as *Miracle* and can not be implemented by any action in our program. Nevertheless it is important to notice that Miracle exists as a mathematical abstraction in our ordering relation. Miracle is the top of our relation, it is the strongest of all actions and no action is more deterministic than Miracle.

Using our ordering relation we can also define equality of actions as given by the following theorem

Theorem 3.1 *For any two actions A and B we say that: action A is equivalent to action B if and only if $A \sqsupseteq B \land B \sqsupseteq A$*

We conclude that the set of observations in our model form a complete lattice with respect to the relation $\sqsupseteq$, having *Chaos* as its bottom element, and $\sqcap$ as its *greatest lower bound operator*.

Let X be a variable that stands for a call to a recursive program, which we are about to define. We may use X more than once to build the body of the recursive program denoted by $F(X)$. We define the semantics of recursion as the *weakest fixed point* [26], given by the following.

$$\mu\, X \bullet F(X) \mathrel{\widehat{=}} \sqcap \{X \mid X \sqsupseteq F(X)\}$$

Property 3.16

L1. $Y \sqsupseteq \mu\, F$ *whenever* $Y \sqsupseteq F(Y)$

L2. $F(\mu\, F) \equiv \mu\, F$

The proof of the above properties is given in UTP. The proviso for the proof is that F is *monotonic*. We show in Appendix C that all the constructs of our language are monotonic; as a consequence F is monotonic □

3.5.12 Timeout

The timeout operator takes a time value and combines two actions. The first action should react to the environment within the given time period, otherwise the second action will take place. We can model this with the help of the external choice.

$$\mathsf{A} \overset{\mathsf{d}}{\rhd} \mathsf{B} = (A \,\Box\, (Wait\ \ \mathsf{d};\ int \rightarrow B)) \setminus \{int\} \tag{3.5.59}$$

The action $\mathsf{A} \overset{\mathsf{d}}{\rhd} \mathsf{B}$ can only behave as A if this action engages in a communication or terminates before the wait period d elapses. The event *int* is taken to be an event that is not used by A or B. The main objective of adding this event is to trigger the external choice and force it to select the second option. The event *int* is hidden from the rest of the environment.

The next properties apply to the timeout operator.

Property 3.17

L1. $Skip \overset{\mathsf{d}}{\rhd} A = Skip$

L2. $Stop \overset{\mathsf{d}}{\rhd} A = Wait\ \ \mathsf{d};\ A$

L3. $(a \rightarrow A) \overset{\mathsf{d}}{\rhd} (b \rightarrow A) = ((a \rightarrow Skip) \overset{\mathsf{d}}{\rhd} (b \rightarrow Skip));\ A$

L4. $A \overset{\mathsf{d}}{\rhd} (B \sqcap C) = (A \overset{\mathsf{d}}{\rhd} B) \sqcap (A \overset{\mathsf{d}}{\rhd} C)$

L5. $(Wait\ \ \mathsf{d};\ A) \overset{\mathsf{d}+\mathsf{d}'}{\rhd} B = Wait\ \ \mathsf{d};\ (A \overset{\mathsf{d}'}{\rhd} B)$

L6. $(Wait\ \ \mathsf{d}+\mathsf{d}';\ A) \overset{\mathsf{d}}{\rhd} B = Wait\ \ \mathsf{d};\ B$

L7. $(A \Box B) \overset{\mathsf{d}}{\rhd} C = (A \overset{\mathsf{d}}{\rhd} C) \Box (B \overset{\mathsf{d}}{\rhd} C)$

L8. $A \overset{\mathsf{0}}{\rhd} B = (A \Box (int \rightarrow B)) \setminus \{int\}$

L9. $Wait\ \ \mathsf{0} \overset{\mathsf{d}}{\rhd} B = Skip$

L10. $A \overset{\mathsf{d}}{\rhd} (B \Box C) = (A \overset{\mathsf{d}}{\rhd} B) \Box (A \overset{\mathsf{d}}{\rhd} C)$

See Appendix D for detailed proofs of the above properties.

3.6 CONCLUDING REMARKS

In this chapter we studied the semantics of the constructs of *Circus* Time Action and, when ever possible, we related them to the UTP constructs and model. The next chapter explores the relation of the UTP model and *Circus* Time Action model in more details; semantic mapping relating predicates from one model to the other are introduced and, then used to create a link between the two theories.

CHAPTER 4

LINKING MODELS

When studying the features and models of a language, it is important to relate them to other models and languages. A special benefit of using Unifying Theories of Programming (UTP) is that these relations can be explored in a natural way. Different models are characterized by different conditions on the predicates that compose the model. The difference in the expressive power of the models is a motivation to explore the relations between them: we want to make the most of all models. Studying the relation that exists between the models makes it possible to create a solid semantic base for frameworks that use integrated techniques and tools.

In this chapter we start by giving a brief introduction to linking theories as presented in the UTP, and explore the benefits of using such links. Afterwards, we present a relation between the *Circus* Time Action model and the untimed model of the Unifying Theories of Programming. An inverse mapping, which links the untimed model to the *Circus* Time Action model is presented as well. We show that the combination of such links is a *Galois Connection*. Finally we study, through an example, the use of such relations to explore program properties common to both models.

4.1 LINKING THEORIES: THE UNIFYING THEORIES OF PROGRAMMING APPROACH

A benefit of using the UTP is the ability to study and relate programming theories and paradigms that are different in many aspects, but share common characteristics. The UTP introduces a general relation between predicates which links an abstract specification $S(a)$ to a concrete design $D(c)$, where a is a set of variables used in the definition of the abstract specification S, and c is the set of variables used to define the concrete design D. The UTP defines the relation with the aid of a mapping function $f(c, a)$, that relates the concrete variables to the abstract equivalent variables. The following is a general mapping from a concrete predicate to a more abstract predicate.

$$L(D(c)) \mathrel{\widehat{=}} \exists\, c \bullet D(c) \wedge f(c, a) \tag{4.1.1}$$

The above predicate lifts the design to the specification level. It gives the strongest specification with alphabet a that satisfies the design D. The inverse relation is defined using universal quantification, as follows

$$R(S(a)) \mathrel{\widehat{=}} \forall\, a \bullet f(c, a) \Rightarrow S(a) \tag{4.1.2}$$

The above predicate gives the weakest design with alphabet c that is guaranteed to satisfy the specification.

The mapping functions described above are used to show the equivalence of two programs in different semantic models in the UTP. A program P in a concrete model

is equivalent to a program Q in an abstract model if, and only if, P and Q satisfy the following condition.

$$[L(P(c)) \Rightarrow Q(a)] iff [P(c) \Rightarrow R(Q(a))] \tag{4.1.3}$$

The square brackets stand for the universal quantification over all the observation variables.

This general approach is presented and explored in detail in the UTP. The links between models was classified in two distinct parts. First the subset theory: in such a mapping, the observations of the abstract model S are a subset of the more concrete observations of model P; the observation variables in the two models are the same and have the same meaning. The most common mapping of such type is the identity function. Inverse functions that map the more abstract model back to a concrete model are also explored. Several different types of subset mapping functions are presented by the UTP and the different characteristics of each mapping are explored in details.

The second type of link between models: this type of link maps observations between models with different alphabets; in such a case, special care is given to the name and meaning of each alphabet variable in the two models; for some variables in a model might not have an equivalent observation in the target model. This type of mapping is more interesting because different programming paradigms need different observation variables. An example of such mapping is explored in details in the next section. We study the mapping of observations from the time model to the UTP model (more abstract model).

4.2 A CONSERVATIVE MAPPING L

Our model was developed with the aim of capturing time information in the semantics of the language, but also preserving the untimed semantics of our programs in the time model. To show the relation between the two models, we create a function L which, given a predicate that represents the time semantics of a *Circus* Time Action A, returns the equivalent behaviour in the original UTP model without time information. This function is defined as follows.

$$L([\![A]\!]_{time}) \mathrel{\widehat{=}} \exists\, tr_t, tr'_t \bullet [\![A]\!]_{time} \wedge f(tr, tr', ref, ref', tr_t, tr'_t) \tag{4.2.1}$$

where f is defined as follows:

$$\begin{aligned} f(tr, tr', ref, ref', tr_t, tr'_t) \;\mathrel{\widehat{=}}\; & tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge \\ & ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{aligned} \tag{4.2.2}$$

The function L maps the timed semantics of an action A, given by $[\![A]\!]_{time}$, to a predicate in the untimed model. This is done by hiding the time traces, tr_t and tr'_t, while introducing the untimed observation variables as follows: tr and tr' applying the *Flat* function to the timed traces tr_t and tr'_t; a projection on the second element of the last entry in tr_t and tr'_t results in the refusal sets ref and ref'.

An important characteristic of L is that, for any action X, $L(X)$ does not change the untimed behaviour of X. In the following proof, we consider the predicate *true*, and

show that

$$L(true) = true \tag{4.2.3}$$

Proof:

$$\begin{array}{ll}
& L(true) \\
= & [4.2.1] \\
& \exists\, tr_t, tr'_t \bullet true \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \\
= & [\text{propositional calculus}] \\
& \exists\, tr_t, tr'_t \bullet \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \\
& [\text{Let } tr_t = \langle (tr, ref) \rangle \text{ and } tr'_t = \langle (tr', ref') \rangle] \\
\Leftarrow & \left(\begin{array}{c} tr = Flat(\langle (tr, ref) \rangle) \wedge \\ tr' = Flat(\langle (tr', ref') \rangle) \wedge \\ ref = snd(last(\langle (tr, ref) \rangle)) \wedge \\ ref' = snd(last(\langle (tr', ref') \rangle)) \end{array} \right) \\
= & [\text{Property of } Flat] \\
& \left(\begin{array}{c} tr = tr \wedge tr' = tr' \wedge \\ ref = snd(last(\langle (tr, ref) \rangle)) \\ \wedge\, ref' = snd(last(\langle (tr', ref') \rangle)) \end{array} \right) \\
= & [\text{Properties of } last \text{ and } snd] \\
& \left(\begin{array}{c} tr = tr \wedge tr' = tr' \wedge \\ ref = ref \wedge ref' = ref' \end{array} \right) \\
= & [\text{Property of equaltity}] \\
& true \qquad \square
\end{array}$$

The case that the action X is *false*

$$L(false) = false \tag{4.2.4}$$

Proof:

$$\begin{array}{ll}
& L(false) \\
= & [4.2.1]
\end{array}$$

$$\exists\, tr_t, tr'_t \bullet false \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array}\right)$$

$=$ [Propositional calculus]

$\exists\, tr_t, tr'_t \bullet false$

$=$ [Predicate calculus]

$false$ $\square$

Before we explore the other properties of the function L we list some helpful algebraic laws.

Property 4.1

L1. *L is idempotent, this is important to show that the function L has no effect when applied more than once to a program*
$L(L(X)) = L(X)$

L2. *L distributes over disjunction*
$L(X \vee Y) = L(X) \vee L(Y)$

L3. *Weak distribution over conjunction, this is because L uses existential quantification*
$L(X \wedge Y) \Rightarrow L(X) \wedge L(Y)$

L4. *Distribution over conjunction*
$L(X \wedge Y) = L(X) \wedge Y$
provided that tr_t and tr'_t are not free in Y, because, tr_t and tr'_t are the quantification variables in L.

L5. *Distribution over conditional choice*
$L(X \lhd b \rhd Y) = L(X) \lhd b \rhd L(Y)$
provided that tr_t and tr_t are not free in the expression b

L6. $L(X;\ Y) = L(X);\ L(Y)$

L7. $L(Expands(tr_t, tr'_t)) = tr \leq tr'$

L8. $L(tr'_t = tr_t) = (tr' = tr \wedge ref' = ref)$

L9. $L(\Pi_t) = \Pi$

L10. $L(trace' = s) = ((tr' - tr) = s)$
For any arbitrary trace s.

L11. $L(wait\_com(c)) = (wait' \wedge c \notin ref' \wedge tr' = tr)$

L12. $L(term\_com(c.e)) = (\neg wait' \wedge tr' - tr = \langle c.e \rangle)$

L13. $L(terminating\_com(c.e)) = (\neg wait' \land tr' - tr = \langle c.e \rangle)$

The proof of the above properties can be found in Appendix E.

Another important desired property of function L is that it should map healthy actions in a timed model to healthy actions in the untimed model. For any timed healthiness condition H_t in the time model, the function L should map H_t healthy predicates to H healthy predicates, where H is a healthiness condition in the untimed model; i.e. for any action X, we should have, $L(H_t(X)) = H(L(X))$. The following characterizes the result of applying the function L to healthy predicates.

Property 4.2

L1. $L(R1_t(X)) = R1(L(X))$

L2. $L(R2_t(X)) = R2(L(X))$

L3. $L(R3_t(X)) = R3(L(X)$

L4. $L(CSP1_t(X)) = CSP1(L(X))$

L5. $L(CSP2_t(X)) = CSP2(L(X))$

L6. $L(CSP3_t(X)) = CSP3(L(X))$

L7. $L(CSP4_t(X)) = CSP4(L(X))$

L8. $L(CSP5_t(X)) = CSP5(L(X))$

The proof of the above properties are detailed in Appendix E.

By applying the function L to *Circus* Time Action constructs we obtain programs in the untimed model. The following properties show the result of applying the function L to the different *Circus* Time Action constructs. Because the syntax of *Circus* Time Action and *Circus* are similar, in the next properties we use the notation $[\![X]\!]_{time}$ to represent the action X in the time model, and $[\![X]\!]$ to stand for the the same action in the untimed model.

Property 4.3

L1. $L([\![x := e]\!]_{time}) = [\![x := e]\!]$

Proof:

$$
\begin{array}{ll}
& L([\![x := e]\!]_{time}) \\
= & \qquad [3.5.7] \\
& L(CSP1_t(R_t \left(\begin{array}{l} ok = ok' \land wait = wait' \land \\ tr'_t = tr_t \land state' = state \oplus \{x \mapsto e\} \end{array} \right))) \\
= & \qquad [property\ 4.2\ L1,L2,L3] \\
& CSP1(R(L \left(\begin{array}{l} ok = ok' \land wait = wait' \land \\ tr'_t = tr_t \land state' = state \oplus \{x \mapsto e\} \end{array} \right)))
\end{array}
$$

$$
\begin{array}{l}
= \qquad \textit{[property 4.1 L4]} \\
CSP1(R \left(\begin{array}{l} ok = ok' \wedge wait = wait' \wedge \\ L(tr'_t = tr_t) \wedge state' = state \oplus \{x \mapsto e\} \end{array} \right)) \\
= \qquad \textit{[property 4.1 L8]} \\
CSP1(R \left(\begin{array}{l} ok = ok' \wedge wait = wait' \wedge \\ tr' = tr \wedge ref' = ref \wedge state' = state \oplus \{x \mapsto e\} \end{array} \right)) \\
= \qquad \textit{[definition of assignment in Circus]} \\
[\![x := e]\!] \qquad \square
\end{array}
$$

L2. $L([\![Skip]\!]_{time}) = [\![Skip]\!]$

L3. $L([\![Stop]\!]_{time}) = [\![Stop]\!]$

L4. $L([\![Chaos]\!]_{time}) = [\![Chaos]\!]$
Law L5 explores the application of the function L to the construct Wait t*; notice that the result is either a Skip or a Stop in the untimed model. This is relevant for our further discussion of the properties of L.*

L5. $L([\![Wait\ \ \mathsf{t}]\!]_{time}) = [\![Stop]\!] \sqcap [\![Skip]\!]$
Proof:
First we show that $L([\![Wait\ \ \mathsf{t}]\!]_{time}) \Leftarrow [\![Stop]\!] \sqcap [\![Skip]\!]$

$$
\begin{array}{l}
L([\![Wait\ \ \mathsf{t}]\!]_{time}) \\
= \qquad \textit{[3.5.8]} \\
L(CSP1_t(R_t(ok' \wedge delay(\mathsf{t}) \wedge (trace' = \langle\rangle)))) \\
= \qquad \textit{[property 4.2 L4]} \\
CSP1(L(R_t(ok' \wedge delay(\mathsf{t}) \wedge (trace' = \langle\rangle)))) \\
\Rightarrow \qquad \textit{[property 4.2 L1,L2 and L3]} \\
CSP1(R(L(ok' \wedge delay(\mathsf{t}) \wedge (trace' = \langle\rangle)))) \\
= \qquad \textit{[property 4.1 L4]} \\
CSP1(R(ok' \wedge L(delay(\mathsf{t}) \wedge (trace' = \langle\rangle)))) \\
= \qquad \textit{[3.5.9 and proposition logic]} \\
CSP1 \left(R \left(ok' \wedge L \left(\begin{array}{c} \left(\begin{array}{c} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathsf{t} \wedge \\ (trace' = \langle\rangle) \end{array} \right) \vee \\ \left(\begin{array}{c} \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = \mathsf{t} \wedge \\ state' = state \wedge \\ (trace' = \langle\rangle) \end{array} \right) \end{array} \right) \right) \right) \\
= \qquad \textit{[4.2.1]}
\end{array}
$$

$$
CSP1\left(R\left(ok' \wedge \left(\exists\, tr_t, tr'_t \bullet \begin{pmatrix} \begin{pmatrix} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \\ (trace' = \langle\rangle) \end{pmatrix} \vee \\ \begin{pmatrix} \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \\ state' = state \wedge \\ (trace' = \langle\rangle) \end{pmatrix} \end{pmatrix} \wedge \begin{pmatrix} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{pmatrix}\right)\right)\right)
$$

$=$ *[3.3.1]*

$$
CSP1\left(R\left(ok' \wedge \left(\exists\, tr_t, tr'_t \bullet \begin{pmatrix} \begin{pmatrix} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \\ (Flat(tr'_t) - Flat(tr_t) = \langle\rangle) \end{pmatrix} \vee \\ \begin{pmatrix} \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \\ state' = state \wedge \\ (Flat(tr'_t) - Flat(tr_t) = \langle\rangle) \end{pmatrix} \end{pmatrix} \wedge \begin{pmatrix} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{pmatrix}\right)\right)\right)
$$

$=$ *[sequence properties]*

$$
CSP1\left(R\left(ok' \wedge \left(\exists\, tr_t, tr'_t \bullet \begin{pmatrix} \begin{pmatrix} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \\ (Flat(tr'_t) = Flat(tr_t)) \end{pmatrix} \vee \\ \begin{pmatrix} \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \\ state' = state \wedge \\ (Flat(tr'_t) = Flat(tr_t)) \end{pmatrix} \end{pmatrix} \wedge \begin{pmatrix} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{pmatrix}\right)\right)\right)
$$

$\Leftarrow$ *[Let $tr_t = \langle (tr, ref) \rangle$]*

[and $tr'_t = \langle (tr', ref') \rangle$]

$$CSP1\left(R\left(ok' \wedge \left(\begin{array}{c}\left(\begin{array}{c}\left(\begin{array}{c}wait' \wedge \\ (\#\langle(tr', ref')\rangle - \#\langle(tr, ref)\rangle) \leq \mathtt{t} \wedge \\ (Flat(\langle(tr', ref')\rangle) = Flat(\langle(tr, ref)\rangle)\end{array}\right) \vee \\ \left(\begin{array}{c}\neg wait' \wedge state = state' \wedge \\ (\#\langle(tr', ref')\rangle - \#\langle(tr, ref)\rangle) = \mathtt{t} \wedge \\ (Flat(\langle(tr', ref')\rangle) = Flat(\langle(tr, ref)\rangle)\end{array}\right)\end{array}\right) \wedge \\ \left(\begin{array}{c}tr = Flat(\langle(tr, ref)\rangle) \wedge \\ tr' = Flat(\langle(tr', ref')\rangle) \wedge \\ ref = snd(last(\langle(tr, ref)\rangle)) \wedge \\ ref' = snd(last(\langle(tr', ref')\rangle))\end{array}\right)\end{array}\right)\right)\right)$$

$=$ *[properties of Flat, snd and last]*

$$CSP1\left(R\left(ok' \wedge \left(\begin{array}{c}\left(\begin{array}{c}\left(\begin{array}{c}wait' \wedge \\ 0 \leq \mathtt{t} \wedge \\ tr' = tr\end{array}\right) \vee \\ \left(\begin{array}{c}\neg wait' \wedge state = state' \wedge \\ 0 = \mathtt{t} \wedge \\ state' = state \wedge \\ tr' = tr\end{array}\right)\end{array}\right) \wedge \\ \left(\begin{array}{c}tr = tr \wedge \\ tr' = tr' \wedge \\ ref = ref \wedge \\ ref' = ref'\end{array}\right)\end{array}\right)\right)\right)$$

$=$ *[Proposition logic]*

$$CSP1\left(R\left(ok' \wedge \left(\begin{array}{c}\left(\begin{array}{c}wait' \wedge \\ 0 \leq \mathtt{t} \wedge \\ tr' = tr\end{array}\right) \vee \\ \left(\begin{array}{c}\neg wait' \wedge state = state' \wedge \\ 0 = \mathtt{t} \wedge \\ state' = state \wedge \\ tr' = tr\end{array}\right)\end{array}\right)\right)\right)$$

$=$ *[Healthiness conditions distribute over $\vee$ and predicate calculus]*

$$CSP1\left(R\left(ok' \wedge \left(\begin{array}{c}wait' \wedge \\ 0 \leq \mathtt{t} \wedge \\ tr' = tr\end{array}\right)\right)\right) \vee CSP1\left(R\left(ok' \wedge \left(\begin{array}{c}\neg wait' \wedge state = state' \wedge \\ 0 = \mathtt{t} \wedge \\ state' = state \wedge \\ tr' = tr\end{array}\right)\right)\right)$$

$=$ *[3.5.1,3.5.4]*

$[\![Stop]\!] \sqcap [\![Skip]\!]$ $\square$

Next we show that $L([\![Wait\ \ \mathtt{t}]\!]_{time}) \Rightarrow [\![Stop]\!] \sqcap [\![Skip]\!]$

$$
\begin{array}{ll}
& L([\![Wait\ \ \mathtt{t}]\!]_{time}) \\
= & \textit{[3.5.8]} \\
& L(CSP1_t(R_t(ok' \wedge delay(\mathtt{t}) \wedge (trace' = \langle\rangle)))) \\
= & \textit{[property 4.2 L4]} \\
& CSP1(L(R_t(ok' \wedge delay(\mathtt{t}) \wedge (trace' = \langle\rangle)))) \\
\Rightarrow & \textit{[property 4.2 L1,L2 and L3]} \\
& CSP1(R(L(ok' \wedge delay(\mathtt{t}) \wedge (trace' = \langle\rangle)))) \\
= & \textit{[property 4.1 L4]} \\
& CSP1(R(ok' \wedge state = state' \wedge L(delay(\mathtt{t}) \wedge (trace' = \langle\rangle)))) \\
= & \textit{[3.5.9 and proposition logic]} \\
& CSP1\left(R\left(ok' \wedge L\left(\begin{array}{c} \left(\begin{array}{c} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \end{array}\right) \vee \\ (trace' = \langle\rangle) \\ \left(\begin{array}{c} \neg wait' \wedge state = state' \wedge \\ (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \end{array}\right) \\ (trace' = \langle\rangle) \end{array}\right)\right)\right) \\
= & \textit{[property 4.1 L2]} \\
& CSP1\left(R\left(ok' \wedge \begin{array}{c} L\left(\begin{array}{c} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \end{array}\right) \vee \\ (trace' = \langle\rangle) \\ L\left(\begin{array}{c} \neg wait' \wedge state = state' \wedge \\ (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \end{array}\right) \\ (trace' = \langle\rangle) \end{array}\right)\right) \\
= & \textit{[property 4.1 L4]} \\
& CSP1\left(R\left(ok' \wedge \begin{array}{c} wait' \wedge L\left(\begin{array}{c} (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \\ (trace' = \langle\rangle) \end{array}\right) \vee \\ \neg wait' \wedge state = state' \wedge L\left(\begin{array}{c} (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \\ (trace' = \langle\rangle) \end{array}\right) \end{array}\right)\right) \\
= & \textit{[property 4.1 L10]} \\
& CSP1\left(R\left(ok' \wedge \begin{array}{c} \left(\begin{array}{c} (wait' \wedge tr' = tr) \wedge \\ L\left(\begin{array}{c} (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \\ (trace' = \langle\rangle) \end{array}\right) \end{array}\right) \vee \\ \left(\begin{array}{c} (\neg wait' \wedge state = state' \wedge tr' = tr) \wedge \\ L\left(\begin{array}{c} (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \\ (trace' = \langle\rangle) \end{array}\right) \end{array}\right) \end{array}\right)\right) \\
\Rightarrow & \textit{[Predicate calculus]} \\
& CSP1\left(R\left(ok' \wedge \left(\begin{array}{c} wait' \wedge tr' = tr \vee \\ \neg wait' \wedge state = state' \wedge tr' = tr \end{array}\right)\right)\right)
\end{array}
$$

$$
\begin{array}{ll}
= & \textit{[Predicate calculus]} \\
\quad CSP1(R(ok' \land wait' \land tr' = tr)) \lor & \\
\quad CSP1(R(ok' \land state = state' \land \neg wait' \land tr' = tr)) & \\
= & \textit{[3.5.1,3.5.4]} \\
[\![Stop]\!] \sqcap [\![Skip]\!] & \Box
\end{array}
$$

An informal explanation to what happens in this case is that by hiding the time information the program that waits for a determined amount of time is similar to a program that can either wait forever (Stop) or terminate immediately Skip.

L6. $L([\![comm]\!]_{time}) = [\![comm]\!]$

L7. $L([\![A \lhd b \rhd B]\!]_{time}) = [\![L(A) \lhd b \rhd L(B)]\!]$

L8. $L([\![p\& \ A]\!]_{time}) = [\![p\& \ L(A)]\!]$

L9. $L([\![A \sqcap B]\!]_{time}) = [\![L(A) \sqcap L(B)]\!]$

L10. $L([\![A \ \Box \ B]\!]_{time}) = [\![L(A) \ \Box \ L(B)]\!]$

L11. $L([\![A; \ B]\!]_{time}) = [\![L(A); \ L(B)]\!]$

L12. $L([\![A \ [\![\ s_A \mid \{\!| \ cs \ |\!\} \mid s_B \ ]\!] \ B]\!]_{time}) = L([\![A]\!]_{time}) \ [\![\ s_A \mid \{\!| \ cs \ |\!\} \mid s_B \ ]\!] \ L([\![B]\!]_{time})$

L13. $L([\![A \setminus cs]\!]_{time}) = L([\![A]\!]_{time}) \setminus cs$

L14. $L([\![\ \mu \, X \bullet A(X)]\!]_{time}) = \mu \, X' \bullet L(A(X'))$

Detailed proof of the properties above are listed in Appendix E.

The function L is an abstraction function; it hides the time information and gives a weaker representation of the same program without time information. As a result of the weakening, the function L can only give a certain best approximation of the same meaning in the weaker theory. So, there can not be an exact inverse of L; nevertheless, it is possible to find a function R which as far as possible undoes the effect of L, with some unavoidable weakening. We introduce the function R as the weak inverse of the function L. Given an untimed action B, the function R returns the weakest timed program with the same behaviour as B.

$$R([\![B]\!]) \ \widehat{=} \ \sqcap \{[\![A]\!]_{time} \mid L([\![A]\!]_{time}) \sqsupseteq [\![B]\!]\} \tag{4.2.5}$$

Because R is a weak inverse of L then there is an unavoidable loss of information when applying R to the result of L. The following theorem captures this aspect.

Theorem 4.1 *Given any Circus Time Action program X then, $R(L(X))$ gives a weaker action than X*

$$X \sqsupseteq R(L(X)) \tag{4.2.6}$$

Proof:

$$
\begin{array}{ll}
& R(L(X)) \\
= & [4.2.5] \\
& \sqcap \{A \mid L(A) \sqsupseteq L(X)\} \\
= & [4.2.1] \\
& \sqcap \left\{ A \mid \begin{array}{l} \exists\, tr_t, tr'_t \bullet A \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \\ \sqsupseteq \\ \exists\, tr_t, tr'_t \bullet X \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right\} \\
\Leftarrow & [\text{Let } tr_t = \langle (tr, ref) \rangle] \\
& [\text{and } tr'_t = \langle (tr', ref') \rangle] \\
& \sqcap \left\{ A \mid \begin{array}{l} A[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t] \wedge \left(\begin{array}{c} tr = Flat(\langle (tr, ref) \rangle) \wedge \\ tr' = Flat(\langle (tr', ref') \rangle) \wedge \\ ref = snd(last(\langle (tr, ref) \rangle)) \wedge \\ ref' = snd(last(\langle (tr', ref') \rangle)) \end{array} \right) \\ \sqsupseteq \\ X[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t] \wedge \left(\begin{array}{c} tr = Flat(\langle (tr, ref) \rangle) \wedge \\ tr' = Flat(\langle (tr, ref) \rangle) \wedge \\ ref = snd(last(\langle (tr, ref) \rangle)) \wedge \\ ref' = snd(last(\langle (tr, ref) \rangle)) \end{array} \right) \end{array} \right\} \\
= & [\text{property of } Flat, \text{ } snd \text{ and } last] \\
& \sqcap \left\{ A \mid \begin{array}{l} A[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t] \wedge \left(\begin{array}{c} tr = tr \wedge tr' = tr' \wedge \\ ref = ref \wedge ref' = ref' \end{array} \right) \\ \sqsupseteq \\ X[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t] \wedge \left(\begin{array}{c} tr = tr \wedge tr' = tr' \wedge \\ ref = ref \wedge ref' = ref' \end{array} \right) \end{array} \right\} \\
= & [\text{property of equality}] \\
& \sqcap \left\{ A \mid \begin{array}{l} A[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t] \\ \sqsupseteq \\ X[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t] \end{array} \right\} \\
\Leftarrow & [\text{predicate calculus}] \\
& \sqcap \{A \mid A = X[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t]\} \\
\Leftarrow & [\text{predicate calculus}] \\
& \{A \mid A = X[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t]\}
\end{array}
$$

$\Leftarrow$ [predicate calculus]
X $\Box$

When applying the weakening function R to a predicate B and, then applying the strengthening function L to the result, this may actually strengthen the predicate B in the untimed model, as expressed in the following theorem

Theorem 4.2 *Given any Circus action program B, by applying the function R and then applying the function L to the result, yields a stronger program.*

$$L(R(B)) \sqsupseteq B \tag{4.2.7}$$

Proof:

$L(R(B))$ [4.2.5]
$= L(\sqcap \{A \mid L([\![A]\!]_{time}) \sqsupseteq [\![B]\!]\})$ $[L([\![A]\!]_{time}) \sqsupseteq [\![B]\!]]$
$\sqsupseteq [\![B]\!]$ $\Box$

Therefore, by applying the function R to a predicate B, and then applying the abstraction function L, we obtain a stronger predicate than the original predicate B. This last observation is important; as this implies that the functions L and R form a *Galois connection* [26]. A Galois connection is a relation between two models (T and S) given by a pair of functions (L, R) such that L maps predicates in T to S and R maps predicates in S to T. The pair (L, R) is a *Galois connection* if for all X and Y

$$L(X) \sqsupseteq Y \text{ iff } X \sqsupseteq R(Y) \tag{4.2.8}$$

R is called a weak inverse of L, and L is called a strong inverse of R. This permits us to explore some properties of the timed language. Let us consider the following definition.

Definition 4.1 *A predicate S is time insensitive if it satisfies the following equation*

$$R(L(S)) = S$$

The above definition states that if, by applying the abstraction function L to a predicate S, and then applying the weak inverse function R to the result, we obtain the same initial predicate S, then the time information in the original predicate S, is irrelevant. An example is the action *Stop*. It waits forever and permits time to pass; however, it is time insensitive in the sense that it does not impose any restriction on the time passage. Therefore removing the time information from *Stop* and then adding arbitrary time results in the same action *Stop*, as shown below.

$R(L([\![Stop]\!]_{time}))$
$=$ [4.2.5]

$$\sqcap \{[\![A]\!]_{time} \mid L([\![A]\!]_{time}) \sqsupseteq L([\![Stop]\!]_{time})\}$$

From the above we notice that the only possible action A is $Stop$ so by substituting A for $Stop$ in the above equation we obtain

$$\begin{array}{ll} & \sqcap \{[\![A]\!]_{time} \mid L([\![Stop]\!]_{time}) \sqsupseteq L([\![Stop]\!]_{time})\} \\ = & \qquad [\text{predicate calculus and set theory}] \\ & [\![Stop]\!]_{time} \end{array}$$

On the other hand, consider the $Wait\ \ 3$ action in the time model; we apply the function L and then apply back the function R as follows

$$\begin{array}{ll} & R(L(Wait\ \ 3)) \\ = & \qquad [4.2.5] \\ & \sqcap \{A \mid L(A) \sqsupseteq L(Wait\ \ 3)\} \\ = & \qquad [\text{Property 4.3 L3}] \\ & \sqcap \{A \mid L(A) \sqsupseteq (Stop \sqcap Skip)\} \\ = & \qquad [\text{From properties 4.3 and set theory}] \\ & \sqcap \{A \mid A = Skip \lor A = Stop \lor \exists\, n \bullet A = Wait\ \ \mathsf{n}\} \\ \neq & \\ & Wait\ \ 3 \end{array}$$

From the above inequality we see clearly, as expected, that the action $Wait\ \ 3$ is not time insensitive.

The relation between the models permits us to explore properties of timed programs that can be expressed in the untimed model. The parts of the program which are not time sensitive can be identified and explored in the untimed model. For the programs with time information, safety properties can still be validated in the untimed model. In the next section, we explore this topic in more detail with the aid of an example.

4.3 SAFETY PROPERTIES OF A ONE PLACE BUFFER

A one place buffer takes a value as input, stores it in a local internal variable, and then offers to communicate the same value on the output channel. A buffer has a main safety requirement: it cannot loose data, in other words, it should not allow the data to be overwritten by new input before an output is issued. A simple buffer can be defined in *Circus* Time Action in the following way

$$Buffer \ \widehat{=}\ in?x \rightarrow out!x \rightarrow Skip$$

To add some timing constraints we state that the input and output operations have a duration of 3 time units: we add a waiting state at the end of each operation. The communication does not consume time.

$$TBuffer \;\widehat{=}\; in?x \rightarrow Wait\ \ 3;\ out!x \rightarrow Wait\ \ 3;\ Skip$$

The safety property of the buffer can be specified by a function on traces. The function states that the projection of the trace over the event *out* should be shorter or equal to the projection of the same trace over the event *in*. It also states that the projection of the *front* of the trace over the event *in* is shorter or equal to the projection of the same trace on the event *out*.

$$\begin{aligned} S_1(trace) \;\widehat{=}\; & front(trace) \upharpoonright \{in\} \leq trace \upharpoonright \{out\} \wedge \\ & trace \upharpoonright \{out\} \leq trace \upharpoonright \{in\} \end{aligned}$$

We would like to check if our timed buffer meets the safety requirement. Because the specification uses the traces to state the property, we can use our mapping function L to obtain the time abstract version of the buffer, and then check if the abstract version of the timed buffer satisfies the specification. We define a relation sat_T between timed programs P and untimed specifications S.

$$P\ \ sat_T\ \ S \;\widehat{=}$$

$$L(P) \Rightarrow S$$

We need to prove that

$$L([\![TBuffer]\!]_{time}) \Rightarrow S_1$$

where

$L([\![TBuffer]\!]_{time})$
$=$ [definition of $TBuffer$]
$L([\![in?x \rightarrow Wait\ \ 3;\ out!x \rightarrow Wait\ \ 3;\ Skip]\!]_{time})$
$=$ [Property 4.3 L11]
$L([\![in?x \rightarrow Wait\ \ 3;\ out!x \rightarrow Wait\ \ 3]\!]_{time});\ L([\![Skip]\!]_{time})$
$=$ [Property 4.3 L11]
$L([\![in?x \rightarrow Wait\ \ 3]\!]_{time});\ L([\![out!x \rightarrow Wait\ \ 3]\!]_{time});\ L([\![Skip]\!]_{time})$
$=$ [Property 4.3 L6,L11]
$in?x \rightarrow L([\![Wait\ \ 3]\!]_{time});\ out!x \rightarrow L([\![Wait\ \ 3]\!]_{time});\ L([\![Skip]\!]_{time})$

$$
\begin{array}{ll}
= & \text{[Property 4.3 L5]} \\
[\![in?x \rightarrow (Stop \sqcap Skip);\ out!x \rightarrow (Stop \sqcap Skip)]\!];\ L([\![Skip]\!]_{time}) & \\
= & \text{[Property 4.3 L2]} \\
[\![in?x \rightarrow (Stop \sqcap Skip);\ out!x \rightarrow (Stop \sqcap Skip)]\!];\ Skip & \Box
\end{array}
$$

Therefore we need to show that

$$[\![in?x \rightarrow (Stop \sqcap Skip);\ out!x \rightarrow (Stop \sqcap Skip)]\!];\ Skip \Rightarrow S_1$$

From the semantic definition of communication and the definition of sequential composition

$$
\begin{array}{lll}
[\![out!x \rightarrow (Stop \sqcap Skip)]\!];\ Skip & \Rightarrow & (trace =<>) \vee \\
 & & (trace =< out >)
\end{array}
$$

In a similar manner

$$
\begin{array}{lll}
[\![in?x \rightarrow (Stop \sqcap Skip);\ out!x \rightarrow (Stop \sqcap Skip)]\!];\ Skip & \Rightarrow & (trace =<>) \vee \\
 & & (trace =< in >) \vee \\
 & & (trace =< in > \frown t' \wedge \\
 & & (t' =<> \vee\ t' =< out >))
\end{array}
$$

This can be simplified to

$$
\begin{array}{lll}
[\![in?x \rightarrow (Stop \sqcap Skip);\ out!x \rightarrow (Stop \sqcap Skip)]\!];\ Skip & \Rightarrow & (trace =<>) \vee \\
 & & (trace =< in >) \vee \\
 & & (trace =< in\ \ out >)
\end{array}
$$

Therefore

$$[\![TBuffer]\!]_{time}\ \ sat_T\ \ S_1 \Box$$

Notice that the abstraction function L, when applied to *Wait* d, substitutes the wait command with non-deterministic choice between *Skip* and *Stop*. This actually introduces a deadlock state into the program. Therefore liveness properties can not be explored with this abstraction function. A more suitable abstraction would be to substitute *Wait* d with a *Skip*.

4.4 NON-CONSERVATIVE MAPPING

A new mapping can be defined using L. It ensures that the only possible waiting state for a program is a waiting state that can wait forever. The definition of this function is as follows:

$$\hat{L}(\llbracket P \rrbracket_{time}) \;\widehat{=}\; L(\llbracket P \rrbracket_{time}) \wedge (wait' \Rightarrow EndlessObs) \tag{4.4.1}$$

where *EndlessObs* observations that wait for an arbitrary n time units.

$$EndlessObs \;\widehat{=}\; \forall\, n \bullet \exists\, tr_o \bullet tr_o = tr_t \frown \langle(\langle\rangle, ref)\rangle^n \wedge \llbracket P \rrbracket_{time}[tr_o/tr'_t] \tag{4.4.2}$$

The notation $\langle e \rangle^n$ stands for a sequence with n occurrences of e.

By applying the above function to *Circus* Time Action constructs, we obtain the following:

Property 4.4

L1. $\hat{L}(\llbracket x := e \rrbracket_{time}) = \llbracket x := e \rrbracket$

L2. $\hat{L}(\llbracket Skip \rrbracket_{time}) = \llbracket Skip \rrbracket$

L3. $\hat{L}(\llbracket Stop \rrbracket_{time}) = \llbracket Stop \rrbracket$

L4. $\hat{L}(\llbracket Chaos \rrbracket_{time}) = \llbracket Chaos \rrbracket$

L5. $\hat{L}(\llbracket comm \rrbracket_{time}) = \llbracket comm \rrbracket$

L6. $\hat{L}(\llbracket Wait\ \ \mathsf{d} \rrbracket_{time}) = \llbracket Skip \rrbracket$

Proof:

$$\begin{array}{ll}
& \hat{L}(\llbracket Wait\ \ \mathsf{d} \rrbracket_{time}) \\
= & \qquad [4.4.1] \\
& L(\llbracket Wait\ \ \mathsf{d} \rrbracket_{time}) \wedge \\
& \left(wait' \Rightarrow \begin{array}{l} \forall\, n \bullet \exists\, tr_o \bullet \\ tr_o = tr_t \frown \langle(\langle\rangle, ref)\rangle^n \wedge \llbracket Wait\ \ \mathsf{d} \rrbracket_{time}[tr_o/tr'_t] \end{array} \right) \\
= & \qquad [3.5.8 \text{ and predicate calculus}] \\
& L(\llbracket Wait\ \ \mathsf{d} \rrbracket_{time}) \wedge \\
& (wait' \Rightarrow false) \\
= & \qquad [\text{Predicate calculus and property 4.3 L3}] \\
& (\llbracket Skip \rrbracket \sqcap \llbracket Stop \rrbracket) \wedge \\
& (\neg wait') \\
= & \qquad [3.5.24 \text{ and predicate calculus}]
\end{array}$$

$$
\begin{array}{ll}
(\llbracket \mathit{Skip} \rrbracket \wedge (\neg \mathit{wait}')) \vee & \\
(\llbracket \mathit{Stop} \rrbracket \wedge (\neg \mathit{wait}')) & \\
= & \textit{[3.5.5 and predicate calculus]} \\
\llbracket \mathit{Skip} \rrbracket & \Box
\end{array}
$$

L7. $\hat{L}(\llbracket A \lhd b \rhd B \rrbracket_{time}) = \hat{L}(\llbracket A \rrbracket_{time}) \lhd b \rhd \hat{L}(\llbracket B \rrbracket_{time})$

L8. $\hat{L}(\llbracket p\& \ A \rrbracket_{time}) = p\& \ \hat{L}(\llbracket A \rrbracket_{time})$

L9. $\hat{L}(\llbracket A \sqcap B \rrbracket_{time}) = \hat{L}(\llbracket A \rrbracket_{time}) \sqcap \hat{L}(\llbracket B \rrbracket_{time})$

L10. $\hat{L}(\llbracket A \ \Box \ B \rrbracket_{time}) = \hat{L}(\llbracket A \rrbracket_{time}) \ \Box \ \hat{L}(\llbracket B \rrbracket_{time})$

L11. $\hat{L}(\llbracket A; \ B \rrbracket_{time}) = \hat{L}(\llbracket A \rrbracket_{time}); \ \hat{L}(\llbracket B \rrbracket_{time})$

L12. $\hat{L}(\llbracket A \backslash cs \rrbracket_{time}) = \hat{L}(\llbracket A \rrbracket_{time}) \backslash cs$

L13. $\hat{L}(\llbracket \mu \, X \bullet A(X) \rrbracket_{time}) = \mu \, X' \bullet \hat{L}(\llbracket A \rrbracket_{time})(X')$

L14. $\hat{L}(\llbracket A \stackrel{\mathsf{d}}{\rhd} B \rrbracket_{time}) = \hat{L}(\llbracket (A \ \Box \ (\mathit{Wait} \ \ \mathsf{d}; \ \mathit{int} \rightarrow B)) \backslash \{\mathit{int}\} \rrbracket_{time})$

L15. $\hat{L}(\llbracket A \ [\![\ s_A \mid \{\!| \ cs \ |\!\} \mid s_B \ ]\!] \ B \rrbracket_{time}) \Rightarrow$
$\hat{L}(\llbracket A \rrbracket_{time}) \ [\![\ s_A \mid \{\!| \ cs \ |\!\} \mid s_B \ ]\!] \ \hat{L}(\llbracket B \rrbracket_{time})$

The function L is conservative, it preserves the behaviour of the original program, the function $\hat{L}$ is not conservative as it does not distribute over parallel composition. To better understand what happens in the case of the parallel composition, consider the following example.

$$
\begin{array}{lcl}
A & \widehat{=} & (a \rightarrow \mathit{Skip}) \stackrel{2}{\rhd} \mathit{Skip} \\
B & \widehat{=} & \mathit{Wait} \ \ 3; \ (a \rightarrow \mathit{Skip})
\end{array}
$$

$$
C \ \widehat{=} \ A \ [\![\ \emptyset \mid \{\!| \ \{a\} \ |\!\} \mid \emptyset \ ]\!] \ B
$$

Notice that

$$
\hat{L}(C) = \mathit{false}
$$

That is because the action A will time out before the action B is ready to synchronize on event a. Whereas in the case of $\hat{L}(A) \ [\![\ \{a\} \ ]\!] \ \hat{L}(B)$ we have the following.

$$
\begin{array}{lcl}
\hat{L}(A) \ [\![\ \{\} \mid \{\!| \ \{a\} \ |\!\} \mid \{\} \ ]\!] \ \hat{L}(B) & = & (((a \rightarrow \mathit{Skip}) \ \Box \ (\mathit{Skip}; \ \mathit{int} \rightarrow \mathit{Skip})) \backslash \{\mathit{int}\}) \\
 & & [\![\{\} \mid \{\!| \ \{a\} \ |\!\} \mid \{\} \ ]\!] \\
 & & (\mathit{Skip}; \ (a \rightarrow \mathit{Skip}))
\end{array}
$$

We can observe that event a can occur immediately. We intend to explore this property in the further and study special cases in which the function $\hat{L}$ distributes over parallel composition.

4.5 EXAMPLE: LIVENESS PROPERTIES OF THE BUFFER

We can now study the liveness property of the buffer. A buffer should always be ready to input when it is empty, and it must be ready for output when it is not empty. We express this requirement with a function S_2. The function makes use of a trace and a refusal set and is defined as follows.

$$\begin{array}{rcl} S_2(trace, ref) & \widehat{=} & trace \upharpoonright \{in\} = trace \upharpoonright \{out\} \Rightarrow \{in.T\} \cap ref = \{\} \wedge \\ & & trace \upharpoonright \{out\} < trace \upharpoonright \{in\} \Rightarrow \{out.T\} \not\subseteq ref \end{array}$$

where T is the type of channels *in* and *out*. The set $\{in.T\}$ is the set of all possible communications over the channel *in*, while $out.T$ is the set of all outputs of type T on channel *out*.

We define a relation sat_{TF} between a timed program and an untimed specification on traces and failures. The relation is defined as follows.

$$P \;\; sat_{TF} \;\; S \mathrel{\widehat{=}}$$

$$\hat{L}(P) \Rightarrow S$$

We need to prove that

$$\hat{L}([\![TBuffer]\!]_{time}) \Rightarrow S_2$$

where

$$\begin{array}{ll} & \hat{L}([\![TBuffer]\!]_{time}) \\ = & \text{[definition of } TBuffer\text{]} \\ & \hat{L}([\![in?x \rightarrow Wait \;\; 3;\; out!x \rightarrow Wait \;\; 3;\; Skip]\!]_{time}) \\ = & \text{[Property 4.4 L11]} \\ & = \hat{L}([\![in?x \rightarrow Wait \;\; 3;\; out!x \rightarrow Wait \;\; 3]\!]_{time});\; \hat{L}([\![Skip]\!]_{time}) \\ = & \text{[Property 4.4 L11]} \\ & \hat{L}([\![in?x \rightarrow Wait \;\; 3]\!]_{time});\; \hat{L}([\![out!x \rightarrow Wait \;\; 3]\!]_{time});\; \hat{L}([\![Skip]\!]_{time}) \\ = & \text{[Property 4.4 L5,L11]} \\ & [\![in?x]\!] \rightarrow \hat{L}([\![Wait \;\; 3]\!]_{time});\; [\![out!x]\!] \rightarrow \hat{L}([\![Wait \;\; 3]\!]_{time});\; \hat{L}([\![Skip]\!]_{time}) \\ = & \text{[Property 4.4 L6 and L2]} \\ & [\![in?x \rightarrow Skip;\; out!x \rightarrow Skip]\!] \end{array}$$

□

From the semantic definition of communication and the definition of sequential composition

$$\begin{aligned}[\![out!x \rightarrow Skip]\!] \;\Rightarrow\; & (trace =<>\wedge \{out\} \notin ref) \vee \\ & (trace =< out > \frown t' \wedge ref'\end{aligned}$$

In a similar manner

$$\begin{aligned}[\![in?x \rightarrow Skip;\ out!x \rightarrow Skip]\!] \;\Rightarrow\; & (trace =<>\wedge\ in \notin ref) \vee \\ & (trace =< in >\wedge\ ref') \vee \\ & (trace =< in > \frown t' \wedge ref'') \vee \\ & (t' =<>\wedge\ out \notin ref') \vee \\ & (t' =< out > \frown t'' \wedge ref'')\end{aligned}$$

This can be simplified to

$$\begin{aligned}[\![in?x \rightarrow Skip;\ out!x \rightarrow Skip]\!] \;\Rightarrow\; & (trace =<>\wedge\ in \notin ref) \vee \\ & (trace =< in >\wedge\ out \notin ref) \vee \\ & (trace =< in\ \ out > \frown t'' \wedge ref')\end{aligned}$$

Therefore

$$\hat{L}([\![TBuffer]\!]_{time})\ \ sat_{TF}\ \ S_2$$

4.6 CONCLUDING REMARKS

The two mappings functions, introduced earlier in this chapter, represent the two extreme observations of the timed model. The first replaces a quantified wait by the option to either wait forever or terminate immediately. The second mapping replaces the quantified wait with an immediate termination; it does not allow the program to wait. The first mapping is suitable for proving properties of traces, while the second mapping is more adequate for proving liveness properties, as it was shown in the buffer example. Some properties, however, can only be proven in the time model for they are related directly to the timing requirements of the system. Deadlock freedom is another property that can not be proven by any of the above mappings because synchronization is time dependent.

In the next chapter we propose the use of the mapping functions in a framework that permits the validation of the timed requirements of a system. The framework proposes a normal form that eliminates the time operators and introduces special timer events

that can be later related to the time requirements. The mapping functions, presented in this chapter, are used to remove the time information and obtain a new untimed action, but preserving the time behaviour effect that can be observed with the aid of the new introduced timer events.

CHAPTER 5

THE VALIDATION FRAMEWORK

In this chapter, we present a framework for specification and validation of real-time programs using the timed and untimed models of *Circus* presented in this thesis. The framework has been originally presented in [51]. The main idea is to use the timed model for specification, and the untimed model to validate the system requirements. A normal form for a timed program is used, as an intermediate step, to obtain an untimed program that can be verified to meet the time requirements. In this way, we reason about time properties without making explicit use of time. The untimed program, however, includes special (*timer*) events.

In the next section we give an overview of the framework structure and the steps to use it. In Section 5.2, we introduce the normal form reduction: a definition for timers and timer events is introduced, and new parallel composition and choice operators are defined to give a special treatment for the timer events. In Section 5.3, we prove the correctness of the normal form reduction. Section 5.4 considers refinement of programs in normal form. In Section 5.5, we link the refinement relation normal form to the refinement relation in the untimed model. In Section 5.6, we explore the approach by applying it to a case study.

5.1 AN OVERVIEW OF THE FRAMEWORK

In this section, we present the general structure of the framework. Figure 5.1 illustrates the steps for using the framework, which can be summarized as follows.

1) We start with a program specification P and the time requirement specification R in the time model, using the timed version of the language. The designer gives a complete description of the system and uses the same notation to describe the desired properties.

2) With the help of the function Φ, defined in the next section, we obtain a normal form program that has the same semantics as the original program (Theorem 5.1). The normal form program specification is composed of two parts: a set of interleaving timers and a program $P' = \Phi(P)$ with no time operators, but, containing internal timer events. The structure of the normal form of the requirement R is the same. Furthermore, the validation step requires that the interleaving timers of the normal forms of P and R must be equivalent.

3) The final transformation step is to apply an abstraction function L to obtain an untimed program. The abstraction function is applied to the program $P' = \Phi(P)$ and to the requirements $R' = \Phi(R)$.

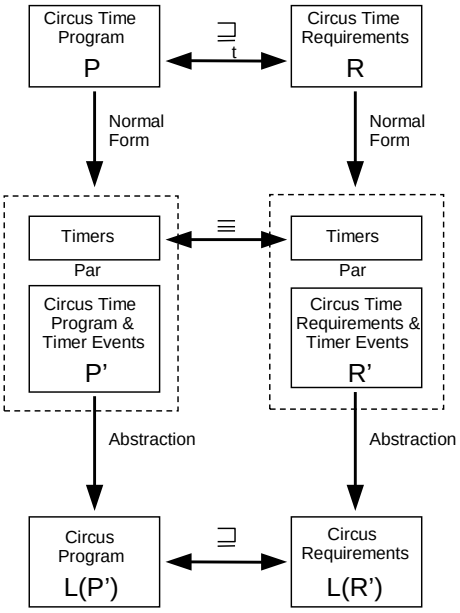

Figure 5.1. A heterogeneous framework for analysis of timed programs in Circus

4) Finally the last step in the validation process is to show that the untimed program $L(\Phi(P))$ satisfies the untimed requirements $L(\Phi(R))$.

In the next section, we present the notation used to express the normal form program.

5.2 CIRCUS TIME ACTION WITH TIMER EVENTS

As explained in the previous section, the first step of our validation framework is a reduction to a normal form. This reduction splits the original program into a set of timers and another program that interacts with the timers through events that represent time actions. Semantically, these events have a specialised behaviour when they appear in an external choice or in a parallel composition. Therefore, strictly, the language used to express the normal form is an extension of *Circus* Time Action with the timer events.

5.2.1 The Normal Form

Usually, timed programs are implemented with timers: the system clock or a dedicated timer. Following this idea, we give a normal form for time operators. They are implemented as a parallel composition of a timer and an untimed program that synchronizes on the timer events.

The normal form of a *Circus* Time Action A is represented as

$$\Phi(A) \ \mathbf{par} \ \mathit{Timers}(k, n)$$

$\mathit{Timers}(k, n)$ stands for a set of interleaving timers with index ranging from k to n. Broadly, Φ maps the original action into one that uses timer events to synchronize

with timers, but $\Phi(A)$ does not include time operators; these are confined to the other component of the normal form: $Timers(k, n)$.

The **par** operator is defined in terms of parallel composition, but deals explicitly with termination of the synchronisation between $\Phi(A)$ and $Timers(k, n)$. Each of the elements of the normal form is detailed in a subsequent section.

5.2.2 The Normal Form Timer

The following is the specification of the timer.

$$Timer(i, d) \mathrel{\widehat=} \mu\, X \bullet \left(\begin{array}{l} setup.i.d \rightarrow \left(\begin{array}{l} (halt.i \rightarrow X) \\ \Box\ Wait\ \ \mathsf{d};\ \left(\begin{array}{l} (out.i.d \rightarrow X) \\ \Box \\ (terminate.i \rightarrow Skip) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \tag{5.2.0}$$

In the above definition, we use parameterized action notation. This type of action is not part of our language, but just a syntactic abbreviation. Based on the above definition we later use, for instance, $Timer(4, t)$ to stand for $Timer(i, d)[4/i, t/d]$. The timer is initiated with the event $setup.i.d$, which takes the timer instance identifier i and the delay d as input. The behaviour of the timer is then to offer the event $halt.i$ while it waits for d time units; at the end, the event $out.i.d$ is offered. When either $halt.i$ or $out.i.d$ takes place, the timer is reset. The timer always offers the event $terminate.i$ that terminates the time suspending its execution.

$Timers(k, n)$ is the interleaving of the n timers needed by the action (one for each time operator).

$$Timers(k, n) \mathrel{\widehat=} \mathop{|||}_{i=k}^{n} (Timer(i, delay(i))) \tag{5.2.0}$$

The function *delay* maps the timer index to the timer delay.

5.2.3 The Normal Form Parallel Composition

The normal form makes use of a special parallel composition operator **par** , that is defined as follows.

$$A\ \mathbf{par}\ Timers(k, n) \mathrel{\widehat=} (A;\ Terminate(k, n) \,[\![\, s_A \mid \{\!|\ TSet\ |\!\} \mid \phi \,]\!]\, Timers(k, n)) \setminus TSet \tag{5.2.0}$$

$Terminate(k, n)$ assures that if the action A terminates normally then all the timers involved in the parallel composition are terminated. This is necessary to terminate the parallel composition otherwise; the timers do not allow the parallel composition to terminate. $Terminate(k, n)$ is defined as follows.

$$Terminate(k, n) \mathrel{\widehat=} \mathop{|||}_{i=k}^{n} (terminate.i \rightarrow Skip) \tag{5.2.0}$$

We use the CSP interleave operator $|||$ as an abbreviation for $[\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}]\!]$. *TSet* is the set of timer events:

$$TSet = timerEvents \cup \bigcup_{i=k}^{n} \{halt.i, terminate.i\}$$

The operator **par** stands for the parallel composition of two actions that synchronize on the timer events. The timer events are hidden and, therefore, treated as internal events.
We also have the following lemma.

Lemma 5.1 *Given any number of interleaving timers* $Timers(k, n)$ *and their associated termination processes* $Terminate(k, n)$*, then the parallel composition with TSet as the synchronization set and any arbitrary* s_A *yields Skip.*

$$(Terminate(k, n) \,[\![\, s_A \mid \{\!|\ TSet\ |\!\} \mid \phi \,]\!]\, Timers(k, n)) \setminus TSet \equiv Skip$$

Proof:

$$
\begin{array}{ll}
& (Terminate(k, n) \,[\![\, s_A \mid \{\!|\ TSet\ |\!\} \mid \phi \,]\!]\, Timers(k, n)) \setminus TSet \\
= & \qquad [5.2.3] \\
& (\mathop{|||}_{i=1}^{n} (terminate.i \rightarrow Skip) \,[\![\, s_A \mid \{\!|\ TSet\ |\!\} \mid \{\} \,]\!]\, Timers(k, n)) \setminus TSet \\
= & \qquad [5.2.2] \\
& \left(\begin{array}{l} \mathop{|||}_{i=k}^{n} (terminate.i \rightarrow Skip) \\ \quad [\![s_A \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ \mathop{|||}_{i=k}^{n} (Timer(i, delay(i))) \end{array} \right) \setminus TSet \\
= & \qquad [5.2.2] \\
& \left(\begin{array}{c} \mathop{|||}_{i=k}^{n} (terminate.i \rightarrow Skip) \\ [\![s_A \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ \mathop{|||}_{i=k}^{n} \mu\, X \bullet \left(\begin{array}{l} setup.i.delay(i) \rightarrow \\ \quad \left(\begin{array}{l} (halt.i \rightarrow X) \\ \quad \Box \left(\begin{array}{l} Wait\ \ \mathsf{delay(i)}; \\ \quad \left(\begin{array}{l} (out.i \rightarrow X) \\ \quad \Box \\ \quad (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \\ \quad \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \setminus TSet \\
= & \qquad [Step\ law\ and\ 5.2.2]
\end{array}
$$

$$
\left(
\begin{array}{l}
\quad \Big|\!\Big|\!\Big|_{i=k}^{n} \ (terminate.i \rightarrow Skip) \\
\quad [\![s_A \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\
\Big|\!\Big|\!\Big|_{i=k}^{n} \left(
\begin{array}{l}
setup.i.delay(i) \rightarrow \\
\left(
\begin{array}{l}
(halt.i \rightarrow (Timer(i, delay(i)))) \\
\Box \left(
\begin{array}{l}
Wait\ \ \mathsf{delay(i)}; \\
\left(
\begin{array}{l}
(out.i \rightarrow (Timer(i, delay(i)))) \\
\Box \\
(terminate.i \rightarrow Skip)
\end{array}
\right)
\end{array}
\right) \\
\Box\ (terminate.i \rightarrow Skip)
\end{array}
\right) \\
\Box\ (terminate.i \rightarrow Skip)
\end{array}
\right)
\end{array}
\right) \setminus TSet
$$

$=$ *[terminate.i ∈ TSet and CSP parallel composition step law]*

$$
\left(\Big|\!\Big|\!\Big|_{i=k}^{n} \ (terminate.i \rightarrow Skip) \right) \setminus TSet
$$

$=$ *[terminate.i ∈ TSet]*

Skip □

5.2.4 The Normal Form Action

The function Φ takes a timed action and returns an action in a normal form that uses the timer events, instead of timed constraints. It is an identity for the basic actions (except for *Wait*), and distributes through the binary operators, except for timeout, external choice and parallel composition.

$$
\begin{aligned}
\Phi(Skip) &= Skip && (5.2.1)\\
\Phi(Stop) &= Stop && (5.2.2)\\
\Phi(Chaos) &= Chaos && (5.2.3)\\
\Phi(x := e) &= x := e && (5.2.4)\\
\Phi(b \ \&\ A) &= b \ \&\ \Phi(A) && (5.2.5)\\
\Phi(A \setminus cs) &= \Phi(A) \setminus cs && (5.2.6)\\
\Phi(c \rightarrow A) &= c \rightarrow \Phi(A) && (5.2.7)\\
\Phi(A \lhd b \rhd B) &= \Phi(A) \lhd b \rhd \Phi(B) && (5.2.8)\\
\Phi(A;\ B) &= \Phi(A);\ \Phi(B) && (5.2.9)\\
\Phi(A \sqcap B) &= \Phi(A) \sqcap \Phi(B) && (5.2.10)\\
\Phi(\mu\, X \bullet A(X))) &= (\mu\, X \bullet \Phi(A(\Phi(X)))) && (5.2.11)\\
\Phi(X) &= X && (5.2.12)
\end{aligned}
$$

The relevant cases are *Wait*, timeout, external choice and parallel composition. For *Wait* d, Φ replaces the wait action by an action that initializes a timer with $setup.i.d$ and then waits for the occurrence of $out.i.d$.

$$
\Phi(\ Wait\ \ \mathsf{d}) = (setup.i.d \rightarrow out.i.d \rightarrow Skip) \qquad (5.2.12)
$$

For the timeout operator, we consider a particular case: $(\mathop{\Box}_{j=1}^{n} c_j \rightarrow A_j) \stackrel{\mathsf{d}}{\triangleright} B$, where $(\mathop{\Box}_{j=1}^{n} c_j \rightarrow A_j)$ is an abbreviation for the term

$$\begin{array}{ll} & c_1 \rightarrow A_1 \\ \Box & \ldots \\ \Box & c_n \rightarrow A_n \end{array}$$

and for all j, c_j is not a timer event. The reason for using a special case is that the timeout has to stop the timer (using the event $halt.i$) after the occurrence of the first event. This does not lead to loss of generality because, using the algebraic laws of *Circus* Time Action presented in Chapter 3, we can transform any action into the required form.

$$\Phi((\mathop{\Box}_{j=1}^{n} c_j \rightarrow A_j) \stackrel{\mathsf{d}}{\triangleright} B) = setup.i.d \rightarrow \left(\begin{array}{c} (\mathop{\Box}_{j=1}^{n} c_j \rightarrow halt.i \rightarrow \Phi(A_j)) \\ \Box \\ (out.i.d \rightarrow \Phi(B)) \end{array} \right) \tag{5.2.12}$$

For external choice, Φ introduces a new external choice operator $\boxtimes$. The new operator gives the semantics of time events in a choice.

$$\Phi(A \Box B) = \Phi(A) \boxtimes \Phi(B) \tag{5.2.12}$$

The new choice operator $\boxtimes$ is used to give priority to the timer events *setup* and *out* over other program events. To understand why we need a new choice operator and why the timer events need to be treated in a special way consider the following example:

$$(Wait \ \ 5 \Box (a \rightarrow Skip))$$

In the timed semantics the above action can engage in a comunication over channel a during the first 5 time units. Afterwards, it should not offer event a anymore; it should terminate immediately. That is, during the first 5 time units, the choice is only determined by the occurrence of the event a. The same semantics is expected in the normal form.

Now recall that the normal form for the $Wait \ \ 5$ is $setup.i.5 \rightarrow out.i.5 \rightarrow Skip$. Therefore, by substituting in the above equation we obtain the following

$$\begin{array}{ll} & \Phi(Wait \ \ 5 \Box (a \rightarrow Skip)) \\ = & \qquad [\text{Assumption that } \Phi \text{ distributes over } \Box] \\ & \Phi(Wait \ \ 5) \Box \Phi(a \rightarrow Skip) \\ = & \qquad [5.2.4 \text{ and } 5.2.7] \\ & ((setup.i.5 \rightarrow out.i.5 \rightarrow Skip) \Box (a \rightarrow Skip)) \end{array}$$

The event *setup* is used only to start the timer and must not be regarded as part of the choice, which cannot be resolved between communicating on a or *setup*. The

desired behaviour is that *setup* occurs first and then the choice is made between doing a communication over a or out. The desired behaviour is as follows:

$$setup.i.5 \rightarrow ((out.i.5 \rightarrow Skip) \Box (a \rightarrow Skip))$$

We define the new choice operator $\boxtimes$ to give the desired semantics.

In CSP, parallel composition can be modelled as several external choice operations [49]. We adopt an approach in which we give the semantics of the choice operator with the aid of the UTP parallel merge operator. The parallel merge as shown in Chapter 3 is a general merge operator that can be specifically tailored with the definition of a new merge function. To show our approach we give the semantics of the UTP choice operator shown in Equation 3.5.25. The new definition for the external choice is as follows:

$$A \Box B \mathrel{\widehat{=}} CSP2 \left(\begin{array}{l} CSP1 \left(\begin{array}{c} (A \|_{CM} B) \vee \\ \left(\begin{array}{l} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array} \right) \end{array} \right) \end{array} \right) \tag{5.2.12}$$

Where CM is a choice parallel merge function defined as follows

$$CM \mathrel{\widehat{=}} \begin{array}{l} \left(\begin{array}{l} (0.ok, 0.wait, 0.tr - tr, 0.ref, 0.state), \\ (1.ok, 1.wait, 1.tr - tr, 1.ref, 1.state) \end{array} \right) \\ ChSynch \\ (ok', wait', tr' - tr, ref', state') \end{array} \tag{5.2.12}$$

The relation $ChSynch$ maps the indexed variables of the involved actions A and B to the dashed observation variables. The relation is given by the following predicates

$$\begin{array}{l} \left(\begin{array}{l} (0.ok, 0.wait, 0.tr, 0.ref, 0.state), \\ (1.ok, 1.wait, 1.tr, 1.ref, 1.state) \end{array} \right) \\ ChSynch \\ (ok', wait', tr', ref', state') \end{array} \Leftrightarrow \begin{array}{c} Diverge \vee \\ WaitokNoEvent \vee \\ DoAnyEvent \vee \\ Terminate \end{array} \tag{5.2.12}$$

The predicate $Diverge$ captures the diverging behaviour of the choice operator. The choice operator can diverges if one of the actions involved in the choice diverges, and that action is selected.

$$Diverge \mathrel{\widehat{=}} ok \wedge \left(\begin{array}{l} \neg ok' \wedge \\ \left(\begin{array}{c} \left(\begin{array}{c} 0.ok = ok' \wedge 0.wait = wait' \wedge 0.tr = tr' \wedge \\ 0.ref = ref' \wedge 0.state = state' \end{array} \right) \vee \\ \left(\begin{array}{c} 1.ok = ok' \wedge 1.wait = wait' \wedge 1.tr = tr' \wedge \\ 1.ref = ref' \wedge 1.state = state' \end{array} \right) \end{array} \right) \end{array} \right) \tag{5.2.12}$$

Next we consider the waiting state of the external choice. In an external choice, a waiting state is only observed if both actions agree on waiting and no external events are observed.

$$WaitokNoEvent \mathrel{\widehat{=}} \neg wait \wedge \left(\begin{array}{l} (ok' \wedge 0.ok = ok' \wedge 1.ok = ok') \wedge \\ (wait' \wedge 0.wait = wait' \wedge 1.wait = wait') \wedge \\ (tr' = \langle\rangle \wedge 0.tr = tr' \wedge 1.tr = tr') \wedge \\ (0.ref = ref' \wedge 1.ref = ref') \wedge \\ (0.state = state' \wedge 1.state = state') \end{array} \right) \tag{5.2.12}$$

Notice that the above predicate captures only the non diverging behaviour, for the diverging behaviour is captured by the predicate *Diverge*. Next we consider the situation is which an event is observed on the output trace indicating that a communication toke place. In such case the action that can observe the same event on the head of its traces is chosen. The predicate is expressed as follows:

$$DoAnyEvent \mathrel{\widehat=} ok \wedge \left(\begin{array}{l} (tr' \neq \langle\rangle) \wedge \\ \left(\begin{array}{c} 0.ok = ok' \wedge 0.wait = wait' \wedge 0.tr = tr' \wedge \\ 0.ref = ref' \wedge 0.state = state' \end{array} \right) \vee \\ \left(\begin{array}{c} 1.ok = ok' \wedge 1.wait = wait' \wedge 1.tr = tr' \wedge \\ 1.ref = ref' \wedge 1.state = state' \end{array} \right) \end{array} \right) \tag{5.2.12}$$

Finally if any of the two actions terminates with no communication observed on the outputs, then the terminating action is chosen. This is expressed in the following predicate:

$$Terminate \mathrel{\widehat=} ok \wedge \left(\begin{array}{l} (\neg wait') \wedge \\ \left(\begin{array}{c} 0.ok = ok' \wedge 0.wait = wait' \wedge 0.tr = tr' \wedge \\ 0.ref = ref' \wedge 0.state = state' \end{array} \right) \vee \\ \left(\begin{array}{c} 1.ok = ok' \wedge 1.wait = wait' \wedge 1.tr = tr' \wedge \\ 1.ref = ref' \wedge 1.state = state' \end{array} \right) \end{array} \right) \tag{5.2.12}$$

In appendix F we present a complete proof of the equivalence of the previous definition of $\Box$ with the original *Circus* definition.

Based on the same principle we give the definition of the special choice operator $\boxtimes$.

$$A \boxtimes B \mathrel{\widehat=} CSP2_t \left(\begin{array}{c} CSP1_t \left(\begin{array}{c} A \|_{TCM} B \vee \\ \left(\begin{array}{l} ok' \wedge wait \wedge \\ (tr'_t = tr_t) \wedge \\ (wait' = wait) \wedge \\ (state' = state) \end{array} \right) \end{array} \right) \end{array} \right) \tag{5.2.12}$$

Where *TCM* is a new parallel merge function that gives the semantics of the new choice operator. It is defined as follows:

$$\begin{array}{rl} TCM \mathrel{\widehat=} & \left(\begin{array}{l} (0.ok, 0.wait, dif(0.tr_t, tr_t), 0.state), \\ (1.ok, 1.wait, dif(1.tr_t, tr_t), 1.state) \end{array} \right) \\ & TChSynch \\ & (ok', wait', dif(tr'_t, tr_t), state') \end{array} \tag{5.2.12}$$

The relation *TChSynch* has the same function as the *ChSynch* relation (Equation 5.2.4) except that, when mapping the indexed variables of the involved actions A and B to the dashed observation variables the new relation, gives the special behaviour needed by the timer events. The relation is given by the following predicate:

$$\begin{array}{l} \left(\begin{array}{l} (0.ok, 0.wait, 0.tr_t, 0.state), \\ (1.ok, 1.wait, 1.tr_t, 1.state) \end{array} \right) \\ TChSynch \\ (ok', wait', tr'_t, state') \end{array} \Leftrightarrow \left(\begin{array}{c} Diverge_t \vee \\ WaitokNoEvent_t \vee \\ DoAnyEvent_t \vee \\ Terminate_t \end{array} \right) \tag{5.2.12}$$

Similar to *Diverge*, the predicate $Diverge_t$ is used to capture the diverging behaviour of the timed choice.

$$Diverge_t \mathrel{\widehat=} ok \land \left(\begin{array}{l} \neg ok' \land \\ \left(\begin{array}{c} \left(\begin{array}{c} 0.ok = ok' \land 0.wait = wait' \land \\ 0.tr_t = tr'_t \land 0.state = state' \end{array} \right) \lor \\ \left(\begin{array}{c} 1.ok = ok' \land 1.wait = wait' \land \\ 1.tr_t = tr'_t \land 1.state = state' \end{array} \right) \end{array} \right) \end{array} \right) \tag{5.2.12}$$

If any of the two actions terminates with no communication observed on the outputs, then the terminating action is chosen. This is expressed in the following predicate:

$$Terminate_t \mathrel{\widehat=} ok \land \left(\begin{array}{l} (\neg wait') \land Flat(tr'_t) = \langle\rangle \land \\ \left(\begin{array}{c} 0.ok = ok' \land 0.wait = wait' \land \\ 0.tr_t = tr'_t \land 0.state = state' \end{array} \right) \lor \\ \left(\begin{array}{c} 1.ok = ok' \land 1.wait = wait' \land \\ 1.tr_t = tr'_t \land 1.state = state' \end{array} \right) \end{array} \right) \tag{5.2.12}$$

Next we consider the waiting state of the external choice. In an external choice, a waiting state is only observed if both actions agree on waiting and no external events are observed.

$$WaitokNoEvent_t \mathrel{\widehat=} \left(\begin{array}{l} (ok' \land 0.ok = ok' \land 1.ok = ok') \land \\ (wait' \land 0.wait = wait' \land 1.wait = wait') \land \\ (Flat(tr'_t) = \langle\rangle \land 0.tr_t = tr'_t \land 1.tr_t = tr'_t) \land \\ (0.state = state' \land 1.state = state') \end{array} \right) \tag{5.2.12}$$

The predicate $DoAnyEvent_t$ is similar to the *DoAnyEvent* predicate used in the external choice, except that now it needs to consider as a special case the timer events.

$$DoAnyEvent_t \mathrel{\widehat=} \left(\begin{array}{c} ok' \land Flat(tr'_t) \neq \langle\rangle \land \\ \left(\begin{array}{c} DoTimePassing \lor \\ DoSetup \lor \\ DoAnyOut \lor \\ DoEventWithOut \lor \\ DoOrderedOut \lor \\ DoEvent \end{array} \right) \end{array} \right) \tag{5.2.12}$$

In the above definition, each of the predicates captures a particular situation. The predicate *DoTimePassing* captures the synchronization of the timed actions and registers the time passing. The time passing is registered if both the involved actions register empty traces at the head of the traces.

$$DoTimePassing \mathrel{\widehat=} \begin{array}{l} fst(head(tr_t)) = \langle\rangle \land \\ fst(head(0.tr_t)) = \langle\rangle \land \\ fst(head(1.tr_t)) = \langle\rangle \land \\ \left(\begin{array}{l} (0.ok, 0.wait, tail(0.tr_t), 0.state), \\ (1.ok, 1.wait, tail(1.tr_t), 1.state) \\ TChSynch \\ (ok', wait', tail(tr'_t), state') \end{array} \right) \end{array} \tag{5.2.12}$$

The predicates *DoSetup* is used to give the behaviour of the program when a *setup* event is observed at the head of the resulting trace. In such a situation the choice is not solved and the program should continue to synchronize the remaining traces to decide and make the choice of the resulting program.

$$DoSetup \mathrel{\widehat=} \left(\begin{array}{l} \exists\, i, d \bullet (head(fst(head(tr'_t))) = setup.i.d) \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} (head(fst(head(0.tr_t))) = setup.i.d) \wedge \\ StepFirst \end{array} \right) \vee \\ \left(\begin{array}{l} (head(fst(head(1.tr_t))) = setup.i.d) \wedge \\ StepSecond \end{array} \right) \end{array} \right) \end{array} \right) \tag{5.2.12}$$

The predicates *StepFirst* and *StepSecond* are defined as follows.

$$StepFirst \mathrel{\widehat=} \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} 0.ok, 0.wait, \\ \langle (tail(fst(head(0.tr_t))), snd(head(0.tr_t)))\rangle \frown tail(0.tr_t), \\ 0.state \end{array} \right), \\ (1.ok, 1.wait, 1.tr_t, 1.state) \end{array} \right) \\ TChSynch \\ \left(\begin{array}{l} ok', wait', \\ \langle (tail(fst(head(tr'_t))), snd(head(tr'_t)))\rangle \frown tail(tr'_t), \\ state' \end{array} \right) \end{array} \right) \tag{5.2.12}$$

and

$$StepSecond \mathrel{\widehat=} \left(\begin{array}{l} \left(\begin{array}{l} (0.ok, 0.wait, 0.tr_t, 0.state), \\ \left(\begin{array}{l} 1.ok, 1.wait, \\ \langle (tail(fst(head(1.tr_t))), snd(head(1.tr_t)))\rangle \frown tail(1.tr_t), \\ 1.state \end{array} \right) \end{array} \right) \\ TChSynch \\ \left(\begin{array}{l} ok', wait', \\ \langle (tail(fst(head(tr'_t))), snd(head(tr'_t)))\rangle \frown tail(tr'_t), \\ state' \end{array} \right) \end{array} \right) \tag{5.2.12}$$

The above definition simply states that whenever we observe the *setup* event at the top of the resulting trace, then the choice is not solved and the remaining traces are synchronized. The synchronization is made with the remaining traces of both actions after removing the *setup* event from the head of the trace. Notice as well that, if both actions can produce *setup* at the top of the trace, then the choice of which should appear first is nondeterministic.

The predicate *DoAnyOut* captures the behaviour of the choice when one of the two

actions need to perform the event *out*, while the other is not ready to do such an event.

$$DoAnyOut \mathrel{\widehat=} \left(\begin{array}{l} \exists\, i, d \bullet (head(fst(head(tr'_t))) = out.i.d) \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} (head(fst(head(0.tr_t))) = out.i.d) \wedge \\ \forall\, j, n \bullet i \neq j \wedge head(fst(head(1.tr_t))) \neq out.j.n \wedge \\ StepFirst \end{array} \right) \vee \\ \left(\begin{array}{l} head(fst(head(1.tr_t))) = out.i.d) \wedge \\ \forall\, j, n \bullet i \neq j \wedge head(fst(head(0.tr_t))) \neq out.j.n \wedge \\ StepSecond \end{array} \right) \end{array} \right) \tag{5.2.12}$$

The above predicate shows the case that only one of the actions in the choice is ready to perform the $out.i.d$ event. The next predicate shows the condition in which both actions can perform the event *out*; in this case the choice is made based on the delay of the timers, and the *out* associated to the timer with the smallest delay is chosen.

$$DoOrderedOut \mathrel{\widehat=} \left(\begin{array}{l} \exists\, i, d \bullet (head(fst(head(tr'_t))) = out.i.d) \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} (head(fst(head(0.tr_t))) = out.i.d) \wedge \\ \exists\, j, n \bullet (head(fst(head(1.tr_t))) = out.j.n) \wedge i \neq j \wedge \\ StepFirst \wedge (d \leq n) \end{array} \right) \vee \\ \left(\begin{array}{l} (head(fst(head(1.tr_t))) = out.i.d) \wedge \\ \exists\, j, n \bullet (head(fst(head(0.tr_t))) = out.j.n) \wedge i \neq j \wedge \\ StepSecond \wedge (d \leq n) \end{array} \right) \end{array} \right) \tag{5.2.12}$$

The above predicate orders the *out* events when they occur on the head of the traces of both actions involved in the choice. The order of the *out* events is determined by the delay of the timers associated with each event; the event associated with the shortest delay occurs first. Case both timers have the same delay, then the order of the events is nondeterministic. Observe as well that the choice is not solved yet and the synchronization of the remaining traces is needed to resolve the choice.

The predicate *DoEvent* is intended to capture the events that are not timer events. It is very similar to the *DoAnyEvent* used in the definition of the choice operator above. The intention of this predicate is to make the choice of which action to take based on the initial events, as long as they are not timer events.

$$DoEvent \mathrel{\widehat=} \left(\begin{array}{l} \exists\, a \bullet (head(fst(head(tr'_t))) = a) \wedge a \notin timerEvents \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} (head(fst(head(0.tr_t))) = a) \wedge \\ (head(fst(head(1.tr_t))) \notin timerEvents) \wedge \\ \left(\begin{array}{l} ok' = 0.ok \wedge wait' = 0.wait \wedge \\ tr'_t = 0.tr_t \wedge state' = 0.state \end{array} \right) \end{array} \right) \vee \\ \left(\begin{array}{l} (head(fst(head(1.tr_t))) = a) \wedge \\ (head(fst(head(0.tr_t))) \notin timerEvents) \wedge \\ \left(\begin{array}{l} ok' = 1.ok \wedge wait' = 1.wait \wedge \\ tr'_t = 1.tr_t \wedge state' = 1.state \end{array} \right) \end{array} \right) \end{array} \right) \tag{5.2.12}$$

The set *timerEvents* contains all the possible *setup* and *out* events:

$$timerEvents \mathrel{\widehat=} \bigcup_{i=1}^{n} \{setup.i.d, out.i.d \mid d = delay(i)\}$$

The function *delay* is an abstract mapping from the timer index i to the timer delay period d. Finally, the predicate *DoEventWithOut* treats the situation in which an event *out* is possible in the presence of another ordinary event. The behaviour of the choice in such a situation is to offer both of them: if the event *out* is selected then the choice is not solved and the other event can occur. Case *out* is not selected and the other event is selected instead, then the event *halt* related to the same timer as the *out* event should be observed in the trace immediately after the event. The predicate is given as follows

$$DoEventWithOut \mathrel{\widehat=} \left(\begin{array}{l} \exists\, a \bullet (head(fst(head(tr_t'))) = a) \land a \notin timerEvents \land \\ \left(\begin{array}{l} \left(\begin{array}{l} (head(fst(head(0.tr_t))) = a) \land \\ \exists\, i, d \bullet head(fst(head(1.tr_t))) = out.i.d \land i \neq j \land \\ \left(\begin{array}{l} ok' = 0.ok \land wait' = 0.wait \land \\ state' = 0.state \land \\ tr_t' = addHead(0.tr_t) \frown tail(0.tr_t) \end{array} \right) \end{array} \right) \lor \\ \left(\begin{array}{l} (head(fst(head(1.tr_t))) = a) \land \\ \exists\, i, d \bullet head(fst(head(0.tr_t))) = out.i.d \land i \neq j \land \\ \left(\begin{array}{l} ok' = 1.ok \land wait' = 1.wait \land \\ state' = 1.state \land \\ tr_t' = addHead(1.tr_t) \frown tail(1.tr_t) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \tag{5.2.12}$$

where

$$addHead(t) \mathrel{\widehat=} \langle (\langle a, halt.i \rangle \frown tail(fst(head(t))), snd(head(t))) \rangle \tag{5.2.12}$$

The following algebraic laws hold for the new operator. In the laws below we take c_i and d_i to be ordinary events (they cannot be timer events) and A and B to be actions defined in the normal form. The first law shows that $\boxtimes$ is *idempotent*.

Law 5.2.1

$$A \boxtimes A = A$$

The following law states $\boxtimes$ *is commutative.*

Law 5.2.2

$$A \boxtimes B = B \boxtimes A$$

The following is the associativity law for $\boxtimes$.

Law 5.2.3

$$A \boxtimes (B \boxtimes C) = (A \boxtimes B) \boxtimes C$$

The next law states that the *setup* event is not determinant in the choice and it occurs without affecting the choice.

Law 5.2.4

$$(setup.i.d \rightarrow A) \boxtimes B = setup.i.d \rightarrow (A \boxtimes B)$$

As in the case of the *setup* event the *out* event is not determinant in the choice, but different from *setup*, the *out* event offers no time event c_i as a choice using the standard external choice operator.

Law 5.2.5

$$(out.j \rightarrow A) \boxtimes (\mathop{\Box}_{i=1}^{n} c_i \rightarrow B_i) = \begin{array}{l} (out.j \rightarrow (A \boxtimes (\mathop{\Box}_{i=1}^{n} c_i \rightarrow B_i))) \\ \Box (\mathop{\Box}_{i=1}^{n} c_i \rightarrow halt.j \rightarrow B_i) \end{array}$$

In the case where the choice operator $\boxtimes$ involves two *out* events, then the events are offered in the order of the smallest delay first. These cases are based on the delay of the timers, where the function $delay(i)$ returns the delay of a timer given by the index i. In the case the two *out* events in the choice correspond to timers with the same delay, then the choice is nondeterministic.

Law 5.2.6

$$\begin{array}{c} (out.i \rightarrow A) \\ \boxtimes \\ (out.j \rightarrow B) \end{array} = \left\{ \begin{array}{ll} out.i \rightarrow (A \boxtimes (out.j \rightarrow B)), & \textit{if delay(i)}\geq\textit{delay(j);} \\ out.j \rightarrow ((out.i \rightarrow A) \boxtimes B), & \textit{if delay(i)}\leq\textit{delay(j);} \end{array} \right.$$

Provided that $i \neq j$

The next law states that the choice operator $\boxtimes$ is solved between timer events.

Law 5.2.7

$$\begin{array}{c} ((out.i \rightarrow A) \Box C) \\ \boxtimes \\ ((out.j \rightarrow B) \Box D) \end{array} = \left(\begin{array}{c} (out.i \rightarrow A) \\ \boxtimes \\ (out.j \rightarrow B) \end{array} \right) \Box (C \Box D)$$

The final law for the operator $\boxtimes$ states that for all other actions in which A or B do not start with the timer events then the choice is the same as the standard external choice operator.

Law 5.2.8

$$A \boxtimes B = A \Box B$$

Provided that setup and out are not in the initial events of A *or* B*.*

The following lemma establishes that new choice operator can always be replaced by a standard choice.

Lemma 5.2 *An action of the form* $A \boxtimes B$ *can always be reduced to* $A' \Box B'$ *such that* A' *and* B' *are actions that do not contain the operator* $\boxtimes$*.*

Proof:
The proof is by structural induction using laws 5.2.1 - 5.2.8. □

For parallel composition, Φ introduces a new parallel operator $[\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]^{nf}$:

$$\Phi(A \,[\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]\, B) = \Phi(A) \,[\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf}\, \Phi(B) \tag{5.2.12}$$

The semantics of this new parallel composition operator is given using the parallel merge strategy used previously in this chapter and in Chapter 3 to give the semantics of the time model parallel composition operator. Given two actions A and B, the following is the semantic definition of the new parallel composition operator:

$$A \,[\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf}\, B \mathrel{\widehat=} A \|_{NPM} B \tag{5.2.12}$$

where NPM stands for the normal form parallel merger function and is defined as follows:

$$NPM(cs, s_A, s_B) \mathrel{\widehat=} \begin{array}{l} (ok' = (0.ok \land 1.ok)) \land \\ (wait' = (0.wait \lor 1.wait)) \land \\ (state' = (0.state - s_B) \oplus (1.state - s_A)) \land \\ (dif(tr'_t, tr_t)\ NTSync(cs)\ dif(0.tr, tr), dif(1.tr, tr)) \end{array} \tag{5.2.12}$$

The synchronization function $NTSync$ takes three time traces and the synchronization set as parameters. It synchronizes the elements from the first time trace with the corresponding elements from the second time trace, based on the synchronization set cs. Case one of the two traces is empty then the shorter trace is complemented with an empty element for the missing time element. In the following, we define the $NTSync$ as a relation between the input traces $0.tr_t$,$1.tr_t$, the synchronization set cs and the final traces tr_t.

$$tr_t\ NTSync(cs)\ 0.tr_t, 1.tr_t \mathrel{\widehat=} \begin{array}{l} (0.tr_t = \langle\rangle \land 1.tr_t = \langle\rangle \land tr_t = \langle\rangle) \lor \\ (0.tr_t \neq \langle\rangle \land 1.tr_t = \langle\rangle \land tr_t\ NTSync(cs)\ 0.tr_t, \langle(\langle\rangle, \{\})\rangle) \lor \\ (0.tr_t = \langle\rangle \land 1.tr_t \neq \langle\rangle \land tr_t\ NTSync(cs)\ \langle(\langle\rangle, \{\})\rangle, 1.tr_t) \lor \\ \left(\begin{array}{l} (0.tr_t = \langle(t_1, r_1)\rangle \frown S_1) \land \\ (1.tr_t = \langle(t_2, r_2)\rangle \frown S_2) \land \\ fst(head(tr_t))\ NPSync(cs)\ t_1, t_2 \land \\ snd(head(tr_t)) = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \backslash cs) \\ tail(tr_t) NTSync S_1, S_2 \end{array} \right) \end{array} \tag{5.2.12}$$

Notice that the above predicate defines the $NTSync$ as a disjunction of three distinct behaviours. The first indicates the case that the input traces are empty, in which case the output trace is also an empty trace. Next the case that one of the two time traces is empty and the other is not, then the synchronization is carried out between either one of the input time traces, that is not empty, and an empty timed trace given by $\langle(\langle\rangle, \{\})\rangle$. Finally, case both input traces are not empty, then we synchronize the first time slots of the two traces. We synchronize the trace element of each time trace using a new synchronization relation $NPSync$. Next we define the relation $NPSync(cs)$ where

cs is the synchronization set. The definition of the synchronization relation is given as follows:

$$tr\ NPSync(cs)\ 0.tr, 1.tr \Leftrightarrow \begin{array}{l} EmptyTraces \lor SynchEvents \lor \\ NoSynchEvents \lor \\ doSetupEvent \lor \\ doOutEvent \end{array} \tag{5.2.12}$$

The above definition determines that the synchronization relation is defined as the conjunction of separate predicates over the input traces $0.tr$ and $1.tr$. The predicate *EmptyTraces* is used to synchronize the traces if both input traces are empty, in which case the result is an empty trace. Case any of the two input traces is not empty then the *Circus* untimed synchronization function *Sync* (described in Section 3.5.9) is used to obtain the output trace.

$$EmptyTraces \mathrel{\widehat=} \begin{array}{l} (0.tr = \langle\rangle \land 1.tr = \langle\rangle \land tr = \langle\rangle) \lor \\ (0.tr \neq \langle\rangle \land 1.tr = \langle\rangle \land tr \in Sync(0.tr, 1.tr, cs)) \lor \\ (0.tr = \langle\rangle \land 1.tr \neq \langle\rangle \land tr \in Sync(0.tr, 1.tr, cs)) \end{array} \tag{5.2.12}$$

The predicate *SynchEvents* describes the situation in which the event observed at the head of the output traces is an element of the synchronization set cs.

$$SynchEvents \mathrel{\widehat=} \begin{array}{l} (head(tr) \in cs) \land \\ (head(0.tr) = head(tr)) \land \\ (head(1.tr) = head(tr)) \land \\ tail(tr)\ NPSync(cs)\ tail(0.tr), tail(1.tr) \end{array} \tag{5.2.12}$$

In the case the event observed at the top of the resulting trace is in the synchronization set cs, then the input traces should both have the same event as the element at the head. The tail of the input traces should synchronize with the remaining part of the resulting trace tr. The predicate *NoSynchEvents* captures the behaviour when the element at the head of the resulting trace is not a member of cs and is not a timer event.

$$NoSynchEvents \mathrel{\widehat=} \begin{array}{l} (head(tr) \notin cs) \land \\ (head(tr) \notin timerEvents) \land \\ \left(\begin{array}{l} \left(\begin{array}{l} (head(0.tr) = head(tr)) \land \\ \forall\, i, d \bullet head(1.tr) \neq setup.i.d \land \\ tail(tr)\ NPSync(cs)\ tail(0.tr), 1.tr \end{array} \right) \lor \\ \left(\begin{array}{l} (head(1.tr) = head(tr)) \land \\ \forall\, i, d \bullet head(0.tr) \neq setup.i.d \land \\ tail(tr)\ NPSync(cs)\ 0.tr, tail(1.tr) \end{array} \right) \end{array} \right) \end{array} \tag{5.2.12}$$

Notice that the above predicate only permits the event at the head of the output trace tr to occur if one of the input traces $0.tr$ and $1.tr$ registers the same event at the head of the trace and the other is not ready to do a *setup* event. The reason behind this restriction is to give priority to the *setup* event in the new parallel composition.

The next predicate, *doSetupEvent*, deals in particular with the situation in which setup is observed at the head of the resulting trace.

$$doSetupEvent \mathrel{\widehat=} \begin{array}{l} \exists\, i, d \bullet (head(tr) = setup.i.d) \land \\ \left(\begin{array}{l} ((head(0.tr) = head(tr)) \land (tail(tr)\ NPSync(cs)\ tail(0.tr), 1.tr)) \lor \\ ((head(1.tr) = head(tr)) \land (tail(tr)\ NPSync(cs)\ 0.tr, tail(1.tr))) \end{array} \right) \end{array} \tag{5.2.12}$$

The above predicate determines that, if *setup* is observed at the top of the resulting trace, then it must be at the top of one of the input traces.

Finally, the predicate *doOutEvent* acts similarly to *doSetupEvent*, except that the events are ordered by the value of their respective timers delays. Case the event *out* is observed at the head of the resulting trace, then it must be at the head of one of the input traces $0.tr$ or $1.tr$. Case the observed event at the top of the trace is the *out* event, and both input traces have *out* as their top event, then the one associated to the timer with the smallest delay is chosen.

$$doOutEvent \mathrel{\widehat=} \begin{array}{l} \exists\, i, d \bullet (head(tr) = out.i.d) \land \\ \left(\begin{array}{l} \left(\begin{array}{l} (head(0.tr) = head(tr)) \land \\ \left(\begin{array}{l} (\forall\, j, n \bullet (head(1.tr \downarrow timerEvents) \neq out.j.n) \land i \neq j) \lor \\ \left(\begin{array}{l} \exists\, j, n \bullet head(1.tr \downarrow timerEvents) = out.j.n \land \\ i \neq j \end{array} \right) \Rightarrow d \leq n \end{array} \right) \land \\ (tail(tr)\ NPSync(cs)\ tail(0.tr), 1.tr) \end{array} \right) \lor \\ \left(\begin{array}{l} (head(1.tr) = head(tr)) \land \\ \left(\begin{array}{l} (\forall\, j, n \bullet (head(0.tr \downarrow timerEvents) \neq out.j.n) \land i \neq j) \lor \\ \left(\begin{array}{l} \exists\, j, n \bullet head(0.tr \downarrow timerEvents) = out.j.n \land \\ i \neq j \end{array} \right) \Rightarrow d \leq n \end{array} \right) \\ (tail(tr)\ NPSync(cs)\ 0.tr, tail(1.tr)) \end{array} \right) \end{array} \right) \end{array} \tag{5.2.12}$$

We define some expansion laws that can be used to convert a parallel action into a sequential action. We use these expansion laws to give an axiomatic semantics for the timer events and the parallel operator introduced by Φ. The first four laws are also valid for the CSP alphabetized parallel operator [45]. In the coming laws c_i and d_i are ordinary events; they cannot be timer events. The first law gives the basic expansion of the parallel composition of two actions that are ready to perform a communication independently.

Law 5.2.9

$$\begin{array}{c} (\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i) \\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]^{nf} \\ (\mathop{\Box}_{j=1}^{m} d_j \rightarrow B_j) \end{array} = \begin{array}{l} \left(\mathop{\Box}_{i=1}^{n} c_i \rightarrow \left(\begin{array}{c} A_i \\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]^{nf} \\ (\mathop{\Box}_{j=1}^{m} d_j \rightarrow B_j) \end{array} \right) \right) \\ \Box \\ \left(\mathop{\Box}_{j=1}^{m} d_j \rightarrow \left(\begin{array}{c} (\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i) \\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]^{nf} \\ B_j \end{array} \right) \right) \end{array}$$

Provided that for all i and j, $(c_i \notin cs) \land (d_j \notin cs)$.

Law 5.2.10 states that parallel composition will stop if both programs need to synchronize, but disagree on the events to synchronize.

Law 5.2.10

$$\begin{array}{c} (\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i) \\ [\![s_A \mid \{\!| \; cs \; |\!\} \mid s_B]\!]^{nf} \\ (\mathop{\Box}_{j=1}^{m} d_j \rightarrow B_j) \end{array} = Stop$$

Provided for all i and j, $(c_i \in cs) \land (d_j \in cs) \land (c_i \neq d_i)$.

The next law shows the expansion of a parallel composition where one of the actions needs to synchronize but also offers the choice of not synchronizing with the other; in this case the free action can proceed while the other waits for the synchronization.

Law 5.2.11

$$\begin{array}{c} \left(\begin{array}{c} (\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i) \\ \Box \\ (\mathop{\Box}_{l=1}^{k} e_l \rightarrow C_l) \end{array} \right) \\ (\mathop{\Box}_{j=1}^{m} d_j \rightarrow B_j) \end{array} [\![s_A \mid \{\!| \; cs \; |\!\} \mid s_B]\!]^{nf} = \begin{array}{c} \left(\mathop{\Box}_{j=1}^{m} d_j \rightarrow \left(\begin{array}{c} (\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i) \\ [\![s_A \mid \{\!| \; cs \; |\!\} \mid s_B]\!]^{nf} \\ B_j \end{array} \right) \right) \\ \Box \\ \left(\mathop{\Box}_{l=1}^{k} e_l \rightarrow \left(\begin{array}{c} C_l \\ [\![s_A \mid \{\!| \; cs \; |\!\} \mid s_B]\!]^{nf} \\ (\mathop{\Box}_{j=1}^{m} d_j \rightarrow B_j) \end{array} \right) \right) \end{array}$$

Provided for all i,j and l, $(c_i \in cs) \land (e_l \notin cs) \land (d_j \notin cs)$.

The next law states that the programs synchronize if they are ready to engage in the same event.

Law 5.2.12

$$\begin{array}{c} (\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i) \\ [\![s_A \mid \{\!| \; cs \; |\!\} \mid s_B]\!]^{nf} \\ (\mathop{\Box}_{i=1}^{n} c_i \rightarrow B_i) \end{array} = \mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i \, [\![\, s_A \mid \{\!| \; cs \; |\!\} \mid s_B \,]\!]^{nf} \, B_i$$

Provided for all i and j, $(c_i \in cs)$.

The following two laws show the behaviour of the parallel composition in case the two actions can either synchronize or proceed independently.

Law 5.2.13

$$
\begin{array}{c}
(\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i) \\
[\![s_A \mid \{\!| \, cs \, |\!\} \mid s_B]\!]^{nf} \\
\left(\begin{array}{l} (\mathop{\Box}_{i=1}^{n} c_i \rightarrow B_i) \\ \Box \\ (\mathop{\Box}_{j=1}^{m} d_j \rightarrow C_i) \end{array} \right)
\end{array}
=
\begin{array}{l}
\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i \, [\![\, s_A \mid \{\!| \, cs \, |\!\} \mid s_B \,]\!]^{nf} \, B_i \\
\sqcap \\
\left(\begin{array}{l} (\mathop{\Box}_{i=1}^{m} c_i \rightarrow A_i) \\ [\![s_A \mid \{\!| \, cs \, |\!\} \mid s_B]\!]^{nf} \\ (\mathop{\Box}_{j=1}^{m} d_j \rightarrow C_i) \end{array} \right)
\end{array}
$$

Provided for all i and j , $(c_i \in cs) \land (d_j \notin cs)$.

Law 5.2.14

$$
\begin{array}{c}
\left(\begin{array}{l} (\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i) \\ \Box \\ (\mathop{\Box}_{l=1}^{k} e_l \rightarrow D_l) \end{array} \right) \\
[\![s_A \mid \{\!| \, cs \, |\!\} \mid s_B]\!]^{nf} \\
\left(\begin{array}{l} (\mathop{\Box}_{i=1}^{n} c_i \rightarrow B_i) \\ \Box \\ (\mathop{\Box}_{j=1}^{m} d_j \rightarrow C_i) \end{array} \right)
\end{array}
=
\begin{array}{l}
\mathop{\Box}_{i=1}^{n} c_i \rightarrow A_i \, [\![\, s_A \mid \{\!| \, cs \, |\!\} \mid s_B \,]\!]^{nf} \, B_i \\
\sqcap \\
\left(\begin{array}{l} (\mathop{\Box}_{i=1}^{m} c_i \rightarrow A_i) \\ [\![s_A \mid \{\!| \, cs \, |\!\} \mid s_B]\!]^{nf} \\ (\mathop{\Box}_{j=1}^{m} d_j \rightarrow C_i) \end{array} \right) \\
\sqcap \\
\left(\begin{array}{l} (\mathop{\Box}_{l=1}^{k} e_l \rightarrow D_l) \\ [\![s_A \mid \{\!| \, cs \, |\!\} \mid s_B]\!]^{nf} \\ (\mathop{\Box}_{i=1}^{n} c_i \rightarrow B_i) \end{array} \right)
\end{array}
$$

Provided for all i, j and l, $(c_i \in cs) \land (d_j \notin cs) \land (e_l \notin cs)$.

The following laws assign a more specialized meaning to the new parallel operator considering the timer events. The next law states that the events *setup*, when in parallel, have to occur before any other events in the action can occur.

Law 5.2.15

$$
\begin{array}{c}
(setup.i.d \rightarrow A) \\
[\![s_A \mid \{\!| \, cs \, |\!\} \mid s_B]\!]^{nf} \\
(setup.j.e \rightarrow B)
\end{array}
=
\left(\begin{array}{c} (setup.i.d \rightarrow setup.j.e \rightarrow Skip) \\ \sqcap \\ (setup.j.e \rightarrow setup.i.d \rightarrow Skip) \end{array} \right) ; \, (A \, [\![s_A \mid \{\!| \, cs \, |\!\} \mid s_B]\!]^{nf} \, B)
$$

Provided $i \neq j$.

The next expansion laws state the order in which the events *out* synchronize, based on the delay of the timers for each event.

Law 5.2.16

$$
\begin{array}{c}
(out.i \rightarrow A) \\
[\![s_A \mid \{\!| \, cs \, |\!\} \mid s_B]\!]^{nf} \\
(out.j \rightarrow B)
\end{array}
= out.i \rightarrow (A \, [\![\, s_A \mid \{\!| \, cs \, |\!\} \mid s_B \,]\!]^{nf} \, (out.j \rightarrow B))
$$

Provided $delay(i) < delay(j)$

Law 5.2.17

$$\begin{array}{c}(out.i \rightarrow A)\\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]^{nf}\\ (out.j \rightarrow B)\end{array} = \left(\begin{array}{c}(out.i \rightarrow out.j)\\ \sqcap \\ (out.j \rightarrow out.i)\end{array}\right) ; (A \, [\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf} \, B)$$

Provided $delay(i) = delay(j)$

The following two laws state the behaviour of the *out* event when in parallel with other events that do not include the *out* event.

Law 5.2.18

$$\begin{array}{c}(out.i \rightarrow A)\\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]^{nf}\\ (\mathop{\Box}_{j=1}^{n} c_j \rightarrow B_j)\end{array} = \begin{array}{l} out.i \rightarrow (A \, [\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf} \, (\mathop{\Box}_{j=1}^{n} c_j \rightarrow B_j))\\ \sqcap \, (\mathop{\Box}_{j=1}^{n} c_j \rightarrow ((out.i \rightarrow A) \, [\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf} \, B_j))\end{array}$$

Provided for all j, $(c_j \notin cs) \land (c_j \neq out.i)$ *for any timer index* i.

Law 5.2.19

$$\begin{array}{c}(out.i \rightarrow A)\\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]^{nf}\\ (\mathop{\Box}_{j=1}^{n} c_j \rightarrow B_j)\end{array} = out.i \rightarrow (A \, [\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf} \, (\mathop{\Box}_{j=1}^{n} c_j \rightarrow B_j))$$

Provided for all j, $(c_j \in cs) \land (c_j \neq out.i)$ *for any timer index* i.

The next expansion laws for parallel composition show the case that two actions in parallel can either do *out* events or other events.

Law 5.2.20

$$\begin{array}{c}(out.i \rightarrow A) \, \Box \, B\\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]^{nf}\\ (out.j \rightarrow C) \, \Box \, D\end{array} = out.i \rightarrow (A \, [\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf} \, ((out.j \rightarrow C) \, \Box \, D))$$

Provided $delay(i) < delay(j)$.

Law 5.2.21

$$\begin{array}{c}(out.i \rightarrow A) \, \Box \, B\\ [\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]^{nf}\\ (out.j \rightarrow C) \, \Box \, D\end{array} = \begin{array}{l} out.i \rightarrow (A \, [\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf} \, ((out.j \rightarrow C) \, \Box \, D))\\ \Box\\ out.j \rightarrow (((out.i \rightarrow A) \, \Box \, B) \, [\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf} \, C)\end{array}$$

Provided $delay(i) = delay(j)$.

Lemma 5.3 *An action of the form* $A \, [\![\, s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]^{nf} \, B$ *can always be reduced to* $A' \, \Box \, B'$ *such that* A' *and* B' *are actions that do not contain the normal form parallel operator.*

Proof:
The proof is by structural induction using laws 5.2.9 - 5.2.21. □

Finally we give the following expansion law that relates that new parallel composition operator to the original parallel composition operator case both operands make no reference to the special timer events.

Law 5.2.22

$$A \,[\![\, s_A \mid \{\!|\, cs \,|\!\} \mid s_B \,]\!]^{nf}\, B = A \,[\![\, s_A \mid \{\!|\, cs \,|\!\} \mid s_B \,]\!]\, B$$

Provided that A and B have no timer events.

Actually, it is not necessary always to serialise the normal form parallelism. If the actions in parallel have no time events, then the normal form parallelism reduces to the standard one.

5.3 CORRECTNESS OF THE NORMAL FORM REDUCTION

The correctness of the Normal Form reduction process is captured by the theorem below.

Theorem 5.1 *Consider a timed action A with an arbitrary number of time operators and associated timers with indices, ranging from k to n. Then*

$$A \equiv \Phi(A) \;\textbf{par}\; \mathit{Timers}(k, n)$$

Proof:
The proof is by structural induction over the language constructs. The base case of the structural induction are the langauge basic operators. We include only the proof for the case of assignment, *Wait* d and Timeout; for the remaining cases refer to Appendix F.
Case: Assignment

$$\Phi(x := e) \;\textbf{par}\; \mathit{Timers}(k, n) \equiv x := e$$

$\Phi(x := e)$ **par** $\mathit{Timers}(k, n)$
$=$ [5.2.4]
$x := e$ **par** $\mathit{Timers}(k, n)$
$=$ [5.2.3]

$$\left(\begin{array}{c} x := e;\ \mathit{Terminate}(k, n) \\ [\![\{x\} \mid \{\!|\, \mathit{TSet} \,|\!\} \mid \{\}]\!] \\ \mathit{Timers}(k, n) \end{array}\right) \setminus \mathit{TSet}$$

$= \mathit{Property}3.14L12 and L15$

$$
\begin{array}{ll}
& x := e;\ \left(\begin{array}{c} Terminate(k,n) \\ [\![\{x\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ Timers(k,n) \end{array} \right) \setminus TSet \\
= & [5.2.3] \\
& x := e;\ \left(\begin{array}{c} \Vert_{i=k}^{n}\ (terminate.i \rightarrow Skip) \\ [\![\{x\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ Timers(k,n) \end{array} \right) \setminus TSet \\
= & [5.2.2] \\
& x := e;\ \left(\begin{array}{c} \Vert_{i=k}^{n}\ (terminate.i \rightarrow Skip) \\ [\![\{x\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ \Vert_{i=k}^{n}\ (Timer(i, delay(i))) \end{array} \right) \setminus TSet \\
= & [5.2.2] \\
& x := e;\ \left(\begin{array}{c} \Vert_{i=k}^{n}\ (terminate.i \rightarrow Skip) \\ [\![\{x\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ \Vert_{i=k}^{n}\ \mu\, X \bullet \left(\begin{array}{l} \left(\begin{array}{l} setup.i.delay(i) \rightarrow \\ \left(\begin{array}{l} (halt.i \rightarrow X) \\ \Box \left(\begin{array}{l} Wait\ \ \mathsf{delay(i)}; \\ \left(\begin{array}{l} (out.i \rightarrow X) \\ \Box \\ (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \setminus TSet \\
= & [\text{Step law and } 5.2.2] \\
& x := e;\ \left(\begin{array}{c} \Vert_{i=k}^{n}\ (terminate.i \rightarrow Skip) \\ [\![\{x\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ \Vert_{i=k}^{n}\ \left(\begin{array}{l} \left(\begin{array}{l} setup.i.delay(i) \rightarrow \\ \left(\begin{array}{l} (halt.i \rightarrow (Timer(i, delay(i)))) \\ \Box \left(\begin{array}{l} Wait\ \ \mathsf{delay(i)}; \\ \left(\begin{array}{l} (out.i \rightarrow (Timer(i, delay(i)))) \\ \Box \\ (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \setminus TSet \\
= & [terminate.i \in TSet \text{ and Property 3.12 L11}] \\
& x := e;\ \left(\Vert_{i=k}^{n}\ (terminate.i \rightarrow Skip) \right) \setminus TSet
\end{array}
$$

$=$ $[terminate.i \in TSet]$

$x := e$ $\Box$

Case: *Wait* d

$$\Phi(\mathit{Wait}\ \mathsf{d})\ \mathbf{par}\ \mathit{Timers}(k,n) \equiv \mathit{Wait}\ \mathsf{d}$$

$\Phi(\mathit{Wait}\ \mathsf{d})\ \mathbf{par}\ \mathit{Timers}(k,n)$

$=$ [5.2.4]

$(\mathit{setup}.k.d \rightarrow \mathit{out}.k \rightarrow \mathit{Skip})\ \mathbf{par}\ \mathit{Timers}(k,n)$

$=$ [5.2.3]

$$\left(\begin{array}{c}((\mathit{setup}.k.d \rightarrow \mathit{out}.k \rightarrow \mathit{Skip});\ \mathit{Terminate}(k,n)) \\ [\![\{\} \mid \{\!|\ \mathit{TSet}\ |\!\} \mid \{\}]\!] \\ \mathit{Timers}(k,n)\end{array}\right) \setminus \mathit{TSet}$$

$=$ [5.2.2]

$$\left(\begin{array}{c}((\mathit{setup}.k.d \rightarrow \mathit{out}.k \rightarrow \mathit{Skip});\ \mathit{Terminate}(k,n)) \\ [\![\{\} \mid \{\!|\ \mathit{TSet}\ |\!\} \mid \{\}]\!] \\ \|\!|_{i=k}^{n}\ (\mathit{Timer}(i,\mathit{delay}(i)))\end{array}\right) \setminus \mathit{TSet}$$

$=$ [5.2.2]

$$\left(\begin{array}{c}((\mathit{setup}.k.d \rightarrow \mathit{out}.k \rightarrow \mathit{Skip});\ \mathit{Terminate}(k,n)) \\ [\![\{\} \mid \{\!|\ \mathit{TSet}\ |\!\} \mid \{\}]\!] \\ \|\!|_{i=k}^{n}\ (\mathit{Timer}(i,\mathit{delay}(i)))\end{array}\right) \setminus \mathit{TSet}$$

$=$ [Property of indexing]

$$\left(\begin{array}{c}((\mathit{setup}.k.d \rightarrow \mathit{out}.k \rightarrow \mathit{Skip});\ \mathit{Terminate}(k,n)) \\ [\![\{\} \mid \{\!|\ \mathit{TSet}\ |\!\} \mid \{\}]\!] \\ \mathit{Timer}(k,\mathit{delay}(k)) \ |\!|\!|\ \|\!|_{i=k+1}^{n}\ (\mathit{Timer}(i,\mathit{delay}(i)))\end{array}\right) \setminus \mathit{TSet}$$

$=$ [5.2.2]

$$\left(\begin{array}{c}((\mathit{setup}.k.d \rightarrow \mathit{out}.k \rightarrow \mathit{Skip});\ \mathit{Terminate}(k,n)) \\ [\![\{\} \mid \{\!|\ \mathit{TSet}\ |\!\} \mid \{\}]\!] \\ \left(\begin{array}{l}\mu\, X \bullet \left(\begin{array}{l}\mathit{setup}.k?\mathit{delay}(k) \rightarrow \\ \left(\begin{array}{l}(\mathit{halt}.\mathit{delay}(k) \rightarrow X) \\ \Box\ \mathit{Wait}\ \mathsf{delay(k)}; \left(\begin{array}{l}(\mathit{out}.k \rightarrow X) \\ \Box \\ (\mathit{terminate}.k \rightarrow \mathit{Skip})\end{array}\right) \\ \Box\ (\mathit{terminate}.k \rightarrow \mathit{Skip})\end{array}\right)\end{array}\right) \\ \Box\ (\mathit{terminate}.k \rightarrow \mathit{Skip}) \\ |\!|\!|\ (\|\!|_{i=k+1}^{n}\ (\mathit{Timer}(i,\mathit{delay}(i))))\end{array}\right)\end{array}\right)$$

$$\setminus TSet$$

$=$ [Step law and 5.2.2]

$$\left(\begin{array}{c}
((setup.k.d \rightarrow out.k \rightarrow Skip);\ Terminate(k,n)) \\
[\![\{\} \mid \{\!\!| TSet |\!\!\} \mid \{\}]\!] \\
\left(\left(\left(\begin{array}{l}
setup.k?delay(1) \rightarrow \\
\left(\begin{array}{l}
(halt.delay(k) \rightarrow Timer(k, delay(k))) \\
\Box\ Wait\ \ \mathsf{delay(k)};\ \left(\begin{array}{l} (out.k \rightarrow Timer(k, delay(k))) \\ \Box \\ (terminate.k \rightarrow Skip) \end{array}\right) \\
\Box\ (terminate.k \rightarrow Skip)
\end{array}\right)
\end{array}\right) \\
\Box\ (terminate.k \rightarrow Skip)
\right) \\
\mid\mid\mid\ (\mathop{\mid\mid\mid}_{i=k+1}^{n}\ (Timer(i, delay(i))))
\right)
\end{array}\right)$$

$$\setminus TSet$$

$=$ [$setup.k.d \in TSet \wedge d = delay(k)$ and Property 3.5.9 L11]

$$\left(\begin{array}{c}
setup.k.d \rightarrow \\
\left(\begin{array}{c}
(out.k \rightarrow Skip);\ Terminate(k,n) \\
[\![\{\} \mid \{\!\!| TSet |\!\!\} \mid \{\}]\!] \\
\left(\left(\begin{array}{l}
(halt.delay(k) \rightarrow Timer(k, delay(k))) \\
\Box\ Wait\ \ \mathsf{delay(k)};\ \left(\begin{array}{l} (out.k \rightarrow Timer(k, delay(k))) \\ \Box \\ (terminate.k \rightarrow Skip) \end{array}\right) \\
\Box\ (terminate.k \rightarrow Skip)
\end{array}\right) \\
\mid\mid\mid\ (\mathop{\mid\mid\mid}_{i=k+1}^{n}\ (Timer(i, delay(i))))
\right)
\end{array}\right)
\end{array}\right)$$

$$\setminus TSet$$

$=$ [Property 3.12 L13]

$$\left(\begin{array}{c}
setup.k.d \rightarrow Wait\ \ \mathsf{d}; \\
\left(\begin{array}{c}
(out.k \rightarrow Skip);\ Terminate(k,n) \\
[\![\{\} \mid \{\!\!| TSet |\!\!\} \mid \{\}]\!] \\
\left(\left(\begin{array}{l}
(out.k \rightarrow Timer(k, delay(k))) \\
\Box \\
(terminate.k \rightarrow Skip)
\end{array}\right) \\
\mid\mid\mid\ (\mathop{\mid\mid\mid}_{i=k+1}^{n}\ (Timer(i, delay(i))))
\right)
\end{array}\right)
\end{array}\right) \setminus TSet$$

$=$ [$out.k \in TSet$ and property 3.12 L11]

$$\left(\begin{array}{c}
setup.k.d \rightarrow Wait\ \ \mathsf{d};\ (out.k \rightarrow Skip); \\
\left(\begin{array}{c}
Terminate(k,n) \\
[\![\{\} \mid \{\!\!| TSet |\!\!\} \mid \{\}]\!] \\
\left(\begin{array}{l}
Timer(k, delay(k)) \\
\mid\mid\mid\ (\mathop{\mid\mid\mid}_{i=k+1}^{n}\ (Timer(i, delay(i))))
\end{array}\right)
\end{array}\right)
\end{array}\right) \setminus TSet$$

$$
\begin{array}{ll}
= & \text{[Property of indexing]} \\
\left(\begin{array}{l} setup.k.d \rightarrow Wait\ \ \mathsf{d};\ (out.k \rightarrow Skip); \\ \left(\begin{array}{c} Terminate(k,n) \\ |[\{\}\mid\{\!|\ TSet\ |\!\}\mid\{\}]| \\ |||\ (\ \underset{i=k+1}{\overset{n}{|||}}\ (Timer(i, delay(i)))) \end{array} \right) \end{array} \right) \setminus TSet & \\
= & \text{[Property 3.14 L12]} \\
\left(\begin{array}{l} (setup.k.d \rightarrow Wait\ \ \mathsf{d};\ (out.k \rightarrow Skip)) \setminus TSet; \\ \left(\begin{array}{c} Terminate(k,n) \\ |[\{\}\mid\{\!|\ TSet\ |\!\}\mid\{\}]| \\ |||\ (\ \underset{i=k}{\overset{n}{|||}}\ (Timer(i, delay(i)))) \end{array} \right) \setminus TSet \end{array} \right) & \\
= & \text{[Lemma 5.1]} \\
((setup.k.d \rightarrow Wait\ \ \mathsf{d};\ (out.k \rightarrow Skip)) \setminus TSet);\ Skip & \\
= & \text{[}Skip\text{ right unit of sequential composition]} \\
((setup.k.d \rightarrow Wait\ \ \mathsf{d};\ (out.k \rightarrow Skip)) \setminus TSet) & \\
= & \text{[Property 3.14 L5]} \\
((Wait\ \ \mathsf{d};\ (out.k \rightarrow Skip)) \setminus TSet) & \\
= & \text{[Property 3.14 L12]} \\
((Wait\ \ \mathsf{d} \setminus TSet);\ (out.k \rightarrow Skip) \setminus TSet) & \\
= & \text{[Property 3.14 L4, L5 and L6]} \\
Wait\ \ \mathsf{d};\ Skip & \\
= & \text{[}Skip\text{ right unit of sequential composition]} \\
Wait\ \ \mathsf{d} & \square
\end{array}
$$

Next we assume that for any actions A and B then $\Phi(A)$ **par** $Timers(k,n) \equiv A$ and $\Phi(B)$ **par** $Timers(j,m) \equiv B$ such that the timer index intervals (k,n) and (j,m) are disjoint. This is the hypothesis of the structural induction proof.

Case: Timeout

$$\Phi((c \rightarrow A) \overset{\mathsf{d}}{\triangleright} B)\ \textbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m)) \equiv ((c \rightarrow A) \overset{\mathsf{d}}{\triangleright} B)$$

$$
\begin{array}{ll}
\Phi((c \rightarrow A) \overset{\mathsf{d}}{\triangleright} B)\ \textbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m)) & \\
= & \text{[5.2.4]} \\
\left(setup.k.d \rightarrow \left(\begin{array}{l} (c \rightarrow halt.k \rightarrow \Phi(A)) \\ \square \\ (out.k \rightarrow int \rightarrow \Phi(B)) \end{array} \right) \setminus \{int\} \right) & \\
\qquad \textbf{par} & \\
(Timers(k,n) \mid\mid\mid Timers(j,m)) & \\
= & \text{[5.2.3]}
\end{array}
$$

$$
\left(\begin{array}{c}
\left(setup.k.d \rightarrow \left(\begin{array}{l}(c \rightarrow halt.k \rightarrow \Phi(A))\\ \Box \\ (out.k \rightarrow int \rightarrow \Phi(B))\end{array}\right) \setminus \{int\}\right); \\
(Terminate(k,n) \;|||\; Terminate(j,m)) \\
[\![\{\} \mid \{\!|\; TSet \;|\!\}\mid \{\}]\!] \\
(Timers(k,n) \;|||\; Timers(j,m))
\end{array}\right) \setminus TSet
$$

$=$ [Decompose timers]

$$
\left(\begin{array}{c}
\left(setup.k.d \rightarrow \left(\begin{array}{l}(c \rightarrow halt.k \rightarrow \Phi(A))\\ \Box \\ (out.k \rightarrow int \rightarrow \Phi(B))\end{array}\right) \setminus \{int\}\right); \\
(Terminate(k,n) \;|||\; Terminate(j,m)) \\
[\![\{\} \mid \{\!|\; TSet \;|\!\}\mid \{\}]\!] \\
(Timer(k,d) \;|||\; Timers(k+1,n) \;|||\; Timers(j,m))
\end{array}\right) \setminus TSet
$$

$=$ [int free in Timer(k,d), Timers(k+1,n) and Timers(j,m)]

$$
\left(\begin{array}{c}
\left(setup.k.d \rightarrow \left(\begin{array}{l}(c \rightarrow halt.k \rightarrow \Phi(A))\\ \Box \\ (out.k \rightarrow int \rightarrow \Phi(B))\end{array}\right)\right); \\
(Terminate(k,n) \;|||\; Terminate(j,m)) \\
[\![\{\} \mid \{\!|\; TSet \;|\!\}\mid \{\}]\!] \\
(Timer(k,d) \;|||\; Timers(k+1,n) \;|||\; Timers(j,m))
\end{array}\right) \setminus (TSet \cup \{int\})
$$

$=$ [5.2.2]

$$
\left(\begin{array}{l}
\left(setup.k.d \rightarrow \left(\begin{array}{l}(c \rightarrow halt.k \rightarrow \Phi(A))\\ \Box \\ (out.k \rightarrow int \rightarrow \Phi(B))\end{array}\right)\right); \\
(Terminate(k,n) \;|||\; Terminate(j,m)) \\
[\![\{\} \mid \{\!|\; TSet \;|\!\}\mid \{\}]\!] \\
\left(setup.k?d \rightarrow \left(\begin{array}{l}(halt.k \rightarrow Timer(k,d)) \\ \Box\; Wait\;\; \mathsf{d}; \left(\begin{array}{l}(out.k \rightarrow Timer(k,d))\\ \Box \\ (terminate.k \rightarrow Skip)\end{array}\right) \\ \Box\; (terminate.i \rightarrow Skip)\end{array}\right)\right) \\
\Box\; (terminate.i \rightarrow Skip) \\
\quad ||| \; (Timers(k+1,n) \;|||\; Timers(j,m))
\end{array}\right) \setminus (TSet \cup \{int\})
$$

$=$ [Property 3.12 L11]

$$\left(\mathit{setup}.k.d \rightarrow \left(\begin{array}{l}\left(\begin{array}{l}c \rightarrow \mathit{halt}.k \rightarrow \\ \left(\begin{array}{l}\Phi(A); \\ (\mathit{Terminate}(k,n) \interleave \mathit{Terminate}(j,m)) \\ [\![\{\} \mid \{\!| \mathit{TSet} |\!\} \mid \{\}]\!] \\ \left(\begin{array}{c}\mathit{Timer}(k,d) \\ \interleave \mathit{Timers}(k+1,n) \\ \interleave \mathit{Timers}(j,m)\end{array}\right)\end{array}\right)\end{array}\right) \\ \Box \\ \left(\begin{array}{l}\mathit{Wait}\ \mathsf{d};\ \mathit{out}.k \rightarrow \mathit{int} \rightarrow \\ \left(\begin{array}{l}\Phi(B); \\ (\mathit{Terminate}(k,n) \interleave \mathit{Terminate}(j,m)) \\ [\![\{\} \mid \{\!| \mathit{TSet} |\!\} \mid \{\}]\!] \\ \left(\begin{array}{c}\mathit{Timer}(k,d) \\ \interleave \mathit{Timers}(k+1,n) \\ \interleave \mathit{Timers}(j,m)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)\right) \setminus (\mathit{TSet} \cup \{\mathit{int}\})$$

$=$ [Property 3.12 L4 and L6]

$$\left(\begin{array}{l}\left(\begin{array}{l}c \rightarrow \\ \left(\begin{array}{l}\Phi(A); \\ (\mathit{Terminate}(k,n) \interleave \mathit{Terminate}(j,m)) \\ [\![\{\} \mid \{\!| \mathit{TSet} |\!\} \mid \{\}]\!] \\ \left(\begin{array}{c}\mathit{Timers}(k,n) \\ \interleave \mathit{Timers}(j,m)\end{array}\right)\end{array}\right) \setminus \mathit{TSet}\end{array}\right) \\ \Box \\ \left(\begin{array}{l}\mathit{Wait}\ \mathsf{d};\ \mathit{int} \rightarrow \\ \left(\begin{array}{l}\Phi(B); \\ (\mathit{Terminate}(k,n) \interleave \mathit{Terminate}(j,m)) \\ [\![\{\} \mid \{\!| \mathit{TSet} |\!\} \mid \{\}]\!] \\ \left(\begin{array}{c}\mathit{Timers}(k,n) \\ \interleave \mathit{Timers}(j,m)\end{array}\right)\end{array}\right) \setminus \mathit{TSet}\end{array}\right) \setminus \{\mathit{int}\}\end{array}\right)$$

$=$ [5.2.3]

$$\left(\left(\begin{array}{l}c \rightarrow (\Phi(A)\ \mathbf{par}\ (\mathit{Timers}(k,n) \interleave \mathit{Timers}(j,m))) \\ \Box \\ \mathit{Wait}\ \mathsf{d};\ \mathit{int} \rightarrow (\Phi(B)\ \mathbf{par}\ (\mathit{Timers}(k,n) \interleave \mathit{Timers}(j,m)))\end{array}\right) \setminus \{\mathit{int}\}\right)$$

$=$ [Hipothesis]

$$\left(\left(\begin{array}{l}c \rightarrow A \\ \Box \\ \mathit{Wait}\ \mathsf{d};\ \mathit{int} \rightarrow B\end{array}\right) \setminus \{\mathit{int}\}\right)$$

$=$ [3.17]

$(c \rightarrow A) \overset{\mathsf{d}}{\triangleright} B$ $\Box$

For the remaining cases of the structural induction proof see Appendix F.

5.4 REFINEMENT OF NORMAL FORM PROGRAMS

As established in the previous section, a program P is equivalent to its normal form:

$$P \equiv \Phi(P) \ \mathbf{par} \ \mathit{Timers}_P$$

For convenience, we omit the parameters of the timers. Consider a possible specification S of P and its normal form:

$$S \equiv \Phi(S) \ \mathbf{par} \ \mathit{Timers}_S$$

The main motivation for such a normal form representation is to isolate explicit time concerns in the timer component of the normal form, with the hope of being able to conduct reasoning using only the action normalised through the function Φ. Therefore it is desirable that

$$(\Phi(P) \sqsupseteq \Phi(S)) \Rightarrow (P \sqsupseteq S)$$

Unfortunately, this is not true in general. When $\mathit{Timers}_P \neq \mathit{Timers}_S$ the corresponding normal forms cannot even be compared. Nevertheless, it is reasonable to assume that the timers of a specification and corresponding implementation are the same. This is because the time delays between the specification and the implementation have to be the same or the comparison between them is not possible. In such cases, the following theorem holds.

Theorem 5.2 *Let* $P \equiv \Phi(P)$ ***par*** *Timers and* $S \equiv \Phi(S)$ ***par*** *Timers. Then*

$$(\Phi(P) \sqsupseteq \Phi(S)) \Rightarrow (P \sqsupseteq S)$$

Proof:
Direct from the monotonicity of the **par** operator. □

Although this helps to simplify reasoning, $\Phi(P)$ and $\Phi(Q)$ are still actions in the *Circus* Time Action model. Our purpose is to reduce the reasoning completely to the untimed model.

5.5 LINKING REFINEMENT OF UNTIMED AND TIMED ACTIONS

In the previous section we have shown that, provided the timers of a specification and an implementation are the same, we can ignore them for reasoning.

Consider $\Phi(P)$ as in the previous section. Although this is an action in the *Circus* Time Action model, it can be abstracted to the untimed model using the function L. Actually, $L(\Phi(P))$ is syntactically identical to $\Phi(P)$, but, semantically, the latter is an untimed program. The same abstraction can be applied to $\Phi(S)$.

The following theorem captures the correctness of the final step of our validation framework, establishing that the reasoning can be conducted in the untimed model.

Theorem 5.3 *For any timed action program* P *and a timed action specification* S*, let* $P' = \Phi(P)$ *and* $S' = \Phi(S)$ *then*

$$(L(P') \sqsupseteq L(S')) \Rightarrow (P' \sqsupseteq S')$$

Proof

$$
\begin{array}{ll}
L(P') \sqsupseteq L(S') & \\
\Rightarrow & [L \text{ and } R \text{ form a Galois connection}] \\
P' \sqsupseteq R(L(S')) & \\
\equiv & [S' \text{ is time insensitive}] \\
P' \sqsupseteq S' \qquad \Box &
\end{array}
$$

In summary, the proposed validation framework can be used as follows. Starting from a specification S and a program P, generate their respective normal forms, with the same set of timers:

$$S = \Phi(S) \textbf{ par } \mathit{Timers}$$

$$P = \Phi(P) \textbf{ par } \mathit{Timers}$$

Abstract $\Phi(S)$ and $\Phi(P)$ into the untimed model and prove that the following holds

$$L(\Phi(P)) \sqsupseteq L(\Phi(S))$$

From Theorem 5.3, it follows that

$$\Phi(P) \sqsupseteq \Phi(S)$$

Then, from Theorem 5.2, it follows that

$$P \sqsupseteq S$$

The fact that the normal form reduction preserves semantics is guaranteed by Theorem 5.2. The next section explores the approach by applying it to a case study.

5.6 AN ALARM SYSTEM

The system used here as case study is also considered in [32]; it is a common burglar alarm controller connected to sensors which detect movements or changes in the environment. When disabled, the controller ignores any disturbance detected; when enabled, the controller will sound an alarm when a sensor signals a disturbance.

There are two timing requirements on the alarm controller.

- The first one states that after enabling the alarm controller, there is a period T_1 before a disturbance is detected. This period permits a person to enable the alarm and get out.

- The second requirement states that, when a disturbance is detected, the controller will wait for a period T_2 before activating the alarm. This will allow a person to enter the building and deactivate the alarm.

5.6.1 The Timed Specification

The event *enable* indicates that the alarm system is enabled. To disable the alarm, the event *disable* is used. When the alarm system is disabled it responds only to the event *enable*. The event *disturbed* indicates that a sensor has detected a disturbance. Finally, *alarm* signals the firing of the alarm.

We can model the alarm controller with the help of several actions that are composed to produce the final system *Alarm*.

$$\begin{array}{rcl} \mathit{Disable} & \widehat{=} & (\mathit{disable} \rightarrow \mathit{Skip}) \\ \mathit{Running} & \widehat{=} & \mathit{Disable} \,\Box\, (\mathit{disturbed} \rightarrow \mathit{Active}) \\ \mathit{Active} & \widehat{=} & \mathit{Disable} \overset{\mathsf{T}_2}{\triangleright} (\mathit{alarm} \rightarrow \mathit{Disable}) \\ \mathit{Alarm} & \widehat{=} & \mu\, X \bullet (\mathit{enable} \rightarrow (\mathit{Disable} \overset{\mathsf{T}_1}{\triangleright} \mathit{Running})) ;\ X \end{array}$$

The action *Disable* simply offers to engage on event *disable*. The action *Running* represents the armed behaviour of the alarm controller: the controller can either be disabled or it can be *disturbed* by a burglar. When the alarm is disturbed, it behaves as *Active*, which models the active state of the alarm. In this state, the controller can again be disabled for the first T_2 time units; after this period an *alarm* is fired. When fired, the controller will only terminate once disabled. The main action *Alarm* is recursive. The controller starts by assuming that the alarm is disabled and offers the *enable* event to start the controller. After the event *enable*, the action can be disabled for the first T_1 time units before it is armed.

5.6.2 Alarm Controller Normal Form

By applying the normal form function Φ to the timed specification of the alarm controller, we obtain the following result.

$$\begin{array}{rcl} \Phi(\mathit{Disable}) & = & \mathit{Disable} \\ \Phi(\mathit{Running}) & = & \mathit{Disable} \,\Box\, (\mathit{disturbed} \rightarrow \Phi(\mathit{Active})) \\ \Phi(\mathit{Active}) & = & \mathit{setup}.2!\,T_2 \rightarrow \left(\begin{array}{c} (\mathit{Disable};\ \mathit{halt}.2 \rightarrow \mathit{Skip}) \\ \Box\, (\mathit{out}.2 \rightarrow \mathit{alarm} \rightarrow \mathit{Disable})) \end{array} \right) \\ \Phi(\mathit{Alarm}) & = & \mu\, X \bullet (\mathit{enable} \rightarrow \mathit{setup}.1!\,T_1 \rightarrow \left(\begin{array}{c} (\mathit{Disable};\ \mathit{halt}.1 \rightarrow \mathit{Skip}) \\ \Box\, (\mathit{out}.1 \rightarrow \Phi(\mathit{Running})) \end{array} \right) ;\ X \end{array}$$

The generated program contains no time information: just timer events. We define one timer for each timeout operator used in the original specification.

$$\begin{array}{rcl} \mathit{Timer}_1 & \widehat{=} & \mathit{Timer}(1, T_1) \\ \mathit{Timer}_2 & \widehat{=} & \mathit{Timer}(2, T_2) \end{array}$$

From Theorem 5.1 we know that $(\Phi(\mathit{Alarm})\ \textbf{\textit{par}}\ (\mathit{Timer}_1 \,|||\, \mathit{Timer}_2)) \equiv \mathit{Alarm}$. This equivalence is the basis for assuring that the validation step to come is correct.

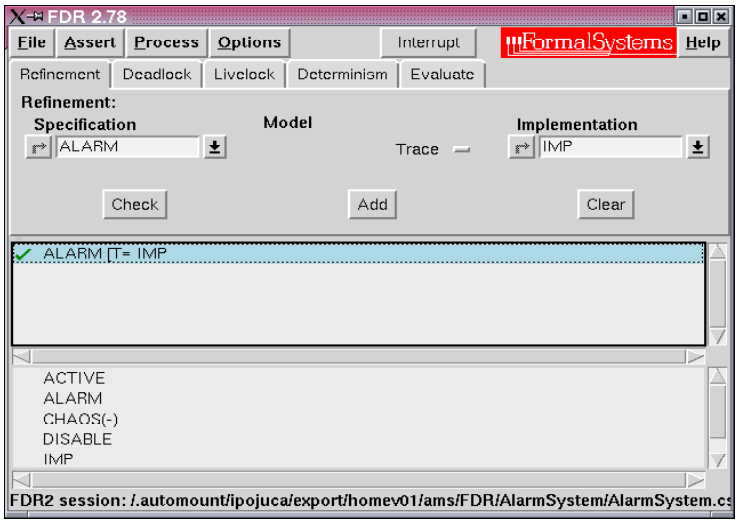

Figure 5.2. Screen dump of the FDR tool used in the validation of the alarm controller example

5.6.3 Implementation

A car alarm system is a possible implementation of the alarm controller specification given above. A car alarm system, when enabled waits for the elapse of T_1 time units before the alarm can be *disturbed*. It should assure also that after detecting a disturbance it waits for at least T_2 time units before the alarm is triggered. These requirements can be expressed in the following action *Imp*:

$$Imp \mathrel{\widehat{=}} \mu\, X \bullet enable \rightarrow Wait\ \ \mathsf{T}_1;\ disturbed \rightarrow Wait\ \ \mathsf{T}_2;\ alarm \rightarrow disable \rightarrow X$$

We can obtain an untimed version of our implementation by applying the Φ function as follows

$$\begin{aligned} \Phi(Imp) \quad = \quad & \mu\, X \bullet enable \rightarrow setup.1!T_1 \rightarrow out.1 \rightarrow disturbed \rightarrow setup.1!T_1 \rightarrow \\ & out.1 alarm \rightarrow disable \rightarrow X \end{aligned}$$

The timers defined for the above program are as follows:

$$\begin{aligned} Timer_1 \quad & \mathrel{\widehat{=}} \quad Timer(1, T_1) \\ Timer_2 \quad & \mathrel{\widehat{=}} \quad Timer(2, T_2) \end{aligned}$$

Notice that the alarm controller specification and the implementation in the normal form use the same timers; therefore we need to prove that the untimed alarm controller $L(\Phi(Imp))$ satisfies its untimed specification $L(\Phi(Alarm))$. Because both actions are in normal form, do not contain time operators, and the semantics of the time events has been given by the normal form expansion laws, we can use the CSP model checking tool FDR [20] in the verification process. Figure 5.2 shows the use of FDR, asserting that the untimed action *ALARM* is refined by the untimed action *Imp*, in the trace model.

From the results in the previous section we can then conclude that the timed specification of the alarm is refined by the implementation.

5.7 LIMITATIONS OF THE FRAMEWORK

Our framework does not handle CSP specifications in general. The limitation occurs with recursive programs that involve interleaving and the timeout operator or a similar form with time operator. We dedicate this section to study the different forms of this limitation.

The first form makes use of the timeout operator and interleaving under recursion. Consider the following action P[1]

$$P = \mu\, X \bullet ((a \rightarrow A) \stackrel{\mathsf{d}}{\rhd} B) \,|||\, X$$

First we recall from the previous chapters that the definition of interleaving in our language is given with the aid of the parallel composition operator. So the above program P is actually defined as follows

$$P = \mu\, X \bullet ((a \rightarrow A) \stackrel{\mathsf{d}}{\rhd} B) \,[\![\{\} \{\} \{\}]\!]\, X$$

Notice that the set of variables that A and B can change should be empty; such a condition is due to the limitation of the parallel merge with state sharing. The parallel merge from Chapter 3 shows clearly that the two parts engaged in the parallel composition should not share variables that can be changed. However, for the case of our example, this information is irrelevant. The behaviour of the above program should be an interleaving of several timeout actions, were some of these actions can timeout and not offer the event a while other actions can timeout and never offer the event a. When we apply the normal form function Φ to the action P we obtain the following

$$\begin{array}{ll}
& \Phi(\mu\, X \bullet ((a \rightarrow A) \stackrel{\mathsf{d}}{\rhd} B) \,[\![\{\} \mid \{\!| \{\} |\!\} \mid \{\}]\!]\, X) \\
= & [5.2.11] \\
& \mu\, X \bullet \Phi(((a \rightarrow A) \stackrel{\mathsf{d}}{\rhd} B) \,[\![\{\} \mid \{\!| \{\} |\!\} \mid \{\}]\!]\, X) \\
= & [5.2.4] \\
& \mu\, X \bullet \Phi(((a \rightarrow A) \stackrel{\mathsf{d}}{\rhd} B)) \,[\![\{\} \mid \{\!| \{\} |\!\} \mid \{\}]\!]^{nf}\, \Phi(X) \\
= & [5.2.12] \\
& \mu\, X \bullet \Phi(((a \rightarrow A) \stackrel{\mathsf{d}}{\rhd} B)) \,[\![\{\} \mid \{\!| \{\} |\!\} \mid \{\}]\!]^{nf}\, X \\
= & [5.2.4 \text{ and } 5.2.7] \\
& \mu\, X \bullet \left(\begin{array}{ll} setup.i.d \rightarrow & (a \rightarrow halt.i \rightarrow \Phi(A)) \\ & \Box \\ & (out.i.d \rightarrow \Phi(B)) \end{array} \right) [\![\{\} \mid \{\!| \{\} |\!\} \mid \{\}]\!]^{nf}\, X
\end{array}$$

Notice that the unfolding of the above recursion would result in an interleaving of

[1] We thank Prof. Jim Woodcock for pointing out this example.

several timer events.

$$
\begin{array}{l}
\left(\begin{array}{ll} setup.i.d \rightarrow & (a \rightarrow halt.i \rightarrow \Phi(A)) \\ & \Box \\ & (out.i.d \rightarrow \Phi(B)) \end{array}\right) \\
\qquad\qquad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}\ ]\!]^{nf} \\
\left(\begin{array}{ll} setup.i.d \rightarrow & (a \rightarrow halt.i \rightarrow \Phi(A)) \\ & \Box \\ & (out.i.d \rightarrow \Phi(B)) \end{array}\right) \\
\qquad\qquad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}\ ]\!]^{nf} \\
\left(\begin{array}{ll} setup.i.d \rightarrow & (a \rightarrow halt.i \rightarrow \Phi(A)) \\ & \Box \\ & (out.i.d \rightarrow \Phi(B)) \end{array}\right) \\
\qquad\qquad\qquad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}]\!]^{nf} \\
\qquad\qquad\qquad\qquad\qquad \cdots\cdots \\
\qquad\qquad\qquad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}]\!]^{nf} \\
\mu\, X \bullet \left(\begin{array}{ll} setup.i.d \rightarrow & (a \rightarrow halt.i \rightarrow \Phi(A)) \\ & \Box \\ & (out.i.d \rightarrow \Phi(B)) \end{array}\right) \quad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}\ ]\!]^{nf}\ X
\end{array}
$$

Now recall from Theorem 5.1, putting the above action in parallel with a timer we should obtain the same behaviour as the program P.

$$
\begin{array}{l}
\left(\begin{array}{ll} setup.i.d \rightarrow & (a \rightarrow halt.i \rightarrow \Phi(A)) \\ & \Box \\ & (out.i.d \rightarrow \Phi(B)) \end{array}\right) \\
\qquad\qquad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}\ ]\!]^{nf} \\
\left(\begin{array}{ll} setup.i.d \rightarrow & (a \rightarrow halt.i \rightarrow \Phi(A)) \\ & \Box \\ & (out.i.d \rightarrow \Phi(B)) \end{array}\right) \\
\qquad\qquad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}\ ]\!]^{nf} \\
\left(\begin{array}{ll} setup.i.d \rightarrow & (a \rightarrow halt.i \rightarrow \Phi(A)) \\ & \Box \\ & (out.i.d \rightarrow \Phi(B)) \end{array}\right) \\
\qquad\qquad\qquad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}]\!]^{nf} \\
\qquad\qquad\qquad\qquad\qquad \cdots\cdots \\
\qquad\qquad\qquad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}]\!]^{nf} \\
\mu\, X \bullet \left(\begin{array}{ll} setup.i.d \rightarrow & (a \rightarrow halt.i \rightarrow \Phi(A)) \\ & \Box \\ & (out.i.d \rightarrow \Phi(B)) \end{array}\right) \quad [\![\{\} \mid \{\!|\ \{\}\ |\!\} \mid \{\}\ ]\!]^{nf}\ X \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{par}\ \ Timer(i, d)
\end{array}
$$

It is clear from the above expanded process that it would require an infinite number of timers (one timer for each process result of unfolding the recursion). Nevertheless, our normal form provides a single time.

We also notice that the above situation has another variation which is in the form

$$P = \mu\, X \bullet ((a \rightarrow A) \,\Box\, (\mathit{Wait}\ \ \mathsf{d};\ B)) \,[\![\, \{\}\{\}\{\} \,]\!]\, X$$

A possible solution to the above problem is to use indexed timers in the unfolding of the recursive action. We leave the solution of this problem as future work.

5.8 CONCLUDING REMARKS

In this chapter we present a framework for the validation of real time properties using the untimed model. The framework is based on the principle of dividing a real time action into an untimed action and a collection of interleaving timers; the same principle is applied to requirements. We then use the untimed version and the untimed tools to validate the desired properties. The correctness and soundness of this method was explored in details. We also point out the limitations of our framework.

CHAPTER 6

CONCLUSION

As a general contribution of this work we presented a proposal for extending the Unifying Theories of Programming (UTP) by adding discrete time. The new model is presented in comparison with the original UTP model; this is done with the aid of a new language: *Circus* Time Action. This language is a subset of *Circus*, which itself integrates Z paragraphs, CSP processes, specification statements and commands, and uses the UTP as its semantic model. We extend *Circus* by adding two time operators: $Wait\ \mathsf{d}$ and Timeout; we give the semantics of all operators in the new time model. New healthiness conditions are proposed and related to the original UTP healthiness conditions for reactive programs and CSP processes. We explore the semantic relation between the two models with the objective of using such a mapping in relating programs and specifications in both languages. Finally we propose a framework for the specification and validation of real time systems using *Circus* Time Action as specification notation. A small case study was used to validate the framework.

In the current chapter we summarize the main contributions of this work, in the context of related work, and propose some topics for future work.

6.1 RELEVANCE AND RESULTS

Some considerations and contributions of our work are detailed in the following.

- A Discrete Time Model for Unifying Theories of Programming. The Unifying Theories of Programming proposed a model for reactive and concurrent programs and specification languages. The UTP ruled out the real time programming languages from the unification proposal due to the complexity of such formalisms. We proposed a discrete model for real time programming languages; the proposed model follows the same principles of the UTP. This new model serves as a semantic model for the definition of real time programming languages in the UTP. The healthiness conditions for reactive programs and CSP processes are rewritten in the new model; *Circus* Time Action is defined using the new time model. Our proposed model has inspired others to define languages in the UTP new model, such as Ri Hoyn Sul and He Jifeng in [54], where the semantics for Timed RSL is given using our timed model. The model has also been a source of inspiration for the development of new models for timed systems, like the one proposed by Qin Shengchao, Jin Song Dong and Wei-Ngan Chin in [42], where the authors propose an extension to our time model by adding observation on sensor-actuator variables along with the time traces; the model was used to give semantics of a subset of TCOZ [33]. An interesting aspect of that work is the definition of two additional time operators, namely *deadline* and *waituntil*; the first is used to define a maximum execution time, where $P \bullet Deadline\ t$ imposes a time constraint on the process P,

which requires it to terminate within time t. The second operator $P \bullet Waituntil\ t$ behaves as follows: if process P terminates within time t, then the program waits until t time units have passed; otherwise, it behaves as P.

- The relation between timed and untimed models. In Chapter 4 we have explored the relation between the proposed time model and the UTP model. The relation involved an abstraction function L that maps predicates in the time model to predicates in the untimed one; we also introduced an inverse relation R. The mapping L and the inverse mapping R were proven to form a *Galois Connection*. We show that the abstract function L can be used to validate time insensitive safety properties. The first mapping L introduces an undesired deadlock state and, therefore, it cannot be used to validate liveness properties. For such properties, a new mapping $\hat{L}$ is introduced. The new mapping eliminates the deadlock state introduced by the mapping L. We explored the use of the function $\hat{L}$ in the validation of liveness properties. However, a new limitation arises in the definition of $\hat{L}$ as it does not distribute over parallel composition. The abstract mapping functions have an important role in the definition of the validation framework.

- The validation framework. In the previous chapter a framework for the validation of real time systems was presented. The framework is based on timers and eliminates the need for special time operators. The method requires using the timed languages to express the program behaviour and then use a normal form to obtain an untimed program with timer events. To ensure the correctness of the framework we have a theorem which states that the time abstraction mapping L, when applied to any time program, yields an untimed behaviour equivalent to the normal form program without the timer events. This guarantees that the untimed behaviour of the original program is unchanged by introducing the timer events. Furthermore, reducing the reasoning from a timed to an untimed model has allowed us to use FDR for mechanical analysis. Another theorem was presented to assure the semantic equivalence between the normal form action (when in parallel with the respective timers) and the original timed action.

6.2 RELATED WORK

The work presented in this thesis is related to various general research areas. This section describes and provides a short characterization of related work.

6.2.1 Unifying Theories of Programming

The UTP idea is fruit of several years of research and studies at Oxford University. The result of these studies is a predicate based semantics for reactive and concurrent systems. The unifying theory is presented in [26]. More recently, Jim Woodcock and Ana Cavalcanti presented in [5] a tutorial detailing proofs and showing the unification process of the theory of design and that of reactive processes to form a theory for CSP. The authors also contribute by relating the theory of CSP in the UTP to the Failure divergence model of CSP. Our work explores an extension to the UTP to support discrete

time reactive concurrent real time systems. In [54], Ri Hyon Sul and He Jifeng use the time model proposed here to give the semantics of a time version of the RAISE Specification Language namely Timed RSL. An interesting contribution was to define an interlock composition operator, which is similar to a parallel composition, but differs in the interaction of the processes with the environment. Another relevant contribution was the use of the strongest fixed point instead of the weakest fixed point. The strongest fixed point is more suitable for reasoning about programs, but it is not implementable; this is the main reason we adopted the weakest fixed point instead. An extension to the time model we proposed was presented by Shengchao Qin, Jin Song Dong and Wei-Ngan Chin in [42]. The extension adds new variables to the model as well as new time operators, as explained in the previous section.

The UTP has been an inspiration for researchers in several places all over the world. The UTP is used to study the behaviour and explore the capacities of different paradigms and languages. In [23] a relational model based on the UTP was given for an object oriented language OOL. The model explores a rich variety of features including subtypes, visibility, inheritance, dynamic binding and polymorphism. Based on the design calculus of the UTP the authors extend the concept of refinement to define both structural and behaviour refinement for OOL; several refinement laws are presented. Based on the OOL semantic model and on the refinement calculus, Jing Yang, Quan Long, Zhiming Liu and Xiaoshan Li present in [29] a predicative semantics for integrating UML models. The proposed semantics is used in the presentation of sequential diagrams and class diagrams of sequential software systems. The refinement process presented in [23] is used to preserve the consistency and the behaviour of the system. More recently, Thiago Santos, Ana Cavalcanti and Augusto Sampaio in [48] present a model for modelling object oriented features in UTP. The main contribution of their work is to break a class declaration in several constructs and to address each language feature in isolation, dealing with dynamic binding and mutual recursion.

Another recent UTP extension is the work presented by Xinbei in [56, 55], where a denotational semantic model for mobile processes is presented. The mobile processes are interpreted as assignment or communication of high order variables. The high order variable values are process constants or parameterized processes. A new set of healthiness conditions for mobile processes is presented.

6.2.2 Time Specification and Validation

Schneider [49] presents a theorem for timewise refinement based on a refinement notation that relates the Timed CSP programs to the untimed CSP specification. He proposes a stepwise development process, where the early steps of the development of a real time system are untimed programs that describe the behaviour of the system without any time constraints. In subsequent steps, time is introduced and the timewise refinement relation can be used to ensure that the correctness with respect to the untimed properties is preserved. In our approach we can adopt a similar stepwise refinement using the mapping functions. A program P can be defined initially without any time operators and, in this case we classify such a program as a *Time insensitive* program, see Definition 4.1; time information can be added resulting in a new version P'. The

timewise refinement states that the program preserves the untimed behaviour; therefore, we use $L(P') \Rightarrow P$ to assure this and then we say that P' is a timewise refinement of P.

Ouaknine [41] developed an algorithm for transforming a dense time model of Timed CSP into a discrete time model that is based on the CSP untimed semantic models and the *tock* event. The event *tock* represents the time passage and has a special semantics; to introduce the semantics of *tock* a new external choice operator is defined. Standard methods for model checking have been applied with the aid of the CSP model checking tool FDR [20]. The use of the *tock* event was also introduced by Schnieder [49], and has the drawback that the time passage is registered by the occurrence of the event *tock*. Therefore a command $Wait\ 3$ is represented as $tock \rightarrow tock \rightarrow tock \rightarrow Skip$; a denotational semantics was given and new definitions for communication, external choice operators and parallel composition were needed. In our approach we first give a semantic representation for the time version of the program and then present a normal form where time operators are interpreted by timer events. The advantage of our model is that we only mark the beginning and end of the wait period with timer events without considering the time passage in between; this reduces the number of possible states in the verification and makes the use of a tool such as FDR much more natural and efficient.

The idea of time specification decomposition into untimed specification and timers was used by Meyer in [35]. Meyer uses the strategy to decompose a Timed CSP specification into an untimed specification in CSP and a collection of Timers. In his approach problems similar to the ones presented in our work are found and solved in a very similar way to the solutions given in our approach. The focus of Meyer's decomposition approach is on its application within the RT-Tester test system, with the objective of generating test cases, test execution and test evaluation. The timer used by Meyer does not consider two aspects that are treated in our timer. First, similar to the timer presented in this work, the timer presented by Meyer stats with a *set* event, equivalent to the *setup* event used in our approach, and the *eld* event, which is similar to the *out* event in the timer defined in Chapter 5. However, his timer lacks the *halt* event which registers that the timer did not reach termination; this is important in the interpretation of the time out and in the validation of the time properties. Because the main concern in the approach used by Meyer is the extraction of test cases, no consideration was made as to the termination of the timers; such considerations are made in our timer, where the *terminate* event is added. The base contribution in this thesis is to extend the structural decomposition approach proposed by Meyer [35] and apply it to the CSP model checking techniques with the aid of a model checking tool such as FDR [20].

6.3 FUTURE WORK

This work has a great deal of opportunities for future research as it reveals some new concepts to the UTP. The following is a list of some topics for future work.

6.3.1 Case study

We plan to apply the framework to a more elaborate case study with different types of real time properties. We consider using the Railway Crossing problem as a case study.

6.3.2 New real time language constructs

In this work we present the semantics of the UTP operators in the time model and we extend them by adding two new time operators, *Wait* and *Timeout*. Real Time programming languages may use other constructs such as TCOZ *Deadline* [33] and Timed CSP *Timed Interrupt* [10]. A future research would be to explore the other time operators in the time model. Some of these operators can be added to the model without any additional constraints; others, however, would require some simple adaptations of the model presented in this work.

6.3.3 Include processes and specification statements

We used a subset of the *Circus* language to show the model extension; this subset, denoted as *Circus* Time Action, includes only actions and some commands of *Circus*. This subset was chosen as close as possible to the UTP model. We hope to extend this subset with the other constructs of the language and add some time operators for processes.

6.3.4 Extending the time model

In our approach we used a discrete time model; it is an interesting future work to explore a continuous time model. A simple extension could be to register the time in which the observation is made instead of registering only the occurrences of an element in the sequence. The traces tr_t and tr'_t could be defined as sequences of elements composed of three entries, as follows:

$$tr_t, tr'_t : \operatorname{seq}^+((\operatorname{seq} Event \times \mathbb{P}\, Event) \times \mathbb{R})$$

where $\mathbb{R}$ is the set of real numbers used to represent time instead of the sequence index. New healthiness conditions should be defined. An interesting result is that the validation framework can be used for any of the model extensions, provided that a semantic mapping can be found between the new model and the model used by our framework.

6.3.5 Refinement calculus

In *Circus* much work has been done on the elaboration of refinement laws [47, 4]. We hope that our work serves as an inspiration to others to extend the *Circus* refinement laws to include both time and untimed programs. It is interesting to see that most algebraic properties of untimed programs are preserved in the timed model; such care was taken in the elaboration of this work to make it possible for future extensions.

6.4 FINAL REMARKS

In this chapter we presented an overview of the challenges involved in this work and the efforts to face them. We also considered future work that can be conducted as continuation of this work. Although we concentrated on the subset of *Circus*, we hope to have contributed by generating a more complete set of tools to deal with the

Circus specification language. We also hope to have contributed to formal specification and validation using *Circus* by proposing a framework that can be used to validate time requirements without explicit time operators, with the aid of the FDR tool.

BIBLIOGRAPHY

[1] A. Bernstein and P.K. Harter. Proving Real-time Properties of Programs with Temporal Logic. In *Proceedings 8thSymposium on Operating System Principles, ACM SIGOPS*, pages 1–11, 1981.

[2] J. P. Bowen and M. G. Hinchey. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 8–16, New York, NY, USA, 2005. ACM Press.

[3] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[4] A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for Circus. *Formal Aspects of Computing*, 15(2-3):146–181, 2003.

[5] A Cavalcanti and J. Woodcock. A Tutorial Introduction to CSP in the Unifying Theories of Programming. In *Pernambuco Schoool of Software Engineering:Refinement, Recife - Pernambuco - Brazil, 23rd of November to 5th of December, 2004*, 2004.

[6] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A Calculus of Duration. *Information Processing Letters.*, 40:269–276, 1991.

[7] B. F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, 1999.

[8] L. Chen. *Timed Processes: Models, Axioms and Decidability.* PhD thesis, The University of Edinburgh, Department of Computer Science, 1993.

[9] D. Craigen, S. Gerhart, and T. Ralston. An International Survey of Industrial Applications of Formal Methods. Technical Report NISTGCR 93/626, U.S. DEPARTMENT OF COMMERCE, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, March 1993.

[10] J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.

[11] R. de Lemos and J. Hall. Ertl: An extension to rtl for requirements analysis for hybrid systems, 1995.

[12] J. S. Dong, P. Hao, S. Qin, J. Sun, and Y. Wang. Timed Patterns: TCOZ to Timed Automata. In *ICFEM 2004*, LNCS. Springer-Verlag, 2004.

[13] R. Duke and G. Smith. Temporal logic and z specifications. *Australian Computer Journal*, 21(2):62–66, 1989.

[14] A. R. Paula et al. Scientific Satellite Computer Subsystem Specification. Technical report, The Brazilian Space Research Institute - INPE, 1995.

[15] A. S. Evans. Visualising Concurrent Z Specifications. In J. P. Bowen and J. A. Hall, editor, *Z User Workshop*, pages 269–281, Cambridge, June 1994. Proceedings of the 8th Z User Meeting, Springer-Verlag Workshop in Computing.

[16] A.S. Evans, D.R.W. Holton, L.M. Lai, and P. Watson. A comparison of real-time formal specification languages. In D.J. Duke and A.S. Evans, editors, *Proceedings of the Northern Formal Methods Workshop, Ilkley, September 1996*, Electronic Workshops in Computer Science, http://ewic.springer.co.uk/, January 1997. Springer Verlag.

[17] A. Farias, A. Mota, and A. Sampaio. Efficient csp-z data abstraction. In Boiten, Derrick, and Smith, editors, *IFM 2004*, volume 2999 of *LNCS*, pages 108–127. Springer-Verlag, 2004.

[18] C. Fischer. Combining CSP and Z. Technical report, University of Oldenburg, 1996.

[19] C. Fischer. *Combination and implementation of processes and data: from csp-oz to java*. PhD thesis, University of Oldenburg, 2000.

[20] Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 2.01*, August 1996.

[21] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A general way to put time in petri nets. In *Proceedings of the Fifth International workshop on Software Specification and Design*, pages 60–67, Pittusburgh, Pennsylvania, USA, 1989. Sponsored by IEEE Computer Society and ACM.

[22] C. Ghezzi and M. Pezze. Cabernet: an environment for the specification and verification of real-time systems. In *In Proceedings of 1992 DECUS Europe Symposium, Cannes (F)*, 1992.

[23] J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. In *To appear in proceedings of the Second ASIAN Symposium on Programming Languages and Systems (APLAS04), Taipei, Taiwan, 4-6 Nov. 2004*. Springer-Verlag, 2004.

[24] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21, 1978.

[25] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[26] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall Series in Computer Science, 1998.

[27] F. Jahanian, A. K. Mok, and D. A. Stuart. Formal specification of real-time systems. Technical Report TR-88-25, Department of Computer Science, University of Texas at Austin, June 1988.

[28] H. Jifeng and V. Verbovskiy. Integrating CSP and DC. R 248, International Institute for Software Technology, The United Nation University, P.O. Box 3058, Macau, January 2002.

[29] Y. Jing, L. Quan, X. Li, and Z. Liu. A Predicative Semantic Model for Integrating UML Models. R 308, International Institute for Software Technology, The United Nation University, P.O. Box 3058, Macau, March 2003. To appear in the proceedings of the 1st International Colloquium on Theoretical Aspects of Computing (ICTAC), September 20-24, 2004 Guiyang, China.

[30] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[31] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1997.

[32] L. Li and H. Jifeng. A Denotational Semantics of Timed RSL using Duration Calculus. R 168, International Institute for Software Technology, The United Nation University, P.O. Box 3058 Macau, July 1999.

[33] B. Mahony and J. Song Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *Proceedings of the 1998 Internaltional Conference on Software Engineering*, pages 95–104, 1998.

[34] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems — Specification*. Springer-Verlag, 1991.

[35] O. Meyer. Structural Decomposition of Timed CSP and its Application in Real-Time Testing. Master's thesis, University of Bremen, July 2001.

[36] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[37] R. Milner. *Communicating Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[38] C. Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, 2nd edition, 1994.

[39] A. Mota and A. Sampaio. Model-Checking CSP-Z, Strategy, Tool Support and Industrial Application. *Science of Computer Programing*, 39(1), 2001.

[40] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[41] J. Ouaknine. *Discrete analysis of continuous behaviour in real-time concurrent systems*. PhD thesis, Oxford University, 2001.

[42] S. Qin, J. S. Dong, and W. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In Araki, Gnesi, and Mandrioli, editors, *FME 2003*, volume 2805 of *LNCS*, pages 321–340. Springer-Verlag, 2003.

[43] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *Proceedings of ICALP '86*, volume 226. Lecture Notes in Computer Science, 1986.

[44] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[45] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, 1998.

[46] S. Silva. *CABERNET user manual Edition 1*, may 1994.

[47] A. Sampaio, J. Woodcock, and A. Cavalcanti. Refinement in Circus. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer, 2002.

[48] T. Santos, A. Cavalcanti, and A. Sampaio. Objetc-orientation in the UTP. In *To appear in proceedings of International Symposium on Unifying Theories of Programming, Darlington - UK, 5th to 7th of February, 2006.*

[49] S. Schneider. *Concurrent and Real-time Systems: The CSP approach*. Wiley, 2000.

[50] A. Sherif. Formal Specification and Validation of Real-Time Systems. Master's thesis, Centro de Informática, UFPE, 2000. http://www.di.ufpe.br/~ams/tese.ps.gz.

[51] A. Sherif, H. Jifeng, A. Cavalcanti, and A. Sampaio. A framework for specification and validation of real-time systems using *Circus* actions. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004: First International Colloquium,Guiyang, China, September 20-24, 2004*, volume 3407, pages 478–493. Springer-Verlag GmbH, 2005.

[52] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, 2000.

[53] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge Tracts in Theoretical Computer Science 3, 1988.

[54] R. Hyon Sul and H. Jifeng. A Complete Verification System for Timed RSL. R 275, International Institute for Software Technology, The United Nation University, P.O. Box 3058, Macau, March 2003.

[55] X. Tang and J. Woodcock. Towards mobile processes in unifying theories. In Jorge Cuellar and Zhiming Liu, editors, *SEFM2004: the 2nd IEEE International Conference on Software Engineering and Formal Methods*, Beijing, China, September 2004. IEEE Computer Society Press. To appear.

[56] X. Tang and J. Woodcock. Travelling processes. In Dexter Kozen, editor, *MPC2004: The 7th International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 381–399, Stirling, Scotland, UK, July 2004. Springer-Verlag,.

[57] J. Woodcock and A. Cavalcanti. Circus: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, July 2001.

[58] J. Woodcock and A. Cavalcanti. The steam boiler in a unified theory of Z and CSP. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, 2001.

[59] J. Woodcock and A. Cavalcanti. The Semantics of Circus. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002, Proceedings*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.

APPENDIX A

NOTATION AND BASIC CONCEPTS

In this appendix we show the basic concepts and notations used throughout the thesis. We start by giving an overview of the notation of sequences and tuples and their related operators which are used in the definition of the semantics of the language. Next we give a definition for time traces and explore the properties specific to his type of sequence.

A.1 SETS

The concept of set is fundamental to mathematics and computer science. A set is a collection of objects or a container of objects. An object(element) x belongs to a set A is symbolically represented by $x \in A$. A set can be described in a number of different ways. The simplest is to list up all of its members if that is possible. For example $\{1, 2, 3\}$ is the set of three numbers 1, 2, and 3. A set can also be described by listing the properties that its members must satisfy. For example, $\{x \mid 1 < x < 2$ and x is a real number. $\}$ represents the set of real numbers between 1 and 2, and $\{x \mid x$ is the square of an integer and $x < 100\}$ represents the set $\{0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100\}$. If a set A has n distinct elements for some natural number n, n is the cardinality (size) of A and A is a finite set. The cardinality of A is denoted by $\mid A \mid$. A set which has no elements is called an empty set. An empty set, denoted by $\emptyset$, is a set that satisfies the following: $\forall\, x \bullet x \notin \emptyset$, where $\notin$ stands for element *not in* a set.

A.1.1 Equality and subsets

Two sets are equal if and only if they have the same elements. More formally, for any sets A and B, $A = B$ if and only if $\forall\, x \bullet x \in A \Leftrightarrow x \in B$. A set A is a subset of a set B if and only if every element in A is also in B. More formally, for any sets A and B, A is a subset of B, and denoted by $A \subseteq B$, if and only if $\forall\, x \bullet x \in A \Rightarrow x \in B$. If $A \subseteq B$, and $A \neq B$, then A is said to be a proper subset of B and it is denoted by $A \subset B$.

A.1.2 Operations on sets

The *union* of sets A and B, denoted by $A \cup B$, is the set defined as

$$A \cup B = \{x \mid x \in A \vee x \in B\} \tag{A.1.0}$$

The *intersection* of sets A and B, denoted by $A \cap B$, is the set defined as

$$A \cap B = \{x \mid x \in A \wedge x \in B\} \tag{A.1.0}$$

The *difference* of sets A from B, denoted by $A - B$, is the set defined as

$$A - B = \{x \mid x \in A \wedge x \notin B\} \tag{A.1.0}$$

A.1.3 Ordered pair

An ordered pair (*tuple*) is a pair of objects with an order associated with them. If objects are represented by x and y, then we write the ordered pair as (x, y). Two ordered pairs (a, b) and (c, d) are equal if and only if $a = c$ and $b = d$. The operation *fst* returns the first element of a pair $fst((a, b)) = a$ and the operation *snd* returns the second element of the pair $snd((a, b)) = b$. The cartesian product of two sets A and B, denoted by $A \times B$ is the set of all the pairs (a, b) such that $a \in A$ and $b \in B$.

A.2 FUNCTION

A function is something that associates each element of a set with an element of another set (which may or may not be the same as the first set). Function, f , from a set A to a set B is a relation from A to B denoted as $f : A \rightarrow B$ and for each element a in A, there is an element b in B, such that (a, b) is in the relation f. The set A in the definition is called the *domain* of the function f and the set B its *codomain* or *range*. A function is also denoted as a set of pairs where each element of the set is a member the function relation, consider the following example

$$\begin{aligned} A &= \{1, 2, 3\} \\ B &= \{x, y, z\} \\ f &= \{(A \times B) \mid 1 \mapsto x, 2 \mapsto y, 3 \mapsto z\} \end{aligned}$$

In the above example f is defined as a function from A to B where each element of A is mapped to an element in B, the elements of the set are called mappings and they show the relation f over the sets A and B. The term $1 \mapsto x$ is the analogue for the pair $(1, x)$. We denote a single pair (a, b) of a function f as a mapping from the element a of the domain of f to the element b of the range of f. The term $f(a)$ is used to denote the member of B to which f maps a (in A). Function composition $f \circ g$ relates to functions f and g such that the domain of f is the range of g it is defined as $f \circ g(x) = f(g(x))$. The function override operator $\oplus$ is used to change the value of an entry in a function, consider the following example

$$\begin{aligned} A &= \{1, 2, 3\} \\ B &= \{x, y, z\} \\ f &= \{(A \times B) \mid 1 \mapsto x, 2 \mapsto y, 3 \mapsto z\} \\ g &= f \oplus \{1 \mapsto y\} \end{aligned}$$

where the both function f and g are mappings from the set A to the set B. The function g is defined as the function f except for the mapping $1 \mapsto x$ is replaced by the mapping $1 \mapsto y$

A.3 SEQUENCE

Sequences can either be finite or infinite. We will primarily write $s = \langle a_1, a_2, a_3 ... ak \rangle$ to denote the sequence s of length k containing the elements a_1, a_2, a_3 .... a_k in the

order in which they appear. We also use the notation $s(i)$ to denote the element at the position i, where $i \in \{1..k\}$. The empty sequence is written as <>.

A.3.1 Concatenation

Let $u = \langle a_1, a_2, a_3...a_k \rangle$ and $v = \langle b_1, b_2, b_3...b_{k'} \rangle$. Their *concatenation* $u \frown v$ is the sequence $\langle a_1, a_2, a_3...a_k, b_1, b_2, b_3...b_{k'} \rangle$ if u is infinite sequence we let $u \frown v = u$. The most important property of concatenation is that the empty trace $\langle\rangle$ is its unit.

Property A.1

L1. $s \frown \langle\rangle = s = \langle\rangle \frown s$

L2. $s \frown (t \frown u) = (s \frown t) \frown u$

L3. $s \frown u = s \frown t \Leftrightarrow t = u$

L4. $s \frown t = u \frown t \Leftrightarrow s = u$

L5. $s \frown t = \langle\rangle \Leftrightarrow s = \langle\rangle \wedge t = \langle\rangle$

A.3.2 Head, tail, front and last

The operator *head* applied to a sequence returns the first element of the sequence; while the operation *tail* when applied to a sequence returns the original sequence without the first element, both operations are defined as follows:

$$head(\langle e \rangle \frown u) = e \tag{A.3.1}$$

$$tail(\langle e \rangle \frown u) = u \tag{A.3.2}$$

The operation *front* applied to a sequence u returns a subsequence of u obtained by eliminating the last element of the sequence. While the operation *last* when applied to a sequence returns the last element of the sequence, both operations are defined as follows:

$$front(u \frown \langle e \rangle) = u \tag{A.3.3}$$

$$last(u \frown \langle e \rangle) = e \tag{A.3.4}$$

For the above sequence operators, the following properties hold.

Property A.2

L1. $head(\langle e \rangle) = last(\langle e \rangle) = e$

L2. $tail(\langle e \rangle) = front(\langle e \rangle) = \langle\rangle$

L3. $s \neq \langle\rangle \Rightarrow head(s \frown t) = head(s) \wedge tail(s \frown t) = tail(s) \frown t$

L4. $t \neq \langle\rangle \Rightarrow last(s \frown t) = last(t) \wedge front(s \frown t) = s \frown front(t)$

L5. $s \neq \langle\rangle \Rightarrow \langle head(s)\rangle \frown (tail(s)) = s \wedge (front(s)) \frown \langle last(s)\rangle = s$

L6. $s = t \equiv (s = t = \langle\rangle \vee (head(s) = head(t) \wedge tail(s) = tail(t)))$

A.3.3 Restriction

The expression $(s \downarrow A)$ denotes the sequence s when restricted to symbols in the set A; it is formed from s simply by omitting all symbols outside A. For example

$$\langle a, b, c, d, a, c\rangle \downarrow a, c = \langle a, c, a, c\rangle$$

Restriction satisfies the following properties

Property A.3

L1. $\langle\rangle \downarrow A = \langle\rangle$

L2. $(s \frown t) \downarrow A = (s \downarrow A) \frown (t \downarrow A)$

L3. $\langle e\rangle \downarrow A = \langle e\rangle$ *if* $e \in A$

L4. $\langle e\rangle \downarrow A = \langle\rangle$ *if* $e \notin A$

L5. $s \downarrow \ = \langle\rangle$

L6. $(s \downarrow A) \downarrow B = s \downarrow (A \cap B)$

From laws *L1*, *L2*, *L3* and *L4* we observe the restriction operator is distributive and with the definition of the operator behaviour on singleton sequences the behaviour of the operator can be deduced for longer sequences. Law *L5* shows that a sequence restriction on an empty set leaves nothing. A successive restriction by two sets in the same as a single restriction by intersection of the two sets, as shown in *L6*. With the aid of the restriction operator and the length operator we define the operator $\upharpoonright$. $s \upharpoonright e$ returns the number of occurrences of the element e in the sequence s.

$$s \upharpoonright e = \#(s \downarrow e)$$

A.3.4 Ordering

We define a prefix relation: given two sequences u and v, then u is a prefix of v $(u \leq v)$ if there can be found a sequence o such that $v = u \frown o$.

$$(u \leq v) \Leftrightarrow \exists\, o \bullet v = (u \frown o) \tag{A.3.4}$$

The following are the properties of sequence prefix

Property A.4

L1. $\langle\rangle \leq s$

L2. $s \leq s$

L3. $s \leq t \wedge t \leq s \Rightarrow s = t$

L4. $s \leq t \wedge t \leq u \Rightarrow s \leq u$

L5. $(\langle e\rangle \frown s) \leq t \equiv t \neq \langle\rangle \wedge e = head(t) \wedge s \leq tail(t)$

L6. $s \leq u \wedge t \leq u \Rightarrow s \leq t \vee t \leq s$

L7. $(\langle e\rangle \frown s)\mathbf{in}t \equiv ((\langle e\rangle \frown s) \leq t) \vee ((t \neq \langle\rangle) \wedge ((\langle e\rangle \frown s)\mathbf{in}tail(t))$

L8. $s \leq t \Rightarrow s \downarrow A \leq t \downarrow A$

L9. $t \leq u \Rightarrow (s \frown t) \leq (s \frown u)$

The $\leq$ is a partial ordering, and its least element is $\langle\rangle$, as stated in laws *L1* to *L4*. The law *L5* gives a method for computing wether $s \leq t$ or not. The prefixing of a given sequence of are totally ordered: this is shown in law *L6*. If s is a subsequence of t (not necessary initial), we say that s is **in** t; this may be defined as $s\mathbf{in}t \mathrel{\widehat{=}} (\exists\, u, v \bullet t = u \frown s \frown v$. The **in** operator satesfies law *L7*. A function from sequences to sequences is said to be *monotonic* if it respects the ordering relation. Law *L8* shows that the restriction operator is monotonic, some operators are monotonic on only one argument as in the case of concatination, which is only monotonic in its second argument as stated by law *L9*.

A.3.5 difference

The sequence difference operator is defined as follows:

$$\langle\rangle - \langle\rangle = \langle\rangle \tag{A.3.5}$$
$$u - \langle\rangle = u \tag{A.3.6}$$
$$u - (u \frown v) = v \tag{A.3.7}$$

More properties of sequence

$$head(s - front(s)) = last(s) \tag{A.3.8}$$
$$tail(s - front(s)) = \langle\rangle \tag{A.3.9}$$

A.3.6 Length

We use the notation $\#s$ to represent the length of a sequence s. The Length operator is defined as follows.

$$\#\langle\rangle = 0 \# \langle e\rangle = 1 \tag{A.3.10}$$
$$\#(s \frown t) = \#s + \#t \tag{A.3.11}$$
$$\tag{A.3.12}$$

APPENDIX B

PROPERTIES OF THE HEALTHINESS CONDITIONS

In this chapter we present some algebraic properties of the healthiness conditions needed in the proofs of the algebraic properties of the time extension to *Circus*.

B.1 PROPERTIES OF $R1_T$

Property B.1

L1. $R1_t$ *is idempotent, for any action* P *the following holds* $R1_t(R1_t(P)) = R1_t(P)$
Proof:

$$
\begin{array}{ll}
R1_t(R1_t(P)) & [3.4.2] \\
= P \land Expands(tr_t, tr'_t) \land Expands(tr_t, tr'_t) & [Predicate\ Calculus] \\
= P \land Expands(tr_t, tr'_t) & [3.4.2] \\
= R1_t(P) \quad \square &
\end{array}
$$

L2. Π_t *is* $R1_t$ *healthy*
$R1_t(\Pi_t) = \Pi_t$
Proof:

$$
\begin{array}{ll}
R1_t(\Pi_t) & \\
= & [3.4.2] \\
\Pi_t \land Expands(tr_t, tr'_t) & \\
= & [3.4.9] \\
\left(\begin{array}{c} (\neg ok \land Expands(tr_t, tr'_t)) \lor \\ \left(\begin{array}{c} ok' \land (tr'_t = tr_t) \land \\ (wait' = wait) \land (state' = state) \end{array} \right) \end{array} \right) \land Expands(tr_t, tr'_t) & \\
= & [Propositional\ calculus] \\
\left(\begin{array}{c} (\neg ok \land Expands(tr_t, tr'_t) \land Expands(tr_t, tr'_t)) \lor \\ \left(\begin{array}{c} ok' \land (tr'_t = tr_t) \land \\ (wait' = wait) \land (state' = state) \\ \land Expands(tr_t, tr'_t) \end{array} \right) \end{array} \right) & \\
= & [Propositional\ calculus] \\
\left(\begin{array}{c} (\neg ok \land Expands(tr_t, tr'_t)) \lor \\ \left(\begin{array}{c} ok' \land (tr'_t = tr_t) \land \\ (wait' = wait) \land (state' = state) \\ \land Expands(tr_t, tr'_t) \end{array} \right) \end{array} \right) & [Property\ 3.2\ L3]
\end{array}
$$

$$= \left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr'_t)) \vee \\ \left(\begin{array}{c} ok' \wedge (tr'_t = tr_t) \wedge \\ (wait' = wait) \wedge (state' = state) \end{array} \right) \end{array} \right) \qquad [3.4.9]$$

$$= \Pi_t \qquad \square$$

L3. *$R1_t$ is closed over $\wedge$, provided that P and Q are healthy then*
$R1_t(P \wedge Q) = P \wedge Q$
Proof:

$R1_t(P \wedge Q)$
$=$ *[3.4.2]*
$(P \wedge Q) \wedge Expands(tr_t, tr'_t)$
$=$ *[Propositional calculus]*
$(P \wedge Expands(tr_t, tr'_t)) \wedge (Q \wedge Expands(tr_t, tr'_t))$
$=$ *[3.4.2]*
$R1_t(P) \wedge R1_t(Q)$
$=$ *[assumption and Property B.1 L1]*
$P \wedge Q$ $\square$

L4. *$R1_t$ is closed over $\vee$, provided that P and Q are healthy then*
$R1_t(P \vee Q) = P \vee Q$
Proof:

$R1_t(P \vee Q)$
$=$ *[3.4.2]*
$(P \vee Q) \wedge Expands(tr_t, tr'_t)$
$=$ *[Propositional calculus]*
$(P \wedge Expands(tr_t, tr'_t)) \vee (Q \wedge Expands(tr_t, tr'_t))$
$=$ *[3.4.2]*
$R1_t(P) \vee R1_t(Q)$
$=$ *[assumption and Property B.1 L1]*
$P \vee Q$ $\square$

L5. *$R1_t$ is closed over the conditional choice operator $\lhd\rhd$, provided that P and Q are healthy then*
$R1_t(P \lhd b \rhd Q) = P \lhd b \rhd Q$
Proof:

$R1_t(P \lhd b \rhd Q)$

$$
\begin{array}{ll}
= & \textit{[3.4.2]} \\
(P \lhd b \rhd Q) \wedge \mathit{Expands}(tr_t, tr_t') & \\
= & \textit{[3.5.22]} \\
((P \wedge b) \vee (Q \wedge \neg b)) \wedge \mathit{Expands}(tr_t, tr_t') & \\
= & \textit{[Propositional calculus]} \\
((P \wedge \mathit{Expands}(tr_t, tr_t') \wedge b) \vee (Q \wedge \mathit{Expands}(tr_t, tr_t') \wedge \neg b)) & \\
= & \textit{[3.5.22]} \\
(P \wedge \mathit{Expands}(tr_t, tr_t')) \lhd b \rhd (Q \wedge \mathit{Expands}(tr_t, tr_t')) & \\
= & \textit{[3.4.2]} \\
R1_t(P) \lhd b \rhd R1_t(Q) & \\
= & \textit{[assumption and Property B.1 L1]} \\
P \lhd b \rhd Q & \Box
\end{array}
$$

L6. *$R1_t$ is closed over sequential composition, provided that P and Q are healthy then $R1_t(P;\ Q) = P;\ Q$*
Proof:

$$
\begin{array}{ll}
R1_t(P;\ Q) & \\
= & \textit{[assumption]} \\
R1_t(R1_t(P);\ R1_t(Q)) & \\
= & \textit{[3.4.2]} \\
((P \wedge \mathit{Expands}(tr_t, tr_t'));\ (Q \wedge \mathit{Expands}(tr_t, tr_t'))) \wedge \mathit{Expands}(tr_t, tr_t') & \\
= & \textit{[3.5.21]} \\
\left(\begin{array}{l} \exists\, tr_o, v_o \bullet \\ (P \wedge \mathit{Expands}(tr_t, tr_t'))[tr_o, v_o / tr_t', v'] \wedge \\ (Q \wedge \mathit{Expands}(tr_t, tr_t'))[tr_o, v_o / tr_t, v] \end{array} \right) \wedge \mathit{Expands}(tr_t, tr_t') & \\
= & \textit{[by substitution]} \\
\left(\begin{array}{l} \exists\, tr_o, v_o \bullet \\ (P[tr_o, v_o / tr_t', v'] \wedge \mathit{Expands}(tr_t, tr_o)) \wedge \\ (Q[tr_o, v_o / tr_t, v] \wedge \mathit{Expands}(tr_o, tr_t')) \end{array} \right) \wedge \mathit{Expands}(tr_t, tr_t') & \\
= & \textit{[Property 3.2 L7]} \\
\left(\begin{array}{l} \exists\, tr_o, v_o \bullet \\ (P[tr_o, v_o / tr_t', v'] \wedge \mathit{Expands}(tr_t, tr_o)) \wedge \\ (Q[tr_o, v_o / tr_t, v] \wedge \mathit{Expands}(tr_o, tr_t')) \end{array} \right) & \\
= & \textit{[Predicate calculus]} \\
\left(\begin{array}{l} \exists\, tr_o, v_o \bullet \\ (P \wedge \mathit{Expands}(tr_t, tr_t'))[tr_o, v_o / tr_t', v'] \wedge \\ (Q \wedge \mathit{Expands}(tr_t, tr_t'))[tr_o, v_o / tr_t, v] \end{array} \right) &
\end{array}
$$

$$
\begin{array}{ll}
= & \textit{[3.4.2]} \\
\left(\begin{array}{l} \exists\, tr_o, v_o \bullet \\ (R1_t(P))[tr_o, v_o / tr'_t, v'] \wedge \\ (R1_t(Q))[tr_o, v_o / tr_t, v] \end{array} \right) & \\
= & \textit{[3.5.21]} \\
R1_t(P);\ R1_t(Q) & \\
= & \textit{[assumption and Property B.1 L1]} \\
P;\ Q & \Box
\end{array}
$$

L7. $R1_t$ *extends over conjunction*
$R1_t(P) \wedge Q = R1_t(P \wedge Q)$
Proof:

$$
\begin{array}{ll}
R1_t(P) \wedge Q & \\
= & \textit{[3.4.2]} \\
P \wedge \mathit{Expands}(tr_t, tr'_t) \wedge Q & \\
= & \textit{[Predicate calculus]} \\
P \wedge Q \wedge \mathit{Expands}(tr_t, tr'_t) & \\
= & \textit{[3.4.2]} \\
R1_t(P \wedge Q) & \Box
\end{array}
$$

L8. $R1_t(P) \wedge \neg R1_t(Q) = R1_t(P \wedge \neg Q)$
Proof:

$$
\begin{array}{ll}
R1_t(P) \wedge \neg R1_t(Q) & \\
= & \textit{[3.4.2]} \\
P \wedge \mathit{Expands}(tr_t, tr'_t) \wedge \neg(Q \wedge \mathit{Expands}(tr_t, tr'_t)) & \\
= & \textit{[De Morgans laws]} \\
P \wedge \mathit{Expands}(tr_t, tr'_t) \wedge (\neg Q \vee \neg \mathit{Expands}(tr_t, tr'_t)) & \\
= & \textit{[Predicate calculus]} \\
(P \wedge \mathit{Expands}(tr_t, tr'_t) \wedge \neg Q) \vee (P \wedge \mathit{Expands}(tr_t, tr'_t) \neg \mathit{Expands}(tr_t, tr'_t)) & \\
= & \textit{[Predicate calculus]} \\
(P \wedge \mathit{Expands}(tr_t, tr'_t) \wedge \neg Q) & \\
= & \textit{[3.4.2]} \\
R1_t(P \wedge \neg Q) & \Box
\end{array}
$$

L9. $R1_t$ *is monotonic with respect to the ordering relation* $\sqsupseteq$
Given to time programs P *and* Q *such that* $P \sqsupseteq Q$ *then*

$R1_t(P) \sqsupseteq R1_t(Q)$ ***Proof:***

$$
\begin{array}{ll}
& R1_t(P) \\
= & \textit{[3.4.2]} \\
& P \wedge \mathit{Expands}(tr_t, tr_t') \\
\sqsupseteq & \textit{[C.0.1]} \\
& Q \wedge \mathit{Expands}(tr_t, tr_t') \\
= & \textit{[3.4.2]} \\
& R1_t(P) \qquad \square
\end{array}
$$

B.2 PROPERTIES OF $R2_T$

Property B.2

L1. $R2_t$ *is idempotent*
$R2_t(R2_t(X(tr_t, tr_t'))) = R2_t(X(tr_t, tr_t'))$
Proof:

$$
\begin{array}{ll}
& R2_t(R2_t(X(tr_t, tr_t'))) \\
= & \textit{[3.4.6]} \\
& R2_t\left(\exists\, ref \bullet X \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, \mathit{dif}(tr_t', tr_t) \\ /tr_t, tr_t' \end{array}\right]\right) \\
= & \textit{[3.4.6]} \\
& \exists\, ref \bullet \left(\exists\, ref \bullet X \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, \mathit{dif}(tr_t', tr_t) \\ /tr_t, tr_t' \end{array}\right]\right) \\
& \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, \mathit{dif}(tr_t', tr_t) \\ /tr_t, tr_t' \end{array}\right] \\
= & \textit{[Substitution]} \\
& \exists\, ref \bullet (X \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, \mathit{dif}(\mathit{dif}(tr_t', tr_t), \\ \langle(\langle\rangle, ref)\rangle) \\ /tr_t, tr_t' \end{array}\right] \\
= & \textit{[Property 3.3 L2]} \\
& \exists\, ref \bullet (X[\langle(\langle\rangle, ref)\rangle, \mathit{dif}(tr_t', tr_t)/tr_t, tr_t'] \\
= & \textit{[3.4.6]} \\
& R2_t(X) \qquad \square
\end{array}
$$

L2. Π_t *is* $R2_t$ *healthy*
$R2_t(\Pi_t) = \Pi_t$
Proof:

$$R2_t(\Pi_t)$$

$$
\begin{array}{ll}
= & \textit{[3.4.11]} \\
R2_t \left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ (ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \end{array} \right) & \\
= & \textit{[3.4.6]} \\
\exists\, ref \bullet \left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ (ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \end{array} \right) \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] & \\
= & \textit{[Substitution]} \\
\exists\, ref \bullet \left(\begin{array}{l} (\neg ok \wedge Expands(\langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t))) \vee \\ \quad \left(\begin{array}{l} ok' \wedge (\langle(\langle\rangle, ref)\rangle = dif(tr_t', tr_t)) \wedge \\ (wait' = wait) \wedge (state' = state) \end{array} \right) \end{array} \right) & \\
= & \textit{[Property 3.3 L9]} \\
\exists\, ref \bullet \left(\begin{array}{l} (\neg ok \wedge Expands(\langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t)) \wedge Expands(tr_t, tr_t')) \vee \\ \qquad \left(\begin{array}{l} ok' \wedge (\langle(\langle\rangle, ref)\rangle = dif(tr_t', tr_t)) \wedge \\ (wait' = wait) \wedge (state' = state) \end{array} \right) \end{array} \right) & \\
= & \textit{[Property 3.3 L1 and 3.2 L8]} \\
\exists\, ref \bullet \left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ \left(\begin{array}{l} ok' \wedge tr_t = tr_t' \wedge \\ (wait' = wait) \wedge (state' = state) \end{array} \right) \end{array} \right) & \\
= & \textit{[Predicate calculus]} \\
\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ \left(\begin{array}{l} ok' \wedge tr_t = tr_t' \wedge \\ (wait' = wait) \wedge (state' = state) \end{array} \right) \end{array} & \\
= & \textit{[3.4.11]} \\
\Pi_t & \square
\end{array}
$$

L3. Commutativity of $R1_t$ and $R2_t$; for any program P
$R1_t(R2_t(P)) = R2_t(R1_t(P))$
Proof:

$$
\begin{array}{ll}
R2_t(R1_t(P)) & \\
= & \textit{[3.4.2]} \\
R2_t(P \wedge Expands(tr_t, tr_t')) & \\
= & \textit{[3.4.6]} \\
\exists\, ref \bullet \left(\begin{array}{l} P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] \wedge \\ Expands(\langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t)) \end{array} \right) & \\
= & \textit{[Property 3.2 L9]}
\end{array}
$$

$$\exists\, ref \bullet P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] \wedge Expands(tr_t, tr'_t)$$

$=$ *[3.4.6]*

$R2_t(P) \wedge Expands(tr_t, tr'_t)$

$=$ *[3.4.2]*

$R1_t(R2_t(P))$ □

L4. $R2_t$ is closed under $\wedge$; given any two $R2_t$ healthy time actions P and Q
$R2_t(P \wedge Q) = P \wedge Q$
Proof:

$R2_t(P \wedge Q)$

$=$ *[Assumption]*

$R2_t(R2_t(P) \wedge R2_t(Q))$

$=$ *[3.4.6]*

$$\exists\, ref \bullet \left(\begin{array}{l} \exists\, ref \bullet P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] \wedge \\ \exists\, ref \bullet Q \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] \end{array} \right) \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right]$$

$=$ *[Substitution]*

$$\exists\, ref \bullet \left(\begin{array}{l} \exists\, ref \bullet P \\ \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(dif(tr'_t, tr_t), \langle(\langle\rangle, ref)\rangle) \\ /tr_t, tr'_t \end{array} \right] \wedge \\ \exists\, ref \bullet Q \\ \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(dif(tr'_t, tr_t), \langle(\langle\rangle, ref)\rangle) \\ /tr_t, tr'_t \end{array} \right] \end{array} \right)$$

$=$ *[Property 3.3 L2]*

$$\left(\begin{array}{l} \exists\, ref \bullet P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] \wedge \\ \exists\, ref \bullet Q \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] \end{array} \right)$$

$=$ *[3.4.6]*

$R2_t(P) \wedge R2_t(Q)$

$=$ *[assumption and Property B.2 L1]*

$P \wedge Q$ □

L5. $R2_t$ is closed over $\vee$; given any two timed programs P and Q
$R2_t(P \vee Q) = P \vee Q$

Proof:

$$
\begin{array}{ll}
& R2_t(P \vee Q) \\
= & \textit{[Assumption]} \\
& R2_t(R2_t(P) \vee R2_t(Q)) \\
= & \textit{[3.4.6]} \\
& \exists\, ref \bullet \left(\begin{array}{l} \exists\, ref \bullet P \\ \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] \vee \\ \exists\, ref \bullet Q \\ \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] \end{array} \right) \\
& \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] \\
= & \textit{[Substitution]} \\
& \exists\, ref \bullet \left(\begin{array}{l} \exists\, ref \bullet P \\ \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(dif(tr_t', tr_t), \langle(\langle\rangle, ref)\rangle) \\ /tr_t, tr_t' \end{array} \right] \vee \\ \exists\, ref \bullet Q \\ \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(dif(tr_t', tr_t), \langle(\langle\rangle, ref)\rangle) \\ /tr_t, tr_t' \end{array} \right] \end{array} \right) \\
= & \textit{[Property 3.3 L2]} \\
& \left(\begin{array}{l} \exists\, ref \bullet P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] \vee \\ \exists\, ref \bullet Q \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] \end{array} \right) \\
= & \textit{[3.4.6]} \\
& R2_t(P) \vee R2_t(Q) \\
= & \textit{[assumption and Property B.2 L1]} \\
& P \vee Q \qquad \Box
\end{array}
$$

L6. *$R2_t$ is closed over $\lhd\rhd$; given any two timed program P and Q*
$R2_t(P \lhd b \rhd Q) = P \lhd b \rhd Q$
Provided that tr_t and tr_t' is free in b. We also make the assumption that P and Q are $R2_t$ healthy actions.
Proof:

$$
\begin{array}{ll}
& R2_t(P \lhd b \rhd Q) \\
= & \textit{[3.4.6]} \\
& \exists\, ref \bullet (P \lhd b \rhd Q) \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right]
\end{array}
$$

$$
\begin{array}{ll}
= & \textit{[3.5.22]} \\
\exists\, ref \bullet ((P \wedge b) \vee (Q \wedge \neg b)) \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] & \\
= & \textit{[Predicate calculus]} \\
\exists\, ref \bullet (P \wedge b) \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] \vee & \\
\exists\, ref \bullet (Q \wedge \neg b)) \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] & \\
= & \textit{[Assumption } tr_t \textit{ and } tr'_t \textit{ free in } b\textit{]} \\
\left(\exists\, ref \bullet P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right]\right) \wedge b \vee & \\
\left(\exists\, ref \bullet Q \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right]\right) \wedge \neg b & \\
= & \textit{[3.4.6]} \\
(R2_t(P) \wedge b) \vee (R2_t(Q) \wedge \neg b) & \\
= & \textit{[3.5.22]} \\
(R2_t(P) \lhd b \rhd R2_t(Q)) & \\
= & \textit{[Assumption]} \\
P \lhd b \rhd Q & \square
\end{array}
$$

L7. $R2_t$ *is closed over sequential composition; given any two* $R2_t$ *healthy timed programs* P *and* Q
$R2_t(P;\ Q) = P;\ Q$
Proof:

$$
\begin{array}{ll}
R2_t(P;\ Q) & \\
= & \textit{[Assumption]} \\
R2_t(R2_t(P);\ R2_t(Q)) & \\
= & \textit{[3.5.21]} \\
R2_t \left(\begin{array}{l} \exists\, tr_o, v_o \bullet \\ R2_t(P)[tr_o, v_o/tr'_t, v'] \wedge \\ R2_t(Q)[tr_o, v_o/tr_t, v] \end{array} \right) & \\
= & \textit{[3.4.6]}
\end{array}
$$

$$
\begin{array}{l}
\exists\, ref \bullet \\
\left(\begin{array}{l}
\exists\, tr_o, v_o \bullet \\
\left(\begin{array}{l}
\exists\, ref \bullet P \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array}\right]
\end{array}\right) \\
\left[\begin{array}{l} tr_o, v_o / tr'_t, v' \end{array}\right] \wedge \\
\left(\begin{array}{l}
\exists\, ref \bullet Q \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array}\right]
\end{array}\right) \\
\left[\begin{array}{l} tr_o, v_o / tr_t, v \end{array}\right]
\end{array}\right) \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array}\right]
\end{array}
$$

$=$ *[Susbtitution]*

$$
\begin{array}{l}
\exists\, ref \bullet \\
\left(\begin{array}{l}
\exists\, tr_o, v_o \bullet \\
\left(\begin{array}{l}
\exists\, ref \bullet P[v_o/v'] \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_o, tr_t) \\ /tr_t, tr'_t \end{array}\right]
\end{array}\right) \wedge \\
\left(\begin{array}{l}
\exists\, ref \bullet Q[v_o/v] \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_o) \\ /tr_t, tr'_t \end{array}\right]
\end{array}\right)
\end{array}\right) \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array}\right]
\end{array}
$$

$=$ *[Substitution]*

$$
\begin{array}{l}
\exists\, ref \bullet \\
\left(\begin{array}{l}
\exists\, tr_o, v_o \bullet \\
\left(\begin{array}{l}
\exists\, ref \bullet P[v_o/v'] \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_o, \langle(\langle\rangle, ref)\rangle) \\ /tr_t, tr'_t \end{array}\right]
\end{array}\right) \wedge \\
\left(\begin{array}{l}
\exists\, ref \bullet Q[v_o/v] \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(dif(tr'_t, tr_t), tr_o) \\ /tr_t, tr'_t \end{array}\right]
\end{array}\right)
\end{array}\right)
\end{array}
$$

$=$ *[Property 3.3 L2]*

$$
\begin{array}{l}
\exists\, ref \bullet \\
\left(\begin{array}{l}
\exists\, tr_o, v_o \bullet \\
\left(\begin{array}{l}
\exists\, ref \bullet P[v_o/v'] \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, tr_o \\ /tr_t, tr'_t \end{array}\right]
\end{array}\right) \wedge \\
\left(\begin{array}{l}
\exists\, ref \bullet Q[v_o/v] \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(dif(tr'_t, tr_t), tr_o) \\ /tr_t, tr'_t \end{array}\right]
\end{array}\right)
\end{array}\right)
\end{array}
$$

$=$ *[let $tr_o = dif(tr_1, tr_t)$ such that $Expands(tr_1, tr'_t) \wedge Expands(tr_t, tr_1)$]*

$$
\begin{array}{l}
\exists\, ref \bullet \\
\left(\begin{array}{l}
\exists\, tr_1, v_o \bullet \\
\left(\begin{array}{l}
\exists\, ref \bullet P[v_o/v'] \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_1, tr_t) \\ /tr_t, tr_t' \end{array}\right]
\end{array}\right) \wedge \\
\left(\begin{array}{l}
\exists\, ref \bullet Q[v_o/v] \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(dif(tr_t', tr_t), dif(tr_1, tr_t)) \\ /tr_t, tr_t' \end{array}\right]
\end{array}\right)
\end{array}\right)
\end{array}
$$

$=$ *[Property 3.3 L11]*

$$
\begin{array}{l}
\exists\, ref \bullet \\
\left(\begin{array}{l}
\exists\, tr_1, v_o \bullet \\
\left(\begin{array}{l}
\exists\, ref \bullet P[v_o/v'] \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_1, tr_t) \\ /tr_t, tr_t' \end{array}\right] \\
\exists\, ref \bullet Q[v_o/v] \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_1) \\ /tr_t, tr_t' \end{array}\right]
\end{array}\right) \wedge
\end{array}\right)
\end{array}
$$

$=$ *[Propositional calculus]*

$$
\begin{array}{l}
\left(\begin{array}{l}
\exists\, tr_1, v_o \bullet \\
\left(\begin{array}{l}
\exists\, ref \bullet P[v_o/v'] \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array}\right] [tr_1/tr_t'] \\
\exists\, ref \bullet Q[v_o/v] \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array}\right] [tr_1/tr_t]
\end{array}\right) \wedge
\end{array}\right)
\end{array}
$$

$=$ *[Predicate calculus]*

$$
\begin{array}{l}
\left(\begin{array}{l}
\exists\, tr_1, v_o \bullet \\
\left(\begin{array}{l}
\exists\, ref \bullet P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array}\right] [v_o/v'][tr_1/tr_t'] \\
\exists\, ref \bullet Q \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array}\right] [v_o/v][tr_1/tr_t]
\end{array}\right) \wedge
\end{array}\right)
\end{array}
$$

$=$ *[3.4.6]*

$$
\left(\begin{array}{l}
\exists\, tr_1, v_o \bullet \\
(R2_t(P)[v_o/v'][tr_1/tr_t']) \wedge \\
(R2_t(Q)[v_o/v][tr_1/tr_t])
\end{array}\right)
$$

$=$ *[assumption and property B.2 L1]*

$$
\left(\begin{array}{l}
\exists\, tr_1, v_o \bullet \\
P[v_o/v'][tr_1/tr_t'] \wedge \\
Q[v_o/v][tr_1/tr_t]
\end{array}\right)
$$

$=$ *[3.5.21]*

$P;\ Q$ □

L8. $R2_t$ is monotonic with respect to the ordering relation $\sqsupseteq$
Given to time actions A and B such that $A \sqsupseteq B$ then

$R2_t(A) \sqsupseteq R2_t(B)$ ***Proof:***

$$
\begin{array}{ll}
& R2_t(A) \\
= & \textit{[3.4.6]} \\
& \exists\, ref \bullet A \left[\langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t)/tr_t, tr'_t \right] \\
\sqsupseteq & \textit{[Predicate calculus]} \\
& \exists\, ref \bullet B \left[\langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t)/tr_t, tr'_t \right] \\
= & \textit{[3.4.6]} \\
& R2_t(B)
\end{array}
$$

B.3 PROPERTIES OF $R3_T$

Property B.3

L1. $R3_t$ *is idempotent, for any timed program* P
$R3_t(R3_t(P)) = R3_t(P)$
Proof:

$$
\begin{array}{ll}
& R3_t(R3_t(P)) \\
= & \textit{[3.4.10]} \\
& \Pi_t \lhd wait \rhd (\Pi_t \lhd wait \rhd P) \\
= & \textit{[Property 3.7 L6]} \\
& \Pi_t \lhd wait \rhd P \\
= & \textit{[3.4.10]} \\
& R3_t(P) \qquad\qquad \Box
\end{array}
$$

L2. Π_t *is* $R3_t$ *healthy*
$R3_t(\Pi_t) = \Pi_t$
Proof:

$$
\begin{array}{ll}
& R3_t(\Pi_t) \\
= & \textit{[3.4.10]} \\
& \Pi_t \lhd wait \rhd \Pi_t \\
= & \textit{[Property 3.7 L1]} \\
& \Pi_t \qquad\qquad \Box
\end{array}
$$

L3. $R3_t$ *is commutative with* $R1_t$*; given any timed program* P
$R1_t(R3_t(P)) = R3_t(R1_t(P))$
Proof:

$$
\begin{array}{ll}
& R3_t(R1_t(P)) \\
= & \textit{[3.4.2]}
\end{array}
$$

$R3_t(P \wedge Expands(tr_t, tr_t'))$
$=$ *[3.4.10]*
$\Pi_t \lhd wait \rhd (P \wedge Expands(tr_t, tr_t'))$
$=$ *[Property B.1 L2 and L1]*
$R1_t(\Pi_t) \lhd wait \rhd (P \wedge Expands(tr_t, tr_t'))$
$=$ *[3.4.2]*
$(\Pi_t \wedge Expands(tr_t, tr_t')) \lhd wait \rhd (P \wedge Expands(tr_t, tr_t'))$
$=$ *[wait free in $Expands(tr_t, tr_t')$]*
$(\Pi_t \lhd wait \rhd P) \wedge Expands(tr_t, tr_t')$
$=$ *[3.4.10]*
$R3_t(P) \wedge Expands(tr_t, tr_t')$
$=$ *[3.4.2]*
$R1_t(R3_t(P))$ □

L4. *$R3_t$ is commutative with $R2_t$; given any timed program P*
$R2_t(R3_t(P)) = R3_t(R2_t(P))$
Proof:

$R3_t(R2_t(P))$
$=$ *[3.4.10]*
$\Pi_t \lhd wait \rhd R2_t(P)$
$=$ *[Property B.2 L2 and L1]*
$R2_t(\Pi_t) \lhd wait \rhd R2_t(P)$
$=$ *[Property B.2 L6]*
$R2_t(\Pi_t \lhd wait \rhd P)$
$=$ *[3.4.10]*
$R2_t(R3_t(P))$ □

L5. *$R3_t$ is closed over $\vee$; given any two $R3_t$ healthy timed programs P and Q*
$R3_t(P \vee Q) = P \vee Q$
Proof:

$R3_t(P \vee Q)$
$=$ *[3.4.10]*
$\Pi_t \lhd wait \rhd (P \vee Q)$
$=$ *[3.5.22]*
$(\Pi_t \wedge wait) \vee (\neg wait \wedge (P \vee Q))$
$=$ *[Propositional calculus]*
$(\Pi_t \wedge wait) \vee (\neg wait \wedge P) \vee (\neg wait \wedge Q)$

$$
\begin{array}{ll}
= & \textit{[Propositional calculus]} \\
((\Pi_t \wedge \mathit{wait}) \vee (\neg \mathit{wait} \wedge P)) \vee ((\Pi_t \wedge \mathit{wait}) \vee (\neg \mathit{wait} \wedge Q)) & \\
= & \textit{[3.5.22]} \\
(\Pi_t \lhd \mathit{wait} \rhd P) \vee (\Pi_t \lhd \mathit{wait} \rhd Q) & \\
= & \textit{[3.4.10]} \\
R3_t(P) \vee R3_t(Q) & \\
= & \textit{[Assumption and property B.3 L1]} \\
P \vee Q & \Box
\end{array}
$$

L6. *$R3_t$ is closed over $\wedge$; given any two $R3_t$ healthy timed programs P and Q*
$R3_t(P \wedge Q) = P \wedge Q$
Proof:

$$
\begin{array}{ll}
P \wedge Q & \\
= & \textit{[Assumption and property B.3 L1]} \\
R3_t(P) \wedge R3_t(Q) & \\
= & \textit{[3.4.10]} \\
(\Pi_t \lhd \mathit{wait} \rhd P) \wedge (\Pi_t \lhd \mathit{wait} \rhd Q) & \\
= & \textit{[3.5.22]} \\
((\Pi_t \wedge \mathit{wait}) \vee (P \wedge \neg \mathit{wait})) \wedge ((\Pi_t \wedge \mathit{wait}) \vee (Q \wedge \neg \mathit{wait})) & \\
= & \textit{[Propositional calculus]} \\
\begin{array}{l}
((\Pi_t \wedge \mathit{wait}) \wedge ((\Pi_t \wedge \mathit{wait}) \vee (Q \wedge \neg \mathit{wait}))) \vee \\
((P \wedge \neg \mathit{wait}) \wedge ((\Pi_t \wedge \mathit{wait}) \vee (Q \wedge \neg \mathit{wait})))
\end{array} & \\
= & \textit{[Propositional calculus]} \\
\begin{array}{l}
((\Pi_t \wedge \mathit{wait}) \wedge (\Pi_t \wedge \mathit{wait})) \vee \\
((\Pi_t \wedge \mathit{wait}) \wedge (Q \wedge \neg \mathit{wait})) \vee \\
((P \wedge \neg \mathit{wait}) \wedge (\Pi_t \wedge \mathit{wait})) \vee \\
((P \wedge \neg \mathit{wait}) \wedge (Q \wedge \neg \mathit{wait}))
\end{array} & \\
= & \textit{[Propositional calculus]} \\
(\Pi_t \wedge \mathit{wait}) \vee ((P \wedge Q) \wedge \neg \mathit{wait}) & \\
= & \textit{[3.4.10]} \\
R3_t(P \wedge Q) & \Box
\end{array}
$$

L7. *$R3_t$ is closed over $\lhd\rhd$; given any two $R3_t$ healthy timed programs P and Q*
$R3_t(P \lhd b \rhd Q) = P \lhd b \rhd Q$
Proof:

$$
\begin{array}{ll}
P \lhd b \rhd Q & \\
= & \textit{[Assumption and property B.3 L1]}
\end{array}
$$

$R3_t(P) \lhd b \rhd R3_t(Q)$
$=$ *[3.4.10]*
$(\Pi_t \lhd wait \rhd P) \lhd b \rhd (\Pi_t \lhd wait \rhd Q)$
$=$ *[Property 3.7 L4]*
$\Pi_t \lhd wait \rhd (P \lhd b \rhd Q)$
$=$ *[3.4.10]*
$R3_t(P \lhd b \rhd Q)$ □

L8. *$R3_t$ is closed over sequential composition; given any two $R3_t$ healthy timed programs P and Q*
$R3_t(P;\ Q) = P;\ Q$
Proof:

$P;\ Q$
$=$ *[Assumption and property B.3 L1]*
$R3_t(P);\ R3_t(Q)$
$=$ *[3.4.10]*
$(\Pi_t \lhd wait \rhd P);\ (\Pi_t \lhd wait \rhd Q)$
$=$ *[Property 3.6 L5]*
$(\Pi_t;\ (\Pi_t \lhd wait \rhd Q)) \lhd wait \rhd (P;\ (\Pi_t \lhd wait \rhd Q))$
$=$ *[Lemma 3.4]*
$(\Pi_t \lhd wait \rhd Q) \lhd wait \rhd (P;\ (\Pi_t \lhd wait \rhd Q))$
$=$ *[3.4.10]*
$(\Pi_t \lhd wait \rhd Q) \lhd wait \rhd (P;\ (R3_t(Q)))$
$=$ *[Assumption and B.3 L1]*
$(\Pi_t \lhd wait \rhd Q) \lhd wait \rhd (P;\ Q)$
$=$ *[Property 3.7 L6]*
$\Pi_t \lhd wait \rhd (P;\ Q)$
$=$ *[3.4.10]*
$R3_t(P;\ Q)$ □

L9. *$R3_t$ is monotonic with respect to the ordering relation $\sqsupseteq$*
Given to time actions A and B such that $A \sqsupseteq B$ then
$R3_t(A) \sqsupseteq R3_t(B)$ ***Proof:***

$R3_t(A)$
$=$ *[3.4.10]*
$(\Pi_t \lhd wait \rhd A)$
$\sqsupseteq$ *[Assumption and C.0.4]*

$$
\begin{array}{ll}
& (\Pi_t \lhd wait \rhd B) \\
= & \qquad [3.4.10] \\
& R3_t(B) \qquad \Box
\end{array}
$$

B.4 PROPERTIES OF $CSP1_T$

Property B.4

L1. $CSP1_t$ *is idempotent, for any timed program* P
$CSP1_t(CSP1_t(P)) = CSP1_t(P)$
Proof:

$$
\begin{array}{ll}
& CSP1_t(CSP1_t(P)) \\
= & \qquad [3.4.14] \\
& (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\
& ((\neg ok \wedge Expands(tr_t, tr_t')) \vee P) \\
= & \qquad [Propositional\ calculus] \\
& (\neg ok \wedge Expands(tr_t, tr_t')) \vee P \\
= & \qquad [3.4.14] \\
& CSP1_t(P) \qquad \Box
\end{array}
$$

L2. Π_t *is* $CSP1_t$ *healthy*
$CSP1_t(\Pi_t) = \Pi_t$
Proof:

$$
\begin{array}{ll}
& CSP1_t(\Pi_t) \\
= & \qquad [3.4.14] \\
& (\neg ok \wedge Expands(tr_t, tr_t')) \vee \Pi_t \\
= & \qquad [3.4.11] \\
& \left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr_t')) \vee (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ (ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \end{array} \right) \\
= & \qquad [Proposition\ calculus] \\
& \left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ (ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \end{array} \right) \\
= & \qquad [3.4.11] \\
& \Pi_t \qquad \Box
\end{array}
$$

L3. $CSP1_t$ *is commutative with* $R1_t$
$CSP1_t(R1_t(P)) = R1_t(CSP1_t(P))$
Proof:

$$
CSP1_t(R1_t(P))
$$

$$
\begin{array}{ll}
= & \textit{[3.4.14]} \\
R1_t(P) \vee (\neg ok \wedge Expands(tr_t, tr'_t)) & \\
= & \textit{[3.4.2]} \\
(P \wedge Expands(tr_t, tr'_t)) \vee & \\
(\neg ok \wedge Expands(tr_t, tr'_t)) & \\
= & \textit{[Proposition calulus]} \\
(P \vee (\neg ok \wedge Expands(tr_t, tr'_t))) \wedge & \\
(Expands(tr_t, tr'_t) \vee (\neg ok \wedge Expands(tr_t, tr'_t))) & \\
= & \textit{[Proposition calculus]} \\
(P \vee (\neg ok \wedge Expands(tr_t, tr'_t))) \wedge & \\
Expands(tr_t, tr'_t) & \\
= & \textit{[3.4.14]} \\
CSP1_t(P) \wedge Expands(tr_t, tr'_t) & \\
= & \textit{[3.4.2]} \\
R1_t(CSP1_t(P)) & \square
\end{array}
$$

L4. $CSP1_t$ *is commutative with* $R2_t$
$CSP1_t(R2_t(P)) = R2_t(CSP1_t(P))$
Proof:

$$
\begin{array}{ll}
R2_t(CSP1_t(P)) & \\
= & \textit{[3.4.14]} \\
R2_t(P \vee (\neg ok \wedge Expands(tr_t, tr'_t))) & \\
= & \textit{[3.4.6]} \\
\exists\, ref \bullet (P \vee (\neg ok \wedge Expands(tr_t, tr'_t))) & \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] & \\
= & \textit{[Predicate calculus and Substitution]} \\
\exists\, ref \bullet & \\
P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] \vee & \\
(\neg ok \wedge Expands(\langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t))) & \\
= & \textit{[Property 3.3 L9]} \\
\exists\, ref \bullet & \\
P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] \vee & \\
(\neg ok \wedge Expands(\langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t)) \wedge Expands(tr_t, tr'_t)) & \\
= & \textit{[Property 3.2 L8]}
\end{array}
$$

$$
\begin{array}{l}
\exists\, ref \bullet \\
P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t) \\ /tr_t, tr'_t \end{array} \right] \vee \\
(\neg ok \wedge Expands(tr_t, tr'_t))
\end{array}
$$

$=$ *[3.4.6]*

$R2_t(P) \vee (\neg ok \wedge Expands(tr_t, tr'_t))$

$=$ *[3.4.14]*

$CSP1_t(R2_t(P))$ □

L5. *$CSP1_t$ is commutative with $R3_t$*
$CSP1_t(R3_t(P)) = R3_t(CSP1_t(P))$
Proof:

$CSP1_t(R3_t(P))$

$=$ *[3.4.10]*

$CSP1_t(\Pi_t \lhd wait \rhd P)$

$=$ *[3.4.14]*

$(\Pi_t \lhd wait \rhd P) \vee$
$(\neg ok \wedge Expands(tr_t, tr'_t))$

$=$ *[Property 3.7 L7]*

$(\Pi_t \vee (\neg ok \wedge Expands(tr_t, tr'_t)))$
$\lhd wait \rhd$
$(P \vee (\neg ok \wedge Expands(tr_t, tr'_t)))$

$=$ *[3.4.14]*

$CSP1_t(\Pi_t) \lhd wait \rhd CSP1_t(P)$

$=$ *[Property B.4 L2]*

$\Pi_t \lhd wait \rhd CSP1_t(P)$

$=$ *[3.4.10]*

$R3_t(CSP1_t(P))$ □

L6. *$CSP1_t$ is closed under disjunction, provided that P and Q are $CSP1_t$*
$CSP1_t(P \vee Q) = P \vee Q$
Proof:

$CSP1_t(P \vee Q)$

$=$ *[3.4.14]*

$(P \vee Q) \vee (\neg ok \wedge Expands(tr_t, tr'_t))$

$=$ *[Propositional calculus]*

$(P \vee (\neg ok \wedge Expands(tr_t, tr'_t))) \vee$
$(Q \vee (\neg ok \wedge Expands(tr_t, tr'_t)))$

$=$ *[3.4.14]*

$CSP1_t(P) \vee CSP1_t(Q)$

$=$ *[Assumption and property B.4 L1]*

$P \vee Q$ $\Box$

L7. *$CSP1_t$ is closed under conjunction, provided that P and Q are $CSP1_t$*
$CSP1_t(P \wedge Q) = P \wedge Q$
Proof:

$CSP1_t(P \wedge Q)$

$=$ *[3.4.14]*

$(P \wedge Q) \vee (\neg ok \wedge Expands(tr_t, tr'_t))$

$=$ *[Propositional calculus]*

$(P \vee (\neg ok \wedge Expands(tr_t, tr'_t))) \wedge$
$(Q \vee (\neg ok \wedge Expands(tr_t, tr'_t)))$

$=$ *[3.4.14]*

$CSP1_t(P) \wedge CSP1_t(Q)$

$=$ *[Assumption and property B.4 L1]*

$P \wedge Q$ $\Box$

L8. *$CSP1_t$ is closed over $\lhd\rhd$, provided that P and Q are $CSP1_t$*
$CSP1_t(P \lhd b \rhd Q) = P \lhd b \rhd Q$
Proof:

$CSP1_t(P \lhd b \rhd Q)$

$=$ *[3.4.14]*

$(P \lhd b \rhd Q) \vee (\neg ok \wedge Expands(tr_t, tr'_t))$

$=$ *[Property 3.9 L8]*

$(P \vee (\neg ok \wedge Expands(tr_t, tr'_t))) \lhd b \rhd (Q \vee (\neg ok \wedge Expands(tr_t, tr'_t)))$

$=$ *[3.4.14]*

$(CSP1_t(P) \lhd b \rhd CSP1_t(Q))$

$=$ *[Assumption and property B.4 L1]*

$(P \lhd b \rhd Q)$ $\Box$

L9. *$CSP1_t$ is closed over sequential composition, provided that P and Q are $CSP1_t$*
$CSP1_t(P;\ Q) = P;\ Q$
Proof:

$P;\ Q$

$=$ *[Assumption]*

$$
\begin{array}{ll}
& CSP1_t(P);\ CSP1_t(Q) \\
= & \textit{[3.4.14]} \\
& (P \vee (\neg ok \wedge Expands(tr_t, tr'_t)));\ (Q \vee (\neg ok \wedge Expands(tr_t, tr'_t))) \\
= & \textit{[Property 3.9 L6]} \\
& (P;\ (Q \vee (\neg ok \wedge Expands(tr_t, tr'_t)))) \vee \\
& (\neg ok \wedge Expands(tr_t, tr'_t));\ (Q \vee (\neg ok \wedge Expands(tr_t, tr'_t))) \\
= & \textit{[Property 3.9 L5]} \\
& (P;\ Q) \vee \\
& (P;\ (\neg ok \wedge Expands(tr_t, tr'_t))) \vee \\
& ((\neg ok \wedge Expands(tr_t, tr'_t));\ Q) \vee \\
& (\neg ok \wedge Expands(tr_t, tr'_t));\ (\neg ok \wedge Expands(tr_t, tr'_t)) \\
= & \textit{[Program can not require none termination and calculus.]} \\
& (P;\ Q) \vee \\
& (\neg ok \wedge Expands(tr_t, tr'_t)) \\
= & \textit{[3.4.14]} \\
& CSP1_t(P;\ Q) \qquad \square
\end{array}
$$

L10. $CSP1_t$ *is monotonic with respect to the ordering relation* $\sqsupseteq$
Given to time actions A *and* B *such that* $A \sqsupseteq B$ *then*
$CSP1_t(A) \sqsupseteq CSP1_t(B)$ ***Proof:***

$$
\begin{array}{ll}
& CSP1_t(A) \sqsupseteq CSP1_t(B) \\
= & \textit{[3.5.58]} \\
& [CSP1_t(A) \Rightarrow CSP1_t(B)] \\
= & \textit{[3.4.14]} \\
& \left[\begin{array}{l} (A \vee (\neg ok \wedge Expands(tr_t, tr'_t))) \\ \Rightarrow \\ (B \vee (\neg ok \wedge Expands(tr_t, tr'_t))) \end{array}\right] \\
= & \textit{[Propositional calculus]} \\
& true \qquad \square
\end{array}
$$

B.5 PROPERTIES OF $CSP2_T$

Property B.5

L1. $CSP2_t$ *is idempotent, for any timed program* P
$CSP2_t(CSP2_t(P)) = CSP2_t(P)$
Proof:

$$
\begin{array}{ll}
& CSP2_t(CSP2_t(P)) \\
= & \textit{[3.4.15]}
\end{array}
$$

$$
\begin{array}{ll}
(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state'));\\
(ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state'); & = \quad \textit{[3.5.21]}\\
P;\ \left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\ (ok \Rightarrow ok_o \wedge wait = wait_o \wedge tr = tr_o \wedge state = state_o) \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr_t' \wedge state_o = state') \end{array}\right) & \\
= \quad \textit{[Propositional calculus]} & \\
(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')) & \\
= \quad \textit{[3.4.15]} & \\
CSP2_t(P) \qquad \square &
\end{array}
$$

L2. Π_t *is* $CSP2_t$ *healthy*
$CSP2_t(\Pi_t) = \Pi_t$
Proof:

$$
\begin{array}{l}
CSP2_t(\Pi_t)\\
= \quad \textit{[3.4.15]}\\
\Pi_t;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')\\
= \quad \textit{[3.4.11]}\\
\left(\begin{array}{c} (\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ (ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state)) \end{array}\right);\\
(ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')\\
= \quad \textit{[Property 3.9 L5]}\\
\left(\begin{array}{l} (\neg ok \wedge Expands(tr_t, tr_t'));\\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state') \end{array}\right) \vee\\
\left(\begin{array}{l} (ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state))\\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state') \end{array}\right)\\
= \quad \textit{[3.5.21]}\\
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\ (\neg ok \wedge Expands(tr_t, tr_o)) \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr_t' \wedge state_o = state') \end{array}\right) \vee\\
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\ (ok_o \wedge (tr_o = tr_t) \wedge (wait_o = wait) \wedge (state_o = state)) \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr_t' \wedge state_o = state') \end{array}\right)\\
= \quad \textit{[Relational calculus and substitution]}\\
\left(\begin{array}{l} \exists\, ok_o \bullet \\ (\neg ok \wedge Expands(tr_t, tr_t')) \wedge \\ (ok_o \Rightarrow ok') \end{array}\right) \vee\\
(ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')\\
= \quad \textit{[one point rule]}
\end{array}
$$

$$\left(\begin{array}{l}(\neg ok \wedge Expands(tr_t, tr'_t)) \wedge \\ ((true \Rightarrow ok') \vee (false \Rightarrow ok'))\end{array}\right) \vee$$
$$(ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state')$$
$$= \qquad \textit{[Propositional calculus]}$$
$$(\neg ok \wedge Expands(tr_t, tr'_t)) \vee (ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state')$$
$$= \qquad \textit{[3.4.11]}$$
$$\Pi_t \qquad \square$$

L3. $CSP2_t$ *is commutative with* $R1_t$
$CSP2_t(R1_t(P)) = R1_t(CSP2_t(P))$
Proof:

$$R1_t(CSP2_t(P))$$
$$= \qquad \textit{[3.4.15]}$$
$$R1_t(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state'))$$
$$= \qquad \textit{[3.4.2]}$$
$$(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state')) \wedge Expands(tr_t, tr'_t)$$
$$= \qquad \textit{[3.5.21]}$$
$$\left(\begin{array}{l}\exists\, ok_o, wait_o, tr_o, state_o \bullet \\ P[ok_o, wait_o, tr_o, state_o / ok', wait', tr'_t, state'] \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr'_t \wedge state_o = state')\end{array}\right) \wedge$$
$$Expands(tr_t, tr'_t)$$
$$= \qquad \textit{[Properties 3.2 L3 and L7]}$$
$$\left(\begin{array}{l}\exists\, ok_o, wait_o, tr_o, state_o \bullet \\ P[ok_o, wait_o, tr_o, state_o / ok', wait', tr'_t, state'] \wedge Expands(tr_t, tr_o) \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr'_t \wedge state_o = state')\end{array}\right)$$
$$= \qquad \textit{[Relational calculus]}$$
$$\left(\begin{array}{l}\exists\, ok_o, wait_o, tr_o, state_o \bullet \\ (P \wedge Expands(tr_t, tr'_t))[ok_o, wait_o, tr_o, state_o / ok', wait', tr'_t, state'] \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr'_t \wedge state_o = state')\end{array}\right)$$
$$= \qquad \textit{[3.4.2]}$$
$$\left(\begin{array}{l}\exists\, ok_o, wait_o, tr_o, state_o \bullet \\ (R1_t(P))[ok_o, wait_o, tr_o, state_o / ok', wait', tr'_t, state'] \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr'_t \wedge state_o = state')\end{array}\right)$$
$$= \qquad \textit{[3.5.21]}$$
$$\left(\begin{array}{l}(R1_t(P)); \\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state')\end{array}\right)$$
$$= \qquad \textit{[3.4.15]}$$
$$CSP2_t(R1_t(P)) \qquad \square$$

L4. $CSP2_t$ is commutative with $R2_t$
$CSP2_t(R2_t(P)) = R2_t(CSP2_t(P))$
Proof:

$$
\begin{array}{ll}
R2_t(CSP2_t(P)) & \\
= & [3.4.15] \\
R2_t(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')) & \\
= & [3.4.6] \\
\exists\, ref \bullet & \\
(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')) & \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] & \\
= & [3.5.21] \\
\exists\, ref \bullet & \\
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\ P[ok_o, wait_o, tr_o, state_o / ok', wait', tr_t', state'] \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr_t' \wedge state_o = state') \end{array} \right) & \\
\left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] & \\
= & [Substitution] \\
\exists\, ref \bullet & \\
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\ P[ok_o, wait_o, tr_o, state_o, \langle(\langle\rangle, ref)\rangle / ok', wait', tr_t', state', tr_t] \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = dif(tr_t', tr_t) \wedge state_o = state') \end{array} \right) & \\
= & [Substitution\ tr_o = dif(tr_t', tr_t)] \\
\exists\, ref \bullet & \\
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\ P[ok_o, wait_o, dif(tr_t', tr_t), state_o, \langle(\langle\rangle, ref)\rangle / ok', wait', tr_t', state', tr_t] \wedge \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = dif(tr_t', tr_t) \wedge state_o = state') \end{array} \right) & \\
= & [Predicate\ calculus] \\
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\ \exists\, ref \bullet \\ P \left[\begin{array}{l} \langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t) \\ /tr_t, tr_t' \end{array} \right] \\ {[ok_o, wait_o, tr_o, state_o / ok', wait', tr_t', state']} \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr_t' \wedge state_o = state') \end{array} \right) & \\
= & [3.4.6] \\
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\ R2_t(P) \\ {[ok_o, wait_o, tr_o, state_o / ok', wait', tr_t', state']} \\ (ok_o \Rightarrow ok' \wedge wait_o = wait' \wedge tr_o = tr_t' \wedge state_o = state') \end{array} \right) &
\end{array}
$$

$=$ *[3.5.21]*

$R2_t(P);\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')$

$=$ *[3.4.15]*

$CSP2_t(R2_t(P))$ □

L5. $CSP2_t$ *is commutative with* $R3_t$
$CSP2_t(R3_t(P)) = R3_t(CSP2_t(P))$
Proof:

$CSP2_t(R3_t(P))$

$=$ *[3.4.15]*

$R3_t(P);\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')$

$=$ *[3.4.10]*

$(\Pi_t \lhd wait \rhd P);\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')$

$=$ *[Property 3.7 L6]*

$(\Pi_t;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state'))$
$\lhd wait \rhd$
$(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state'))$

$=$ *[3.4.15]*

$CSP2_t(\Pi_t) \lhd wait \rhd CSP2_t(P)$

$=$ *[Property B.5 L2]*

$\Pi_t \lhd wait \rhd CSP2_t(P)$

$=$ *[3.4.10]*

$R3_t(CSP2_t(P))$ □

L6. $CSP2_t$ *is commutative with* $CSP1_t$
$CSP1_t(CSP2_t(P)) = CSP2_t(CSP1_t(P))$
Proof:

$CSP2_t(CSP1_t(P))$

$=$ *[3.4.14]*

$CSP2_t(P \vee (\neg ok \wedge Expands(tr_t, tr_t')))$

$=$ *[3.4.15]*

$(P \vee (\neg ok \wedge Expands(tr_t, tr_t')));$
$(ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')$

$=$ *[Property 3.9 L5]*

$(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')) \vee$
$(\neg ok \wedge Expands(tr_t, tr_t'));\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')$

$=$ *[Relational calculus]*

$(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')) \vee$
$(\neg ok \wedge Expands(tr_t, tr_t'))$
$=$ *[3.4.15]*
$(CSP2_t(P)) \vee (\neg ok \wedge Expands(tr_t, tr_t'))$
$=$ *[3.4.14]*
$CSP1_t(CSP2_t(P))$ □

L7. *$CSP2_t$ is closed under disjunction, provided that P and Q are $CSP2_t$*
$CSP2_t(P \vee Q) = P \vee Q$
Proof:

$CSP2_t(P \vee Q)$
$=$ *[3.4.15]*
$(P \vee Q);\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')$
$=$ *[Property 3.9 L5]*
$(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')) \vee$
$(Q;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state'))$
$=$ *[3.4.15]*
$CSP2_t(P) \vee CSP2_t(Q)$
$=$ *[Assumption and Property B.5 L1]*
$P \vee Q$ □

L8. *$CSP2_t$ is not closed under conjunction, provided that P and Q are $CSP2_t$*
$CSP1_t(P \wedge Q) \Rightarrow (P \wedge Q)$
Proof:

$CSP1_t(P \wedge Q)$
$=$ *[3.4.15]*
$(P \wedge Q);\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')$
$=$ *[3.5.21]*
$\exists\, obs_o \bullet (P \wedge Q)[obs_o/obs'] \wedge$
$(ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')[obs_o/obs]$
$=$ *[Propositional calculus]*
$\exists\, obs_o \bullet$
$(P[obs_o/obs'] \wedge (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')[obs_o/obs]) \wedge$
$(Q[obs_o/obs'] \wedge (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')[obs_o/obs])$
$\Rightarrow$ *[Propositional calculus]*
$\exists\, obs_o \bullet$
$(P[obs_o/obs'] \wedge (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')[obs_o/obs]) \wedge$
$\exists\, obs_o \bullet$
$(Q[obs_o/obs'] \wedge (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr_t' \wedge state = state')[obs_o/obs])$

$$
\begin{array}{ll}
= & [3.5.21] \\
(P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state')) \wedge & \\
(Q;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state')) & \\
= & [3.4.15] \\
CSP2_t(P) \wedge CSP2_t(Q) & \\
= & [\textit{Assumption and Property B.5 L1}] \\
P \wedge Q & \Box
\end{array}
$$

L9. *$CSP2_t$ is closed over $\lhd\rhd$, provided that P and Q are $CSP2_t$*
$CSP2_t(P \lhd b \rhd Q) = P \lhd b \rhd Q$
Proof:

$$
\begin{array}{ll}
CSP2_t(P \lhd b \rhd Q) & \\
= & [3.4.15] \\
(P \lhd b \rhd Q);\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state') & \\
= & [\textit{Property 3.7 L6}] \\
P;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state') & \\
\lhd\ b\ \rhd & \\
Q;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state') & \\
= & [3.4.15] \\
CSP2_t(P) \lhd b \rhd CSP2_t(Q) & \\
= & [\textit{Assumption and Property B.5 L1}] \\
P \lhd b \rhd Q & \Box
\end{array}
$$

L10. *$CSP2_t$ is closed over sequential composition, provided that Q is $CSP2_t$*
$CSP2_t(P;\ Q) = P;\ Q$
Proof:

$$
\begin{array}{ll}
P;\ Q & \\
= & [\textit{Assumption and Property B.5 L1}] \\
P;\ CSP2_t(Q) & \\
= & [3.4.15] \\
P;\ (Q;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state')) & \\
= & [\textit{Property 3.6 L4}] \\
(P;\ Q);\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state') & \\
= & [3.4.15] \\
CSP2_t(P;\ Q) & \Box
\end{array}
$$

L11. *$CSP2_t$ is monotonic with respect to the ordering relation $\sqsupseteq$*
Given to time actions A and B such that $A \sqsupseteq B$ then

$CSP2_t(A) \sqsupseteq CSP2_t(B)$ ***Proof:***

$CSP2_t(A)$
$=$ *[3.4.15]*
$A;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state')$
$\sqsupseteq$ *[C.0.8]*
$B;\ (ok \Rightarrow ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge state = state')$
$=$ *[3.4.15]*
$CSP2_t(B)$ □

B.6 PROPERTIES OF $CSP3_T$

Property B.6

L1. *$CSP3_t$ is idempotent, for any timed program P.*
$CSP3_t(CSP3_t(P)) = CSP3_t(P)$
Proof:

$CSP3_t(CSP3_t(P))$
$=$ *[3.4.16]*
$Skip;\ (Skip;\ P)$
$=$ *[Property 3.6 L4]*
$(Skip;\ Skip);\ P$
$=$ *[Assumption and Lemma 3.6]*
$Skip;\ P$
$=$ *[3.4.16]*
$CSP3_t(P)$ □

L2. *$CSP3_t$ is closed over $\vee$, provided that P and Q are $CSP3_t$ healthy*
$CSP3_t(P \vee Q) = P \vee Q$
Proof:

$CSP3_t(P \vee Q)$
$=$ *[3.4.16]*
$Skip;\ (P \vee Q)$
$=$ *[Property 3.6 L6]*
$(Skip;\ P) \vee (Skip;\ Q)$
$=$ *[3.4.16]*
$CSP3_t(P) \vee CSP3_t(Q)$
$=$ *[assumption and property B.6 L1]*
$P \vee Q$ □

L3. *$CSP3_t$ is closed over conditional choice operator $\lhd\rhd$, provided that P and Q are $CSP3_t$ healthy*
$CSP3_t(P \lhd b \rhd Q) = P \lhd b \rhd Q$
Proof:

$CSP3_t(P \lhd b \rhd Q)$
$=$ *[3.4.16]*
$Skip;\ (P \lhd b \rhd Q)$
$=$ *[Property 3.6 L6]*
$(Skip;\ P) \lhd b \rhd (Skip;\ Q)$
$=$ *[3.4.16]*
$CSP3_t(P) \lhd b \rhd CSP3_t(Q)$
$=$ *[assumption and property B.6 L1]*
$P \lhd b \rhd Q$ $\Box$

L4. *$CSP3_t$ is closed over sequential composition, provided that P is $CSP3_t$ healthy*
$CSP3_t(P;\ Q) = P;\ Q$
Proof:

$P;\ Q$
$=$ *[assumption]*
$CSP3_t(P);\ Q$
$=$ *[3.4.16]*
$(Skip;\ P);\ Q$
$=$ *[Property 3.6 L4]*
$Skip;\ (P;\ Q)$
$=$ *[3.4.16]*
$CSP3_t(P;\ Q)\Box$

L5. *If P and Q are $CSP3_t$ healthy it implies that their conjunction is also $CSP3_t$ healthy.*
$(P \wedge Q) \Rightarrow CSP3_t(P \wedge Q)$
Proof:

$P \wedge Q$
$=$ *[assumption]*
$CSP3_t(P) \wedge CSP3_t(Q)$
$=$ *[3.4.16]*
$(Skip;\ P) \wedge (Skip;\ Q)$
$=$ *[3.5.21]*

$(\exists\, v_o \bullet \mathit{Skip}[v_o/v'] \wedge P[v_o/v]) \wedge (\exists\, v_o \bullet \mathit{Skip}[v_o/v'] \wedge Q[v_o/v])$
$\Rightarrow$ *[Predicate calculus]*
$\exists\, v_o \bullet \mathit{Skip}[v_o/v'] \wedge P[v_o/v] \wedge \mathit{Skip}[v_o/v'] \wedge Q[v_o/v]$
$=$ *[Propositional calculus]*
$\exists\, v_o \bullet \mathit{Skip}[v_o/v'] \wedge P[v_o/v] \wedge Q[v_o/v]$
$=$ *[Predicate calculus]*
$\exists\, v_o \bullet \mathit{Skip}[v_o/v'] \wedge (P \wedge Q)[v_o/v]$
$=$ *[3.5.21]*
$\mathit{Skip};\ (P \wedge Q)$
$=$ *[3.4.16]*
$CSP3_t(P \wedge Q)$ □

L6. *$CSP3_t$ is monotonic with respect to the ordering relation $\sqsupseteq$*
Given to time actions A and B such that $A \sqsupseteq B$ then $CSP3_t(A) \sqsupseteq CSP3_t(B)$ ***Proof:***

$CSP3_t(A)$
$=$ *[3.4.16]*
$\mathit{Skip};\ A$
$\sqsupseteq$ *[C.0.9]*
$\mathit{Skip};\ B$
$=$ *[3.4.16]*
$CSP3_t(B)$□

B.7 PROPERTIES OF $CSP4_T$

Property B.7

L1. *$CSP4_t$ is idempotent, for any timed program P*
$CSP4_t(CSP4_t(P)) = CSP4_t(P)$
Proof:

$CSP4_t(CSP4_t(P))$
$=$ *[3.4.17]*
$(P;\ \mathit{Skip});\ \mathit{Skip}$
$=$ *[Property 3.6 L4]*
$P;\ (\mathit{Skip};\ \mathit{Skip})$
$=$ *[Lemma 3.6]*
$P;\ \mathit{Skip}$
$=$ *[3.4.17]*
$CSP4_t(P)$ □

L2. *$CSP4_t$ is closed over disjunction, provided that P and Q are $CSP4_t$ healthy*
$CSP4_t(P \vee Q) = P \vee Q$
Proof:

$$
\begin{array}{ll}
& P \vee Q \\
= & \textit{[Assumption]} \\
& CSP4_t(P) \vee CSP4_t(Q) \\
= & \textit{[3.4.17]} \\
& (P;\ \mathit{Skip}) \vee (Q;\ \mathit{Skip}) \\
= & \textit{[Property 3.9 L5]} \\
& (P \vee Q);\ \mathit{Skip} \\
= & \textit{[3.4.17]} \\
& CSP4_t(P \vee Q) \qquad \Box
\end{array}
$$

L3. *$CSP4_t$ is closed over conditional choice operator $\lhd\rhd$, provided that P and Q are $CSP4_t$ healthy*
$CSP4_t(P \lhd b \rhd Q) = P \lhd b \rhd Q$
Proof:

$$
\begin{array}{ll}
& CSP4_t(P \lhd b \rhd Q) \\
= & \textit{[3.4.17]} \\
& (P \lhd b \rhd Q);\ \mathit{Skip} \\
= & \textit{[Property 3.6 L5]} \\
& (P;\ \mathit{Skip}) \lhd b \rhd (Q;\ \mathit{Skip}) \\
= & \textit{[3.4.17]} \\
& CSP4_t(P) \lhd b \rhd CSP4_t(Q) \\
= & \textit{[assumption and property B.7 L1]} \\
& P \lhd b \rhd Q \qquad \Box
\end{array}
$$

L4. *$CSP4_t$ is closed over sequential composition, provided that Q is $CSP4_t$ healthy*
$CSP4_t(P;\ Q) = P;\ Q$
Proof:

$$
\begin{array}{ll}
& P;\ Q \\
= & \textit{[assumption]} \\
& P;\ CSP4_t(Q) \\
= & \textit{[3.4.17]} \\
& P;\ (Q;\ \mathit{Skip}) \\
= & \textit{[Property 3.6 L4]} \\
& (P;\ Q);\ \mathit{Skip}
\end{array}
$$

$=$ *[3.4.17]*
$CSP4_t(P;\ Q)\Box$

L5. If P and Q are $CSP4_t$ healthy it implies that their conjunction is also $CSP4_t$ healthy.
$(P \wedge Q) \Rightarrow CSP4_t(P \wedge Q)$
Proof:

$P \wedge Q$
$=$ *[assumption]*
$CSP4_t(P) \wedge CSP4_t(Q)$
$=$ *[3.4.17]*
$(P;\ Skip) \wedge (Q;\ Skip)$
$=$ *[3.5.21]*
$(\exists\, v_o \bullet P[v_o/v'] \wedge Skip[v_o/v]) \wedge (\exists\, v_o \bullet Q[v_o/v'] \wedge Skip[v_o/v])$
$\Rightarrow$ *[Predicate calculus]*
$\exists\, v_o \bullet P[v_o/v'] \wedge Skip[v_o/v] \wedge Q[v_o/v'] \wedge Skip[v_o/v]$
$=$ *[Propositional calculus]*
$\exists\, v_o \bullet P[v_o/v'] \wedge Q[v_o/v'] \wedge Skip[v_o/v]$
$\Rightarrow$ *[Predicate calculus]*
$\exists\, v_o \bullet (P \wedge Q)[v_o/v'] \wedge Skip[v_o/v]$
$=$ *[3.5.21]*
$(P \wedge Q);\ Skip$
$=$ *[3.4.17]*
$CSP4_t(P \wedge Q)$ $\Box$

L6. $CSP4_t$ is monotonic with respect to the ordering relation $\sqsupseteq$
Given to time actions A and B such that $A \sqsupseteq B$ then
$CSP4_t(A) \sqsupseteq CSP4_t(B)$ ***Proof:***

$CSP4_t(A)$
$=$ *[3.4.17]*
$A;\ Skip$
$\sqsupseteq$ *[C.0.8]*
$B;\ Skip$
$=$ *[3.4.17]*
$CSP4_t(B)\Box$

APPENDIX C

MONOTONICITY OF CIRCUS TIME ACTION OPERATORS

In this chapter we present the detailed proof of the monotonicity of the *Circus* Time Actionconstructs. Given any two timed actions A and B such that $A \sqsupseteq B$, then given any timed program P the following laws should be satisfied

Law C.0.1

$$(A \wedge P) \sqsupseteq (B \wedge P)$$

Provided that $A \sqsupseteq B$.

Proof:

$(A \wedge P) \sqsupseteq (B \wedge P)$
$=$ [3.5.58]
$[(A \wedge P) \Rightarrow (B \wedge P)]$
$=$ [Propositional calculus]
$[((A \wedge P) \Rightarrow B) \wedge ((A \wedge P) \Rightarrow P)]$
$=$ [Propositional calculus]
$[((A \Rightarrow B) \vee (A \Rightarrow P)) \wedge ((A \Rightarrow P) \vee (P \Rightarrow P))]$
$=$ [Assumption and propositional calculus]
$[(true \vee (A \Rightarrow P)) \wedge ((A \Rightarrow P) \vee true)]$
$=$ [Propositional calculus]
$[true \wedge true]$
$=$ [Propositional calculus]
$true$ □

Law C.0.2

$$(A \vee P) \sqsupseteq (B \vee P)$$

Provided that $A \sqsupseteq B$.

Proof:

$(A \vee P) \sqsupseteq (B \vee P)$
$=$ [3.5.58]

$[(A \lor P) \Rightarrow (B \lor P)]$
$=$ [Propositional calculus]
$[(A \Rightarrow (B \lor P)) \land (P \Rightarrow (B \lor P))]$
$=$ [Propositional calculus]
$[((A \Rightarrow B) \lor (A \Rightarrow P)) \land ((P \Rightarrow B) \lor (P \Rightarrow P))]$
$=$ [Assumption and propositional calculus]
$[(true \lor (A \Rightarrow P)) \land ((P \Rightarrow B) \lor true)]$
$=$ [Propositional calculus]
$true$ □

Law C.0.3

$$(A \lhd b \rhd P) \sqsupseteq (B \lhd b \rhd P)$$

Provided that $A \sqsupseteq B$.

Proof:
First lets consider b to be *true*

$(A \lhd true \rhd P) \sqsupseteq (B \lhd true \rhd P)$
$=$ [Property 3.7 L5]
$A \sqsupseteq B$
$=$ [Assumption]
$true$ □

Next take b to be *false*

$(A \lhd false \rhd P) \sqsupseteq (B \lhd false \rhd P)$
$=$ [Property 3.7 L5]
$P \sqsupseteq P$
$=$ [3.5.58]
$[P \Rightarrow P]$
$=$ [Predicate calculus]
$true$ □

Law C.0.4

$$(P \lhd b \rhd A) \sqsupseteq (P \lhd b \rhd A)$$

Provided that $A \sqsupseteq B$.

Proof:
Same as C.0.3 □

Law C.0.5

$$(b \& A) \sqsupseteq (b \& B)$$

Provided that $A \sqsupseteq B$.

Proof:

$$
\begin{array}{ll}
(b \& A) \sqsupseteq (b \& B) = & [3.5.23] \\
(A \lhd b \rhd Stop) \sqsupseteq (B \lhd b \rhd Stop) & \\
= & [C.0.3] \\
true & \Box
\end{array}
$$

Law C.0.6

$$(A \sqcap P) \sqsupseteq (B \sqcap P)$$

Provided that $A \sqsupseteq B$.

Proof:

$$
\begin{array}{ll}
(A \sqcap P) \sqsupseteq (B \sqcap P) & \\
= & [3.5.24] \\
(A \vee P) \sqsupseteq (B \vee P) & \\
= & [C.0.2] \\
true & \Box
\end{array}
$$

Law C.0.7

$$(P \sqcap A) \sqsupseteq (P \sqcap B)$$

Provided that $A \sqsupseteq B$.

Proof:
From Law C.0.6 and internal choice is commutative from Property 3.9 L3.□

Law C.0.8

$$(A;\ P) \sqsupseteq (B;\ P)$$

Provided that $A \sqsupseteq B$.

Proof:

$$
\begin{array}{ll}
A;\ P & \\
= & [3.5.21] \\
\exists\, obs_o \bullet [\![A]\!]_{time}[obs_o/obs'] \wedge [\![P]\!]_{time}[obs_o/obs] & \\
\sqsupseteq & [\text{Predicate calculus and C.0.1}] \\
\exists\, obs_o \bullet [\![B]\!]_{time}[obs_o/obs'] \wedge [\![P]\!]_{time}[obs_o/obs] & \\
= & [3.5.21] \\
B;\ P & \Box
\end{array}
$$

Law C.0.9

$$(P;\ A) \sqsupseteq (P;\ A)$$

Provided that $A \sqsupseteq B$.

Proof:

$$\begin{array}{ll}
& P;\ A \\
= & [3.5.21] \\
& \exists\, obs_o \bullet [\![P]\!]_{time}[obs_o/obs'] \wedge [\![A]\!]_{time}[obs_o/obs] \\
\sqsupseteq & [\text{Predicate calculus and C.0.1}] \\
& \exists\, obs_o \bullet [\![P]\!]_{time}[obs_o/obs'] \wedge [\![B]\!]_{time}[obs_o/obs] \\
= & [3.5.21] \\
& P;\ B \qquad \square
\end{array}$$

Law C.0.10

$$A \setminus cs \sqsupseteq B \setminus cs$$

Provided that $A \sqsupseteq B$.

Proof:

$$\begin{array}{ll}
& A \setminus cs \\
= & [3.5.55] \\
& R\left(\begin{array}{l} \exists\, s_t \bullet [\![A]\!]_{time}[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s, tr_t) \downarrow_t (Event - cs) \end{array}\right);\ Skip \\
\sqsupseteq & [R \text{ is monotonic, predicate calculus and C.0.1}] \\
& R\left(\begin{array}{l} \exists\, s_t \bullet [\![B]\!]_{time}[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s, tr_t) \downarrow_t (Event - cs) \end{array}\right);\ Skip \\
= & [3.5.55] \\
& B \setminus cs \qquad \square
\end{array}$$

Law C.0.11

$$(A \,\Box\, P) \sqsupseteq (B \,\Box\, P)$$

Provided that $A \sqsupseteq B$.

Proof:

$$\begin{array}{ll}
& A \,\Box\, P \\
= & [3.5.30] \\
& CSP2_t(ExtChoice1(A, P) \vee ExtChoice2(A, P));\ Skip
\end{array}$$

$$
\begin{array}{ll}
= & [3.5.27 \text{ and } 3.5.28] \\
CSP2_t \left(\begin{array}{l} (A \land P \land Stop) \lor \\ (DifDetected(A, P) \land (A \lor P)) \end{array} \right) & \\
= & [3.5.29] \\
CSP2_t \left(\begin{array}{l} (A \land P \land Stop) \lor \\ \left(\begin{array}{c} ((A \land P \land Stop) \lor Skip); \\ (\neg wait' \land tr_t' = tr_t) \lor \\ \left(\begin{array}{c} trace' \neq \langle\rangle \land \\ (fst(head(Difference(tr_t, tr_t'))) \neq \langle\rangle) \land \\ Expands(tr_t, tr_t') \end{array} \right) \end{array} \right) \\ \land (A \lor P) \end{array} \right) & \\
\sqsupseteq & [CSP2_t \text{ is monotonic,predicate calculus, C.0.1}] \\
= & [3.5.29] \\
CSP2_t \left(\begin{array}{l} (B \land P \land Stop) \lor \\ \left(\begin{array}{c} ((B \land P \land Stop) \lor Skip); \\ (\neg wait' \land tr_t' = tr_t) \lor \\ \left(\begin{array}{c} trace' \neq \langle\rangle \land \\ (fst(head(Difference(tr_t, tr_t'))) \neq \langle\rangle) \land \\ Expands(tr_t, tr_t') \end{array} \right) \end{array} \right) \\ \land (B \lor P) \end{array} \right) & \\
= & [3.5.29,3.5.27 \text{ and } 3.5.28] \\
CSP2_t(ExtChoice1(B, P) \lor ExtChoice2(B, P));\ Skip & \\
= & [3.5.30] \\
A \,\Box\, P & \Box
\end{array}
$$

Law C.0.12

$$(P \,\Box\, A) \sqsupseteq (P \,\Box\, B)$$

Provided that $A \sqsupseteq B$.

Proof:
From Law C.0.11 and internal choice is commutative from property 3.10 L3.□

Law C.0.13

$$(A \,[\![\, s_A \mid \{\!|\, cs \,|\!\} \mid s_P \,]\!]\, P) \sqsupseteq (B \,[\![\, s_B \mid \{\!|\, cs \,|\!\} \mid s_P \,]\!]\, P)$$

Provided that $A \sqsupseteq B$.

Proof:
First we recall the restrictions imposed on the parallel composition defined in Chapter 3 where the sets of s_A and s_P must be disjoint and represent the variables each action can change, we also request that the state variables *state* are divided between the s_A and s_P

from this restriction we can affirm $s_A = s_B$. From the definition of Parallel merge 3.5.32 we need to proof that

$$((A;\ U0(m))\|(P;\ U1(m)));\ TM(cs, s_A, s_P) \sqsupseteq ((B;\ U0(m))\|(P;\ U1(m)));\ TM(cs, s_B, s_P)$$

From the definition of parallel composition in the UTP

$$((A;\ U0(m)) \wedge (P;\ U1(m)));\ TM(cs, s_A, s_P) \sqsupseteq ((B;\ U0(m)) \wedge (P;\ U1(m)));\ TM(cs, s_B, s_P)$$

Notice that the above is true because, sequential composition and the state separation stated above ($s_A = s_B$). Further we shown in Chapter 3 that the parallel composition is commutative. Therefore, the parallel composition is monotonic $\Box$

APPENDIX D

PROOFS OF CHAPTER 3

In this appendix we present the proof of the properties of the concept of time trace introduced in 3.

D.1 FLAT RELATION

Property 3.1

L1. $Flat(t_a) \frown Flat(t_b) = Flat(t_a \frown t_b)$
Proof:
From definition of *Flat* □

L2. $Flat(t_a) = fst(head(t_a)) \frown Flat(tail(t_a))$
Provided that $t_a \geq 2$.
Proof:
From the definition of *Flat*, and the condition $t_a \geq 2$ is because *Flat* is not defined for the empty sequence □

L3. $Flat(t_a) = \langle\rangle \Leftrightarrow \forall\, i : 1..\#t_a \bullet fst(t_a)(i) =<>$ **Proof:**
From the definition of *Flat* □

L4. $t_a \leq t_b \Rightarrow Flat(t_a) \leq Flat(t_b)$ **Proof:**
By structural induction over time traces □

L5. $t_a = t_b \Rightarrow Flat(t_a) = Flat(t_b)$ **Proof:**
By structural induction over time traces □

D.2 EXPANDS

Property 3.2

L1. $Expands(t_a, t_b) \Rightarrow (Flat(t_a) \leq Flat(t_b))$

L2. $(t_a \leq t_b) \Rightarrow Expands(t_a, t_b)$

L3. $(t_a = t_b) \Rightarrow Expands(t_a, t_b)$

L4. $((Flat(t_a) = Flat(t_b)) \wedge Expands(t_a, t_b)) \Rightarrow (t_a = t_b)$

L5. $(t_a = t_b) \Rightarrow (last(t_a) = last(t_b))$

L6. $(t_a \leq t_b) \Rightarrow (last(t_a) = t_b(\#t_a))$

L7. $Expands(t_a, t_b) \wedge Expands(t_b, t_c) = Expands(t_a, t_c)$

L8. $Expands(\langle(\langle\rangle, ref)\rangle, t_b) = true$ for any arbitrary ref
Proof:

$Expands(\langle(\langle\rangle, ref)\rangle, t_b)$
$=$ [3.4.3]
$(front(\langle(\langle\rangle, ref)\rangle) \leq tr_b) \wedge (fst(last(\langle(\langle\rangle, ref)\rangle)) \leq fst(tr_b(\#\ \langle(\langle\rangle, ref)\rangle)))$
$=$ [Properties of *front*,*last* and *fst*]
$\langle\rangle \leq tr_b \wedge (\langle\rangle \leq fst(tr_b(1)))$
$=$ [Property of $\leq$]
$true \wedge true = true$ □

L9. $Expands(tr_a, tr_b) \Leftrightarrow Expand(\langle(\langle\rangle, ref)\rangle, dif(tr_b, tr_b))$
Proof:

$Expands(tr_a, tr_b) \Leftrightarrow Expand(\langle(\langle\rangle, ref)\rangle, dif(tr_b, tr_a))$
$=$ [assumption]
$true \Leftrightarrow Expand(\langle(\langle\rangle, ref)\rangle, dif(tr_b, tr_a))$
$=$ [Property 3.2 L8]
$true \Leftrightarrow true$
$=$ [Propositional Calculus]
$true$ □

D.3 DIFFERENCE

Property 3.3

L1. $dif(tr, tr) = \langle(\langle\rangle, snd(last(tr)))\rangle$ for any time trace tr
Proof:

$dif(tr, tr)$
$=$ [3.4.7]
$\langle(fst(head(tr - front(tr))) - fst(last(tr)),$
$snd(head(tr - front(tr))))\rangle \frown tail(tr - front(tr))$
$=$ [A.3.8 and A.3.9]
$\langle(fst(last(tr)) - fst(last(tr)), snd(last(tr)))\rangle \frown \langle\rangle$
$=$ [Property of $\frown$]
$\langle(\langle\rangle, snd(last(tr)))\rangle$ □

L2. $dif(tr, \langle(\langle\rangle, ref)) = tr$ for any arbitrary ref
Proof:

$dif(tr, \langle(\langle\rangle, ref)\rangle)$

$$
\begin{array}{ll}
= & [3.4.7] \\
\langle (fst(head(tr - front(\langle(\langle\rangle, ref)\rangle))) - fst(last(\langle(\langle\rangle, ref)\rangle))), & \\
snd(head(tr - front(\langle(\langle\rangle, ref)\rangle)))\rangle \frown tail(tr - front(\langle(\langle\rangle, ref)\rangle)) & \\
= & [\text{Property A.2 L1 and L2}] \\
\langle (fst(head(tr - \langle\rangle)) - fst(\langle(\langle\rangle, ref)\rangle), & \\
snd(head(tr - \langle\rangle))\rangle \frown tail(tr - \langle\rangle) & \\
= & [A.3.6] \\
\langle (fst(head(tr)) - fst(\langle(\langle\rangle, ref)\rangle), snd(head(tr)))\rangle \frown tail(tr) & \\
= & [\text{definition of } fst] \\
\langle (fst(head(tr)) - \langle\rangle, snd(head(tr)))\rangle \frown tail(tr) & \\
= & [A.3.6] \\
\langle (fst(head(tr)), snd(head(tr)))\rangle \frown tail(tr) & \\
= & [\text{definition of } fst \text{ and } snd] \\
\langle head(tr)\rangle \frown tail(tr) & \\
= & [\text{Property A.2 L5}] \\
tr & \square
\end{array}
$$

L3. $Flat(dif(tr'_t, tr_t)) = Flat(tr'_t) - Flat(tr_t)$

L4. $dif(t_a, t_b) = dif(t_c, t_b) \Leftrightarrow t_a = t_c$
Proof:

$$
\begin{array}{ll}
dif(t_a, t_b) = dif(t_c, t_b) & \\
= & [3.4.7] \\
\left(\begin{array}{l} \langle (fst(head(t_a - front(t_b))) - fst(last(t_b)), \\ snd(head(t_a - front(t_b))))\rangle \frown tail(t_a - front(t_b)) \end{array} \right) & \\
= & \\
\left(\begin{array}{l} \langle (fst(head(t_a - front(t_c))) - fst(last(t_c)), \\ snd(head(t_a - front(t_c))))\rangle \frown tail(t_a - front(t_c)) \end{array} \right) & \\
= & [\text{Sequence equivalence}] \\
\left(\begin{array}{l} \left\langle \left(\begin{array}{l} fst(head(t_a - front(t_b))) - fst(last(t_b)), \\ snd(head(t_a - front(t_b))) \end{array} \right) \right\rangle \\ = \\ \left\langle \left(\begin{array}{l} fst(head(t_a - front(t_c))) - fst(last(t_c)), \\ snd(head(t_a - front(t_c))) \end{array} \right) \right\rangle \end{array} \right) \wedge & \\
tail(t_a - front(t_b)) = tail(t_a - front(t_c)) & \\
= & [\text{tuple equivelence}]
\end{array}
$$

$$
\begin{array}{l}
fst(head(t_a - front(t_b))) - fst(last(t_b)) = fst(head(t_a - front(t_c))) - fst(last(t_c)) \wedge \\
snd(head(t_a - front(t_b))) = snd(head(t_a - front(t_c))) \wedge \\
tail(t_a - front(t_b)) = tail(t_a - front(t_c))
\end{array}
$$

$=$ [equivalence and subtraction]

$$
\begin{array}{l}
front(t_b) = front(t_c) \wedge \\
last(t_b) = last(t_c)
\end{array}
$$

$=$ [Property of sequence]

$t_b = t_c$ □

L5. $(\exists\, t \bullet dif(tr'_t, tr_t) = t) \Leftrightarrow Expands(tr_t, tr'_t)$
Proof:

$$\exists\, t \bullet dif(t_a, t_b) = t$$

$=$ [3.4.7]

$$\exists\, t \bullet \left(\begin{array}{l} \langle (fst(head(tr'_t - front(tr_t))) - fst(last(tr_t)), \\ snd(head(tr'_t - front(tr_t)))) \rangle \frown tail(tr'_t - front(tr_t)) \end{array} \right) = t$$

$=$ [Property of sequences]

$$\exists\, t \bullet \left(\begin{array}{l} head(t) = \left\langle \left(\begin{array}{l} fst(head(tr'_t - front(tr_t))) - fst(last(tr_t)), \\ snd(head(tr'_t - front(tr_t))) \end{array} \right) \right\rangle \wedge \\ tail(t) = tail(tr'_t - front(tr_t)) \end{array} \right)$$

$=$ [Property of sequences]

$$\exists\, t \bullet \left(\begin{array}{l} head(t) = \left\langle \left(\begin{array}{l} fst(tr'_t(\#tr_t)) - fst(last(tr_t)), \\ snd(head(tr'_t - front(tr_t))) \end{array} \right) \right\rangle \wedge \\ tail(t) = tail(tr'_t - front(tr_t)) \end{array} \right)$$

$\Leftrightarrow$ [Sequence subtraction]

$$fst(last(tr_t)) \leq fst(tr'_t(\#tr_t)) \wedge front(tr_t) \leq tr'_t$$

$=$ [3.4.3]

$Expands(tr_t, tr'_t)$ □

L6. $Flat(dif(t_a, t_b)) = \langle\rangle \Leftrightarrow \begin{array}{l} \forall\, i : 1..\#dif(t_a, t_b) \bullet \\ fst(dif(t_a, t_b)(i)) = \langle\rangle \end{array}$
Proof:from the definition of *dif* □

L7. $\begin{array}{l} \forall\, i : 1..\#dif(t_a, t_b) \bullet \\ snd(dif(t_a, t_b)(i)) = X \end{array} \Leftrightarrow \forall\, i : \#t_b..\#t_a \bullet snd(t_a(i)) = X$
Proof: from the definition of *dif* □

L8. $(\#tr'_t = \#tr_t \wedge Flat(tr'_t) = Flat(tr_t) \wedge Expands(tr_t, tr'_t)) = tr_t = tr'_t$
Proof: from the definition of *Flat* □

L9. $fst(hd(dif(tr'_t, tr_t))) \neq \langle\rangle \Rightarrow trace' \neq \langle\rangle$
Proof:

$$
\begin{array}{ll}
& fst(hd(dif(tr'_t, tr_t))) \neq \langle\rangle \\
= & \text{[definition of } hd\text{]} \\
& fst(dif(tr'_t, tr_t)(1)) \neq \langle\rangle \\
= & \text{[Property of negation]} \\
& \neg(fst(dif(tr'_t, tr_t)(1)) = \langle\rangle) \\
\Rightarrow & \text{[Predicate calculus]} \\
& \neg(\exists\, i : 1..\#dif(tr'_t, tr_t) \bullet fst(dif(tr'_t, tr_t)(i)) = \langle\rangle) \\
= & \text{[Predicate calculus]} \\
& \forall\, i : 1..\#dif(tr'_t, tr_t) \bullet fst(dif(tr'_t, tr_t)(i)) \neq \langle\rangle \\
= & \text{[Property 3.3 L7]} \\
& Flat(dif(tr'_t, tr_t)) \neq \langle\rangle \\
= & \text{[Property 3.3 L3]} \\
& Flat(tr'_t) - Flat(tr_t) \neq \langle\rangle \\
= & \text{[3.3.1]} \\
& trace' \neq \langle\rangle \qquad \square
\end{array}
$$

L10. $dif(dif(t_a, t_c), dif(t_b, t_c)) = dif(t_a, t_b)$ Provided that $Expands(t_a, t_c) \wedge Expands(t_b, t_c) \wedge Expands(t_a, t_b)$ is true.

D.4 SEQUENTIAL COMPOSITION

Property 3.6

L1. $Stop;\ A = Stop$
Provided that A is healthy
Proof:

$$
\begin{array}{ll}
& Stop;\ A \\
= & \text{[3.5.3]} \\
& CSP1_t(R3_t(ok' \wedge wait' \wedge trace' = \langle\rangle));\ A \\
= & \text{[Assumption and Property B.4 L9]} \\
& CSP1_t((R3_t(ok' \wedge wait' \wedge trace' = \langle\rangle));\ A) \\
= & \text{[3.4.10]} \\
& CSP1_t\big(\big(\ (\Pi_t \lhd wait \rhd (ok' \wedge wait' \wedge trace' = \langle\rangle))\ \big);\ A\big) \\
= & \text{[Predicate calculus]}
\end{array}
$$

$$
\begin{array}{ll}
& CSP1_t\left(\left(\ \Pi_t \lhd wait \rhd (ok' \wedge wait' \wedge trace' = \langle\rangle)\ \right);\ A\right) \\
= & \text{[Property 3.6 L5]} \\
& CSP1_t\left(\begin{array}{l}(\Pi_t;\ A) \\ \lhd\ wait\ \rhd \\ ((ok' \wedge wait' \wedge trace' = \langle\rangle);\ A)\end{array}\right) \\
= & \text{[Lemma 3.4]} \\
& CSP1_t(A \lhd wait \rhd ((ok' \wedge wait' \wedge trace' = \langle\rangle);\ A)) \\
= & \text{[Assumption } A \text{ is } R3_t \text{ healthy]} \\
& CSP1_t\left(\begin{array}{l}(\Pi_t \lhd wait \rhd A) \\ \lhd\ wait\ \rhd \\ ((ok' \wedge wait' \wedge trace' = \langle\rangle);\ A)\end{array}\right) \\
= & \text{[Property 3.7 L6]} \\
& CSP1_t(\Pi_t \lhd wait \rhd ((ok' \wedge wait' \wedge trace' = \langle\rangle);\ A)) \\
= & \text{[Relational calculus]} \\
& CSP1_t(\Pi_t \lhd wait \rhd ((ok' \wedge trace' = \langle\rangle);\ (wait \wedge A))) \\
= & \text{[Assumption } A \text{ is } R3_t \text{ healthy]} \\
& CSP1_t\left(\Pi_t \lhd wait \rhd \left(\begin{array}{l}(ok' \wedge trace' = \langle\rangle); \\ (wait \wedge (\Pi_t \lhd wait \rhd A))\end{array}\right)\right) \\
= & \text{[Property 3.7 L5]} \\
& CSP1_t\left(\Pi_t \lhd wait \rhd \left(\begin{array}{l}(ok' \wedge trace' = \langle\rangle); \\ (wait \wedge \Pi_t)\end{array}\right)\right) \\
= & \text{[Relational calculus]} \\
& CSP1_t\left(\Pi_t \lhd wait \rhd \left(\begin{array}{l}(wait' \wedge trace' = \langle\rangle); \\ (ok \wedge \Pi_t)\end{array}\right)\right) \\
= & \text{[Lemma 3.1]} \\
& CSP1_t\left(\Pi_t \lhd wait \rhd \left(\begin{array}{l}(wait' \wedge trace' = \langle\rangle); \\ (ok \wedge ok' \wedge wait = wait' \wedge tr = tr' \wedge state = state')\end{array}\right)\right) \\
= & \text{[Relational calculus]} \\
& CSP1_t(\Pi_t \lhd wait \rhd ((ok' \wedge wait' \wedge trace' = \langle\rangle))) \\
= & \text{[3.4.10]} \\
& CSP1_t(R3_t(ok' \wedge wait' \wedge trace' = \langle\rangle)) \\
= & \text{[3.5.5]} \\
& Stop \qquad \square
\end{array}
$$

L2. $(x := e);\ (x := f(x)) = x := f(e)$
where f(x) is a function of x.
Proof:

$$
\begin{array}{ll}
& (x := e);\ (x := f(x)) \\
= & [3.5.7] \\
& CSP1\left(R_t\left(\begin{array}{c} ok = ok' \wedge \\ wait = wait' \wedge \\ tr'_t = tr_t \wedge \\ state' = state \oplus \{x \mapsto val(e, state)\} \end{array}\right)\right); \\
& CSP1\left(R_t\left(\begin{array}{c} ok = ok' \wedge \\ wait = wait' \wedge \\ tr'_t = tr_t \wedge \\ state' = state \oplus \{x \mapsto val(f(x), state)\} \end{array}\right)\right) \\
= & [\text{Properties B.1 L6, B.2 L7, B.3 L8 and B.4 L9}] \\
& CSP1\left(R_t\left(\left(\begin{array}{c} ok = ok' \wedge \\ wait = wait' \wedge \\ tr'_t = tr_t \wedge \\ state' = state \oplus \{x \mapsto val(e, state)\} \end{array}\right); \left(\begin{array}{c} ok = ok' \wedge \\ wait = wait' \wedge \\ tr'_t = tr_t \wedge \\ state' = state \oplus \{x \mapsto val(f(x), state)\} \end{array}\right)\right)\right) \\
= & [3.5.21] \\
& CSP1\left(R_t\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\ \left(\begin{array}{c} ok = ok_o \wedge \\ wait = waito \wedge \\ tr_t = tr_o \wedge \\ stateo = state \oplus \{x \mapsto val(e, state)\} \end{array}\right) \wedge \\ \left(\begin{array}{c} ok_o = ok' \wedge \\ wait_o = wait' \wedge \\ tr_o = tr'_t \wedge \\ state' = state_o \oplus \{x \mapsto val(f(x), state_o)\} \end{array}\right) \end{array}\right)\right) \\
= & [\text{Substitution}] \\
& CSP1\left(R_t\left(\begin{array}{c} ok = ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge \\ state' = state \oplus \{x \mapsto val(e, state)\} \oplus \\ \{x \mapsto val(f(x), state \oplus \{x \mapsto val(e, state)\})\} \end{array}\right)\right) \\
= & [\text{Substitute } x \text{ in } state \text{ for } val(e, state)] \\
& CSP1\left(R_t\left(\begin{array}{c} ok = ok' \wedge wait = wait' \wedge tr_t = tr'_t \wedge \\ state' = state \oplus \{x \mapsto val(f(e), state)\} \end{array}\right)\right) \\
= & [3.5.7] \\
& x := f(e) \qquad \square
\end{array}
$$

L3. $Wait\ \mathsf{n};\ Wait\ \mathsf{m} = Wait\ \mathsf{n} + \mathsf{m}$

Proof:

$$
\begin{array}{l}
Wait\ \mathsf{n};\ Wait\ \mathsf{m} \\
= \hfill [3.5.8] \\
CSP1_t\left(R_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land (\#tr_t' - \#tr_t) < n \land \\ trace' = \langle\rangle\end{array}\right) \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land (\#tr_t' - \#tr_t) = n \land \\ trace' = \langle\rangle \land state' = state\end{array}\right)\end{array}\right)\right); \\
CSP1_t\left(R_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land (\#tr_t' - \#tr_t) < m \land \\ trace' = \langle\rangle\end{array}\right) \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land (\#tr_t' - \#tr_t) = m \land \\ trace' = \langle\rangle \land state' = state\end{array}\right)\end{array}\right)\right) \\
= \hfill [\text{Properties B.1 L6, B.2 L7, B.3 L8 and B.4 L9}] \\
CSP1_t\left(R_t\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \\ (\#tr_t' - \#tr_t) < n \land \\ trace' = \langle\rangle\end{array}\right) \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \\ (\#tr_t' - \#tr_t) = n \land \\ trace' = \langle\rangle \land \\ state' = state\end{array}\right)\end{array}\right); \\ \left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \\ (\#tr_t' - \#tr_t) < m \land \\ trace' = \langle\rangle\end{array}\right) \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \\ (\#tr_t' - \#tr_t) = m \land \\ trace' = \langle\rangle \land \\ state' = state\end{array}\right)\end{array}\right)\end{array}\right)\right) \\
= \hfill [\text{Properties B.3 L3,L4 and B.4 L5}] \\
CSP1_t\left(R12_t\left(\begin{array}{l}R3_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \\ (\#tr_t' - \#tr_t) < n \land \\ trace' = \langle\rangle\end{array}\right) \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \\ (\#tr_t' - \#tr_t) = n \land \\ trace' = \langle\rangle \land \\ state' = state\end{array}\right)\end{array}\right); \\ R3_t\left(\begin{array}{l}\left(\begin{array}{l}ok' \land wait' \land \\ (\#tr_t' - \#tr_t) < m \land \\ trace' = \langle\rangle\end{array}\right) \lor \\ \left(\begin{array}{l}ok' \land \neg wait' \land \\ (\#tr_t' - \#tr_t) = m \land \\ trace' = \langle\rangle \land \\ state' = state\end{array}\right)\end{array}\right)\end{array}\right)\right)
\end{array}
$$

$$
\begin{array}{ll}
= & [3.4.10] \\
& CSP1_t \left(R12_t \left(\begin{array}{l}
\left(\begin{array}{l} \Pi_t \lhd wait \rhd \\ \left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \\ \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \end{array} \right) \end{array} \right) ; \\
\left(\begin{array}{l} \Pi_t \lhd wait \rhd \\ \left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \\ \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \end{array} \right) \end{array} \right)
\end{array} \right) \right) \\
= & [\text{Property 3.6 L5}] \\
& CSP1_t \left(R_t \left(\begin{array}{l}
\left(\begin{array}{l} \Pi_t; \\ \left(\begin{array}{l} \Pi_t \lhd wait \rhd \\ \left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \\ \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \\
\lhd wait \rhd \\
\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \\ \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \end{array} \right) ; \\
\left(\begin{array}{l} \Pi_t \lhd wait \rhd \\ \left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \\ \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \end{array} \right) \end{array} \right)
\end{array} \right) \right)
\end{array}
$$

$$
\begin{array}{ll}
= & \text{[Lemma 3.5]} \\
& CSP1_t \left(R_t \left(\begin{array}{l}
\left(\begin{array}{l}
\Pi_t \lhd \mathit{wait} \rhd \\
\left(\left(\begin{array}{l} ok' \wedge \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ \mathit{trace}' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ \mathit{trace}' = \langle\rangle \wedge \\ \mathit{state}' = \mathit{state} \end{array} \right) \right)
\end{array} \right) \\
\lhd \mathit{wait} \rhd \\
\left(\begin{array}{l}
\left(\left(\begin{array}{l} ok' \wedge \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ \mathit{trace}' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ \mathit{trace}' = \langle\rangle \wedge \\ \mathit{state}' = \mathit{state} \end{array} \right) \right) \\
; \\
\left(\begin{array}{l}
\Pi_t \lhd \mathit{wait} \rhd \\
\left(\left(\begin{array}{l} ok' \wedge \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ \mathit{trace}' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ \mathit{trace}' = \langle\rangle \wedge \\ \mathit{state}' = \mathit{state} \end{array} \right) \right)
\end{array} \right)
\end{array} \right) \right) \\
= & \text{[Property 3.7 L6]} \\
& CSP1_t \left(R_t \left(\begin{array}{l}
\left(\begin{array}{l}
\Pi_t \lhd \mathit{wait} \rhd \\
\left(\left(\begin{array}{l} ok' \wedge \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ \mathit{trace}' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ \mathit{trace}' = \langle\rangle \wedge \\ \mathit{state}' = \mathit{state} \end{array} \right) \right) \\
; \\
\left(\begin{array}{l}
\Pi_t \lhd \mathit{wait} \rhd \\
\left(\left(\begin{array}{l} ok' \wedge \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ \mathit{trace}' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg \mathit{wait}' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ \mathit{trace}' = \langle\rangle \wedge \\ \mathit{state}' = \mathit{state} \end{array} \right) \right)
\end{array} \right)
\end{array} \right) \right) \\
= & \text{[3.4.10 and property B.3 L1]}
\end{array}
$$

$$
CSP1_t \left(R_t \left(\left(\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \right) ; \left(\Pi_t \lhd wait \rhd \left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \right) \right) \right) \right) \right)
$$

$=$ [3.5.22 and property 3.9 L6]

$$
CSP1_t \left(R_t \left(\left(\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \right) ; \left(\Pi_t \wedge wait \right) \right) \vee \left(\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \right) ; \left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \right) \wedge \neg wait \right) \right) \right) \right) \right)
$$

$=$ [Relational calculus]

$$
CSP1_t \left(R_t \left(\begin{array}{l}
\left(\begin{array}{l} \Pi_t \\ \wedge\ wait' \end{array} \right) ; \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) \leq n \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \\
\vee \\
\left(\begin{array}{l} \Pi_t \\ \wedge\ \neg wait' \end{array} \right) ; \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) \\
\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right)
\end{array} \right) \right)
$$

$=$ [Lemma 3.4 and propositional calculus]

$$
CSP1_t \left(R_t \left(\begin{array}{l}
\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right) ; \\
\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < m \wedge \\ trace' = \langle\rangle \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array} \right)
\end{array} \right) \right)
$$

$=$ [3.5.21]

$$
CSP1_t\left(\begin{array}{l} R_t\left(\begin{array}{l}
\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array}\right) \vee \\
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\
\left(\begin{array}{l} ok_o \wedge \neg wait_o \wedge \\ (\#tr_o - \#tr_t) = n \wedge \\ Flat(tr_o) - \mathit{flat}(tr_t) = \langle\rangle \wedge \\ state_o = state \end{array}\right) \wedge \\
\left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_o) < m \wedge \\ Flat(tr'_t) - Flat(tr_o) = \langle\rangle \end{array}\right) \vee \\
\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_o) = m \wedge \\ Flat(tr'_t) - Flat(tr_o) = \langle\rangle \wedge \\ state' = state_o \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right)
$$

$=$ [propositional calculus]

$$
CSP1_t\left(\begin{array}{l} R_t\left(\begin{array}{l}
\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array}\right) \vee \\
\left(\begin{array}{l} \exists\, ok_o, wait_o, tr_o, state_o \bullet \\
\left(\begin{array}{l} \left(\begin{array}{l} ok_o \wedge \neg wait_o \wedge \\ (\#tr_o - \#tr_t) = n \wedge \\ Flat(tr_o) - \mathit{flat}(tr_t) = \langle\rangle \wedge \\ state_o = state \end{array}\right) \wedge \\
\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_o) < m \wedge \\ Flat(tr'_t) - Flat(tr_o) = \langle\rangle \end{array}\right) \end{array}\right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} ok_o \wedge \neg wait_o \wedge \\ (\#tr_o - \#tr_t) = n \wedge \\ Flat(tr_o) - \mathit{flat}(tr_t) = \langle\rangle \wedge \\ state_o = state \end{array}\right) \wedge \\
\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_o) = m \wedge \\ Flat(tr'_t) - Flat(tr_o) = \langle\rangle \wedge \\ state' = state_o \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right)
$$

$=$ [Substitution and condition imposed by $R1_t$]

$$
CSP1_t\left(R_t\left(\begin{array}{l}
\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array}\right) \vee \\
\exists\, ok_o, wait_o \bullet \\
\left(\left(\begin{array}{l} ok_o \wedge \neg wait_o \wedge \\ ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n + m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right)\right) \vee \\
\left(\begin{array}{l} ok_o \wedge \neg wait_o \wedge \\ ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n + m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right)
\end{array}\right)\right)
$$

$=$ [ok_o and $wait_o$ not free]

$$
CSP1_t\left(R_t\left(\begin{array}{l}
\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n \wedge \\ trace' = \langle\rangle \end{array}\right) \vee \\
\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n + m \wedge \\ trace' = \langle\rangle \end{array}\right)\right) \vee \\
\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right)
\end{array}\right)\right)
$$

$=$ [Propositional calculus]

$$
CSP1_t\left(R_t\left(\begin{array}{l}
\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) < n + m \wedge \\ trace' = \langle\rangle \end{array}\right)\right) \vee \\
\left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = m \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right)
\end{array}\right)\right)
$$

$=$ [3.5.8]

$Wait\ \ \mathsf{n} + \mathsf{m}$ □

L4. $A;\ (B;\ C) = (A;\ B);\ C$

Proof:

$A;\ (B;\ C)$ [3.5.21]

$= \exists\, obs_1 \bullet A[obs_1/obs'] \wedge (B;\ C)[obs_1/obs]$ [3.5.21]

$$= \left(\begin{array}{c} \exists\, obs_1 \bullet A[obs_1/obs'] \wedge \\ \left(\begin{array}{c} \exists\, obs_2 \bullet B[obs_2/obs'] \wedge \\ C[obs_2/obs] \end{array} \right)[obs_1/obs] \end{array} \right) \quad \left[\begin{array}{c} obs_1 \neq obs_2 \text{ and} \\ \text{and predicate calculus} \end{array} \right]$$

$$= \exists\, obs_1, obs_2 \bullet \left(\begin{array}{c} A[obs_1/obs'] \wedge \\ B[obs_1/obs][obs_2/obs'] \wedge \\ C[obs_2/obs] \end{array} \right) \qquad [obs_2 \text{ not free in } A]$$

$$= \exists\, obs_1, obs_2 \bullet \left(\begin{array}{c} (A[obs_1/obs'] \wedge B[obs_1/obs])[obs_2/obs'] \wedge \\ C[obs_2/obs] \end{array} \right) \qquad [3.5.21]$$

$$= \exists\, obs_2 \bullet \left(\begin{array}{c} (A;\ B)[obs_2/obs'] \wedge \\ C[obs_2/obs] \end{array} \right) \qquad [3.5.21]$$

$$= (A;\ B);\ C \qquad \square$$

L5. $(A \lhd b \rhd B);\ C = (A;\ C) \lhd b \rhd (B;\ C)$

Proof:

$$\begin{array}{ll}
(A \lhd b \rhd B);\ C & [3.5.21] \\
= \exists\, obs_o \bullet (A \lhd b \rhd B)[obs_o/obs'] \wedge C[obs_o/obs] & [3.5.22] \\
= \exists\, obs_o \bullet \left(\begin{array}{c} \left(\begin{array}{c} (A \wedge b) \vee \\ (B \wedge \neg b) \end{array} \right)[obs_o/obs'] \wedge \\ C[obs_o/obs] \end{array} \right) & [\text{definition of substituition}] \\
= \exists\, obs_o \bullet \left(\begin{array}{c} (A \wedge b)[obs_o/obs'] \vee \\ (B \wedge \neg b)[obs_o/obs'] \end{array} \right) \wedge C[obs_o/obs] & [obs' \text{ in not free in } b] \\
= \exists\, obs_o \bullet \left(\begin{array}{c} (A[obs_o/obs'] \wedge b) \vee \\ (B[obs_o/obs'] \wedge \neg b) \end{array} \right) \wedge C[obs_o/obs] & [\text{predicate calcuals}] \\
= \exists\, obs_o \bullet \left(\begin{array}{c} (A[obs_o/obs'] \wedge C[obs_o/obs] \wedge b) \vee \\ (B[obs_o/obs'] \wedge C[obs_o/obs] \wedge \neg b) \end{array} \right) & [\text{predicate calcuals}] \\
= \left(\begin{array}{c} (\exists\, obs_o \bullet (A[obs_o/obs'] \wedge C[obs_o/obs]) \wedge b) \vee \\ (\exists\, obs_o \bullet (B[obs_o/obs'] \wedge C[obs_o/obs]) \wedge \neg b) \end{array} \right) & [3.5.21] \\
= ((A;\ C) \wedge b) \vee ((B;\ C) \wedge \neg b) & [3.5.22] \\
= (A;\ C) \lhd b \rhd (B;\ C) \qquad \square &
\end{array}$$

D.5 CONDITIONAL CHOICE

Property 3.7

L1. $A \lhd b \rhd A = A$

Proof:

$$\begin{array}{ll}
A \lhd b \rhd A & [3.5.22] \\
= (b \wedge A) \vee (\neg b \wedge A) & [\text{propositional calculus}] \\
= (b \vee \neg b) \wedge A & [\text{propositional calculus}] \\
= (true) \wedge A & [\text{propositional calculus}]
\end{array}$$

$= A \qquad \square$

L2. $A \lhd b \rhd B = B \lhd \neg b \rhd A$
Proof:

$$
\begin{array}{ll}
A \lhd b \rhd B & [3.5.22] \\
= (b \wedge A) \vee (\neg b \wedge B) & [\text{propositional calculus}] \\
= (\neg b \wedge B) \vee (\neg\neg b \wedge A) & [3.5.22] \\
= B \lhd \neg b \rhd A \qquad \square &
\end{array}
$$

L3. $(A \lhd b \rhd B) \lhd c \rhd C = A \lhd (b \wedge c) \rhd (B \lhd c \rhd C)$
Proof:

$$
\begin{array}{ll}
A \lhd (b \wedge c) \rhd (B \lhd c \rhd C) & [3.5.22] \\
= A \lhd (b \wedge c) \rhd ((c \wedge B) \vee (\neg c \wedge C)) & [3.5.22] \\
= \begin{array}{l} (b \wedge c \wedge A) \vee \\ ((\neg b \vee \neg c) \wedge ((c \wedge B) \vee (\neg c \wedge C))) \end{array} & [\text{propositional calculus}] \\
= \begin{array}{l} (b \wedge c \wedge A) \vee \\ ((\neg b \vee \neg c) \wedge (c \wedge B)) \vee \\ ((\neg b \vee \neg c) \wedge (\neg c \wedge C)) \end{array} & [\text{propositional calculus}] \\
= \begin{array}{l} (b \wedge c \wedge A) \vee \\ (\neg b \wedge c \wedge B) \vee \\ (\neg c \wedge c \wedge B) \vee \\ (\neg b \wedge \neg c \wedge C) \vee \\ (\neg c \wedge \neg c \wedge C) \end{array} & [\text{propositional calculus}] \\
= \begin{array}{l} (b \wedge c \wedge A) \vee \\ (\neg b \wedge c \wedge B) \vee \\ (\neg b \wedge \neg c \wedge C) \vee \\ (\neg c \wedge C) \end{array} & [\text{propositional calculus}] \\
= \begin{array}{l} (b \wedge c \wedge A) \vee \\ (\neg b \wedge c \wedge B) \vee \\ (\neg c \wedge C) \end{array} & [\text{propositional calculus}] \\
= (((b \wedge A) \vee (\neg b \wedge B)) \wedge c) \vee (\neg c \wedge C) & [3.5.22] \\
= ((b \wedge A) \vee (\neg b \wedge B)) \lhd c \rhd C & [3.5.22] \\
= (A \lhd b \rhd B) \lhd c \rhd C \qquad \square &
\end{array}
$$

L4. $A \lhd b \rhd (B \lhd c \rhd C) = (A \lhd b \rhd B) \lhd c \rhd (A \lhd b \rhd C)$
Proof:

$$
\begin{array}{ll}
(A \lhd b \rhd B) \lhd c \rhd (A \lhd b \rhd C) & [3.5.22] \\
= ((b \wedge A) \vee (\neg b \wedge B)) \lhd c \rhd ((b \wedge A) \vee (\neg b \wedge C)) & [3.5.22]
\end{array}
$$

$$
\begin{array}{ll}
= & \begin{array}{l}(b \wedge c \wedge A) \vee \\ (\neg b \wedge c \wedge B) \vee \\ (b \wedge \neg c \wedge A) \vee \\ (\neg b \wedge \neg c \wedge C)\end{array} & \text{[propositional calculus]} \\
= & \begin{array}{l}((b \wedge A) \wedge (c \vee \neg c)) \vee \\ (\neg b \wedge ((c \wedge B) \vee (\neg c \wedge C)))\end{array} & \text{[propositional calculus]} \\
= & (b \wedge A) \vee (\neg b(c \wedge B) \vee (\neg c \wedge C)) & \text{[3.5.22]} \\
= & (b \wedge A) \vee (\neg b(B \lhd c \rhd C)) & \text{[3.5.22]} \\
= & A \lhd b \rhd (B \lhd c \rhd C) & \Box
\end{array}
$$

L5. $A \lhd \mathit{true} \rhd B = A = B \lhd \mathit{false} \rhd A$
Proof:

$$
\begin{array}{ll}
& A \lhd \mathit{true} \rhd B & \text{[3.5.22]} \\
= & (\mathit{true} \wedge A) \vee (\neg \mathit{true} \wedge B) & \text{[propositional calculus]} \\
= & (\mathit{true} \wedge A) \vee (\mathit{false} \wedge B) & \text{[propositional calculus]} \\
= & A \vee \mathit{false} & \text{[propositional calculus]} \\
= & A & \Box
\end{array}
$$

in a similar manner

$$
\begin{array}{ll}
& B \lhd \mathit{false} \rhd A & \text{[3.5.22]} \\
= & (\mathit{false} \wedge B) \vee (\neg \mathit{false} \wedge A) & \text{[propositional calculus]} \\
= & (\mathit{false} \wedge B) \vee (\mathit{true} \wedge A) & \text{[propositional calculus]} \\
= & \mathit{false} \vee A & \text{[propositional calculus]} \\
= & A & \Box
\end{array}
$$

L6. $A \lhd b \rhd (B \lhd b \rhd C) = A \lhd b \rhd C$
Proof:

$$
\begin{array}{ll}
& A \lhd b \rhd (B \lhd b \rhd C) & \text{[3.5.22]} \\
= & (b \wedge A) \vee (\neg b \wedge (B \lhd b \rhd C)) & \text{[3.5.22]} \\
= & (b \wedge A) \vee (\neg b \wedge b \wedge B) \vee (\neg b \wedge \neg b \wedge C) & \text{[propositional calculus]} \\
= & (b \wedge A) \vee (\neg b \wedge C) & \text{[3.5.22]} \\
= & A \lhd b \rhd C & \Box
\end{array}
$$

L7. $A \lhd b \rhd (A \lhd c \rhd B) = A \lhd (b \vee c) \rhd B$
Proof:

$$
\begin{array}{ll}
& A \lhd b \rhd (A \lhd c \rhd B) & \text{[Property 3.7 L2]}
\end{array}
$$

$= (B \lhd \neg c \rhd A) \lhd \neg b \rhd A$ [Property 3.7 L3]
$= B \lhd (\neg c \wedge \neg b) \rhd (A \lhd \neg b \rhd A)$ [Property 3.7 L1]
$= B \lhd (\neg c \wedge \neg b) \rhd A$ [propositional calculus]
$= B \lhd (\neg(c \vee b)) \rhd A$ [Property 3.7 L2]
$= A \lhd (b \vee c) \rhd B$ □

L8. $((A \sqcap B) \lhd b \rhd C) = (A \lhd b \rhd C) \sqcap (B \lhd b \rhd C)$
Proof:

$((A \sqcap B) \lhd b \rhd C)$ [3.5.24]
$= ((A \vee B) \lhd b \rhd C)$ [3.5.22]
$= (((A \vee B) \wedge b) \vee (C \wedge \neg b))$ [propositional calculus]
$= ((A \wedge b) \vee (B \wedge b)) \vee (C \wedge \neg b)$ [propositional calculus]
$= ((A \wedge b) \vee (C \wedge \neg b)) \vee ((B \wedge b) \vee (C \wedge \neg b))$ [3.5.22]
$= (A \lhd b \rhd C) \vee (B \lhd b \rhd C)$ [3.5.24]
$= (A \lhd b \rhd C) \sqcap (B \lhd b \rhd C)$ □

D.6 GUARDED ACTION

Property 3.8

L1. $false \& A = Stop$
Proof:

$false \& A$ [3.5.23]
$= (A \lhd false \rhd Stop)$ [Property 3.7 L5]
$= Stop$ □

L2. $true \& A = A$
Proof:

$true \& A$ [3.5.23]
$= (A \lhd true \rhd Stop)$ [Property 3.7 L5]
$= A$ □

L3. $b \& Stop = Stop$
Proof:

$b \& Stop$ [3.5.23]
$= (Stop \lhd b \rhd Stop)$ [Property 3.7 L1]
$= Stop$ □

L4. $b\&(q\&A) = (b \wedge q)\&A$
Proof:

$$
\begin{array}{ll}
& b\&(q\&A) \qquad [3.5.23] \\
= & ((q\&A) \lhd b \rhd Stop) \qquad [3.5.23] \\
= & ((A \lhd q \rhd Stop) \lhd b \rhd Stop) \qquad [\text{Property 3.7 L3}] \\
= & (A \lhd (b \wedge q) \rhd (Stop \lhd b \rhd Stop)) \qquad [\text{Property 3.7 L1}] \\
= & (A \lhd (b \wedge q) \rhd Stop) \qquad [3.5.23] \\
= & (b \wedge q)\&A \qquad \Box
\end{array}
$$

L5. $b\&(A \sqcap B) = (b\&A) \sqcap (b\&B)$
Proof:

$$
\begin{array}{ll}
& b\&(A \sqcap B) \qquad [3.5.23] \\
= & ((A \sqcap B) \lhd b \rhd Stop) \qquad [\text{Property 3.7 L8}] \\
= & (A \lhd b \rhd Stop) \sqcap (B \lhd b \rhd Stop) \qquad [3.5.23] \\
= & (b\&A) \sqcap (b\&B) \qquad \Box
\end{array}
$$

L6. $b\&(A;\ B) = (b\&A);\ B$
Proof:

$$
\begin{array}{ll}
& b\&(A;\ B) \qquad [3.5.23] \\
= & ((A;\ B) \lhd b \rhd Stop) \qquad [\text{Property 3.6 L2}] \\
= & ((A;\ B) \lhd b \rhd (Stop;\ B)) \qquad [\text{Property 3.6 L5}] \\
= & (A \lhd b \rhd Stop);\ B \qquad [3.5.23] \\
= & (b\&A);\ B \qquad \Box
\end{array}
$$

L7. $b\&A = (b\&Skip);\ A$
Proof:

$$
\begin{array}{ll}
& b\&A \qquad [\text{Property 3.6 L1}] \\
= & b\&(Skip;\ A) \qquad [\text{Property 3.8 L6}] \\
= & (b\&Skip);\ A \Box
\end{array}
$$

D.7 INTERNAL CHOICE

Property 3.9

L1. $Chaos \sqcap A = Chaos$
Proof:
We make the assumption that A satisfies the healthiness condition R_t.

$$
\begin{array}{ll}
& Chaos \sqcap A \qquad [3.5.6]
\end{array}
$$

$= R_t(true) \sqcap A$ [3.5.24]
$= R_t(true) \vee A$ [from the assumption]
$= R_t(true) \vee R_t(A)$ [R_t Closed under $\vee$]
$= R_t(true \vee A)$ [propositional calculus]
$= R_t(true)$ [3.5.6]
$Chaos$ $\Box$

L2. $A \sqcap A = A$
Proof:

$A \sqcap A$ [3.5.24]
$= A \vee A$ [propositional calculus]
$= A$ $\Box$

L3. $A \sqcap B = B \sqcap A$
Proof:

$A \sqcap B$ [3.5.24]
$= A \vee B$ [propositional calculus]
$= B \vee A$ [3.5.24]
$= B \sqcap A \Box$

L4. $A \sqcap (B \sqcap C) = (A \sqcap B) \sqcap C$
Proof:

$A \sqcap (B \sqcap C)$ [3.5.24]
$= A \vee (B \vee C)$ [propositional calculus]
$= (A \vee B) \vee C$ [3.5.24]
$= (A \sqcap B) \sqcap C \Box$

L5. $(A \sqcap B);\ C = (A;\ C) \sqcap (B;\ C)$
Proof:

$(A \sqcap B);\ C$ [3.5.24]
$= (A \vee B);\ C)$ [3.5.21]
$= \exists\, obs_o \bullet (A \vee B)[obs_o/obs'] \wedge C[obs_o/obs]$ [propositional calculus]
$= \exists\, obs_o \bullet \left(\begin{array}{c} (A[obs_o/obs'] \vee B[obs_o/obs']) \wedge \\ C[obs_o/obs] \end{array} \right)$ [propositional calculus]
$= \exists\, obs_o \bullet \left(\begin{array}{c} (A[obs_o/obs'] \wedge C[obs_o/obs]) \vee \\ (B[obs_o/obs'] \wedge C[obs_o/obs]) \end{array} \right)$ [predicate calculus]

$$= \left(\begin{array}{c} \exists\, obs_o \bullet (A[obs_o/obs'] \wedge C[obs_o/obs]) \vee \\ \exists\, obs_o \bullet (B[obs_o/obs'] \wedge C[obs_o/obs]) \end{array} \right) \quad [3.5.21]$$

$$= (A;\ C) \vee (B;\ C) \quad [3.5.24]$$

$$= (A;\ C) \sqcap (B;\ C) \quad \Box$$

L6. $A;\ (B \sqcap C) = (A;\ C) \sqcap (A;\ B)$
Proof:

$$A;\ (B \sqcap C) \quad [3.5.24]$$

$$= A;\ (B \vee C) \quad [3.5.21]$$

$$= \exists\, obs_o \bullet A[obs_o/obs'] \wedge (B \vee C)[obs_o/obs] \quad [\text{propositional calculus}]$$

$$= \exists\, obs_o \bullet \left(\begin{array}{c} (A[obs_o/obs'] \wedge B[obs_o/obs]) \vee \\ (A[obs_o/obs'] \wedge C[obs_o/obs]) \end{array} \right) \quad [\text{predicate calculus}]$$

$$= \left(\begin{array}{c} \exists\, obs_o \bullet (A[obs_o/obs'] \wedge B[obs_o/obs]) \vee \\ \exists\, obs_o \bullet (A[obs_o/obs'] \wedge C[obs_o/obs]) \end{array} \right) \quad [3.5.21]$$

$$= (A;\ B) \vee (A;\ C) \quad [3.5.24]$$

$$= (A;\ B) \sqcap (A;\ C) \quad \Box$$

L7. $A \lhd b \rhd (B \sqcap C) = (A \lhd b \rhd B) \sqcap (A \lhd b \rhd C)$
Proof:

$$A \lhd b \rhd (B \sqcap C) \quad [3.5.24]$$

$$= A \lhd b \rhd (B \vee C) \quad [3.5.22]$$

$$= (A \wedge b) \vee ((B \vee C) \wedge \neg b) \quad [\text{propositional calculus}]$$

$$= ((A \wedge b) \vee (B \wedge \neg b)) \vee ((A \wedge b) \vee (C \wedge \neg b)) \quad [3.5.22]$$

$$= (A \lhd b \rhd B) \vee (A \lhd b \rhd C) \quad [3.5.24]$$

$$= (A \lhd b \rhd B) \sqcap (A \lhd b \rhd C) \quad \Box$$

L8. $A \sqcap (B \lhd b \rhd C) = (A \sqcap B) \lhd b \rhd (A \sqcap C)$
Proof:

$$A \sqcap (B \lhd b \rhd C) \quad [3.5.24]$$

$$= A \vee (B \lhd b \rhd C) \quad [3.5.22]$$

$$= A \vee (b \wedge B) \vee (\neg b \wedge C) \quad [\text{propositional calculus}]$$

$$= (true \wedge A) \vee (b \wedge B) \vee (\neg b \wedge C) \quad [\text{propositional calculus}]$$

$$= ((b \vee \neg b) \wedge A) \vee (b \wedge B) \vee (\neg b \wedge C) \quad [\text{propositional calculus}]$$

$$= (b \wedge A) \vee (\neg b \wedge A) \vee (b \wedge B) \vee (\neg b \wedge C) \quad [\text{propositional calculus}]$$

$$= (b \wedge (A \vee B)) \vee (\neg b \wedge (A \vee C)) \quad [3.5.24]$$

$$= (b \wedge (A \sqcap B)) \vee (\neg b \wedge (A \sqcap C)) \quad [3.5.22]$$

$$= (A \sqcap B) \lhd b \rhd (A \sqcap C) \quad \Box$$

L9. $(c \to A) \sqcap (c \to B) = c \to (A \sqcap B)$
Proof:

$$
\begin{array}{ll}
(c \to A) \sqcap (c \to B) & [3.5.20] \\
= ((c \to \mathit{Skip});\ A) \sqcap ((c \to \mathit{Skip});\ B) & [\text{Property 3.9 L6}] \\
= (c \to \mathit{Skip});\ (A \sqcap B) & [3.5.20] \\
= c \to (A \sqcap B) \quad \square &
\end{array}
$$

D.8 PARALLEL COMPOSITION

D.8.1 Merge function validity

The following is the proof of the validity of our merge function. The first property the parallel merge function TM needs to satisfy to be valid is the symmetry property. The merge function should be symmetric on its input.

D.8.1.1 TM is symmetric

$$TM(cs, s_A, s_B)[0.m, 1.m/1.m, 0.m] = TM(cs, s_B, s_A) \tag{D.8.0}$$

Proof:

$$
\begin{array}{ll}
TM(cs, s_A, s_B)[0.m, 1.m/1.m, 0.m] & \\
= & [3.5.40] \\
\left(\begin{array}{c}
ok' = (0.ok \wedge 1.ok) \wedge \\
wait' = (0.wait \vee 1.wait) \wedge \\
dif(tr_t', tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array} \right) \wedge \\
state' = (0.state - s_B) \oplus (1.state - s_A)
\end{array} \right) [0.m, 1.m/1.m, 0.m] & \\
= & [\text{Substitution}] \\
\left(\begin{array}{c}
ok' = (1.ok \wedge 0.ok) \wedge \\
wait' = (1.wait \vee 0.wait) \wedge \\
dif(tr_t', tr_t) \in TSync \left(\begin{array}{c} dif(1.tr_t, tr_t), \\ dif(0.tr_t, tr_t), \\ cs \end{array} \right) \wedge \\
state' = (1.state - s_B) \oplus (0.state - s_A)
\end{array} \right) & \\
= & [\text{Propositional calculus}]
\end{array}
$$

$$
\left(\begin{array}{c} ok' = (0.ok \wedge 1.ok) \wedge \\ wait' = (0.wait \vee 1.wait) \wedge \\ dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(1.tr_t, tr_t), \\ dif(0.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state' = (1.state - s_B) \oplus (0.state - s_A) \end{array}\right)
$$

$=$ [$TSync$ symmetric from 3.5.41]

$$
\left(\begin{array}{c} ok' = (0.ok \wedge 1.ok) \wedge \\ wait' = (0.wait \vee 1.wait) \wedge \\ dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state' = (1.state - s_B) \oplus (0.state - s_A) \end{array}\right)
$$

$=$ [s_A and s_B are disjoint]

$$
\left(\begin{array}{c} ok' = (0.ok \wedge 1.ok) \wedge \\ wait' = (0.wait \vee 1.wait) \wedge \\ dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state' = (0.state - s_A) \oplus (1.state - s_B) \end{array}\right)
$$

$=$ [3.5.40]

$TM(cs, s_B, s_A)$ □

D.8.1.2 TM is associative For to prove the associativity of the parallel merge function we need to create a special three way merge function TM_3 defined as follows

$$
TM_3(cs, s_A, s_B, s_C) = \begin{array}{l} \exists\, ok_x, wait_x, tr_x, state_x \bullet \\ TM(cs, s_A, s_B) \left[\begin{array}{l} ok_x, wait_x, tr_x, state_x/ \\ ok', wait', tr'_t, state' \end{array}\right] \wedge \\ TM(cs, s_A \cap s_B, s_C) \left[\begin{array}{l} ok_x, wait_x, tr_x, state_x, \\ 2.ok, 2.wait, 2.tr_t, 2.state/ \\ 0.ok, 0.wait, 0.tr_t, 0.state, \\ 1.ok, 1.wait, 1.tr_t, 1.state \end{array}\right] \end{array} \quad \text{(D.8.0)}
$$

Note that the result of the first function is used as the first indexed variables of the second function. We also call the attention to the term $s_A \cap s_B$ used to represent the set of variables both A and B can change. We also recall that the sets s_A, s_B and s_C have to be disjoint. To show that the parallel merge function is associative we need to prove the following

$$
(0.m, 1.m.2.m := 1.m, 2.m, 0.m);\ TM_3(cs, s_B, s_C, s_A) = TM_3(cs, s_A, s_B, s_C) \quad \text{(D.8.0)}
$$

Proof:

$$
(0.m, 1.m.2.m := 1.m, 2.m, 0.m);\ TM_3(cs, s_B, s_C, s_A)
$$

$=$ [Relational calculus]

$TM_3(cs, s_B, s_C, s_A)[1.m, 2.m, 0.m/0.m, 1.m.2.m]$

$=$ [D.8.1.2]

$$\exists\, ok_x, wait_x, tr_x, state_x \bullet$$
$$TM(cs, s_B, s_C) \left[\begin{array}{l} ok_x, wait_x, tr_x, state_x / \\ ok', wait', tr'_t, state' \end{array} \right] \wedge$$
$$TM(cs, s_B \cap s_C, s_A) \left[\begin{array}{l} ok_x, wait_x, tr_x, state_x, \\ 2.ok, 2.wait, 2.tr_t, 2.state / \\ 0.ok, 0.wait, 0.tr_t, 0.state, \\ 1.ok, 1.wait, 1.tr_t, 1.state \end{array} \right] [1.m, 2.m, 0.m/0.m, 1.m.2.m]$$

$=$ [Substitution]

$$\exists\, ok_x, wait_x, tr_x, state_x \bullet$$
$$TM(cs, s_B, s_C) \left[\begin{array}{l} ok_x, wait_x, tr_x, state_x, \\ 1.ok, 1, wait, 1.tr_t, 1.state, \\ 2.ok, 2, wait, 2.tr_t, 2.state / \\ ok', wait', tr'_t, state', \\ 0.ok, 0, wait, 0.tr_t, 0.state, \\ 1.ok, 1, wait, 1.tr_t, 1.state \end{array} \right] \wedge$$
$$TM(cs, s_B \cap s_C, s_A) \left[\begin{array}{l} ok_x, wait_x, tr_x, state_x, \\ 0.ok, 0.wait, 0.tr_t, 0.state / \\ 0.ok, 0.wait, 0.tr_t, 0.state, \\ 1.ok, 1.wait, 1.tr_t, 1.state \end{array} \right]$$

$=$ [3.5.40 and substitution]

$$
\begin{array}{l}
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c} ok' = (0.ok \wedge 1.ok) \wedge \\ wait' = (0.wait \vee 1.wait) \wedge \\ dif(tr_t', tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state' = (0.state - s_C) \oplus (1.state - s_B) \end{array}\right) \\
\left[\begin{array}{l} ok_x, wait_x, tr_x, state_x, \\ 1.ok, 1, wait, 1.tr_t, 1.state, \\ 2.ok, 2, wait, 2.tr_t, 2.state/ \\ ok', wait', tr_t', state', \\ 0.ok, 0, wait, 0.tr_t, 0.state, \\ 1.ok, 1, wait, 1.tr_t, 1.state \end{array}\right] \wedge \\
\left(\begin{array}{c} ok' = (0.ok \wedge 1.ok) \wedge \\ wait' = (0.wait \vee 1.wait) \wedge \\ dif(tr_t', tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state' = (0.state - s_A) \oplus (1.state - (s_B \cap s_C)) \end{array}\right) \\
\left[\begin{array}{l} ok_x, wait_x, tr_x, state_x, \\ 0.ok, 0.wait, 0.tr_t, 0.state/ \\ 0.ok, 0.wait, 0.tr_t, 0.state, \\ 1.ok, 1.wait, 1.tr_t, 1.state \end{array}\right] \\
= \qquad \text{[Substitution]} \\
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c} ok_x = (1.ok \wedge 2.ok) \wedge \\ wait_x = (1.wait \vee 2.wait) \wedge \\ dif(tr_x, tr_t) \in TSync \left(\begin{array}{c} dif(1.tr_t, tr_t), \\ dif(2.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state_x = (1.state - s_C) \oplus (2.state - s_B) \end{array}\right) \wedge \\
\left(\begin{array}{c} ok' = (ok_x \wedge 0.ok) \wedge \\ wait' = (wait_x \vee 0.wait) \wedge \\ dif(tr_t', tr_t) \in TSync \left(\begin{array}{c} dif(tr_x, tr_t), \\ dif(0.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state' = (state_x - s_A) \oplus (0.state - (s_B \cap s_C)) \end{array}\right) \\
= \qquad \text{[Substitution]}
\end{array}
$$

$$
\begin{array}{l}
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c}
ok' = (1.ok \wedge 2.ok \wedge 0.ok) \wedge \\
wait' = (1.wait \vee 2.wait \vee 0.wait) \wedge \\
dif(tr_x, tr_t) \in TSync \left(\begin{array}{c} dif(1.tr_t, tr_t), \\ dif(2.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(tr_x, tr_t), \\ dif(0.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
state' = (((1.state - s_C) \oplus (2.state - s_B)) - s_A) \oplus (0.state - (s_B \cap s_C))
\end{array}\right) \\
= \qquad \text{[Property of } \oplus\text{]} \\
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c}
ok' = (1.ok \wedge 2.ok \wedge 0.ok) \wedge \\
wait' = (1.wait \vee 2.wait \vee 0.wait) \wedge \\
dif(tr_x, tr_t) \in TSync \left(\begin{array}{c} dif(1.tr_t, tr_t), \\ dif(2.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(tr_x, tr_t), \\ dif(0.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
state' = ((1.state - (s_C \cap s_A)) \oplus (2.state - (s_B \cap s_A))) \oplus (0.state - (s_B \cap s_C))
\end{array}\right) \\
= \qquad \text{[Three way synch]} \\
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c}
ok' = (1.ok \wedge 2.ok \wedge 0.ok) \wedge \\
wait' = (1.wait \vee 2.wait \vee 0.wait) \wedge \\
dif(tr'_t, tr_t) \in TSync3 \left(\begin{array}{c} dif(1.tr_t, tr_t), \\ dif(2.tr_t, tr_t), \\ dif(0.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
state' = ((1.state - (s_C \cap s_A)) \oplus (2.state - (s_B \cap s_A))) \oplus (0.state - (s_B \cap s_C))
\end{array}\right) \\
= \qquad [TSync3 \text{ is symmetric}] \\
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c}
ok' = (1.ok \wedge 2.ok \wedge 0.ok) \wedge \\
wait' = (1.wait \vee 2.wait \vee 0.wait) \wedge \\
dif(tr'_t, tr_t) \in TSync3 \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ dif(2.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
state' = ((1.state - (s_C \cap s_A)) \oplus (2.state - (s_B \cap s_A))) \oplus (0.state - (s_B \cap s_C))
\end{array}\right) \\
= \qquad \text{[Three way synch]}
\end{array}
$$

$$
\begin{array}{l}
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c}
ok' = (1.ok \land 2.ok \land 0.ok) \land \\
wait' = (1.wait \lor 2.wait \lor 0.wait) \land \\
dif(tr_x, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \land \\
dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(tr_x, tr_t), \\ dif(2.tr_t, tr_t), \\ cs \end{array}\right) \land \\
state' = ((1.state - (s_C \cap s_A)) \oplus (2.state - (s_B \cap s_A))) \oplus (0.state - (s_B \cap s_C))
\end{array}\right) \\
= \qquad \text{[Assumption that } s_A, \text{ } s_B \text{ and } s_C \text{ are disjoint]} \\
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c}
ok' = (1.ok \land 2.ok \land 0.ok) \land \\
wait' = (1.wait \lor 2.wait \lor 0.wait) \land \\
dif(tr_x, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \land \\
dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(tr_x, tr_t), \\ dif(2.tr_t, tr_t), \\ cs \end{array}\right) \land \\
state' = (0.state - (s_B \cap s_C)) \oplus (1.state - (s_C \cap s_A)) \oplus (2.state - (s_B \cap s_A))
\end{array}\right) \\
= \qquad \text{[Property of } \oplus\text{]} \\
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c}
ok' = (1.ok \land 2.ok \land 0.ok) \land \\
wait' = (1.wait \lor 2.wait \lor 0.wait) \land \\
dif(tr_x, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \land \\
dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(tr_x, tr_t), \\ dif(2.tr_t, tr_t), \\ cs \end{array}\right) \land \\
state' = (((0.state - s_B) \oplus (1.state - s_A)) - s_C) \oplus (2.state - (s_B \cap s_A))
\end{array}\right) \\
= \qquad \text{[introduce } ok_x, \text{ } state_x \text{ and } wait_x\text{]}
\end{array}
$$

$$
\begin{array}{l}
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c}
ok_x = (0.ok \wedge 1.ok) \wedge \\
ok' = (ok_x \wedge 2.ok) \wedge \\
wait_x = (0.wait \vee 1.wait) \wedge \\
wait' = (wait_x \vee 2.wait) \wedge \\
dif(tr_x, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(tr_x, tr_t), \\ dif(2.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
state_x = ((0.state - s_B) \oplus (1.state - s_A)) \\
state' = (state_x - s_C) \oplus (2.state - (s_B \cap s_A))
\end{array}\right) \\
= \qquad \text{[Propositional calculus]} \\
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
\left(\begin{array}{c}
ok_x = (0.ok \wedge 1.ok) \wedge \\
wait_x = (0.wait \vee 1.wait) \wedge \\
dif(tr_x, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
state_x = ((0.state - s_B) \oplus (1.state - s_A))
\end{array}\right) \wedge \\
\left(\begin{array}{c}
ok' = (ok_x \wedge 2.ok) \wedge \\
wait' = (wait_x \vee 2.wait) \wedge \\
dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(tr_x, tr_t), \\ dif(2.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
state' = (state_x - s_C) \oplus (2.state - (s_B \cap s_A))
\end{array}\right) \\
= \qquad \text{[Predicate calculus]} \\
\exists\, ok_x, wait_x, tr_x, state_x \bullet \\
TM(cs, s_A, s_B) \left[\begin{array}{l} ok_x, wait_x, tr_x, state_x / \\ ok', wait', tr'_t, state' \end{array}\right] \wedge \\
TM(cs, s_A \cap s_B, s_C) \left[\begin{array}{l} ok_x, wait_x, tr_x, state_x, \\ 2.ok, 2.wait, 2.tr_t, 2.state / \\ 0.ok, 0.wait, 0.tr_t, 0.state, \\ 1.ok, 1.wait, 1.tr_t, 1.state \end{array}\right] \\
= \qquad \text{[D.8.1.2]} \\
TM3(cs, s_A, s_B, s_C) \qquad \square
\end{array}
$$

D.8.1.3 TM and identity Finally, according to the UTP, our merge function needs to satisfy the following property

$$(0.m, 1.m := m, m);\ TM(cs, s_A, s_B) = \Pi_t$$

We use the following equivalent property.

$$(0.m, 1.m := m, m);\ TM(cs, s_A, s_B);\ Skip = Skip$$

The equivalence of the two properties is due to Lemma 3.4.
Proof:

$$(0.m, 1.m := m, m);\ TM(cs, s_A, s_B);\ Skip$$

$$= \quad \text{[Relational calculus]}$$

$$TM(cs, s_A, s_B)[m, m/0.m, 1.m];\ Skip$$

$$= \quad \text{[3.5.40]}$$

$$\left(\begin{array}{c} ok' = (0.ok \wedge 1.ok) \wedge \\ wait' = (0.wait \vee 1.wait) \wedge \\ dif(tr'_t, tr_t) \in TSync\left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state' = (0.state - s_B) \oplus (1.state - s_A) \end{array}\right)[m, m/0.m, 1.m];\ Skip$$

$$= \quad \text{[Susbtitution]}$$

$$\left(\begin{array}{c} ok' = (ok \wedge ok) \wedge \\ wait' = (wait \vee wait) \wedge \\ dif(tr'_t, tr_t) \in TSync\left(\begin{array}{c} dif(tr_t, tr_t), \\ dif(tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state' = (state - s_B) \oplus (state - s_A) \end{array}\right);\ Skip$$

$$= \quad \text{[Propositional calculus]}$$

$$\left(\begin{array}{c} ok' = ok \wedge \\ wait' = wait \wedge \\ dif(tr'_t, tr_t) \in TSync\left(\begin{array}{c} dif(tr_t, tr_t), \\ dif(tr_t, tr_t), \\ cs \end{array}\right) \wedge \\ state' = (state - s_B) \oplus (state - s_A) \end{array}\right);\ Skip$$

$$= \quad \text{[Property 3.3 L1]}$$

$$\left(\begin{array}{c} ok' = ok \wedge \\ wait' = wait \wedge \\ dif(tr'_t, tr_t) \in TSync\left(\begin{array}{c} \langle(\langle\rangle, snd(last(tr_t)))\rangle, \\ \langle(\langle\rangle, snd(last(tr_t)))\rangle, \\ cs \end{array}\right) \wedge \\ state' = (state - s_B) \oplus (state - s_A) \end{array}\right);\ Skip$$

$$= \quad \text{[3.5.41]}$$

$$\left(\begin{array}{c} ok' = ok \wedge \\ wait' = wait \wedge \\ dif(tr'_t, tr_t) \in \left(\begin{array}{l} \left\{ \begin{array}{l} (t', r') \mid t' \in Sync(\langle\rangle, \langle\rangle, cs) \\ r' = \begin{array}{l} ((snd(last(tr_t)) \cup snd(last(tr_t))) \cap cs) \cup \\ ((snd(last(tr_t)) \cap snd(last(tr_t))) \setminus cs) \end{array} \end{array} \right\} \\ ^\frown TSync(\langle\rangle, \langle\rangle, Event) \end{array}\right) \wedge \\ state' = (state - s_B) \oplus (state - s_A) \end{array}\right) ; \ Skip$$

$=$ [Set theory]

$$\left(\begin{array}{c} ok' = ok \wedge \\ wait' = wait \wedge \\ dif(tr'_t, tr_t) \in \left(\begin{array}{l} \left\{ \begin{array}{l} (t', r') \mid t' \in Sync(\langle\rangle, \langle\rangle, cs) \\ r' = snd(last(tr_t)) \end{array} \right\} \\ ^\frown TSync(\langle\rangle, \langle\rangle, Event) \end{array}\right) \wedge \\ state' = (state - s_B) \oplus (state - s_A) \end{array}\right) ; \ Skip$$

$=$ [3.5.47 and 3.5.42]

$$\left(\begin{array}{c} ok' = ok \wedge \\ wait' = wait \wedge \\ dif(tr'_t, tr_t) \in \left(\begin{array}{l} \left\{ \begin{array}{l} (t', r') \mid t' \in \{\langle\rangle\} \\ r' = snd(last(tr_t)) \end{array} \right\} \\ ^\frown\{\} \end{array}\right) \wedge \\ state' = (state - s_B) \oplus (state - s_A) \end{array}\right) ; \ Skip$$

$=$ [Set theory]

$$\left(\begin{array}{c} ok' = ok \wedge \\ wait' = wait \wedge \\ dif(tr'_t, tr_t) = \langle(\langle\rangle, snd(last(tr_t)))\rangle \wedge \\ state' = (state - s_B) \oplus (state - s_A) \end{array}\right) ; \ Skip$$

$=$ [Property 3.3 L1]

$$\left(\begin{array}{c} ok' = ok \wedge \\ wait' = wait \wedge \\ tr'_t = tr_t \wedge \\ state' = (state - s_B) \oplus (state - s_A) \end{array}\right) ; \ Skip$$

$=$ [s_A and s_B are disjoint *]

$$\left(\begin{array}{c} ok' = ok \wedge \\ wait' = wait \wedge \\ tr'_t = tr_t \wedge \\ state' = state \end{array}\right) ; \ Skip$$

$=$ [Relational calculus]

$Skip$ □

D.9 HIDING

Property 3.13

L1. $t_a \downarrow_t Events = t_a$
Proof:

$$
\begin{array}{ll}
& t_a \downarrow_t Events = t_b \\
= & [\text{A.3}] \\
& \forall\, i : 1..\#t_a \bullet \left(\begin{array}{l} fst(t_b(i)) = fst(t_a(i)) \downarrow Events \wedge \\ snd(t_a(i)) = (snd(tr_b(i)) \cup (Events - Events)) \wedge \\ \#t_a = \#t_b \end{array} \right) \\
= & [\text{Set theory and sequence restriction property}] \\
& \forall\, i : 1..\#t_a \bullet \left(\begin{array}{l} fst(t_b(i)) = fst(t_a(i)) \\ snd(t_a(i)) = (snd(tr_b(i)) \cup (\{\})) \wedge \\ \#t_a = \#t_b \end{array} \right) \\
= & [\text{Set theory}] \\
& \forall\, i : 1..\#t_a \bullet \left(\begin{array}{l} fst(t_b(i)) = fst(t_a(i)) \\ snd(t_a(i)) = snd(tr_b(i)) \wedge \\ \#t_a = \#t_b \end{array} \right) \\
= & [t_a = t_b] \\
& t_a \downarrow_t Events = t_a \qquad \square
\end{array}
$$

L2. $\langle (t, r) \rangle \downarrow_t cs = \langle ((t \downarrow cs), (r \cup (Events - cs))) \rangle$
Proof:
Form the definition of time trace restriction $\square$

L3. $t_a \frown \langle (t, r) \rangle \downarrow_t cs = (t_a \downarrow_t cs) \frown (\langle (t, r) \rangle \downarrow_t cs)$
Proof:
Form the definition of time trace restriction $\square$

L4. $\#(t_a \downarrow_t cs) = \#t_a$
Proof:
Form the definition of time trace restriction $\square$

L5. $t_a \downarrow_t cs_1 \downarrow_t cs_2 = t_a \downarrow_t (cs_1 \cap cs_2)$
Proof:

$$(t_a \downarrow_t cs_1) \downarrow_t cs_2$$

$$
\begin{array}{ll}
= & [\text{A.3}] \\
\exists\, t_b \bullet & \\
\forall\, i : 1..\#t_a \bullet \left(\begin{array}{l} fst(t_b(i)) = fst(t_a(i)) \downarrow cs_1 \land \\ snd(t_a(i)) = (snd(tr_b(i)) \cup (Events - cs_1)) \land \\ \#t_a = \#t_b \end{array} \right) \land & \\
\forall\, i : 1..\#t_b \bullet \left(\begin{array}{l} fst(t_c(i)) = fst(t_b(i)) \downarrow cs_2 \land \\ snd(t_b(i)) = (snd(tr_c(i)) \cup (Events - cs_2)) \land \\ \#t_b = \#t_c \end{array} \right) & \\
= & [\text{Predicate calculus and } \#t_a = \#t_b] \\
\exists\, t_b \bullet & \\
\forall\, i : 1..\#t_a \bullet \left(\begin{array}{l} fst(t_b(i)) = fst(t_a(i)) \downarrow cs_1 \land \\ snd(t_a(i)) = (snd(tr_b(i)) \cup (Events - cs_1)) \land \\ \#t_a = \#t_b \land \\ fst(t_c(i)) = fst(t_b(i)) \downarrow cs_2 \land \\ snd(t_b(i)) = (snd(tr_c(i)) \cup (Events - cs_2)) \land \\ \#t_b = \#t_c \end{array} \right) & \\
= & [\text{Substitiotion}] \\
\forall\, i : 1..\#t_a \bullet \left(\begin{array}{l} fst(t_c(i)) = fst(t_a(i)) \downarrow cs_1 \downarrow cs_2 \land \\ snd(t_a(i)) = ((snd(tr_c(i)) \cup (Events - cs_2)) \cup (Events - cs_1)) \land \\ \#t_a = \#t_c \end{array} \right) & \\
= & [\text{set theory and sequence restriction property}] \\
\forall\, i : 1..\#t_a \bullet \left(\begin{array}{l} fst(t_c(i)) = fst(t_a(i)) \downarrow (cs_1 \cap cs_2) \land \\ snd(t_a(i)) = (snd(tr_c(i)) \cup (Events - (cs_1 \cap cs_2))) \land \\ \#t_a = \#t_c \end{array} \right) & \\
= & [\text{A.3}] \\
t_a \downarrow_t (cs_1 \cap cs_2) & \square
\end{array}
$$

Property 3.14

L1. $A \setminus \{\} = A$
Provided that A is healthy *Circus* Time Action
Proof:

$$
\begin{array}{ll}
A \setminus \{\} & \\
= & [3.5.55] \\
R_t \left(\begin{array}{l} \exists\, s_t \bullet A[s_t/tr'_t] \land \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - \{\}) \end{array} \right) ;\ Skip & \\
= & [\text{Set theoery}] \\
R_t \left(\begin{array}{l} \exists\, s_t \bullet A[s_t/tr'_t] \land \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t Event \end{array} \right) ;\ Skip &
\end{array}
$$

$$
\begin{array}{ll}
= & \text{[Property 3.13 L1]} \\
R_t \left(\begin{array}{l} \exists\, s_t \bullet A[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \end{array} \right) ;\ Skip & \\
= & \text{[Property 3.3 L4]} \\
= & \text{[Property 3.13 L1]} \\
R_t(\exists\, s_t \bullet A[s_t/tr'_t] \wedge tr'_t = s_t);\ Skip & \\
= & \text{[Susbtitution]} \\
R_t(A);\ Skip & \\
= & \text{[Assumption } A \text{ is } R_t \text{ and } CSP4_t \text{ healthy]} \\
A & \square
\end{array}
$$

L2. $A \setminus cs_1 \setminus cs_2 = A \setminus (cs_1 \cup cs_1)$
Provided that A is a healthy *Circus* Time Action
Proof:

$$
\begin{array}{ll}
A \setminus cs_1 \setminus cs_2 & \\
= & \text{[3.5.55]} \\
\left(R_t \left(\begin{array}{l} \exists\, s_1 \bullet A[s_1/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_1, tr_t) \downarrow_t (Event - cs_1) \end{array} \right) ;\ Skip \right) \setminus cs_2 & \\
= & \text{[Property 3.14 L12]} \\
\left(R_t \left(\begin{array}{l} \exists\, s_1 \bullet A[s_1/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_1, tr_t) \downarrow_t (Event - cs_1) \end{array} \right) \setminus cs_2;\ Skip \setminus cs_2 \right) & \\
= & \text{[Property 3.14 L8]} \\
\left(R_t \left(\begin{array}{l} \exists\, s_1 \bullet A[s_1/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_1, tr_t) \downarrow_t (Event - cs_1) \end{array} \right) \setminus cs_2;\ Skip \right) & \\
= & \text{[hiding distributes over } R_t\text{]} \\
R_t \left(\left(\begin{array}{l} \exists\, s_1 \bullet A[s_1/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_1, tr_t) \downarrow_t (Event - cs_1) \end{array} \right) \setminus cs_2 \right) ;\ Skip & \\
= & \text{[3.5.55]} \\
R_t \left(\begin{array}{l} \exists\, s_2 \bullet \left(\begin{array}{l} \exists\, s_1 \bullet A[s_1/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_1, tr_t) \downarrow_t (Event - cs_1) \end{array} \right) [s_2/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_2, tr_t) \downarrow_t (Event - cs_2) \end{array} \right) ;\ Skip & \\
= & \text{[Substitution]} \\
R_t \left(\begin{array}{l} \exists\, s_2 \bullet \left(\begin{array}{l} \exists\, s_1 \bullet A[s_1/tr'_t] \wedge \\ dif(s_2, tr_t) = dif(s_1, tr_t) \downarrow_t (Event - cs_1) \end{array} \right) \wedge \\ dif(tr'_t, tr_t) = dif(s_2, tr_t) \downarrow_t (Event - cs_2) \end{array} \right) ;\ Skip & \\
= & \text{[Substituition]}
\end{array}
$$

$$
\begin{array}{ll}
& R_t\left(\begin{array}{l}\exists\, s_1 \bullet A[s_1/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_1, tr_t) \downarrow_t (Event - cs_1) \downarrow_t (Event - cs_2)\end{array}\right);\ Skip \\
= & \text{[Property 3.13 L5]} \\
= & \text{[Substituition]} \\
& R_t\left(\begin{array}{l}\exists\, s_1 \bullet A[s_1/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_1, tr_t) \downarrow_t (Event - cs_1) \cap (Event - cs_2)\end{array}\right);\ Skip \\
= & \text{[Set theory]} \\
& R_t\left(\begin{array}{l}\exists\, s_1 \bullet A[s_1/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_1, tr_t) \downarrow_t (Event - (cs_1 \cap cs_2))\end{array}\right);\ Skip \\
= & \text{[3.5.55]} \\
A \setminus (cs_1 \cap cs_2) & \Box
\end{array}
$$

L3. $(A \sqcap B) \setminus cs = (A \setminus cs) \sqcap (B \setminus cs)$
Proof:

$$
\begin{array}{ll}
(A \sqcap B) \setminus cs & \\
= & \text{[3.5.55]} \\
& R_t\left(\begin{array}{l}\exists\, s_t \bullet (A \sqcap B)[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right);\ Skip \\
= & \text{[3.5.24]} \\
& R_t\left(\begin{array}{l}\exists\, s_t \bullet (A \vee B)[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right);\ Skip \\
= & \text{[Predicate calculus]} \\
& R_t\left(\begin{array}{l}\exists\, s_t \bullet (A[s_t/tr'_t] \vee B[s_t/tr'_t]) \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right);\ Skip \\
= & \text{[Predicate calculus and propositional calculus]} \\
& R_t\left(\begin{array}{l}\left(\begin{array}{l}\exists\, s_t \bullet A[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right) \vee \\ \left(\begin{array}{l}\exists\, s_t \bullet B[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right)\end{array}\right);\ Skip \\
= & \text{[}R_t\text{ closed over }\vee\text{]} \\
& \left(\begin{array}{l} R_t\left(\begin{array}{l}\exists\, s_t \bullet A[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right) \vee \\ R_t\left(\begin{array}{l}\exists\, s_t \bullet B[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right)\end{array}\right);\ Skip \\
= & \text{[Property 3.9 L5]} \\
& R_t\left(\begin{array}{l}\exists\, s_t \bullet A[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right);\ Skip \vee \\
& R_t\left(\begin{array}{l}\exists\, s_t \bullet B[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right);\ Skip
\end{array}
$$

$$
\begin{array}{ll}
= & [3.5.55] \\
(A \setminus cs) \vee (B \setminus cs) & \\
= & [3.5.24] \\
(A \setminus cs) \sqcap (B \setminus cs) & \Box
\end{array}
$$

L4. $(c \rightarrow A) \setminus cs = c \rightarrow (A \setminus cs)$ if $c \notin cs$
Proof:

$$
\begin{array}{ll}
(c \rightarrow A) \setminus cs & \\
= & [3.5.20] \\
((c \rightarrow \mathit{Skip});\ A) \setminus cs & \\
= & [\text{Property 3.14 L12}] \\
((c \rightarrow \mathit{Skip}) \setminus cs);\ (A \setminus cs) & \\
= & [3.5.17] \\
\left(\left(CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} wait\_com(c) \vee \\ terminating\_com(c.e) \end{array}\right)\right)\right) \setminus cs\right);\ (A \setminus cs) & \\
= & [\text{hiding distributes over } CSP1_t \text{ and } R_t] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} wait\_com(c) \vee \\ terminating\_com(c.e) \end{array}\right) \setminus cs\right);\ (A \setminus cs) & \\
= & [\text{Property 3.14 L3}] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} wait\_com(c) \setminus cs \vee \\ terminating\_com(c.e) \setminus cs \end{array}\right)\right);\ (A \setminus cs) & \\
= & [3.5.55] \\
CSP1_t\left(ok' \wedge R_t\left(\left(R_t\left(\begin{array}{c} \exists\, s_t \bullet wait\_com(c)[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) \\ = \\ dif(s_t, tr_t) \\ \downarrow_t \\ (Event - cs) \\ terminating\_com(c.e) \setminus cs \end{array}\right);\ \mathit{Skip}\right) \vee \right)\right);\ (A \setminus cs) & \\
= & [3.5.13 \text{ and substituition}] \\
CSP1_t\left(ok' \wedge R_t\left(\left(\begin{array}{l} R_t\left(\begin{array}{l} \exists\, s_t \bullet \\ \left(\begin{array}{l} wait' \wedge \\ Flat(s_t) - Flat(tr_t) = \langle\rangle \wedge \\ \forall\, i : 1..dif(s_t, tr_t) \bullet \\ c \notin snd(dif(s'_t, tr_t)(i)) \end{array}\right) \wedge \\ dif(tr'_t, tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array}\right) \\ ;\ \mathit{Skip} \\ terminating\_com(c.e) \setminus cs \end{array}\right) \vee \right)\right) & \\
;\ (A \setminus cs) & \\
= & [A.3]
\end{array}
$$

$$
CSP1_t \left(ok' \wedge R_t \left(\begin{array}{l} \left(\begin{array}{l} R_t \left(\begin{array}{l} \exists\, s_t \bullet \\ \left(\begin{array}{l} wait' \wedge \\ Flat(s_t) - Flat(tr_t) = \langle\rangle \wedge \\ \ \forall\, i : 1..dif(s_t, tr_t) \bullet \\ \ c \notin snd(dif(s'_t, tr_t)(i)) \end{array} \right) \wedge \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ \left(\begin{array}{l} fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)) = \\ (snd(dif(tr'_t, tr_t)(i)) \\ \cup cs) \wedge \\ \#dif(s_t, tr_t) = \\ \#dif(tr'_t, tr_t) \end{array} \right) \end{array} \right) \\ ;\ Skip \end{array} \right) \vee \\ \qquad terminating\_com(c.e) \setminus cs \end{array} \right) \right)
$$

$$
;\ (A \setminus cs)
$$

$=$ [Predicate calculus]

$$
CSP1_t \left(ok' \wedge R_t \left(\begin{array}{l} \left(\begin{array}{l} R_t \left(\begin{array}{l} \exists\, s_t \bullet \\ \left(\begin{array}{l} wait' \wedge \\ Flat(s_t) - Flat(tr_t) = \langle\rangle \wedge \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ c \notin snd(dif(s'_t, tr_t)(i)) \\ fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \end{array} \right) \end{array} \right) \\ ;\ Skip \end{array} \right) \vee \\ \qquad terminating\_com(c.e) \setminus cs \end{array} \right) \right)
$$

$$
;\ (A \setminus cs)
$$

$=$ [Properties 3.3 L3 and L7]

$$
\begin{array}{l}
CSP1_t \left(\begin{array}{l} ok' \wedge R_t \left(\begin{array}{l} \left(\begin{array}{l} R_t \left(\begin{array}{l} \exists\, s_t \bullet \\ \left(\begin{array}{l} wait' \wedge \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ fst(dif(s_t, tr_t)(i)) = \langle\rangle \\ c \notin snd(dif(s'_t, tr_t)(i)) \\ fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \end{array} \right) \end{array} \right) \\ ;\ Skip \end{array} \right) \\ terminating\_com(c.e) \setminus cs \end{array} \right) \end{array} \right) \vee \\ ;\ (A \setminus cs) \end{array} \right) \\
= \qquad \text{[Substitution]} \\
CSP1_t \left(\begin{array}{l} ok' \wedge R_t \left(\begin{array}{l} \left(\begin{array}{l} R_t \left(\begin{array}{l} wait' \wedge \\ \forall\, i : 1..\#dif(tr'_t, tr_t) \bullet \\ c \notin (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ fst(dif(tr'_t, tr_t)(i)) = \\ \langle\rangle \downarrow (Event - cs) \end{array} \right) \end{array} \right) \\ ;\ Skip \end{array} \right) \vee \\ terminating\_com(c.e) \setminus cs \end{array} \right) \\
;\ (A \setminus cs) \\
= \qquad \text{[}c \notin cs \text{ and sequence restriction property]} \\
CSP1_t \left(\begin{array}{l} ok' \wedge R_t \left(\begin{array}{l} \left(\begin{array}{l} R_t \left(\begin{array}{l} wait' \wedge \\ \forall\, i : 1..\#dif(tr'_t, tr_t) \bullet \\ c \notin snd(dif(tr'_t, tr_t)(i)) \wedge \\ fst(dif(tr'_t, tr_t)(i)) = \langle\rangle \end{array} \right) \end{array} \right) \\ ;\ Skip \end{array} \right) \vee \\ terminating\_com(c.e) \setminus cs \end{array} \right) \\
;\ (A \setminus cs) \\
= \qquad \text{[Predicate calculus]} \\
CSP1_t \left(\begin{array}{l} ok' \wedge R_t \left(\begin{array}{l} \left(\begin{array}{l} R_t \left(\begin{array}{l} wait' \wedge \\ \forall\, i : 1..\#dif(tr'_t, tr_t) \bullet \\ c \notin snd(dif(tr'_t, tr_t)(i)) \wedge \\ \forall\, i : 1..\#dif(tr'_t, tr_t) \bullet \\ fst(dif(tr'_t, tr_t)(i)) = \langle\rangle \end{array} \right) \end{array} \right) \\ ;\ Skip \end{array} \right) \vee \\ terminating\_com(c.e) \setminus cs \end{array} \right) \\
;\ (A \setminus cs) \\
= \qquad \text{[3.5.14 and property 3.3 L3]}
\end{array}
$$

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{l}\left(\begin{array}{l} R_t\left(\begin{array}{l} wait' \wedge possible(tr_t, tr'_t, cs) \wedge \\ trace' = \langle\rangle \end{array}\right) \end{array}\right) \\ \qquad\qquad ;\ Skip \end{array}\right) \vee \\ terminating\_com(c.e) \setminus cs \end{array}\right) \\
;\ (A \setminus cs) \\
= \qquad\qquad [3.5.13] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{l} (R_t(wait\_com(c));\ Skip) \vee \\ terminating\_com(c.e) \setminus cs \end{array}\right)\right) \\
;\ (A \setminus cs) \\
= \qquad\qquad [3.5.16] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} (R_t(wait\_com(c));\ Skip) \vee \\ ((wait\_com(c); \\ term\_com(c)) \vee term\_com(c)) \setminus cs \end{array}\right)\right) \\
;\ (A \setminus cs) \\
= \qquad\qquad [\text{Property 3.14 L3 and L12}] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} (R_t(wait\_com(c));\ Skip) \vee \\ ((wait\_com(c) \setminus cs); \\ (term\_com(c) \setminus cs)) \vee \\ (term\_com(c) \setminus cs) \end{array}\right)\right) \\
;\ (A \setminus cs) \\
= \qquad\qquad [\text{From the previous steps}] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} (R_t(wait\_com(c));\ Skip) \vee \\ (R_t(wait\_com(c));\ Skip); \\ (term\_com(c) \setminus cs)) \vee \\ (term\_com(c) \setminus cs) \end{array}\right)\right) \\
;\ (A \setminus cs) \\
= \qquad\qquad [3.5.15] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} (R_t(wait\_com(c));\ Skip) \vee \\ \left(\begin{array}{c} (R_t(wait\_com(c));\ Skip); \\ \left(\begin{array}{l} \neg wait' \wedge \\ trace' = \langle c\rangle \\ \wedge\ \#tr'_t = \#tr_t \end{array}\right) \setminus cs \end{array}\right) \vee \\ \left(\begin{array}{l} \neg wait' \wedge \\ trace' = \langle c\rangle \\ \wedge\ \#tr'_t = \#tr_t \end{array}\right) \setminus cs \end{array}\right)\right) \\
;\ (A \setminus cs) \\
= \qquad\qquad [\text{Property 3.3 L3,L7 and L9}]
\end{array}
$$

$$
\begin{array}{l}
CSP1_t \left(ok' \wedge R_t \left(\begin{array}{c} (R_t(wait\_com(c));\ Skip) \vee \\ \left(\begin{array}{l} (R_t(wait\_com(c));\ Skip); \\ \left(\begin{array}{l} \neg wait' \wedge \\ fst(dif(tr_t', tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(tr_t', tr_t) = 1 \end{array} \right) \setminus cs \end{array} \right) \vee \\ \left(\begin{array}{l} \neg wait' \wedge \\ fst(dif(tr_t', tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(tr_t', tr_t) = 1 \end{array} \right) \setminus cs \end{array} \right) \right) \\
;\ (A \setminus cs) \\
= \qquad [3.5.55] \\
CSP1_t \left(ok' \wedge R_t \left(\begin{array}{c} (R_t(wait\_com(c));\ Skip) \vee \\ \left(\begin{array}{l} (R_t(wait\_com(c));\ Skip); \\ R_t \left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(s, tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(s, tr_t) = 1 \\ dif(tr_t', tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array} \right) ;\ Skip \end{array} \right) \vee \\ R_t \left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(s, tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(s, tr_t) = 1 \\ dif(tr_t', tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array} \right) ;\ Skip \end{array} \right) \right) \\
;\ (A \setminus cs) \\
= \qquad [A.3]
\end{array}
$$

$$
\begin{array}{l}
CSP1_t \left(ok' \wedge R_t \left(\begin{array}{l}
\quad (R_t(wait\_com(c));\ Skip) \vee \\
\left(\begin{array}{l}
(R_t(wait\_com(c));\ Skip); \\
R_t \left(\begin{array}{l}
\exists\, s \bullet \\
\neg wait' \wedge \\
fst(dif(s, tr_t)(1)) = \langle c \rangle \\
\wedge\ \#dif(s, tr_t) = 1 \\
\forall\, i : 1..\#dif(s_t, tr_t) \bullet \\
\left(\begin{array}{l}
fst(dif(tr'_t, tr_t)(i)) = \\
fst(dif(s_t, tr_t)(i)) \\
\downarrow (Event - cs) \wedge \\
snd(dif(s_t, tr_t)(i)) = \\
(snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\
\#dif(s_t, tr_t) = \\
\#dif(tr'_t, tr_t)
\end{array} \right)
\end{array} \right) ;\ Skip
\end{array} \right) \vee \\
\quad R_t \left(\begin{array}{l}
\exists\, s \bullet \\
\neg wait' \wedge \\
fst(dif(s, tr_t)(1)) = \langle c \rangle \\
\wedge\ \#dif(s, tr_t) = 1 \\
\forall\, i : 1..\#dif(s_t, tr_t) \bullet \\
\left(\begin{array}{l}
fst(dif(tr'_t, tr_t)(i)) = \\
fst(dif(s_t, tr_t)(i)) \\
\downarrow (Event - cs) \wedge \\
snd(dif(s_t, tr_t)(i)) = \\
(snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\
\#dif(s_t, tr_t) = \\
\#dif(tr'_t, tr_t)
\end{array} \right)
\end{array} \right) ;\ Skip
\end{array} \right) \right) \\
;\ (A \setminus cs)
\end{array}
$$

$=$ [Substitution and predicate calculus]

$$
\begin{array}{l}
CSP1_t \left(ok' \wedge R_t \left(\begin{array}{l}
\quad (R_t(wait\_com(c));\ Skip) \vee \\
\left(\begin{array}{l}
(R_t(wait\_com(c));\ Skip); \\
R_t \left(\begin{array}{l}
\exists\, s \bullet \\
\neg wait' \wedge \\
fst(dif(tr'_t, tr_t)(1)) = \langle c \rangle \downarrow (Event - cs) \wedge \\
snd(dif(s_t, tr_t)(1)) = \\
(snd(dif(tr'_t, tr_t)(1)) \cup cs) \wedge \\
\#dif(tr'_t, tr_t) = 1
\end{array} \right) ;\ Skip
\end{array} \right) \vee \\
\quad R_t \left(\begin{array}{l}
\exists\, s \bullet \\
\neg wait' \wedge \\
fst(dif(tr'_t, tr_t)(1)) = \langle c \rangle \downarrow (Event - cs) \wedge \\
snd(dif(s_t, tr_t)(1)) = \\
(snd(dif(tr'_t, tr_t)(1)) \cup cs) \wedge \\
\#dif(tr'_t, tr_t) = 1
\end{array} \right) ;\ Skip
\end{array} \right) \right) \\
;\ (A \setminus cs)
\end{array}
$$

$=$ [sequence restriction and $c \notin cs$]

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c}
\begin{array}{c}(R_t(wait\_com(c));\ Skip) \vee \\ (R_t(wait\_com(c));\ Skip);\end{array} \\
R_t\left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(tr_t', tr_t)(1)) = \langle c\rangle \wedge \\ snd(dif(s_t, tr_t)(1)) = \\ (snd(dif(tr_t', tr_t)(1)) \cup cs) \wedge \\ \#dif(tr_t', tr_t) = 1 \end{array}\right);\ Skip \\
\vee \\
R_t\left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(tr_t', tr_t)(1)) = \langle c\rangle \wedge \\ snd(dif(s_t, tr_t)(1)) = \\ (snd(dif(tr_t', tr_t)(1)) \cup cs) \wedge \\ \#dif(tr_t', tr_t) = 1 \end{array}\right);\ Skip
\end{array}\right)\end{array}\right) \\
;\ (A \setminus cs)
\end{array}
$$

$=$ [Arbitrary refusal]

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c}
\begin{array}{c}(R_t(wait\_com(c));\ Skip) \vee \\ (R_t(wait\_com(c));\ Skip);\end{array} \\
R_t\left(\begin{array}{l} \neg wait' \wedge \\ fst(dif(tr_t', tr_t)(1)) = \langle c\rangle \wedge \\ (\exists\, ref \bullet snd(dif(tr_t', tr_t)(1)) = ref) \wedge \\ \#dif(tr_t', tr_t) = 1 \end{array}\right);\ Skip \\
\vee \\
R_t\left(\begin{array}{l} \neg wait' \wedge \\ fst(dif(tr_t', tr_t)(1)) = \langle c\rangle \wedge \\ (\exists\, ref \bullet snd(dif(tr_t', tr_t)(1)) = ref) \wedge \\ \#dif(tr_t', tr_t) = 1 \end{array}\right);\ Skip
\end{array}\right)\end{array}\right) \\
;\ (A \setminus cs)
\end{array}
$$

$=$ [Properties 3.3 L3 and L7]

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c}
\begin{array}{c}(R_t(wait\_com(c));\ Skip) \vee \\ (R_t(wait\_com(c));\ Skip);\end{array} \\
R_t\left(\begin{array}{l} \neg wait' \wedge \\ trace' = \langle c\rangle \wedge \\ \#tr_t' = \#tr_t \end{array}\right);\ Skip \\
\vee \\
R_t\left(\begin{array}{l} \neg wait' \wedge \\ trace' = \langle c\rangle \wedge \\ \#tr_t' = \#tr_t \end{array}\right);\ Skip
\end{array}\right)\end{array}\right) \\
;\ (A \setminus cs)
\end{array}
$$

$=$ [3.5.15]

$$
\begin{array}{l}
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c}(R_t(wait\_com(c));\ Skip) \vee \\ (\ R_t(wait\_com(c));\ Skip;\ R_t(term\_com(c));\ Skip\ ) \vee \\ R_t(term\_com(c));\ Skip\end{array}\right)\right) \\
;\ (A \setminus cs)
\end{array}
$$

$=$ [R_t closed under ; and $\vee$ and $CSP4_t$]

$$
\begin{array}{l}
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} R_t(wait\_com(c));\ Skip \vee \\ R_t(wait\_com(c);\ (term\_com(c)));\ Skip \vee \\ R_t(term\_com(c));\ Skip \end{array}\right)\right) \\
;\ (A \setminus cs) \\
= \qquad\qquad [R_t \text{ closed under } ; \text{ and } \vee \text{ and property 3.9 L5}] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} wait\_com(c) \vee \\ wait\_com(c);\ term\_com(c) \vee \\ term\_com(c) \end{array}\right)\right);\ Skip \\
;\ (A \setminus cs) \\
= \qquad\qquad [3.5.16 \text{ and } 3.5.17] \\
(c \rightarrow Skip);\ Skip;\ (A \setminus cs) \\
= \qquad\qquad [\text{Property 3.6 L4 and Lemma 3.7}] \\
(c \rightarrow Skip);\ (A \setminus cs) \\
= \qquad\qquad [3.5.20] \\
c \rightarrow (A \setminus cs) \qquad\qquad \square
\end{array}
$$

L5. $(c \rightarrow A) \setminus cs = A \setminus cs$ if $c \in cs$
Proof:

$$
\begin{array}{l}
(c \rightarrow A) \setminus cs \\
= \qquad\qquad [3.5.20] \\
((c \rightarrow Skip);\ A) \setminus cs \\
= \qquad\qquad [\text{Property 3.14 L12}] \\
((c \rightarrow Skip) \setminus cs);\ (A \setminus cs) \\
= \qquad\qquad [3.5.17] \\
\left(\left(CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} wait\_com(c) \vee \\ terminating\_com(c.e) \end{array}\right)\right)\right) \setminus cs\right);\ (A \setminus cs) \\
= \qquad\qquad [\text{hiding distributes over } CSP1_t \text{ and } R_t] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} wait\_com(c) \vee \\ terminating\_com(c.e) \end{array}\right) \setminus cs\right);\ (A \setminus cs) \\
= \qquad\qquad [\text{Property 3.14 L3}] \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} wait\_com(c) \setminus cs \vee \\ terminating\_com(c.e) \setminus cs \end{array}\right)\right);\ (A \setminus cs) \\
= \qquad\qquad [3.5.55] \\
CSP1_t\left(ok' \wedge R_t\left(\left(\begin{array}{c} R_t\left(\begin{array}{c} \exists\, s_t \bullet wait\_com(c)[s_t/tr'_t] \wedge \\ dif(tr'_t, tr_t) \\ = \\ dif(s_t, tr_t) \\ \downarrow_t \\ (Event - cs) \end{array}\right);\ Skip \\ terminating\_com(c.e) \setminus cs \end{array}\right) \vee\right)\right);\ (A \setminus cs) \\
= \qquad\qquad [3.5.13 \text{ and substituition}]
\end{array}
$$

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l} R_t\left(\begin{array}{l}\exists\, s_t \bullet \\ \left(\begin{array}{l} wait' \wedge \\ Flat(s_t) - Flat(tr_t) = \langle\rangle \wedge \\ \forall\, i : 1..dif(s_t, tr_t) \bullet \\ c \notin snd(dif(s'_t, tr_t)(i)) \end{array}\right) \wedge \\ dif(tr'_t, tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array}\right) \\ ;\ Skip \end{array}\right) \\ terminating\_com(c.e) \setminus cs \end{array}\right) \vee \end{array}\right) \\ ;\ (A \setminus cs) \end{array}\right) \\
= \qquad \text{[A.3]} \\
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l} R_t\left(\begin{array}{l}\exists\, s_t \bullet \\ \left(\begin{array}{l} wait' \wedge \\ Flat(s_t) - Flat(tr_t) = \langle\rangle \wedge \\ \forall\, i : 1..dif(s_t, tr_t) \bullet \\ c \notin snd(dif(s'_t, tr_t)(i)) \end{array}\right) \wedge \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ \left(\begin{array}{l} fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \end{array}\right) \end{array}\right) \\ ;\ Skip \end{array}\right) \\ terminating\_com(c.e) \setminus cs \end{array}\right) \vee \end{array}\right) \\ ;\ (A \setminus cs) \end{array}\right) \\
= \qquad \text{[Predicate calculus]} \\
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l} R_t\left(\begin{array}{l}\exists\, s_t \bullet \\ \left(\begin{array}{l} wait' \wedge \\ Flat(s_t) - Flat(tr_t) = \langle\rangle \wedge \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ c \notin snd(dif(s'_t, tr_t)(i)) \\ fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \end{array}\right) \\ ;\ Skip \end{array}\right) \\ terminating\_com(c.e) \setminus cs \end{array}\right) \vee \end{array}\right) \\ ;\ (A \setminus cs) \end{array}\right) \\
= \qquad \text{[Properties 3.3 L3 and L7]}
\end{array}
$$

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{l} \left(\begin{array}{l} R_t\left(\begin{array}{l} \exists\, s_t \bullet \\ \left(\begin{array}{l} wait' \wedge \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ fst(dif(s_t, tr_t)(i)) = \langle\rangle \\ c \notin snd(dif(s'_t, tr_t)(i)) \\ fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \end{array}\right) \\ ;\ Skip \\ terminating\_com(c.e) \setminus cs \end{array}\right) \end{array}\right) \vee \end{array}\right) \\ ;\ (A \setminus cs) \end{array}\right) \\
= \qquad \text{[Substitution]} \\
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{l} \left(\begin{array}{l} R_t\left(\begin{array}{l} wait' \wedge \\ \forall\, i : 1..\#dif(tr'_t, tr_t) \bullet \\ c \notin (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ fst(dif(tr'_t, tr_t)(i)) = \langle\rangle \downarrow (Event - cs) \end{array}\right) \\ ;\ Skip \\ terminating\_com(c.e) \setminus cs \end{array}\right) \vee \end{array}\right) \\ ;\ (A \setminus cs) \end{array}\right) \\
= \qquad \text{[}c \in cs \text{ and sequence restriction property]} \\
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{l} \left(\begin{array}{l} R_t\left(\begin{array}{l} wait' \wedge \\ \forall\, i : 1..\#dif(tr'_t, tr_t) \bullet \\ false \wedge \\ fst(dif(tr'_t, tr_t)(i)) = \langle\rangle \end{array}\right) \\ ;\ Skip \\ terminating\_com(c.e) \setminus cs \end{array}\right) \vee \end{array}\right) \\ ;\ (A \setminus cs) \end{array}\right) \\
= \qquad \text{[Predicate calculus]} \\
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c} (R_t(false);\ Skip) \vee \\ terminating\_com(c.e) \setminus cs \end{array}\right) \end{array}\right) \\ ;\ (A \setminus cs) \\
= \qquad \text{[3.5.16]} \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} (R_t(false);\ Skip) \vee \\ ((wait\_com(c);\ term\_com(c)) \vee term\_com(c)) \setminus cs \end{array}\right)\right) \\ ;\ (A \setminus cs) \\
= \qquad \text{[Property 3.14 L3 and L12]} \\
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} (R_t(false);\ Skip) \vee \\ ((wait\_com(c) \setminus cs);\ (term\_com(c) \setminus cs)) \vee \\ (term\_com(c) \setminus cs) \end{array}\right)\right) \\ ;\ (A \setminus cs) \\
= \qquad \text{[From the previous steps]}
\end{array}
$$

$$
\begin{array}{l}
CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c}(R_t(false);\ Skip) \vee \\ (R_t(false);\ Skip);\ (term\_com(c) \setminus cs)) \vee \\ (term\_com(c) \setminus cs)\end{array}\right)\right) \\
;\ (A \setminus cs) \\
= \qquad [3.5.15] \\
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c}(R_t(false);\ Skip) \vee \\ \left(\begin{array}{l}(R_t(false);\ Skip); \\ \left(\begin{array}{l}\neg wait' \wedge \\ trace' = \langle c \rangle \\ \wedge\ \#tr_t' = \#tr_t\end{array}\right) \setminus cs\end{array}\right) \vee \\ \left(\begin{array}{l}\neg wait' \wedge \\ trace' = \langle c \rangle \\ \wedge\ \#tr_t' = \#tr_t\end{array}\right) \setminus cs\end{array}\right)\end{array}\right) \\
;\ (A \setminus cs) \\
= \qquad [\text{Property 3.3 L3,L7 and L9}] \\
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c}(R_t(false);\ Skip) \vee \\ \left(\begin{array}{l}(R_t(wait\_com(c));\ Skip); \\ \left(\begin{array}{l}\neg wait' \wedge \\ fst(dif(tr_t', tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(tr_t', tr_t) = 1\end{array}\right) \setminus cs\end{array}\right) \vee \\ \left(\begin{array}{l}\neg wait' \wedge \\ fst(dif(tr_t', tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(tr_t', tr_t) = 1\end{array}\right) \setminus cs\end{array}\right)\end{array}\right) \\
;\ (A \setminus cs) \\
= \qquad [3.5.55] \\
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c}(R_t(false);\ Skip) \vee \\ \left(\begin{array}{l}(R_t(false);\ Skip); \\ R_t\left(\begin{array}{l}\exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(s, tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(s, tr_t) = 1 \\ dif(tr_t', tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right);\ Skip\end{array}\right) \vee \\ R_t\left(\begin{array}{l}\exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(s, tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(s, tr_t) = 1 \\ dif(tr_t', tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs)\end{array}\right);\ Skip\end{array}\right)\end{array}\right) \\
;\ (A \setminus cs) \\
= \qquad [A.3]
\end{array}
$$

$$
\begin{array}{l}
CSP1_t \left(ok' \wedge R_t \left(\begin{array}{c} (R_t(false);\ Skip) \vee \\ \left(\begin{array}{l} (R_t(false);\ Skip); \\ R_t \left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(s, tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(s, tr_t) = 1 \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ \left(\begin{array}{l} fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)(i)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \\ \#dif(tr'_t, tr_t) \end{array} \right) \end{array} \right) ;\ Skip \end{array} \right) \vee \\ R_t \left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(s, tr_t)(1)) = \langle c \rangle \\ \wedge\ \#dif(s, tr_t) = 1 \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ \left(\begin{array}{l} fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)(i)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \\ \#dif(tr'_t, tr_t) \end{array} \right) \end{array} \right) ;\ Skip \end{array} \right) \end{array} \right) \right) \\
;\ (A \setminus cs)
\end{array}
$$

$=$ [Substitution and predicate calculus]

$$
\begin{array}{l}
CSP1_t \left(ok' \wedge R_t \left(\begin{array}{c} (R_t(false);\ Skip) \vee \\ \left(\begin{array}{l} (R_t(false);\ Skip); \\ R_t \left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(tr'_t, tr_t)(1)) = \langle c \rangle \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)(1)) = \\ (snd(dif(tr'_t, tr_t)(1)) \cup cs) \wedge \\ \#dif(tr'_t, tr_t) = 1 \end{array} \right) ;\ Skip \end{array} \right) \vee \\ R_t \left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(tr'_t, tr_t)(1)) = \langle c \rangle \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)(1)) = \\ (snd(dif(tr'_t, tr_t)(1)) \cup cs) \wedge \\ \#dif(tr'_t, tr_t) = 1 \end{array} \right) ;\ Skip \end{array} \right) \end{array} \right) \right) \\
;\ (A \setminus cs)
\end{array}
$$

$=$ [sequence restriction and $c \in cs$]

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c} (R_t(false);\ Skip) \vee \\ \left(\begin{array}{l} (R_t(false);\ Skip); \\ R_t\left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(tr'_t, tr_t)(1)) = \langle\rangle \wedge \\ snd(dif(s_t, tr_t)(1)) = \\ (snd(dif(tr'_t, tr_t)(1)) \cup cs) \wedge \\ \#dif(tr'_t, tr_t) = 1 \end{array}\right);\ Skip \end{array}\right) \vee \\ R_t\left(\begin{array}{l} \exists\, s \bullet \\ \neg wait' \wedge \\ fst(dif(tr'_t, tr_t)(1)) = \langle\rangle \wedge \\ snd(dif(s_t, tr_t)(1)) = \\ (snd(dif(tr'_t, tr_t)(1)) \cup cs) \wedge \\ \#dif(tr'_t, tr_t) = 1 \end{array}\right);\ Skip \end{array}\right) \end{array}\right) \\
;\ (A \setminus cs)
\end{array}
$$

= [Arbitrary refusal *ref*]

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c} (R_t(false);\ Skip) \vee \\ \left(\begin{array}{l} (R_t(false);\ Skip); \\ R_t\left(\begin{array}{l} \neg wait' \wedge \\ fst(dif(tr'_t, tr_t)(1)) = \langle\rangle \wedge \\ (\exists\, ref \bullet snd(dif(tr'_t, tr_t)(1)) = ref) \wedge \\ \#dif(tr'_t, tr_t) = 1 \end{array}\right);\ Skip \end{array}\right) \vee \\ R_t\left(\begin{array}{l} \neg wait' \wedge \\ fst(dif(tr'_t, tr_t)(1)) = \langle\rangle \wedge \\ (\exists\, ref \bullet snd(dif(tr'_t, tr_t)(1)) = ref) \wedge \\ \#dif(tr'_t, tr_t) = 1 \end{array}\right);\ Skip \end{array}\right) \end{array}\right) \\
;\ (A \setminus cs)
\end{array}
$$

= [Properties 3.3 L3 and L7]

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c} (R_t(false);\ Skip) \vee \\ \left(\begin{array}{l} (R_t(false);\ Skip); \\ R_t\left(\begin{array}{l} \neg wait' \wedge \\ trace' = \langle\rangle \wedge \\ \#tr'_t = \#tr_t \end{array}\right);\ Skip \end{array}\right) \vee \\ R_t\left(\begin{array}{l} \neg wait' \wedge \\ trace' = \langle\rangle \wedge \\ \#tr'_t = \#tr_t \end{array}\right);\ Skip \end{array}\right) \end{array}\right) \\
;\ (A \setminus cs)
\end{array}
$$

= [Property of time traces]

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} ok' \wedge R_t\left(\begin{array}{c} (R_t(false);\ Skip) \vee \\ \left(\begin{array}{l} (R_t(false);\ Skip); \\ R_t\left(\begin{array}{l} \neg wait' \wedge \\ tr'_t = tr_t \end{array}\right);\ Skip \end{array}\right) \vee \\ R_t\left(\begin{array}{l} \neg wait' \wedge \\ tr'_t = tr_t \end{array}\right);\ Skip \end{array}\right) \end{array}\right) \\
;\ (A \setminus cs)
\end{array}
$$

$$
\begin{array}{ll}
= & [R_t \text{ closed under ; and } \vee \text{ and } CSP4_t] \\
\multicolumn{2}{l}{CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} R_t(false);\ Skip \vee \\ R_t(false;\ (\neg wait' \wedge tr_t' = tr_t));\ Skip \vee \\ R_t(\neg wait' \wedge tr_t' = tr_t);\ Skip \end{array}\right)\right)} \\
\multicolumn{2}{l}{;\ (A \setminus cs)} \\
= & [R_t \text{ closed under ; and } \vee \text{ and property 3.9 L5}] \\
\multicolumn{2}{l}{CSP1_t\left(ok' \wedge R_t\left(\begin{array}{c} false \vee \\ false;\ (\neg wait' \wedge tr_t' = tr_t) \vee \\ (\neg wait' \wedge tr_t' = tr_t) \end{array}\right)\right);\ Skip} \\
\multicolumn{2}{l}{;\ (A \setminus cs)} \\
= & [\text{Relational and propositional caclulus}] \\
\multicolumn{2}{l}{CSP1_t(ok' \wedge R_t(\neg wait' \wedge tr_t' = tr_t);\ Skip);\ (A \setminus cs)} \\
= & [\text{Relational calculus}] \\
\multicolumn{2}{l}{Skip;\ (A \setminus cs)} \\
= & [\text{hide is } CSP3_t \text{ healthy}] \\
\multicolumn{2}{l}{(A \setminus cs)} \qquad \square
\end{array}
$$

L6. $(Wait\ \ \mathsf{n}) \setminus cs = Wait\ \ \mathsf{n}$
Proof:

$$
\begin{array}{ll}
\multicolumn{2}{l}{(Wait\ \ \mathsf{n}) \setminus cs} \\
= & [3.5.8] \\
\multicolumn{2}{l}{CSP1_t\left(R_t\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr_t' - \#tr_t) \leq n \wedge \\ trace' = \langle\rangle \end{array}\right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr_t' - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right)\right)\right) \setminus cs} \\
= & [\text{hiding distributes over healthiness conditions}] \\
\multicolumn{2}{l}{CSP1_t\left(R_t\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr_t' - \#tr_t) \leq n \wedge \\ trace' = \langle\rangle \end{array}\right) \vee \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr_t' - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right)\right) \setminus cs\right)\right)} \\
= & [\text{Property 3.14 L3}]
\end{array}
$$

$$
CSP1_t\left(\begin{array}{l} \\ R_t\left(\begin{array}{l} \\ \left(\begin{array}{l} ok' \wedge wait' \wedge \\ (\#tr'_t - \#tr_t) \leq n \wedge \\ trace' = \langle\rangle \end{array}\right) \setminus cs \vee \\ \left(\begin{array}{l} ok' \wedge \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = n \wedge \\ trace' = \langle\rangle \wedge \\ state' = state \end{array}\right) \setminus cs \\ \\ \end{array}\right) \\ \\ \end{array}\right)
$$

$=$ [3.5.55]

$$
CSP1_t\left(\begin{array}{l} \\ R_t\left(\begin{array}{l} \\ \left(\begin{array}{l} \exists\, s_t \bullet \\ ok' \wedge wait' \wedge \\ (\#s_t - \#tr_t) \leq n \wedge \\ \qquad\qquad\qquad\qquad dif(tr'_t, tr_t) \\ \qquad\qquad\qquad\qquad\quad = \\ Flat(s_t) - Flat(tr_t) = \langle\rangle \wedge \quad dif(s_t, tr_t) \\ \qquad\qquad\qquad\qquad\quad \downarrow_t \\ \qquad\qquad\qquad\qquad (Event - cs) \end{array}\right) \vee \\ \left(\begin{array}{l} \exists\, s_t \bullet \\ ok' \wedge \neg wait' \wedge \\ (\#s_t - \#tr_t) = n \wedge \\ Flat(s_t) - Flat(tr_t) = \langle\rangle \wedge \\ state' = state \wedge \\ \quad dif(tr'_t, tr_t) \\ \qquad = \\ \quad dif(s_t, tr_t) \\ \qquad \downarrow_t \\ \quad (Event - cs) \end{array}\right) \\ \\ \end{array}\right) \\ \\ \end{array}\right)
$$

$=$ [Predicate and propositional calculus]

$$
CSP1_t\left(\begin{array}{l} \\ R_t\left(\begin{array}{l} \\ \left(\begin{array}{l} ok' \wedge \\ \exists\, s_t \bullet \\ Flat(s_t) - Flat(tr_t) = \langle\rangle \wedge \\ \quad dif(tr'_t, tr_t) \\ \qquad = \\ \quad dif(s_t, tr_t) \quad \wedge \\ \qquad \downarrow_t \\ (Event - cs) \\ \left(\begin{array}{l} wait' \wedge \\ (\#s_t - \#tr_t) \leq n \end{array}\right) \vee \left(\begin{array}{l} \neg wait' \wedge \\ state' = state \wedge \\ (\#s_t - \#tr_t) = n \end{array}\right) \end{array}\right) \\ \\ \end{array}\right) \\ \\ \end{array}\right)
$$

$=$ [Property 3.3 L3 and L7]

$$
CSP1_t\left(\begin{array}{l} R_t\left(\begin{array}{l}\left(\begin{array}{l} ok' \wedge \\ \exists\, s_t \bullet \\ \left(\begin{array}{l}\forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ fst(dif(s_t, tr_t)(i)) = \langle\rangle\end{array}\right) \wedge \begin{array}{c} dif(tr'_t, tr_t) \\ = \\ dif(s_t, tr_t) \\ \downarrow_t \\ (Event - cs)\end{array} \wedge \\ \left(\left(\begin{array}{l} wait' \wedge \\ (\#s_t - \#tr_t) \leq n\end{array}\right) \vee \left(\begin{array}{l}\neg wait' \wedge \\ state' = state \wedge \\ (\#s_t - \#tr_t) = n\end{array}\right)\right)\end{array}\right)\end{array}\right)\end{array}\right)
$$

$=$ [A.3]

$$
CSP1_t\left(\begin{array}{l} R_t\left(\begin{array}{l}\left(\begin{array}{l} ok' \wedge \\ \exists\, s_t \bullet \\ \left(\begin{array}{l}\forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ fst(dif(s_t, tr_t)(i)) = \langle\rangle\end{array}\right) \wedge \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \wedge \\ \left(\left(\begin{array}{l} wait' \wedge \\ (\#s_t - \#tr_t) \leq n\end{array}\right) \vee \left(\begin{array}{l}\neg wait' \wedge \\ state' = state \wedge \\ (\#s_t - \#tr_t) = n\end{array}\right)\right)\end{array}\right)\end{array}\right)\end{array}\right)
$$

$=$ [Predicate calculus]

$$
CSP1_t\left(\begin{array}{l} R_t\left(\begin{array}{l}\left(\begin{array}{l} ok' \wedge \\ \exists\, s_t \bullet \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ fst(dif(s_t, tr_t)(i)) = \langle\rangle \wedge \\ fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s_t, tr_t)(i)) \\ \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \wedge \\ \left(\left(\begin{array}{l} wait' \wedge \\ (\#s_t - \#tr_t) \leq n\end{array}\right) \vee \left(\begin{array}{l}\neg wait' \wedge \\ state' = state \wedge \\ (\#s_t - \#tr_t) = n\end{array}\right)\right)\end{array}\right)\end{array}\right)\end{array}\right)
$$

$=$ [Substitution and property 3.3 L4]

$$
CSP1_t\left(\begin{array}{l} R_t\left(\begin{array}{l}\left(\begin{array}{l} ok' \wedge \\ \exists\, s_t \bullet \\ \quad \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ \quad fst(dif(tr'_t, tr_t)(i)) = \langle\rangle \downarrow (Event - cs) \wedge \\ \quad snd(dif(s_t, tr_t)) = (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \quad \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \wedge \\ \quad \left(\begin{array}{l}\left(\begin{array}{l} wait' \wedge \\ (\#tr'_t - \#tr_t) \leq n \end{array}\right) \vee \left(\begin{array}{l} \neg wait' \wedge \\ state' = state \wedge \\ (\#tr'_t - \#tr_t) = n \end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)
$$

$=$ [Let $ref = snd(dif(s_t, tr_t))$]

$$
CSP1_t\left(\begin{array}{l} R_t\left(\begin{array}{l}\left(\begin{array}{l} ok' \wedge \\ \exists\, ref \bullet \\ \quad fst(dif(tr'_t, tr_t)(i)) = \langle\rangle \downarrow (Event - cs) \wedge \\ \quad ref = (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \quad \left(\begin{array}{l}\left(\begin{array}{l} wait' \wedge \\ (\#tr'_t - \#tr_t) \leq n \end{array}\right) \vee \left(\begin{array}{l} \neg wait' \wedge \\ state' = state \wedge \\ (\#tr'_t - \#tr_t) = n \end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)
$$

$=$ [Property of sequence restriction]

$$
CSP1_t\left(\begin{array}{l} R_t\left(\begin{array}{l}\left(\begin{array}{l} ok' \wedge \\ fst(dif(tr'_t, tr_t)(i)) = \langle\rangle \wedge \\ \exists\, ref \bullet \\ \quad ref = (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \quad \left(\begin{array}{l}\left(\begin{array}{l} wait' \wedge \\ (\#tr'_t - \#tr_t) \leq n \end{array}\right) \vee \left(\begin{array}{l} \neg wait' \wedge \\ state' = state \wedge \\ (\#tr'_t - \#tr_t) = n \end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)
$$

$=$ [refusals are arbitrary in *wait* and 3.3.1]

$$
CSP1_t\left(\begin{array}{l} R_t\left(\begin{array}{l}\left(\begin{array}{l} ok' \wedge \\ trace' = \langle\rangle \wedge \\ \quad \left(\begin{array}{l}\left(\begin{array}{l} wait' \wedge \\ (\#tr'_t - \#tr_t) \leq n \end{array}\right) \vee \left(\begin{array}{l} \neg wait' \wedge \\ state' = state \wedge \\ (\#tr'_t - \#tr_t) = n \end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)
$$

$=$ [3.5.8]

$Wait\ \mathsf{n}$ □

L7. $Stop \setminus cs = Stop$

Proof:

$Stop \setminus cs$

$=$ [3.5.5]

$CSP1_t(R3_t(R1_t((ok' \wedge wait' \wedge trace' = \langle\rangle)))) \setminus cs$

$=$ [Hide distributes over healthiness conditions]

$$
\begin{array}{ll}
& CSP1_t(R3_t(R1_t((ok' \wedge wait' \wedge trace' = \langle\rangle) \setminus cs))) \\
= & [ok' \text{ and } wait' \text{ free in hiding}] \\
& CSP1_t(R3_t(R1_t((ok' \wedge wait' \wedge (trace' = \langle\rangle) \setminus cs)))) \\
= & [3.5.55] \\
& CSP1_t\left(R3_t\left(R1_t\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ \left(\begin{array}{l} \exists\, s_t \bullet \\ Flat(s) - Flat(tr_t) = \langle\rangle \wedge \\ \qquad dif(tr'_t, tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array}\right) \end{array}\right)\right)\right)\right)\right) \\
= & [\text{Property 3.3 L3 and L7}] \\
& CSP1_t\left(R3_t\left(R1_t\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ \left(\begin{array}{l} \exists\, s_t \bullet \\ \forall\, i : 1..\#dif(s, tr_t) \bullet \\ fst(dif(s, tr_t)(i)) = \langle\rangle \wedge \\ \qquad dif(tr'_t, tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array}\right) \end{array}\right)\right)\right)\right)\right) \\
= & [A.3] \\
& CSP1_t\left(R3_t\left(R1_t\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ \left(\begin{array}{l} \exists\, s_t \bullet \\ \forall\, i : 1..\#dif(s, tr_t) \bullet \\ fst(dif(s, tr_t)(i)) = \langle\rangle \wedge \\ \qquad \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ \qquad fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s, tr_t)(i)) \downarrow (Event - cs) \wedge \\ \qquad snd(dif(s_t, tr_t)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \end{array}\right) \end{array}\right)\right)\right)\right)\right) \\
= & [\text{Predicate calculus}] \\
& CSP1_t\left(R3_t\left(R1_t\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ \left(\begin{array}{l} \exists\, s_t \bullet \\ \forall\, i : 1..\#dif(s, tr_t) \bullet \\ fst(dif(s, tr_t)(i)) = \langle\rangle \wedge \\ fst(dif(tr'_t, tr_t)(i)) = \\ fst(dif(s, tr_t)(i)) \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)(i)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \end{array}\right) \end{array}\right)\right)\right)\right)\right) \\
= & [\text{Substitution}]
\end{array}
$$

$$
\begin{array}{l}
CSP1_t\left(\begin{array}{l} R3_t\left(\begin{array}{l} R1_t\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ \left(\begin{array}{l} \exists\, s_t \bullet \\ \forall\, i : 1..\#dif(tr'_t, tr_t) \bullet \\ fst(dif(tr'_t, tr_t)(i)) = \\ \langle\rangle \downarrow (Event - cs) \wedge \\ snd(dif(s_t, tr_t)(i)) = \\ (snd(dif(tr'_t, tr_t)(i)) \cup cs) \wedge \\ \#dif(s_t, tr_t) = \#dif(tr'_t, tr_t) \end{array}\right) \end{array}\right)\right)\right)\right)\right) \\
= \qquad \text{[substitute } ref = snd(dif(s_t, tr_t)(i))\text{]} \\
CSP1_t\left(\begin{array}{l} R3_t\left(\begin{array}{l} R1_t\left(\left(\left(\begin{array}{l} ok' \wedge wait' \wedge \\ \left(\begin{array}{l} \exists\, ref \bullet \\ \forall\, i : 1..\#dif(tr'_t, tr_t) \bullet \\ fst(dif(tr'_t, tr_t)(i)) = \\ \langle\rangle \downarrow (Event - cs) \wedge \\ ref = (snd(dif(tr'_t, tr_t)(i)) \cup cs) \end{array}\right) \end{array}\right)\right)\right)\right)\right) \\
= \qquad \text{[3.3.1 and Arbitrary refusals]} \\
CSP1_t(R3_t(R1_t((ok' \wedge wait' \wedge trace' = \langle\rangle)))) \\
= \qquad [3.5.5] \\
Stop \qquad \square
\end{array}
$$

L8. $Skip \setminus cs = Skip$
Proof:

$$
\begin{array}{l}
Skip \setminus cs \\
= \qquad [3.5.2] \\
R_t(\exists\, ref \bullet ref = snd(last(tr_t)) \wedge \Pi_t) \setminus cs \\
= \qquad [3.5.55] \\
R_t\left(\begin{array}{l} R_t\left(\begin{array}{l} \exists\, s_t, ref \bullet \\ ref = snd(last(tr_t)) \wedge \\ \Pi_t[s_t/tr'_t] \wedge \\ \qquad dif(tr'_t, tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array}\right) ;\ Skip \end{array}\right) \\
= \qquad \text{[}R_t\text{ is idempotent,closed under ; and } Skip \text{ is } R_t \text{ healthy]} \\
R_t\left(\begin{array}{l} \exists\, s_t, ref \bullet \\ ref = snd(last(tr_t)) \wedge \\ \Pi_t[s_t/tr'_t] \wedge \\ \qquad dif(tr'_t, tr_t) = \\ dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array}\right) ;\ Skip \\
= \qquad [3.4.11]
\end{array}
$$

$$
R_t\left(\begin{array}{l}
\exists\, s_t, ref \bullet \\
ref = snd(last(tr_t)) \wedge \\
\left(\begin{array}{l}(\neg ok \wedge Expands(tr_t, s_t)) \vee \\ (ok' \wedge s_t = tr_t \wedge wait' = wait \wedge state' = state)\end{array}\right) \wedge \\
\quad dif(tr'_t, tr_t) = \\
dif(s_t, tr_t) \downarrow_t (Event - cs)
\end{array}\right) ; \; Skip
$$

$=$ [Propositional and predicate calculus]

$$
R_t\left(\begin{array}{l}
\left(\begin{array}{l}
\exists\, s_t, ref \bullet \\
ref = snd(last(tr_t)) \wedge \\
\neg ok \wedge Expands(tr_t, s_t) \wedge \\
\quad dif(tr'_t, tr_t) = \\
dif(s_t, tr_t) \downarrow_t (Event - cs)
\end{array}\right) \vee \\
\left(\begin{array}{l}
\exists\, s_t, ref \bullet \\
ref = snd(last(tr_t)) \wedge \\
ok' \wedge s_t = tr_t \wedge wait' = wait \wedge state' = state \wedge \\
\quad dif(tr'_t, tr_t) = \\
dif(s_t, tr_t) \downarrow_t (Event - cs)
\end{array}\right)
\end{array}\right) ; \; Skip
$$

$=$ [Substitution]

$$
R_t\left(\begin{array}{l}
\left(\begin{array}{l}
\exists\, s_t, ref \bullet \\
ref = snd(last(tr_t)) \wedge \\
\neg ok \wedge Expands(tr_t, s_t) \wedge \\
\quad dif(tr'_t, tr_t) = \\
dif(s_t, tr_t) \downarrow_t (Event - cs)
\end{array}\right) \vee \\
\left(\begin{array}{l}
\exists\, ref \bullet \\
ref = snd(last(tr_t)) \wedge \\
ok' \wedge wait' = wait \wedge state' = state \wedge \\
\quad dif(tr'_t, tr_t) = \\
dif(tr_t, tr_t) \downarrow_t (Event - cs)
\end{array}\right)
\end{array}\right) ; \; Skip
$$

$=$ [Property 3.3 L1 and L5]

$$
R_t\left(\begin{array}{l}
\left(\begin{array}{l}
\exists\, s_t, ref \bullet \\
ref = snd(last(tr_t)) \wedge \\
\neg ok \wedge \\
\exists\, t \bullet dif(s_t, tr_t) = t \wedge \\
\quad dif(tr'_t, tr_t) = \\
dif(s_t, tr_t) \downarrow_t (Event - cs)
\end{array}\right) \vee \\
\left(\begin{array}{l}
\exists\, ref \bullet \\
ref = snd(last(tr_t)) \wedge \\
ok' \wedge wait' = wait \wedge state' = state \wedge \\
\quad dif(tr'_t, tr_t) = \\
\langle(\langle\rangle, snd(last(tr_t)))\rangle \downarrow_t (Event - cs)
\end{array}\right)
\end{array}\right) ; \; Skip
$$

$=$ [Property 3.13 L2 and substitution]

$$R_t\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\exists\, ref \bullet \\ ref = snd(last(tr_t)) \wedge \\ \neg ok \wedge \\ \exists\, t \bullet \\ \quad dif(tr'_t, tr_t) = t \downarrow_t (Event - cs)\end{array}\right) \vee \\ \left(\begin{array}{l}\exists\, ref \bullet \\ ref = snd(last(tr_t)) \wedge \\ ok' \wedge wait' = wait \wedge state' = state \wedge \\ dif(tr'_t, tr_t) = \langle(\langle\rangle, ref \cup cs)\rangle\end{array}\right)\end{array}\right) ;\ Skip$$

$=$ [Predicate calculus]

$$R_t\left(\begin{array}{l}\exists\, ref \bullet \\ ref = snd(last(tr_t)) \wedge \\ \left(\begin{array}{l}\neg ok \wedge \\ \exists\, t \bullet \\ \quad dif(tr'_t, tr_t) = t \downarrow_t (Event - cs)\end{array}\right) \vee \\ \left(\begin{array}{l} ok' \wedge wait' = wait \wedge state' = state \wedge \\ dif(tr'_t, tr_t) = \langle(\langle\rangle, ref \cup cs)\rangle\end{array}\right)\end{array}\right) ;\ Skip$$

$=$ [Substitute t for $t1$ such that $t1 = t \downarrow_t (Event - cs)$]

$$R_t\left(\begin{array}{l}\exists\, ref \bullet \\ ref = snd(last(tr_t)) \wedge \\ \left(\begin{array}{l}\neg ok \wedge \\ \exists\, t1 \bullet dif(tr'_t, tr_t) = t1\end{array}\right) \vee \\ \left(\begin{array}{l} ok' \wedge wait' = wait \wedge state' = state \wedge \\ dif(tr'_t, tr_t) = \langle(\langle\rangle, ref \cup cs)\rangle\end{array}\right)\end{array}\right) ;\ Skip$$

$=$ [Properties 3.3 L1 and L5]

$$R_t\left(\begin{array}{l}\exists\, ref \bullet \\ ref = snd(last(tr_t)) \wedge \\ \left(\begin{array}{l}\neg ok \wedge Expands(tr_t, tr'_t)\end{array}\right) \vee \\ \left(\begin{array}{l} ok' \wedge wait' = wait \wedge state' = state \wedge \\ tr_t = tr'_t\end{array}\right)\end{array}\right) ;\ Skip$$

$=$ [3.4.11]

$$R_t\left(\begin{array}{l}\exists\, ref \bullet \\ ref = snd(last(tr_t)) \wedge \\ \Pi_t\end{array}\right) ;\ Skip$$

$=$ [3.5.2]

$Skip;\ Skip$

$=$ [Lemma 3.7]

$Skip$ □

L9. $(A \lhd b \rhd B) \setminus cs = (A \setminus cs) \lhd b \rhd (B \setminus cs)$

Proof:

$(A \lhd b \rhd B) \setminus cs$
$=$ [3.5.22]
$((b \wedge A) \vee (\neg b \wedge B)) \setminus cs$
$=$ [Property 3.14 L3]
$((b \wedge A) \setminus cs \vee (\neg b \wedge B) \setminus cs)$
$=$ [b makes no reference to dashed varaibales]
$(b \wedge (A \setminus cs)) \vee (\neg b \wedge (B \setminus cs))$
$=$ [3.5.22]
$(A \setminus cs) \lhd b \rhd (B \setminus cs)$ $\Box$

L10. $((a \rightarrow Skip) \Box (b \rightarrow Skip)) \setminus \{a\} = (Skip \Box (b \rightarrow Skip))$

L11. $((a \rightarrow A) \Box (Wait\ \ \mathsf{n};\ B)) \setminus a = A \setminus \{a\}$

L12. $(A;\ B) \setminus cs = (A \setminus cs);\ (B \setminus cs)$

L13. $Chaos \setminus cs = Chaos$

L14. $(x := e) \setminus cs = x := e$

L15. $((a \rightarrow A) \Box (b \rightarrow B)) \setminus cs = ((a \rightarrow (A \setminus cs)) \Box (b \rightarrow (B \setminus cs)))$
Provided that $(a \notin cs) \wedge (b \notin cs)$

D.10 TIMEOUT

Property 3.17

L1. $Skip \overset{\mathsf{d}}{\rhd} A = Skip$
Proof:

$Skip \overset{\mathsf{d}}{\rhd} A$
$=$ [3.5.59]
$(Skip \Box (Wait\ \ \mathsf{d};\ int \rightarrow A)) \setminus \{int\}$
$=$ [$(Skip \Box Wait\ \ \mathsf{d}) = Skip$]
$Skip\{int\}$
$=$ [Property 3.14 L8]
$Skip$ $\Box$

L2. $Stop \overset{\mathsf{d}}{\rhd} A = Wait\ \ \mathsf{d};\ A$
Proof:

$$
\begin{array}{ll}
& Stop \stackrel{\mathsf{d}}{\triangleright} A \\
= & [3.5.59] \\
& (Stop \;\Box\; (Wait\ \ \mathsf{d};\ int \to A)) \setminus \{int\} \\
= & [\text{Property 3.10 L1}] \\
& (Wait\ \ \mathsf{d};\ int \to A) \setminus \{int\} \\
= & [\text{Property 3.14 L12}] \\
& Wait\ \ \mathsf{d} \setminus \{int\};\ (int \to A) \setminus \{int\} \\
= & [\text{Property 3.14 L6}] \\
& Wait\ \ \mathsf{d};\ (int \to A) \setminus \{int\} \\
= & [\text{Property 3.14 L5}] \\
& Wait\ \ \mathsf{d};\ A \setminus \{int\} \\
= & [\{int\} \text{ not in } A] \\
& Wait\ \ \mathsf{d};\ A \qquad \Box
\end{array}
$$

L3. $(a \to A) \stackrel{\mathsf{d}}{\triangleright} (b \to A) = ((a \to Skip) \stackrel{\mathsf{d}}{\triangleright} (b \to Skip));\ A$
Proof:

$$
\begin{array}{ll}
& (a \to A) \stackrel{\mathsf{d}}{\triangleright} (b \to A) \\
= & [3.5.59] \\
& ((a \to A) \;\Box\; (Wait\ \ \mathsf{d};\ int \to (b \to A))) \setminus \{int\} \\
= & [3.5.20] \\
& (((a \to Skip);\ A) \;\Box\; (Wait\ \ \mathsf{d};\ (int \to Skip);\ ((b \to Skip);\ A))) \setminus \{int\} \\
= & [\text{Property 3.6 L4}] \\
& (((a \to Skip);\ A) \;\Box\; ((Wait\ \ \mathsf{d};\ (int \to Skip);\ (b \to Skip));\ A)) \setminus \{int\} \\
= & [\text{Property 3.10 L6}] \\
& (((a \to Skip) \;\Box\; ((Wait\ \ \mathsf{d};\ (int \to Skip);\ (b \to Skip))));\ A) \setminus \{int\} \\
= & [\text{Property 3.14 L12}] \\
& ((a \to Skip) \;\Box\; ((Wait\ \ \mathsf{d};\ (int \to Skip);\ (b \to Skip)))) \setminus \{int\};\ (A \setminus \{int\}) \\
= & [A \text{ does not communicate on } int] \\
& ((a \to Skip) \;\Box\; ((Wait\ \ \mathsf{d};\ (int \to Skip);\ (b \to Skip)))) \setminus \{int\};\ A \\
= & [3.5.59] \\
& ((a \to Skip) \stackrel{\mathsf{d}}{\triangleright} (b \to Skip));\ A \qquad \Box
\end{array}
$$

L4. $A \stackrel{\mathsf{d}}{\triangleright} (B \sqcap C) = (A \stackrel{\mathsf{d}}{\triangleright} B) \sqcap (A \stackrel{\mathsf{d}}{\triangleright} C)$
Proof:

$$
\begin{array}{ll}
& A \stackrel{\mathsf{d}}{\triangleright} (B \sqcap C) \\
= & [3.5.59] \\
& (A \,\Box\, (Wait\ \ \mathsf{d};\ int \to (B \sqcap C))) \setminus \{int\} \\
= & [3.5.20] \\
& (A \,\Box\, (Wait\ \ \mathsf{d};\ (int \to Skip);\ (B \sqcap C))) \setminus \{int\} \\
= & [\text{Property 3.9 L6}] \\
& (A \,\Box\, (Wait\ \ \mathsf{d};\ (((int \to Skip);\ B) \sqcap ((int \to Skip);\ C)))) \setminus \{int\} \\
= & [\text{Property 3.9 L6}] \\
& (A \,\Box\, ((Wait\ \ \mathsf{d};\ ((int \to Skip);\ B)) \sqcap (Wait\ \ \mathsf{d};\ ((int \to Skip);\ C)))) \setminus \{int\} \\
= & [\text{Property 3.10 L5}] \\
& \left(\begin{array}{l} (A \,\Box\, (Wait\ \ \mathsf{d};\ ((int \to Skip);\ B))) \\ \sqcap\, (A \,\Box\, (Wait\ \ \mathsf{d};\ ((int \to Skip);\ C))) \end{array} \right) \setminus \{int\} \\
= & [\text{Property 3.14 L3}] \\
& ((A \,\Box\, (Wait\ \ \mathsf{d};\ ((int \to Skip);\ B)))) \setminus \{int\} \\
& \sqcap\, ((A \,\Box\, (Wait\ \ \mathsf{d};\ ((int \to Skip);\ C)))) \setminus \{int\} \\
= & [3.5.59] \\
& (A \stackrel{\mathsf{d}}{\triangleright} B) \sqcap (A \stackrel{\mathsf{d}}{\triangleright} C) \qquad \Box
\end{array}
$$

L5. $(A \sqcap B) \stackrel{\mathsf{d}}{\triangleright} C = (A \stackrel{\mathsf{d}}{\triangleright} C) \sqcap (B \stackrel{\mathsf{d}}{\triangleright} C)$
Proof:

$$
\begin{array}{ll}
& (A \sqcap B) \stackrel{\mathsf{d}}{\triangleright} C \\
= & [3.5.59] \\
& ((A \sqcap B) \,\Box\, (Wait\ \ \mathsf{d};\ int \to C)) \setminus \{int\} \\
= & [\text{Property 3.10 L5}] \\
& ((A \,\Box\, (Wait\ \ \mathsf{d};\ int \to C)) \sqcap (B \,\Box\, (Wait\ \ \mathsf{d};\ int \to C))) \setminus \{int\} \\
= & [\text{Property 3.14 L3}] \\
& (A \,\Box\, (Wait\ \ \mathsf{d};\ int \to C)) \setminus \{int\} \\
& \sqcap\, (B \,\Box\, (Wait\ \ \mathsf{d};\ int \to C)) \setminus \{int\} \\
= & [3.5.59] \\
& (A \stackrel{\mathsf{d}}{\triangleright} C) \sqcap (B \stackrel{\mathsf{d}}{\triangleright} C) \qquad \Box
\end{array}
$$

L6. $(A \stackrel{\mathsf{d}+\delta}{\triangleright} B) \stackrel{\mathsf{d}}{\triangleright} C = A \stackrel{\mathsf{d}}{\triangleright} C$ providing that $\delta > 0$
Proof:

$$
\begin{array}{ll}
& (A \overset{\mathsf{d}+\delta}{\triangleright} B) \overset{\mathsf{d}}{\triangleright} C \\
= & [3.5.59] \\
& ((A \,\Box\, (Wait\ \ \mathsf{d}+\delta;\ int \to B)) \setminus \{int\}) \overset{\mathsf{d}}{\triangleright} C \\
= & [3.5.59] \\
& \left(\begin{array}{l} ((A \,\Box\, (Wait\ \ \mathsf{d}+\delta;\ int_1 \to B)) \setminus \{int_1\}) \\ \Box\, (Wait\ \ \mathsf{d};\ int_2 \to C) \end{array} \right) \setminus \{int_2\} \\
= & [\text{Property of Hiding}] \\
& \left(\begin{array}{l} (A \,\Box\, (Wait\ \ \mathsf{d}+\delta;\ int_1 \to B)) \\ \Box\, (Wait\ \ \mathsf{d};\ int_2 \to C) \end{array} \right) \setminus \{int_1, int_2\} \\
= & [\text{Property 3.6 L4}] \\
& \left(\begin{array}{l} (A \,\Box\, (Wait\ \ \mathsf{d};\ Wait\ \ \delta;\ int_1 \to B)) \\ \Box\, (Wait\ \ \mathsf{d};\ int_2 \to C) \end{array} \right) \setminus \{int_1, int_2\} \\
= & [\text{Property 3.10 L4 and L8}] \\
& (A \,\Box\, Wait\ \ \mathsf{d};\ ((Wait\ \ \delta;\ int_1 \to B) \,\Box\, (int_2 \to C))) \setminus \{int_1, int_2\} \\
= & [\text{Property of Hiding}] \\
& (A \,\Box\, Wait\ \ \mathsf{d};\ ((Wait\ \ \delta;\ int_1 \to B) \,\Box\, (int_2 \to C)) \setminus \{int_1, int_2\}) \\
= & [\text{Property 3.14 L11}] \\
& (A \,\Box\, Wait\ \ \mathsf{d};\ (C) \setminus \{int_1, int_2\}) \\
= & [\text{Property 3.14 L5}] \\
& (A \,\Box\, Wait\ \ \mathsf{d};\ (int_1 \to C)) \setminus \{int_1\} \\
= & [3.5.59] \\
& A \overset{\mathsf{d}}{\triangleright} C
\end{array}
$$

L7. $(Wait\ \ \mathsf{d};\ A) \overset{\mathsf{d}+\mathsf{d}'}{\triangleright} B = Wait\ \ \mathsf{d};\ (A \overset{\mathsf{d}'}{\triangleright} B)$
Proof:

$$
\begin{array}{ll}
& (Wait\ \ \mathsf{d};\ A) \overset{\mathsf{d}+\mathsf{d}'}{\triangleright} B \\
= & [3.5.59] \\
& ((Wait\ \ \mathsf{d};\ A) \,\Box\, (Wait\ \ \mathsf{d}+\mathsf{d}';\ int \to B)) \setminus \{int\} \\
= & [\text{Property 3.10 L8}] \\
& (Wait\ \ \mathsf{d};\ (A \,\Box\, (Wait\ \ \mathsf{d}';\ int \to B))) \setminus \{int\} \\
= & [\text{Property 3.14 L12}] \\
& Wait\ \ \mathsf{d} \setminus \{int\};\ (A \,\Box\, (Wait\ \ \mathsf{d}';\ int \to B)) \setminus \{int\} \\
= & [\text{Property 3.14 L6}]
\end{array}
$$

$$
\begin{array}{ll}
& \mathit{Wait}\ \ \mathsf{d};\ (A \,\Box\, (\mathit{Wait}\ \ \mathsf{d}';\ \mathit{int} \to B)) \setminus \{\mathit{int}\} \\
= & [3.5.59] \\
& \mathit{Wait}\ \ \mathsf{d};\ (A \overset{\mathsf{d}'}{\triangleright} B) \qquad \Box
\end{array}
$$

L8. $(\mathit{Wait}\ \ \mathsf{d} + \mathsf{d}';\ A) \overset{\mathsf{d}}{\triangleright} B = \mathit{Wait}\ \ \mathsf{d};\ B$
Proof:

$$
\begin{array}{ll}
& (\mathit{Wait}\ \ \mathsf{d} + \mathsf{d}';\ A) \overset{\mathsf{d}}{\triangleright} B \\
= & [3.5.59] \\
& ((\mathit{Wait}\ \ \mathsf{d} + \mathsf{d}';\ A) \,\Box\, (\mathit{Wait}\ \ \mathsf{d};\ \mathit{int} \to B)) \setminus \{\mathit{int}\} \\
= & [\text{Property 3.10 L3 and L8}] \\
& (\mathit{Wait}\ \ \mathsf{d};\ ((\mathit{Wait}\ \ \mathsf{d}';\ A) \,\Box\, (\mathit{int} \to B))) \setminus \{\mathit{int}\} \\
= & [\text{Property 3.14 L12}] \\
& (\mathit{Wait}\ \ \mathsf{d} \setminus \{\mathit{int}\});\ ((\mathit{Wait}\ \ \mathsf{d}';\ A) \,\Box\, (\mathit{int} \to B)) \setminus \{\mathit{int}\} \\
= & [\text{Property 3.14 L6}] \\
& \mathit{Wait}\ \ \mathsf{d};\ ((\mathit{Wait}\ \ \mathsf{d}';\ A) \,\Box\, (\mathit{int} \to B)) \setminus \{\mathit{int}\} \\
= & [\text{Property 3.14 L11}] \\
& \mathit{Wait}\ \ \mathsf{d};\ B \qquad \Box
\end{array}
$$

L9. $(A \,\Box\, B) \overset{\mathsf{d}}{\triangleright} C = (A \overset{\mathsf{d}}{\triangleright} C) \,\Box\, (B \overset{\mathsf{d}}{\triangleright} C)$
Proof:

$$
\begin{array}{ll}
& (A \overset{\mathsf{d}}{\triangleright} C) \,\Box\, (B \overset{\mathsf{d}}{\triangleright} C) \\
= & [3.5.59] \\
& (A \,\Box\, (\mathit{Wait}\ \ \mathsf{d};\ \mathit{int}_1 \to C)) \setminus \{\mathit{int}_1\} \\
& \Box \\
& (B \,\Box\, (\mathit{Wait}\ \ \mathsf{d};\ \mathit{int}_2 \to C)) \setminus \{\mathit{int}_2\} \\
= & [\text{Property of hiding}] \\
& \left(\begin{array}{l} (A \,\Box\, (\mathit{Wait}\ \ \mathsf{d};\ \mathit{int}_1 \to C)) \\ \Box \\ (B \,\Box\, (\mathit{Wait}\ \ \mathsf{d};\ \mathit{int}_2 \to C)) \end{array} \right) \setminus \{\mathit{int}_1, \mathit{int}_2\} \\
= & [\text{Property 3.10 L4}] \\
& \left(\begin{array}{l} (A \,\Box\, B) \,\Box \\ ((\mathit{Wait}\ \ \mathsf{d};\ \mathit{int}_1 \to C) \,\Box\, (\mathit{Wait}\ \ \mathsf{d};\ \mathit{int}_2 \to C)) \end{array} \right) \setminus \{\mathit{int}_1, \mathit{int}_2\} \\
= & [\text{Property 3.10 L8}] \\
& \left(\begin{array}{l} (A \,\Box\, B) \,\Box \\ \mathit{Wait}\ \ \mathsf{d};\ ((\mathit{int}_1 \to C) \,\Box\, (\mathit{int}_2 \to C)) \end{array} \right) \setminus \{\mathit{int}_1, \mathit{int}_2\}
\end{array}
$$

$$
\begin{array}{ll}
= & \text{[Property 3.10 L6]} \\
\left(\begin{array}{l} (A \,\Box\, B) \,\Box \\ \mathit{Wait}\ \ \mathsf{d};\ (((\mathit{int}_1 \rightarrow \mathit{Skip}) \,\Box\, (\mathit{int}_2 \rightarrow \mathit{Skip}));\ C) \end{array} \right) \setminus \{\mathit{int}_1, \mathit{int}_2\} & \\
= & [\mathit{int}_1 \text{ and } \mathit{int}_2 \text{ are internal}] \\
\left(\begin{array}{l} (A \,\Box\, B) \,\Box \\ \mathit{Wait}\ \ \mathsf{d};\ (\mathit{int} \rightarrow \mathit{Skip});\ C \end{array} \right) \setminus \{\mathit{int}\} & \\
= & \text{[3.5.59]} \\
(A \,\Box\, B) \stackrel{\mathsf{d}}{\triangleright} C &
\end{array}
$$

L10. $A \stackrel{0}{\triangleright} B = (A \,\Box\, (\mathit{int} \rightarrow B)) \setminus \{\mathit{int}\}$
Proof:

$$
\begin{array}{ll}
A \stackrel{0}{\triangleright} B & \\
= & \text{[3.5.59]} \\
(A \,\Box\, (\mathit{Wait}\ \ 0;\ \mathit{int} \rightarrow B)) \setminus \{\mathit{int}\} & \\
= & \text{[Property 3.5 L3]} \\
(A \,\Box\, (\mathit{Skip};\ \mathit{int} \rightarrow B)) \setminus \{\mathit{int}\} & \\
= & [\text{Communication is } \mathit{CSP}3_t \text{ healthy}] \\
(A \,\Box\, (\mathit{int} \rightarrow B)) \setminus \{\mathit{int}\} \qquad \Box &
\end{array}
$$

L11. $\mathit{Wait}\ \ 0 \stackrel{\mathsf{d}}{\triangleright} B = \mathit{Skip}$
Proof:

$$
\begin{array}{ll}
\mathit{Wait}\ \ 0 \stackrel{\mathsf{d}}{\triangleright} B & \\
= & \text{[3.5.59]} \\
(\mathit{Wait}\ \ 0 \,\Box\, (\mathit{Wait}\ \ \mathsf{d};\ \mathit{int} \rightarrow B)) \setminus \{\mathit{int}\} & \\
= & \text{[Property 3.5 L3]} \\
(\mathit{Skip} \,\Box\, (\mathit{Wait}\ \ \mathsf{d};\ \mathit{int} \rightarrow B)) \setminus \{\mathit{int}\} & \\
= & \text{[Property 3.10 L11]} \\
(\mathit{Skip}) \setminus \{\mathit{int}\} & \\
= & \text{[Property 3.14 L8]} \\
\mathit{Skip} \qquad \Box &
\end{array}
$$

L12. $A \stackrel{\mathsf{d}}{\triangleright} (B \,\Box\, C) = (A \stackrel{\mathsf{d}}{\triangleright} B) \,\Box\, (A \stackrel{\mathsf{d}}{\triangleright} C)$

APPENDIX E

PROOFS OF CHAPTER 4

In this appendix we give the proofs of the properties and laws introduced in Chapter 4.

E.1 GENERAL PROPERTIES OF FUNCTION L

The following are some general algebraic properties of L needed in the subsequent proofs.

Property 4.1

L1. L is idempotent
$L(L(X)) = L(X)$
Proof:

$$
\begin{array}{ll}
& L(L(X)) \\
= & \qquad [4.2.1 \text{ and } 4.2.2] \\
& \exists\, tr_t, tr'_t \bullet L(X) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \\
= & \qquad [4.2.1 \text{ and } 4.2.2] \\
& \exists\, tr_t, tr'_t \bullet \exists\, tr_t, tr'_t \bullet X \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \\
= & \qquad [\text{Predicate calculus}] \\
& \exists\, tr_t, tr'_t \bullet X \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \\
= & \qquad [4.2.1 \text{ and } 4.2.2] \\
& L(X) \qquad \Box
\end{array}
$$

L2. L distributes over disjunction
$L(X \vee Y) = L(X) \vee L(Y)$
Proof:

$$
\begin{array}{ll}
& L(X \vee Y) \\
= & \qquad [4.2.1 \text{ and } 4.2.2] \\
& \exists\, tr_t, tr'_t \bullet (X \vee Y) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \\
= & \qquad [\text{Propositional Calculus}] \\
& \exists\, tr_t, tr'_t \bullet \left(\begin{array}{l} X \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \vee \\ Y \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right) \\
= & \qquad [\text{Predicate calculus}] \\
& \left(\begin{array}{l} \exists\, tr_t, tr'_t \bullet \left(X \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \right) \vee \\ \exists\, tr_t, tr'_t \bullet \left(Y \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \right) \end{array} \right) \\
= & \qquad [4.2.1 \text{ and } 4.2.2] \\
& L(X) \vee L(Y) \qquad \Box
\end{array}
$$

L3. Weak distribution over conjunction
$L(X \wedge Y) \Rightarrow L(X) \wedge L(Y)$
Proof:

$$
\begin{array}{ll}
& L(X \wedge Y) \\
= & \qquad [4.2.1 \text{ and } 4.2.2] \\
& \exists\, tr_t, tr'_t \bullet (X \wedge Y) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right)
\end{array}
$$

$$
\begin{array}{ll}
= & \text{[Propositional Calculus]} \\
\exists\, tr_t, tr'_t \bullet \left(\begin{array}{l} X \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \\ Y \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right) \wedge & \\
\Rightarrow & \text{[Predicate calculus]} \\
\exists\, tr_t, tr'_t \bullet \left(X \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \right) \wedge & \\
\exists\, tr_t, tr'_t \bullet \left(Y \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \right) & \\
= & \text{[4.2.1 and 4.2.2]} \\
L(X) \wedge L(Y) & \square
\end{array}
$$

L4. Distribution over conjunction
$L(X \wedge Y) = L(X) \wedge Y$
provided that tr_t and tr'_t are not free in Y
Proof:

$$
\begin{array}{ll}
L(X \wedge Y) & \\
= & \text{[4.2.1 and 4.2.2]} \\
\exists\, tr_t, tr'_t \bullet (X \wedge Y) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) & \\
= & \text{[Assumption]} \\
\left(\exists\, tr_t, tr'_t \bullet X \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \right) \wedge Y & \\
= & \text{[4.2.1 and 4.2.2]} \\
L(X) \wedge Y & \square
\end{array}
$$

L5. Distribution over conditional choice

$L(X \lhd b \rhd Y) = L(X) \lhd b \rhd L(Y)$
provided that tr_t and tr_t are not free in the expression b
Proof:

$$
\begin{array}{ll}
& L(X \lhd b \rhd Y) \\
= & \qquad [3.5.22] \\
& L((X \wedge b) \vee (Y \wedge \neg b)) \\
= & \qquad [\text{Property 4.1 L2}] \\
& L(X \wedge b) \vee L(Y \wedge \neg b) \\
= & \qquad [\text{Property 4.1 L4}] \\
& (L(X) \wedge b) \vee (L(Y) \wedge \neg b) \\
= & \qquad [3.5.22] \\
& L(X) \lhd b \rhd L(Y) \qquad \Box
\end{array}
$$

L6. $L(X;\ Y) = L(X);\ L(Y)$
Proof:

$$
\begin{array}{ll}
& L(X);\ L(Y) \\
= & \qquad [4.2.1 \text{ and } 4.2.2] \\
& \left(\exists\, tr_t, tr'_t \bullet X \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \right); \\
& \left(\exists\, tr_t, tr'_t \bullet Y \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \right) \\
= & \qquad [3.5.21] \\
& \exists\, tr_{t_o}, tr_o, ref_o, v_o \bullet \\
& \left(\begin{array}{l} \exists\, tr_t, tr_{t_o} \bullet \\ X[tr_{t_o}, v_o / tr'_t, v'] \wedge \\ \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr_o = Flat(tr_{t_o}) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref_o = snd(last(tr_{t_o})) \end{array} \right) \end{array} \right) \wedge \\
& \left(\begin{array}{l} \exists\, tr_{t_o}, tr'_t \bullet \\ Y[tr_{t_o}, v_o / tr_t, v] \wedge \\ \left(\begin{array}{c} tr_o = Flat(tr_{t_o}) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref_o = snd(last(tr_{t_o})) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right)
\end{array}
$$

$$
\begin{array}{ll}
= & \text{[Predicate calculus]} \\
\exists\, tr_t, tr'_t \bullet & \\
\left(\begin{array}{l} \exists\, tr_{t_o}, tr_o, ref_o, v_o \bullet \\ \left(\begin{array}{l} X[tr_{t_o}, v_o / tr'_t, v'] \wedge \\ Y[tr_{t_o}, v_o / tr_t, v] \wedge \\ \left(\begin{array}{c} tr_o = Flat(tr_{t_o}) \wedge \\ ref_o = snd(last(tr_{t_o})) \end{array}\right) \end{array}\right) \end{array}\right) \wedge & \\
\left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array}\right) & \\
= & [tr_o \text{ and } ref_o \text{ not free}] \\
\exists\, tr_t, tr'_t \bullet & \\
\left(\begin{array}{l} \exists\, tr_{t_o}, v_o \bullet \\ \left(\begin{array}{l} X[tr_{t_o}, v_o / tr'_t, v'] \wedge \\ Y[tr_{t_o}, v_o / tr_t, v] \wedge \end{array}\right) \end{array}\right) \wedge & \\
\left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array}\right) & \\
= & \text{[3.5.21]} \\
\exists\, tr_t, tr'_t \bullet & \\
(X;\ Y) \wedge & \\
\left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array}\right) & \\
= & \text{[4.2.1 and 4.2.2]} \\
L(X;\ Y) & \square
\end{array}
$$

L7. $L(Expands(tr_t, tr'_t)) = tr \leq tr'$

Proof:

First we prove that $L(Expands(tr_t, tr'_t)) \Rightarrow tr \leq tr'$

$$
\begin{array}{ll}
L(Expands(tr_t, tr'_t)) & \\
= & \text{[4.2.1]} \\
\left(\begin{array}{l} \exists\, tr_t, tr'_t \bullet Expands(tr_t, tr'_t) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array}\right) \end{array}\right) & \\
\Rightarrow & \text{[Property 3.2 L1]}
\end{array}
$$

$$\exists\, tr_t, tr'_t \bullet Flat(tr_t) \leq Flat(tr'_t) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array}\right)$$

$=$ [Substitution]

$$\forall\, tr_t, tr'_t \bullet tr \leq tr' \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array}\right)$$

$=$ [Predicate calculus]

$$tr \leq tr' \wedge \forall\, tr_t, tr'_t \bullet \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array}\right)$$

$\Rightarrow$ [Propositional calculus]

$tr \leq tr'$ $\Box$

Next we prove that $L(Expands(tr_t, tr'_t)) \Leftarrow tr \leq tr'$

$L(Expands(tr_t, tr'_t))$

$=$ [4.2.1]

$$\exists\, tr_t, tr'_t \bullet Expands(tr_t, tr'_t) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array}\right)$$

$=$ [3.4.3]

$$\exists\, tr_t, tr'_t \bullet \left(\begin{array}{c} (front(tr_t) \leq tr'_t) \wedge \\ (fst(last(tr_t)) \leq fst(tr'_t(\#\ tr_t))) \wedge \\ tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t) \end{array}\right)$$

$\Leftarrow$ [Let $tr_t = \langle (tr, ref) \rangle$ and $tr'_t = \langle (tr', ref') \rangle$]

$$\left(\begin{array}{c} (front(\langle (tr, ref) \rangle) \leq \langle (tr', ref') \rangle) \wedge \\ (fst(last(\langle (tr, ref) \rangle)) \leq fst(\langle (tr', ref') \rangle(\#\ \langle (tr, ref) \rangle))) \wedge \\ tr = Flat(\langle (tr, ref) \rangle) \wedge tr' = Flat(\langle (tr', ref') \rangle) \wedge \\ ref = snd(last(\langle (tr, ref) \rangle)) \wedge ref' = snd(last(\langle (tr', ref') \rangle)) \end{array}\right)$$

$=$ [properties of *front*,*snd* and *Flat*]

$$\left(\begin{array}{c} (\langle\rangle \leq \langle (tr', ref') \rangle) \wedge \\ (tr \leq tr') \wedge \\ tr = tr \wedge tr' = tr' \wedge \\ ref = ref \wedge ref' = ref' \end{array}\right)$$

$$
\begin{array}{ll}
= & \text{[Propositional calculus]} \\
(\langle\rangle \leq \langle(tr', ref')\rangle) \wedge (tr \leq tr') & \\
= & \text{[sequence properties]} \\
(tr \leq tr') & \Box
\end{array}
$$

L8. $L(tr'_t = tr_t) = (tr' = tr \wedge ref' = ref)$

Proof:

The first part of the proof shows that $L(tr'_t = tr_t) \Leftarrow (tr' = tr \wedge ref' = ref)$

$$
\begin{array}{ll}
L(tr'_t = tr_t) & \\
= & \text{[4.2.1]} \\
\exists\, tr_t, tr'_t \bullet (tr'_t = tr_t) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) & \\
\Leftarrow & \text{[Let } tr_t = \langle(tr, ref)\rangle \text{ and } tr'_t = \langle(tr', ref')\rangle] \\
(\langle(tr', ref')\rangle = \langle(tr, ref)\rangle) \wedge \left(\begin{array}{c} tr = tr \wedge tr' = tr' \wedge \\ ref = ref \wedge ref' = ref' \end{array} \right) & \\
= & \text{[Property of equality]} \\
(\langle(tr', ref')\rangle = \langle(tr, ref)\rangle) & \\
= & \text{[sequence properties]} \\
(tr' = tr \wedge ref' = ref) &
\end{array}
$$

Next we show that $L(tr'_t = tr_t) \Rightarrow (tr' = tr \wedge ref' = ref)$

$$
\begin{array}{ll}
L(tr'_t = tr_t) & \\
= & \text{[4.2.1]} \\
\exists\, tr_t, tr'_t \bullet (tr'_t = tr_t) \wedge \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) & \\
= & \text{[property 3.2 L4,L5]} \\
\left(\begin{array}{l} \exists\, tr_t, tr'_t \bullet \left(\begin{array}{c} Flat(tr_t) = Flat(tr'_t) \wedge \\ last(tr_t) = last(tr'_t) \end{array} \right) \wedge \\ \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right) & \\
= & \text{[Substitution]}
\end{array}
$$

$$\left(\begin{array}{l}(tr = tr' \wedge ref = ref') \wedge \exists\, tr_t, tr_t' \bullet \\ \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr_t') \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr_t')) \end{array}\right)\end{array}\right)$$

$\Rightarrow$ [propositional calculus]

$(tr' = tr \wedge ref' = ref)$ $\Box$

L9. $L(\Pi_t) = \Pi$

Proof:

$L(\Pi_t) =$ [3.4.11]

$$L\left(\begin{array}{l}(\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ (ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state))\end{array}\right)$$

$=$ [Property 4.1 L2]

$$\left(\begin{array}{l}L(\neg ok \wedge Expands(tr_t, tr_t')) \vee \\ L(ok' \wedge (tr_t' = tr_t) \wedge (wait' = wait) \wedge (state' = state))\end{array}\right)$$

$=$ [Property 4.1 L4]

$$\left(\begin{array}{l}(\neg ok \wedge L(Expands(tr_t, tr_t'))) \vee \\ (ok' \wedge (wait' = wait) \wedge (state' = state) \wedge L(tr_t' = tr_t))\end{array}\right)$$

$=$ [Property 4.1 L7]

$$\left(\begin{array}{l}(\neg ok \wedge tr \leq tr') \vee \\ (ok' \wedge (wait' = wait) \wedge (state' = state) \wedge L(tr_t' = tr_t))\end{array}\right)$$

$=$ [Property 4.1 L8]

$$\left(\begin{array}{l}(\neg ok \wedge tr \leq tr') \vee \\ (ok' \wedge (wait' = wait) \wedge (state' = state) \wedge (tr' = tr) \wedge (ref' = ref))\end{array}\right)$$

$=$ [3.4.9]

Π $\Box$

L10. $L(trace' = s) = ((tr' - tr) = s)$

For any arbitrary trace s.

Proof:

First we show that $L(trace' = s) \Rightarrow ((tr' - tr) = s)$

$L(trace' = s)$

$=$ [3.3.1]

$L(Flat(tr_t') - Flat(tr_t) = s)$

$=$ [4.2.1]

$$\left(\begin{array}{l}\exists\, tr_t, tr_t' \bullet Flat(tr_t') - Flat(tr_t) = s \wedge \\ tr = Flat(tr_t) \wedge tr' = Flat(tr_t') \wedge \\ ref = snd(last(tr_t)) \wedge ref' = snd(last(tr_t'))\end{array}\right)$$

$$
\begin{array}{ll}
= & \text{[Susbtitution]} \\
\left(\begin{array}{l} \exists\, tr_t, tr'_t \bullet tr' - tr = s \wedge \\ tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{array} \right) & \\
= & \text{[Predicate calculus]} \\
tr' - tr = s \wedge \left(\begin{array}{l} \exists\, tr_t, tr'_t \bullet \\ tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{array} \right) & \\
\Rightarrow & \text{[Predicate calculus]} \\
tr' - tr = s &
\end{array}
$$

Next we show that $L(trace' = s) \Leftarrow (tr' - tr = s)$

$$
\begin{array}{ll}
L(trace' = s) & \\
= & [3.3.1] \\
L(Flat(tr'_t) - Flat(tr_t) = s) & \\
= & [4.2.1] \\
\left(\begin{array}{l} \exists\, tr_t, tr'_t \bullet Flat(tr'_t) - Flat(tr_t) = s \wedge \\ tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{array} \right) & \\
\Leftarrow & \text{[Let } tr_t = \langle (tr, ref) \rangle \text{ and } tr'_t = \langle (tr', ref') \rangle] \\
\left(\begin{array}{l} Flat(\langle (tr', ref') \rangle) - Flat(\langle (tr, ref) \rangle) = s \wedge \\ tr = Flat(\langle (tr, ref) \rangle) \wedge tr' = Flat(\langle (tr', ref') \rangle) \wedge \\ ref = snd(last(\langle (tr, ref) \rangle)) \wedge ref' = snd(last(\langle (tr', ref') \rangle)) \end{array} \right) & \\
= & \text{[Properties of } Flat\text{, snd and last]} \\
\left(\begin{array}{l} tr' - tr = s \wedge tr = tr \wedge tr' = tr' \wedge \\ ref = ref \wedge ref' = ref' \end{array} \right) & \\
= & \text{[Property of equality]} \\
tr' - tr = s & \Box
\end{array}
$$

L11. $L(wait\_com(c)) = (wait' \wedge c \notin ref' \wedge tr' = tr)$

Proof:

First we show that $L(wait\_com(c)) \Rightarrow (wait' \wedge c \notin ref' \wedge tr' = tr)$

$$
\begin{array}{ll}
L(wait\_com(c)) & \\
= & [3.5.13] \\
L(wait' \wedge possible(tr_t, tr'_t, c) \wedge trace' = \langle\rangle) &
\end{array}
$$

$$
\begin{array}{ll}
= & [3.5.14] \\
L(wait' \wedge \forall\, i : \#tr_t..\#tr'_t \bullet c \notin snd(tr'_t(i)) \wedge trace' = \langle\rangle) & \\
= & [\text{property 4.1 L4}] \\
wait' \wedge L(\forall\, i : \#tr_t..\#tr'_t \bullet c \notin snd(tr'_t(i)) \wedge trace' = \langle\rangle) & \\
= & [4.2.1] \\
wait' \wedge \left(\begin{array}{c} \exists\, tr_t, tr'_t \bullet \\ \forall\, i : \#tr_t..\#tr'_t \bullet c \notin snd(tr'_t(i)) \wedge \\ trace' = \langle\rangle \wedge \\ \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right) & \\
\Rightarrow \qquad [((\forall\, i : \#tr_t..\#tr'_t c \notin snd(tr'_t(i))) \wedge ref' = snd(last(tr'_t))) \Rightarrow c \notin ref'] & \\
wait' \wedge c \notin ref' \wedge \left(\begin{array}{c} \exists\, tr_t, tr'_t \bullet \\ \forall\, i : \#tr_t..\#tr'_t \bullet c \notin snd(tr'_t(i)) \wedge \\ trace' = \langle\rangle \wedge \\ \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right) & \\
\Rightarrow & [\text{Predicate calculus and Property 4.1 L10}] \\
wait' \wedge c \notin ref' \wedge tr' = tr & \square
\end{array}
$$

Next we show that $L(wait\_com(c)) \Leftarrow (wait' \wedge c \notin ref' \wedge tr' = tr)$

$$
\begin{array}{ll}
L(wait\_com(c)) & \\
= & [3.5.13] \\
L(wait' \wedge possible(tr_t, tr'_t, c) \wedge trace' = \langle\rangle) & \\
= & [3.5.14] \\
L(wait' \wedge \forall\, i : \#tr_t..\#tr'_t \bullet c \notin snd(tr'_t(i)) \wedge trace' = \langle\rangle) & \\
= & [\text{property 4.1 L4}] \\
wait' \wedge L(\forall\, i : \#tr_t..\#tr'_t \bullet c \notin snd(tr'_t(i)) \wedge trace' = \langle\rangle) & \\
= & [4.2.1] \\
wait' \wedge \left(\begin{array}{c} \exists\, tr_t, tr'_t \bullet \\ \forall\, i : \#tr_t..\#tr'_t \bullet c \notin snd(tr'_t(i)) \wedge \\ trace' = \langle\rangle \wedge \\ \left(\begin{array}{c} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right) &
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \text{[Let } tr_t = \langle(\langle\rangle, ref)\rangle \text{ and } tr'_t = \langle(\langle\rangle, ref')\rangle\text{]} \\
wait' \wedge \left(\begin{array}{c} \forall\, i : \#\langle(\langle\rangle, ref)\rangle..\#\langle(\langle\rangle, ref')\rangle \bullet c \notin snd(\langle(\langle\rangle, ref')\rangle(i)) \wedge \\ trace' = \langle\rangle \wedge \\ \left(\begin{array}{c} tr = Flat(\langle(\langle\rangle, ref)\rangle) \wedge \\ tr' = Flat(\langle(\langle\rangle, ref')\rangle) \wedge \\ ref = snd(last(\langle(\langle\rangle, ref)\rangle)) \wedge \\ ref' = snd(last(\langle(\langle\rangle, ref')\rangle)) \end{array} \right) \end{array} \right) & \\
= & \text{[Predicate calculus and Property 4.1 L10]} \\
wait' \wedge c \notin ref' \wedge tr' = tr & \square
\end{array}
$$

L12. $L(term\_com(c.e)) = (\neg wait' \wedge tr' - tr = \langle c.e\rangle)$

Proof:

First we show that $L(term\_com(c.e)) \Rightarrow (\neg wait' \wedge tr' - tr = \langle c.e\rangle)$

$$
\begin{array}{ll}
L(term\_com(c)) & \\
= & [3.5.15] \\
L(\neg wait' \wedge trace' = \langle c.e\rangle \wedge \#tr'_t = \#tr_t) & \\
= & \text{[Property 4.1 L4]} \\
\neg wait' \wedge L(trace' = \langle c.e\rangle \wedge \#tr'_t = \#tr_t) & \\
= & \text{[Property 4.1 L10]} \\
\neg wait' \wedge tr' - tr = \langle c.e\rangle \wedge L(\#tr'_t = \#tr_t) & \\
\Rightarrow & \text{[Propositional calculus]} \\
\neg wait' \wedge tr' - tr = \langle c.e\rangle & \square
\end{array}
$$

Next we show that $L(term\_com(c)) \Leftarrow (\neg wait' \wedge tr' - tr = \langle c\rangle)$

$$
\begin{array}{ll}
L(term\_com(c)) & \\
= & [3.5.15] \\
L(\neg wait' \wedge trace' = \langle c.e\rangle \wedge \#tr'_t = \#tr_t) & \\
= & \text{[Property 4.1 L4]} \\
\neg wait' \wedge L(trace' = \langle c.e\rangle \wedge \#tr'_t = \#tr_t) & \\
= & [4.2.1] \\
\begin{array}{l} \neg wait' \wedge \\ \quad \exists\, tr_t, tr'_t \bullet \\ \quad Flat(tr_t) - Flat(tr'_t) = \langle c.e\rangle \wedge \\ \quad \#tr'_t = \#tr_t \\ \quad tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \\ \quad ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{array} & \\
\Leftarrow & \text{[Let } tr_t = \langle(tr, ref)\rangle \text{ and } tr'_t = \langle(tr', ref')\rangle\text{]}
\end{array}
$$

$$
\begin{array}{l}
\neg wait' \wedge \\
\quad Flat(\langle (tr, ref) \rangle) - Flat(\langle (tr', ref') \rangle) = \langle c.e \rangle \wedge \\
\quad \#\langle (tr', ref') \rangle = \#\langle (tr, ref) \rangle \\
\quad tr = Flat(\langle (tr, ref) \rangle) \wedge tr' = Flat(\langle (tr', ref') \rangle) \\
\quad ref = snd(last(\langle (tr, ref) \rangle)) \wedge ref' = snd(last(\langle (tr', ref') \rangle))
\end{array}
$$

$=$ [Property of Flat and propositional calculus]

$$
\begin{array}{l}
\neg wait' \wedge \\
\quad tr - tr' = \langle c.e \rangle \wedge \\
\quad tr = tr \wedge tr' = tr' \\
\quad ref = ref \wedge ref' ref'
\end{array}
$$

$=$ [propositional calculus]

$\neg wait' \wedge tr' - tr = \langle c.e \rangle$ $\square$

L13. $L(terminating\_com(c.e)) = (\neg wait' \wedge tr' - tr = \langle c.e \rangle)$
Proof:

$L(terminating\_com(c.e))$

$=$ [3.5.16]

$$
L\left(\begin{array}{l} (wait\_com(c);\ term\_com(c.e)) \vee \\ (term\_com(c.e)) \end{array} \right)
$$

$=$ [Property 4.1 L2]

$$
\begin{array}{l}
L(wait\_com(c);\ term\_com(c.e)) \vee \\
L(term\_com(c.e))
\end{array}
$$

$=$ [Property 4.1 L6]

$$
\begin{array}{l}
(L(wait\_com(c));\ L(term\_com(c.e))) \vee \\
L(term\_com(c.e))
\end{array}
$$

$=$ [Property 4.1 L11 and L12]

$$
\left(\begin{array}{l} (wait' \wedge c.e \notin ref' \wedge tr' = tr); \\ (\neg wait' \wedge tr' - tr = \langle c.e \rangle) \end{array} \right) \vee
(\neg wait' \wedge tr' - tr = \langle c.e \rangle)
$$

$=$ [3.5.21]

$$
\left(\begin{array}{l} \exists\, wait_o, ref_o, tr_o \bullet \\ (wait_o \wedge c.e \notin ref_o \wedge tr_o = tr) \wedge \\ (\neg wait' \wedge tr' - tr_o = \langle c.e \rangle) \end{array} \right) \vee
(\neg wait' \wedge tr' - tr = \langle c.e \rangle)
$$

$=$ [Substitution]

$$
\left(\begin{array}{l} \exists\, wait_o, ref_o, tr_o \bullet \\ (wait_o \wedge c.e \notin ref_o) \wedge \\ (\neg wait' \wedge tr' - tr = \langle c.e \rangle) \end{array} \right) \vee
(\neg wait' \wedge tr' - tr = \langle c.e \rangle)
$$

$=$ [Propositional calculus]

$(\neg wait' \wedge tr' - tr = \langle c.e \rangle)$ $\Box$

E.2 FUNCTION L AND THE HEALTHINESS CONDITIONS

Property 4.2

L1. $L(R1_t(X)) = R1(L(X))$

Proof:

We first consider $L(R1_t(X)) \Leftarrow R1(L(X))$

$L(R1_t(X))$

$=$ [3.4.2]

$L(X \wedge Expands(tr_t, tr'_t))$

$=$ [4.2.1]

$$\exists\, tr_t, tr'_t \bullet \begin{pmatrix} (X \wedge Expands(tr_t, tr'_t)) \wedge \\ f(tr, tr', ref, ref', tr_t, tr'_t) \end{pmatrix}$$

$=$ [predicate calculus]

$$\exists\, tr_t, tr'_t \bullet \begin{array}{c} (X \wedge f(tr, tr', ref, ref', tr_t, tr'_t)) \wedge \\ (Expands(tr_t, tr'_t) \wedge f(tr, tr', ref, ref', tr_t, tr'_t)) \end{array}$$

$=$ [3.4.3 and 4.2.2]

$$\exists\, tr_t, tr'_t \bullet \begin{pmatrix} (X \wedge f(tr, tr', ref, ref', tr_t, tr'_t)) \wedge \\ (front(tr_t) \leq tr'_t) \wedge \\ (fst(last(tr_t)) \leq fst(tr'_t(\#\ tr_t))) \wedge \\ tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{pmatrix}$$

$\Leftarrow$ [let $tr_t = \langle (tr, ref) \rangle$ and $tr'_t = \langle (tr', ref') \rangle$]

$$(X \wedge f(tr, tr', ref, ref', tr_t, tr'_t))[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t] \wedge \begin{pmatrix} (front(\langle (tr, ref) \rangle) \leq \langle (tr', ref') \rangle) \wedge \\ (fst(last(\langle (tr, ref) \rangle)) \leq fst(\langle (tr', ref') \rangle(\#\ \langle (tr, ref) \rangle))) \wedge \\ tr = Flat(\langle (tr, ref) \rangle) \wedge tr' = Flat(\langle (tr', ref') \rangle) \wedge \\ ref = snd(last(\langle (tr, ref) \rangle)) \wedge ref' = snd(last(\langle (tr', ref') \rangle)) \end{pmatrix}$$

$=$ [properties of fst,snd,$last$,$front$ and $Flat$]

$$(X \wedge f(tr, tr', ref, ref', tr_t, tr'_t))[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t] \wedge \begin{pmatrix} (\langle\rangle \leq \langle (tr', ref') \rangle) \wedge \\ (tr \leq tr') \wedge \\ tr = tr \wedge tr' = tr' \wedge \\ ref = ref \wedge ref' ref' \end{pmatrix}$$

$=$ [predicate calculus]

$$(X \wedge f(tr, tr', ref, ref', tr_t, tr'_t))[\langle (tr, ref) \rangle, \langle (tr', ref') \rangle / tr_t, tr'_t] \wedge (tr \leq tr')$$

$$
\begin{array}{ll}
= & \text{[introduce } \exists\text{]} \\
(\exists\, tr_t, tr_t' \bullet X \wedge f(tr, tr', ref, ref', tr_t, tr_t')) \wedge tr \leq tr' & \\
= & \text{[4.2.1]} \\
L(X) \wedge tr \leq tr' & \\
= & \text{[3.4.1]} \\
R1(L(X)) & \Box
\end{array}
$$

Next we prove the case $L(R1_t(X)) \Rightarrow R1(L(X))$

$$
\begin{array}{ll}
L(R1_t(X)) = & \text{[3.4.2]} \\
L(X \wedge Expands(tr_t, tr_t')) & \\
\Rightarrow & \text{[property 4.1 L3]} \\
L(X) \wedge L(Expands(tr_t, tr_t')) & \\
= & \text{[property 4.1 L7]} \\
L(X) \wedge tr \leq tr' & \\
= & \text{[3.4.1]} \\
R1(L(X)) & \Box
\end{array}
$$

L2. $L(R2_t(X)) = R2(L(X))$

Proof:

First we show that $L(R2_t(X)) \Leftarrow R2(L(X))$

$$
\begin{array}{ll}
L(R2_t(X)) & \\
= & \text{[3.4.6]} \\
L(\exists\, ref \bullet X[\langle(\langle\rangle, ref)\rangle, dif(tr_t', tr_t)/tr_t, tr_t']) & \\
= & \text{[definition of /]} \\
L(\exists\, tr_t, tr_t', ref \bullet X \wedge tr_t = \langle(\langle\rangle, ref)\rangle \wedge tr_t' = dif(tr_t', tr_t)) & \\
= & [ref \text{ free in } L] \\
\exists\, ref \bullet L(\exists\, tr_t, tr_t' \bullet X \wedge tr_t = \langle(\langle\rangle, ref)\rangle \wedge tr_t' = dif(tr_t', tr_t)) & \\
= & \text{[4.2.1]} \\
\exists\, ref, tr_t, tr_t' \bullet \left(\begin{array}{l} X \wedge tr_t = \langle(\langle\rangle, ref)\rangle \wedge \\ tr_t' = dif(tr_t', tr_t) \wedge \\ f(tr, tr', ref, ref', tr_t, tr_t') \end{array} \right) & \\
= & \text{[predicate calculus]} \\
\exists\, ref, tr_t, tr_t' \bullet \left(\begin{array}{l} X \wedge f(tr, tr', ref, ref', tr_t, tr_t') \wedge \\ tr_t = \langle(\langle\rangle, ref)\rangle \wedge \\ tr_t' = dif(tr_t', tr_t) \wedge \\ f(tr, tr', ref, ref', tr_t, tr_t') \end{array} \right) & \\
= & \text{[4.2.2,3.4.7]}
\end{array}
$$

$$
\exists\, ref, tr_t, tr'_t \bullet \left(\begin{array}{l}
X \wedge f(tr, tr', ref, ref', tr_t, tr'_t) \wedge \\
tr_t = \langle(\langle\rangle, ref)\rangle \wedge \\
tr'_t = \left(\begin{array}{l} \langle(fst(head(tr'_t - front(tr_t))) \\ -fst(last(tr_t)), \\ snd(head(tr'_t - front(tr_t))))\rangle^\frown \\ tail(tr'_t - front(tr_t)) \end{array}\right) \wedge \\
tr = Flat(tr_t) \wedge \\
tr' = Flat(tr'_t) \wedge \\
ref = snd(last(tr_t)) \wedge \\
ref' = snd(last(tr'_t))
\end{array}\right)
$$

$\Leftarrow$ [let $tr_t = \langle(tr, ref)\rangle$ and $tr'_t = \langle(tr', ref')\rangle$]

$$
\exists\, ref, tr_t, tr'_t \bullet \left(\begin{array}{l}
\left(\begin{array}{l} X \wedge \\ f(tr, tr', ref, ref', tr_t, tr'_t) \end{array}\right) \left[\begin{array}{c} \langle(tr, ref)\rangle, \\ \langle(tr', ref')\rangle / \\ tr_t, tr'_t \end{array}\right] \wedge \\
\langle(tr, ref)\rangle = \langle(\langle\rangle, ref)\rangle \wedge \\
\langle(tr', ref')\rangle = \left(\begin{array}{l} \langle(fst(head(\langle(tr', ref')\rangle - \\ front(\langle(tr, ref)\rangle))) - \\ fst(last(\langle(tr, ref)\rangle)), \\ snd(head(\langle(tr', ref')\rangle - \\ front(\langle(tr, ref)\rangle)))))\rangle^\frown \\ tail(\langle(tr', ref')\rangle - front(\langle(tr, ref)\rangle)) \end{array}\right) \wedge \\
tr = Flat(\langle(tr, ref)\rangle) \wedge \\
tr' = Flat(\langle(tr', ref')\rangle) \wedge \\
ref = snd(last(\langle(tr, ref)\rangle)) \wedge \\
ref' = snd(last(\langle(tr', ref')\rangle))
\end{array}\right)
$$

$=$ [properties of *last*,*front*,*head*,*Flat*,*fst* and *snd*]

$$
\exists\, ref \bullet \left(\begin{array}{l}
\left(\begin{array}{l} X \wedge \\ f(tr, tr', ref, ref', tr_t, tr'_t) \end{array}\right) \left[\begin{array}{c} \langle(tr, ref)\rangle, \\ \langle(tr', ref')\rangle / \\ tr_t, tr'_t \end{array}\right] \wedge \\
\langle(tr, ref)\rangle = \langle(\langle\rangle, ref)\rangle \wedge \\
\langle(tr', ref')\rangle = \left(\begin{array}{l} \langle(fst(head(\langle(tr', ref')\rangle - \langle\rangle)) - tr, \\ snd(head(\langle(tr', ref')\rangle - \langle\rangle)))\rangle^\frown \\ tail(\langle(tr', ref')\rangle - \langle\rangle) \end{array}\right) \wedge \\
tr = tr \wedge tr' = tr' \wedge \\
ref = ref \wedge ref' = ref'
\end{array}\right)
$$

$=$ [predicate calculus and sequence subtraction]

$$
\exists\, ref \bullet \left(\begin{array}{l}
\left(\begin{array}{l} X \wedge \\ f(tr, tr', ref, ref', tr_t, tr'_t) \end{array}\right) \left[\begin{array}{c} \langle(tr, ref)\rangle, \\ \langle(tr', ref')\rangle / \\ tr_t, tr'_t \end{array}\right] \wedge \\
\langle(tr, ref)\rangle = \langle(\langle\rangle, ref)\rangle \wedge \\
\langle(tr', ref')\rangle = \langle(tr' - tr, ref')\rangle \frown tail(\langle(tr', ref')\rangle)
\end{array}\right)
$$

$=$ [predicate calculus and *tail*]

$$
\begin{array}{ll}
\exists\, ref \bullet \left(\begin{array}{l} \left(\begin{array}{l} X \wedge \\ f(tr, tr', ref, ref', tr_t, tr_t') \end{array} \right) \left[\begin{array}{c} \langle (tr, ref) \rangle, \\ \langle (tr', ref') \rangle / \\ tr_t, tr_t' \end{array} \right] \wedge \\ \langle (tr, ref) \rangle = \langle (\langle\rangle, ref) \rangle \wedge \\ \langle (tr', ref') \rangle = \langle (tr' - tr, ref') \rangle \frown \langle\rangle \end{array} \right) & \\
= & \text{[predicate calculus and sequence properties]} \\
\exists\, ref \bullet \left(\begin{array}{l} \left(\begin{array}{l} X \wedge \\ f(tr, tr', ref, ref', tr_t, tr_t') \end{array} \right) \left[\begin{array}{c} \langle (tr, ref) \rangle, \\ \langle (tr', ref') \rangle / \\ tr_t, tr_t' \end{array} \right] \wedge \\ tr = \langle\rangle \wedge \\ tr' = tr' - tr \end{array} \right) & \\
= & \text{[Introduce } \exists\text{]} \\
\exists\, ref \bullet \left(\begin{array}{l} (\exists\, tr_t, tr_t' \bullet (X \wedge f(tr, tr', ref, ref', tr_t, tr_t'))) \wedge \\ tr = \langle\rangle \wedge \\ tr' = tr' - tr \end{array} \right) & \\
= & [4.2.1] \\
\exists\, ref \bullet \left(\begin{array}{l} L(X) \wedge \\ tr = \langle\rangle \wedge \\ tr' = tr' - tr \end{array} \right) & \\
= & \text{[property of substitution]} \\
\exists\, ref \bullet L(X)[\langle\rangle, (tr' - tr)/tr, tr'] & \\
= & [3.4.5] \\
R2(L(X)) & \square
\end{array}
$$

Next we show that $L(R2_t(X)) \Rightarrow R2(L(X))$

$$
\begin{array}{ll}
L(R2_t(X)) & \\
= & [4.2.1] \\
\exists\, tr_t, tr_t' \bullet \left(\begin{array}{l} R2_t(X) \wedge \\ tr = Flat(tr_t) \wedge \\ tr' = Flat(tr_t') \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr_t')) \end{array} \right) & \\
= & [3.4.6] \\
\exists\, tr_t, tr_t' \bullet \left(\begin{array}{l} X[\langle (\langle\rangle, ref) \rangle, dif(tr_t', tr_t)/tr_t, tr_t'] \wedge \\ tr = Flat(tr_t) \wedge \\ tr' = Flat(tr_t') \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr_t')) \end{array} \right) & \\
= & \text{[Predicate calculus]}
\end{array}
$$

$$
\begin{array}{ll}
 & \exists\, tr_t, tr'_t \bullet \left(\begin{array}{l} \exists\, tr_t, tr'_t \bullet X \wedge tr_t = \langle (\langle\rangle, ref) \rangle \wedge \\ tr'_t = dif(tr'_t, tr_t) \wedge \\ tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \right) \\
= & \qquad \text{[Propositional and substitution]} \\
 & \exists\, tr_t, tr'_t \bullet \begin{array}{l} X \wedge tr_t = \langle (\langle\rangle, ref) \rangle \wedge \\ tr'_t = dif(tr'_t, tr_t) \wedge \\ tr = Flat(\langle (\langle\rangle, ref) \rangle) \wedge \\ tr' = Flat(dif(tr'_t, tr_t)) \wedge \\ tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{array} \\
= & \qquad \text{[propositional calculus]} \\
 & \exists\, tr_t, tr'_t \bullet \begin{array}{l} \exists\, tr_t, tr'_t \bullet X \wedge \\ tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \wedge \\ tr_t = \langle (\langle\rangle, ref) \rangle \wedge \\ tr'_t = dif(tr'_t, tr_t) \wedge \\ tr = Flat(\langle (\langle\rangle, ref) \rangle) \wedge \\ tr' = Flat(dif(tr'_t, tr_t)) \end{array} \\
= & \qquad \text{[4.2.1]} \\
 & \exists\, tr_t, tr'_t \bullet \begin{array}{l} L(X) \wedge \\ tr_t = \langle (\langle\rangle, ref) \rangle \wedge \\ tr'_t = dif(tr'_t, tr_t) \wedge \\ tr = Flat(\langle (\langle\rangle, ref) \rangle) \wedge \\ tr' = Flat(dif(tr'_t, tr_t)) \end{array} \\
= & \qquad \text{[from definition of Flat and substitution]} \\
 & \exists\, tr_t, tr'_t \bullet \begin{array}{l} L(X) \wedge \\ tr_t = \langle (\langle\rangle, ref) \rangle \wedge \\ tr'_t = dif(tr'_t, \langle (\langle\rangle, ref) \rangle) \wedge \\ tr = \langle\rangle \wedge \\ tr' = Flat(dif(tr'_t, \langle (\langle\rangle, ref) \rangle)) \end{array} \\
= & \qquad \text{[Property 3.3 L2]}
\end{array}
$$

$$
\begin{array}{ll}
 & \exists\, tr'_t \bullet \begin{array}{l} L(X) \wedge \\ tr_t = \langle(\langle\rangle, ref)\rangle \wedge \\ tr = \langle\rangle \wedge \\ tr' = Flat(tr'_t) \end{array} \\
= & \quad [\text{Rename } tr'_t \text{ to } tr' \text{ such that } Flat(tr'_t) = tr' - tr \text{ and introduce } tr] \\
 & \exists\, tr, tr' \bullet \begin{array}{l} L(X) \wedge \\ tr_t = \langle(\langle\rangle, ref)\rangle \wedge \\ tr = \langle\rangle \wedge \\ tr' = (tr' - tr) \end{array} \\
= & \quad [3.4.5] \\
 & \begin{array}{l} R2(L(X)) \wedge \\ tr_t = \langle(\langle\rangle, ref)\rangle \end{array} \\
\Rightarrow & \quad [\text{propositional calculus}] \\
 & R2(L(X)) \qquad \Box
\end{array}
$$

L3. $L(R3_t(X)) = R3(L(X))$
Proof:

$$
\begin{array}{ll}
 & L(R3_t(X)) \\
= & \quad [3.4.10] \\
 & L(\Pi_t \lhd wait \rhd X) \\
= & \quad [\text{property 4.1 L5}] \\
 & L(\Pi_t) \lhd wait \rhd L(X) \\
= & \quad [\text{property 4.1 L9}] \\
 & \Pi \lhd wait \rhd L(X) \\
= & \quad [3.4.8] \\
 & R3(L(X)) \qquad \Box
\end{array}
$$

L4. $L(CSP1_t(X)) = CSP1(L(X))$
Proof:

$$
\begin{array}{ll}
 & L(CSP1_t(X)) \\
= & \quad [3.4.14] \\
 & L((\neg ok \wedge Expands(tr_t, tr'_t)) \vee X) \\
= & \quad [\text{property 4.1 L2}] \\
 & L((\neg ok \wedge Expands(tr_t, tr'_t)) \vee L(X) \\
= & \quad [\text{property 4.1 L4}] \\
 & (\neg ok \wedge L(Expands(tr_t, tr'_t))) \vee L(X) \\
= & \quad [\text{property 4.1 L7}]
\end{array}
$$

$$
\begin{array}{ll}
(\neg ok \wedge tr' \leq tr) \vee L(X) & \\
= & [3.4.13] \\
CSP1(L(X)) & \Box
\end{array}
$$

L5. $L(CSP2_t(X)) = CSP2(L(X))$
Proof:

$$
\begin{array}{ll}
L(CSP2_t(X)) & \\
= & [3.4.15] \\
L\left(X;\ \left(\begin{array}{c}(ok \Rightarrow ok') \wedge \\ (tr'_t = tr_t) \wedge \\ (wait' = wait) \wedge \\ (state' = state)\end{array}\right)\right) & \\
= & [\text{property 4.1 L6}] \\
L(X);\ L\left(\begin{array}{c}(ok \Rightarrow ok') \wedge \\ (tr'_t = tr_t) \wedge \\ (wait' = wait) \wedge \\ (state' = state)\end{array}\right) & \\
= & [\text{property 4.1 L4}] \\
L(X);\ \left(\begin{array}{c}(ok \Rightarrow ok') \wedge \\ (state' = state) \wedge \\ (wait' = wait) \wedge \\ L(tr'_t = tr_t)\end{array}\right) & \\
= & [\text{property 4.1 L8}] \\
L(X);\ \left(\begin{array}{c}(ok \Rightarrow ok') \wedge \\ (state' = state) \wedge \\ (wait' = wait) \wedge \\ (tr = tr') \wedge (ref = ref')\end{array}\right) & \\
= & [3.4.15] \\
CSP2(L(X)) & \Box
\end{array}
$$

L6. $L(CSP3_t(X)) = CSP3(L(X))$
Proof:

$$
\begin{array}{ll}
L(CSP3_t(X)) & \\
= & [3.4.16] \\
L(Skip;\ X) & \\
= & [\text{property 4.1 L6}] \\
L(Skip);\ L(X) & \\
= & [\text{property 4.3 L2}]
\end{array}
$$

$$
\begin{array}{ll}
& \mathit{Skip};\ L(X) \\
= & \qquad\qquad [\text{definition of } \mathit{CSP}3 \text{ in UTP}] \\
& \mathit{CSP}3(L(X))\Box
\end{array}
$$

L7. $L(\mathit{CSP}4_t(X)) = \mathit{CSP}4(L(X))$
Proof:

$$
\begin{array}{ll}
& L(\mathit{CSP}4_t(X)) \\
= & \qquad\qquad [3.4.17] \\
& L(X;\ \mathit{Skip}) \\
= & \qquad\qquad [\text{property 4.1 L6}] \\
& L(X);\ L(\mathit{Skip}) \\
= & \qquad\qquad [\text{property 4.3 L2}] \\
& L(X);\ \mathit{Skip} \\
= & \qquad\qquad [\text{definition of } \mathit{CSP}4 \text{ in UTP}] \\
& \mathit{CSP}4(L(X))\Box
\end{array}
$$

L8. $L(\mathit{CSP}5_t(X)) = \mathit{CSP}5(L(X))$
Proof:

$$
\begin{array}{ll}
& L(\mathit{CSP}5_t(X)) \\
= & \qquad\qquad [3.4.18] \\
& L(X \,[\![\, \Delta X \mid \{\!|\ \phi\ |\!\} \mid \phi \,]\!]\, \mathit{Skip}) \\
= & \qquad\qquad [\text{property 4.3 L12}] \\
& L(X) \,[\![\, \Delta X \mid \{\!|\ \phi\ |\!\} \mid \phi \,]\!]\, L(\mathit{Skip}) \\
= & \qquad\qquad [\text{property 4.3 L2}] \\
& L(X) \,[\![\, \Delta X \mid \{\!|\ \phi\ |\!\} \mid \phi \,]\!]\, \mathit{Skip} \\
= & \qquad\qquad [\text{Definition of } \mathit{CSP}5 \text{ in UTP}] \\
& \mathit{CSP}5(L(X)) \qquad\qquad \Box
\end{array}
$$

E.3 APPLYING L TO CIRCUS TIME ACTION CONSTRUCTS

Property 4.3

L1. $L([\![x := e]\!]_{time}) = [\![x := e]\!]$
Proof:

$$
\begin{array}{ll}
& L([\![x := e]\!]_{time}) \\
= & \qquad\qquad [3.5.7] \\
& L(\mathit{CSP}1_t(R_t \left(\begin{array}{l} ok = ok' \wedge wait = wait' \wedge \\ tr'_t = tr_t \wedge state' = state \oplus \{x \mapsto e\} \end{array} \right)))
\end{array}
$$

$$
\begin{array}{ll}
= & \text{[property 4.2 L1,L2,L3]} \\
CSP1(R(L\left(\begin{array}{l} ok = ok' \land wait = wait' \land \\ tr'_t = tr_t \land state' = state \oplus \{x \mapsto e\} \end{array}\right))) & \\
= & \text{[property 4.1 L4]} \\
CSP1(R\left(\begin{array}{l} ok = ok' \land wait = wait' \land \\ L(tr'_t = tr_t) \land state' = state \oplus \{x \mapsto e\} \end{array}\right)) & \\
= & \text{[property 4.1 L8]} \\
CSP1(R\left(\begin{array}{l} ok = ok' \land wait = wait' \land \\ tr' = tr \land ref' = ref \land state' = state \oplus \{x \mapsto e\} \end{array}\right)) & \\
= & \text{[definition of assignment in } \textit{Circus}\text{]} \\
[\![x := e]\!] & \Box
\end{array}
$$

L2. $L([\![Skip]\!]_{time}) = [\![Skip]\!]$
Proof:

$$
\begin{array}{ll}
L([\![Skip]\!]_{time}) & \\
= & \text{[3.5.2]} \\
L(R_t(\exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t)) & \\
= & \text{[properties 4.2 L1,L2,L3]} \\
R(L(R_t(\exists\, ref \bullet ref = snd(last(tr_t)) \land \Pi_t))) & \\
= & \text{[property 4.1 L9]} \\
R(\exists\, ref \bullet \Pi) & \\
= & \text{[3.5.1]} \\
[\![Skip]\!] & \Box
\end{array}
$$

L3. $L([\![Stop]\!]_{time}) = [\![Stop]\!]$
Proof:

$$
\begin{array}{ll}
L([\![Stop]\!]_{time}) & \\
= & \text{[3.5.5]} \\
L(CSP1_t(R3_t(ok' \land wait' \land trace' = \langle\rangle))) & \\
= & \text{[property 4.2 L4]} \\
CSP1(L(R3_t(ok' \land wait' \land trace' = \langle\rangle))) & \\
= & \text{[property 4.2 L3]} \\
CSP1(R3(L(ok' \land wait' \land trace' =<>))) & \\
= & \text{[property 4.1 L4]} \\
CSP1(R3(ok' \land wait' \land L(trace' =<>))) & \\
= & \text{[property 4.1 L10]} \\
CSP1(R3(ok' \land wait' \land tr = tr')) &
\end{array}
$$

$$
\begin{array}{ll}
= & [3.5.3] \\
[\![Stop]\!] & \Box
\end{array}
$$

L4. $L([\![Chaos]\!]_{time}) = [\![Chaos]\!]$
Proof:

$$
\begin{array}{ll}
L([\![Chaos]\!]_{time}) & \\
= & [3.5.6] \\
L(R_t(true)) & \\
= & [\text{properties 4.2 L1,L2 and L3}] \\
R(L(true)) & \\
= & [4.2.3] \\
R(true) & \\
= & [\text{definition of } Chaos \text{ in UTP}] \\
[\![Chaos]\!] & \Box
\end{array}
$$

L5. $L([\![Wait\ \ \mathsf{t}]\!]_{time}) = [\![Stop]\!] \sqcap [\![Skip]\!]$
Proof:
First we show that $L([\![Wait\ \ \mathsf{t}]\!]_{time}) \Leftarrow [\![Stop]\!] \sqcap [\![Skip]\!]$

$$
\begin{array}{ll}
L([\![Wait\ \ \mathsf{t}]\!]_{time}) & \\
= & [3.5.8] \\
L(CSP1_t(R_t(ok' \wedge delay(\mathsf{t}) \wedge (trace' = \langle\rangle)))) & \\
= & [\text{property 4.2 L4}] \\
CSP1(L(R_t(ok' \wedge delay(\mathsf{t}) \wedge (trace' = \langle\rangle)))) & \\
\Rightarrow & [\text{property 4.2 L1,L2 and L3}] \\
CSP1(R(L(ok' \wedge delay(\mathsf{t}) \wedge (trace' = \langle\rangle)))) & \\
= & [\text{property 4.1 L4}] \\
CSP1(R(ok' \wedge L(delay(\mathsf{t}) \wedge (trace' = \langle\rangle)))) & \\
= & [3.5.9 \text{ and proposition logic}] \\
CSP1\left(R\left(ok' \wedge L\left(\begin{array}{c} \left(\begin{array}{c} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathsf{t} \wedge \\ (trace' = \langle\rangle) \end{array}\right) \vee \\ \left(\begin{array}{c} \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = \mathsf{t} \wedge \\ state' = state \wedge \\ (trace' = \langle\rangle) \end{array}\right) \end{array}\right)\right)\right) & \\
= & [4.2.1]
\end{array}
$$

$$
CSP1\left(R\left(ok' \wedge \left(\exists\, tr_t, tr'_t \bullet \begin{pmatrix} \begin{pmatrix} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \\ (trace' = \langle\rangle) \end{pmatrix} \vee \\ \begin{pmatrix} \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \\ state' = state \wedge \\ (trace' = \langle\rangle) \end{pmatrix} \end{pmatrix} \wedge \begin{pmatrix} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{pmatrix}\right)\right)\right)
$$

$=$ [3.3.1]

$$
CSP1\left(R\left(ok' \wedge \left(\exists\, tr_t, tr'_t \bullet \begin{pmatrix} \begin{pmatrix} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \\ (Flat(tr'_t) - Flat(tr_t) = \langle\rangle) \end{pmatrix} \vee \\ \begin{pmatrix} \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \\ state' = state \wedge \\ (Flat(tr'_t) - Flat(tr_t) = \langle\rangle) \end{pmatrix} \end{pmatrix} \wedge \begin{pmatrix} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{pmatrix}\right)\right)\right)
$$

$=$ [sequence properties]

$$
CSP1\left(R\left(ok' \wedge \left(\exists\, tr_t, tr'_t \bullet \begin{pmatrix} \begin{pmatrix} wait' \wedge \\ (\#tr'_t - \#tr_t) < \mathtt{t} \wedge \\ (Flat(tr'_t) = Flat(tr_t)) \end{pmatrix} \vee \\ \begin{pmatrix} \neg wait' \wedge \\ (\#tr'_t - \#tr_t) = \mathtt{t} \wedge \\ state' = state \wedge \\ (Flat(tr'_t) = Flat(tr_t)) \end{pmatrix} \end{pmatrix} \wedge \begin{pmatrix} tr = Flat(tr_t) \wedge \\ tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge \\ ref' = snd(last(tr'_t)) \end{pmatrix}\right)\right)\right)
$$

$\Leftarrow$ [Let $tr_t = \langle (tr, ref) \rangle$]
[and $tr'_t = \langle (tr', ref') \rangle$]

$$
CSP1\left(R\left(ok' \wedge \left(\begin{array}{c}\left(\begin{array}{c}\left(\begin{array}{c} wait' \wedge \\ (\#\langle(tr', ref')\rangle - \#\langle(tr, ref)\rangle) \leq \mathtt{t} \wedge \\ (Flat(\langle(tr', ref')\rangle) = Flat(\langle(tr, ref)\rangle)) \end{array}\right) \vee \\ \left(\begin{array}{c} \neg wait' \wedge state = state' \wedge \\ (\#\langle(tr', ref')\rangle - \#\langle(tr, ref)\rangle) = \mathtt{t} \wedge \\ (Flat(\langle(tr', ref')\rangle) = Flat(\langle(tr, ref)\rangle)) \end{array}\right) \end{array}\right) \wedge \\ \left(\begin{array}{c} tr = Flat(\langle(tr, ref)\rangle) \wedge \\ tr' = Flat(\langle(tr', ref')\rangle) \wedge \\ ref = snd(last(\langle(tr, ref)\rangle)) \wedge \\ ref' = snd(last(\langle(tr', ref')\rangle)) \end{array}\right) \end{array}\right)\right)\right)
$$

$=$ [properties of *Flat*, *snd* and *last*]

$$
CSP1\left(R\left(ok' \wedge \left(\begin{array}{c}\left(\begin{array}{c}\left(\begin{array}{c} wait' \wedge \\ 0 \leq \mathtt{t} \wedge \\ tr' = tr \end{array}\right) \vee \\ \left(\begin{array}{c} \neg wait' \wedge state = state' \wedge \\ 0 = \mathtt{t} \wedge \\ state' = state \wedge \\ tr' = tr \end{array}\right) \end{array}\right) \wedge \\ \left(\begin{array}{c} tr = tr \wedge \\ tr' = tr' \wedge \\ ref = ref \wedge \\ ref' = ref' \end{array}\right) \end{array}\right)\right)\right)
$$

$=$ [Proposition logic]

$$
CSP1\left(R\left(ok' \wedge \left(\begin{array}{c}\left(\begin{array}{c} wait' \wedge \\ 0 \leq \mathtt{t} \wedge \\ tr' = tr \end{array}\right) \vee \\ \left(\begin{array}{c} \neg wait' \wedge state = state' \wedge \\ 0 = \mathtt{t} \wedge \\ state' = state \wedge \\ tr' = tr \end{array}\right) \end{array}\right)\right)\right)
$$

$=$ [Healthiness conditions distribute over $\vee$ and predicate calculus]

$$
CSP1\left(R\left(ok' \wedge \left(\begin{array}{c} wait' \wedge \\ 0 \leq \mathtt{t} \wedge \\ tr' = tr \end{array}\right)\right)\right) \vee
$$

$$
CSP1\left(R\left(ok' \wedge \left(\begin{array}{c} \neg wait' \wedge state = state' \wedge \\ 0 = \mathtt{t} \wedge \\ state' = state \wedge \\ tr' = tr \end{array}\right)\right)\right)
$$

$=$ [3.5.1,3.5.4]

$[\![Stop]\!] \sqcap [\![Skip]\!]$ $\square$

Next we show that $L(\llbracket Wait\ \ \mathsf{t} \rrbracket_{time}) \Rightarrow \llbracket Stop \rrbracket \sqcap \llbracket Skip \rrbracket$

$$
\begin{array}{ll}
& L(\llbracket Wait\ \ \mathsf{t} \rrbracket_{time}) \\
= & [3.5.8] \\
& L(CSP1_t(R_t(ok' \land delay(\mathsf{t}) \land (trace' = \langle\rangle)))) \\
= & [\text{property 4.2 L4}] \\
& CSP1(L(R_t(ok' \land delay(\mathsf{t}) \land (trace' = \langle\rangle)))) \\
\Rightarrow & [\text{property 4.2 L1,L2 and L3}] \\
& CSP1(R(L(ok' \land delay(\mathsf{t}) \land (trace' = \langle\rangle)))) \\
= & [\text{property 4.1 L4}] \\
& CSP1(R(ok' \land state = state' \land L(delay(\mathsf{t}) \land (trace' = \langle\rangle)))) \\
= & [3.5.9 \text{ and proposition logic}] \\
& CSP1\left(R\left(ok' \land L\left(\begin{array}{c} \left(\begin{array}{c} wait' \land \\ (\#tr'_t - \#tr_t) < \mathsf{t} \land \end{array}\right) \lor \\ (trace' = \langle\rangle) \\ \left(\begin{array}{c} \neg wait' \land state = state' \land \\ (\#tr'_t - \#tr_t) = \mathsf{t} \land \end{array}\right) \\ (trace' = \langle\rangle) \end{array}\right)\right)\right) \\
= & [\text{property 4.1 L2}] \\
& CSP1\left(R\left(ok' \land \begin{array}{c} L\left(\begin{array}{c} wait' \land \\ (\#tr'_t - \#tr_t) < \mathsf{t} \land \end{array}\right) \lor \\ (trace' = \langle\rangle) \\ L\left(\begin{array}{c} \neg wait' \land state = state' \land \\ (\#tr'_t - \#tr_t) = \mathsf{t} \land \end{array}\right) \\ (trace' = \langle\rangle) \end{array}\right)\right) \\
= & [\text{property 4.1 L4}] \\
& CSP1\left(R\left(ok' \land \begin{array}{c} wait' \land L\left(\begin{array}{c} (\#tr'_t - \#tr_t) < \mathsf{t} \land \\ (trace' = \langle\rangle) \end{array}\right) \lor \\ \neg wait' \land state = state' \land L\left(\begin{array}{c} (\#tr'_t - \#tr_t) = \mathsf{t} \land \\ (trace' = \langle\rangle) \end{array}\right) \end{array}\right)\right) \\
= & [\text{property 4.1 L10}] \\
& CSP1\left(R\left(ok' \land \begin{array}{c} \left(\begin{array}{c} (wait' \land tr' = tr) \land \\ L\left(\begin{array}{c} (\#tr'_t - \#tr_t) < \mathsf{t} \land \\ (trace' = \langle\rangle) \end{array}\right) \end{array}\right) \lor \\ \left(\begin{array}{c} (\neg wait' \land state = state' \land tr' = tr) \land \\ L\left(\begin{array}{c} (\#tr'_t - \#tr_t) = \mathsf{t} \land \\ (trace' = \langle\rangle) \end{array}\right) \end{array}\right) \end{array}\right)\right) \\
\Rightarrow & [\text{Predicate calculus}] \\
& CSP1\left(R\left(ok' \land \left(\begin{array}{c} wait' \land tr' = tr \lor \\ \neg wait' \land state = state' \land tr' = tr \end{array}\right)\right)\right)
\end{array}
$$

$$
\begin{array}{ll}
= & \text{[Predicate calculus]} \\
\quad CSP1(R(ok' \land wait' \land tr' = tr)) \lor & \\
\quad CSP1(R(ok' \land state = state' \land \neg wait' \land tr' = tr)) & \\
= & \text{[3.5.1,3.5.4]} \\
[\![Stop]\!] \sqcap [\![Skip]\!] & \Box
\end{array}
$$

L6. $L([\![comm]\!]_{time}) = [\![comm]\!]$
Proof:

$$
\begin{array}{ll}
L([\![c!e \rightarrow Skip]\!]_{time}) & \\
= & \text{[3.5.18]} \\
L([\![c.e \rightarrow Skip]\!]_{time}) & \\
= & \text{[3.5.17]} \\
L(CSP1_t(ok' \land R_t(wait\_com(c) \lor terminating\_com(c.e)))) & \\
= & \text{[property 4.2 L4]} \\
CSP1(L(ok' \land R_t(wait\_com(c) \lor terminating\_com(c.e)))) & \\
= & \text{[property 4.1 L4]} \\
CSP1(ok' \land L(R_t(wait\_com(c) \lor terminating\_com(c.e)))) & \\
= & \text{[properties 4.2 L1,L2 and L3]} \\
CSP1(ok' \land R(L(wait\_com(c) \lor terminating\_com(c.e)))) & \\
= & \text{[properties 4.1 L2]} \\
CSP1(ok' \land R(L(wait\_com(c)) \lor L(terminating\_com(c.e)))) & \\
= & \text{[properties 4.1 L11]} \\
CSP1(ok' \land R((wait' \land c \notin ref' \land tr' = tr) \lor L(terminating\_com(c.e)))) & \\
= & \text{[properties 4.1 L13]} \\
CSP1(ok' \land R((wait' \land c \notin ref' \land tr' = tr) \lor (\neg wait' \land tr' - tr = \langle c.e \rangle))) & \\
= & \text{[3.5.10]} \\
[\![c!e \rightarrow Skip]\!] & \Box
\end{array}
$$

L7. $L([\![A \lhd b \rhd B]\!]_{time}) = [\![L(A) \lhd b \rhd L(B)]\!]$
Proof:
Based on property 4.1 L4 $\Box$

L8. $L([\![p\& \ A]\!]_{time}) = [\![p\& \ L(A)]\!]$
Proof:

$$
L([\![p\& \ A]\!]_{time})
$$

$=$ [3.5.23]
$L([\![A \lhd p \rhd Stop]\!]_{time})$
$=$ [Property 4.3 L7]
$[\![L(A) \lhd p \rhd L(Stop)]\!]$
$=$ [Property 4.3 L3]
$[\![L(A) \lhd p \rhd Stop]\!]$
$=$ [3.5.23]
$[\![p \& L(A)]\!]$ $\Box$

L9. $L([\![A \sqcap B]\!]_{time}) = [\![L(A) \sqcap L(B)]\!]$
Proof:

$L(A \sqcap B)$
$=$ [3.9]
$L(A \vee B)$
$=$ [Property 4.1 L2]
$L(A) \vee L(B)$
$=$ [refintchoice]
$L(A) \sqcap L(B) \Box$

L10. $L([\![A \Box B]\!]_{time}) = [\![L(A) \Box L(B)]\!]$
Proof:
We start by showing the $L([\![A \Box B]\!]_{time}) \Rightarrow [\![L(A) \Box L(B)]\!]$

$L([\![A \Box B]\!]_{time})$
$=$ [3.5.30]
$L(CSP2_t(ExtChoice1(A, B) \vee ExtChoice2(A, B)))$
$=$ [Property 4.2 L4]
$CSP2(L(ExtChoice1(A, B) \vee ExtChoice2(A, B)))$
$=$ [Property 4.1 L2]
$CSP2(L(ExtChoice1(A, B)) \vee L(ExtChoice2(A, B)))$
$=$ [3.5.27 and 3.5.28]
$CSP2(L(A \wedge B \wedge Stop) \vee L(DifDetected(A, B) \wedge (A \vee B)))$
$=$ [3.5.29]

$$
CSP2\left(\begin{array}{l}
L(A \wedge B \wedge Stop) \vee \\
L\left(\begin{array}{c}
\left(\begin{array}{c}
\neg ok' \vee \\
\left(\begin{array}{c}
\left(\begin{array}{c}
(ok \wedge \neg wait) \wedge \\
\left(\begin{array}{c}
\left(\begin{array}{c} A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right) \vee \\
Skip
\end{array}\right)
\end{array}\right) ; \\
(ok' \wedge \neg wait' \wedge tr'_t = tr_t) \vee \\
\left(\begin{array}{c} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right)
\end{array}\right)
\end{array}\right) \wedge \\
(A \vee B)
\end{array}\right)
\end{array}\right)
$$

$=$ [Predicate calculus]

$$
CSP2\left(\begin{array}{l}
L(A \wedge B \wedge Stop) \vee \\
L\left(\begin{array}{c}
(\neg ok' \wedge (A \vee B)) \vee \\
\left(\begin{array}{c}
\left(\begin{array}{c}
\left(\begin{array}{c}
(ok \wedge \neg wait) \wedge \\
A \wedge B \wedge ok' \wedge \\
wait' \wedge trace' = \langle\rangle
\end{array}\right) \vee \\
(ok \wedge \neg wait \wedge Skip)
\end{array}\right) ; \\
(ok' \wedge \neg wait' \wedge tr'_t = tr_t) \vee \\
\left(\begin{array}{c} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right)
\end{array}\right) \wedge \\
(A \vee B)
\end{array}\right)
\end{array}\right)
$$

$=$ [Predicate calculus and Property 3.9 L5,L6]

$$
CSP2\left(\begin{array}{l}
L(A \wedge B \wedge Stop) \vee \\
L\left(\begin{array}{c}
(\neg ok' \wedge A) \vee \\
(\neg ok' \wedge B) \vee \\
\left(\begin{array}{c}
\left(\begin{array}{c}
\left(\begin{array}{c}
ok \wedge \neg wait \wedge \\
A \wedge B \wedge ok' \wedge \\
wait' \wedge trace' = \langle\rangle
\end{array}\right) ; \\
(ok' \wedge \neg wait' \wedge tr'_t = tr_t) \vee \\
\left(\begin{array}{c} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right)
\end{array}\right) \vee \\
(ok \wedge \neg wait \wedge Skip); \\
\left(\begin{array}{c}
(ok' \wedge \neg wait' \wedge tr'_t = tr_t) \vee \\
\left(\begin{array}{c} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right)
\end{array}\right)
\end{array}\right) \wedge \\
(A \vee B)
\end{array}\right)
\end{array}\right)
$$

$=$ [Predicate calculus and Property 3.9 L5,L6]

$$
CSP2\left(\begin{array}{l}
L(A \wedge B \wedge Stop) \vee \\
L\left(\begin{array}{c}
(\neg ok' \wedge A) \vee \\
(\neg ok' \wedge B) \vee \\
\left(\begin{array}{c}
\left(\begin{array}{c}
\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \\
(ok' \wedge \neg wait' \wedge tr_t' = tr_t)
\end{array}\right) \vee \\
\left(\begin{array}{c}
\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \\
\left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr_t', tr_t))) \neq \langle\rangle \end{array}\right)
\end{array}\right) \vee \\
\left(\begin{array}{c}
(ok \wedge \neg wait \wedge Skip); \\
(ok' \wedge \neg wait' \wedge tr_t' = tr_t)
\end{array}\right) \vee \\
\left(\begin{array}{c}
(ok \wedge \neg wait \wedge Skip); \\
\left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr_t', tr_t))) \neq \langle\rangle \end{array}\right)
\end{array}\right)
\end{array}\right) \wedge \\
(A \vee B)
\end{array}\right)\right)
$$

$=$ [Predicate calculus]

$$
CSP2\left(\begin{array}{l}
L(A \wedge B \wedge Stop) \vee \\
L\left(\begin{array}{c}
(\neg ok' \wedge A) \vee \\
(\neg ok' \wedge B) \vee \\
\left(\begin{array}{c}
\left(\begin{array}{c}
\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \\
(ok' \wedge \neg wait' \wedge tr_t' = tr_t)
\end{array}\right) \wedge (A \vee B) \vee \\
\left(\begin{array}{c}
\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \\
\left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr_t', tr_t))) \neq \langle\rangle \end{array}\right)
\end{array}\right) \wedge (A \vee B) \vee \\
\left(\begin{array}{c}
(ok \wedge \neg wait \wedge Skip); \\
(ok' \wedge \neg wait' \wedge tr_t' = tr_t)
\end{array}\right) \wedge (A \vee B) \vee \\
\left(\begin{array}{c}
(ok \wedge \neg wait \wedge Skip); \\
\left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr_t', tr_t))) \neq \langle\rangle \end{array}\right)
\end{array}\right) \wedge (A \vee B)
\end{array}\right)
\end{array}\right)
$$

$=$ [Property 4.1 L3]

$$
CSP2\left(\begin{array}{l}
L(A \wedge B \wedge Stop) \vee \\
\qquad L(\neg ok' \wedge A) \vee \\
\qquad L(\neg ok' \wedge B) \vee \\
\quad L\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \right.\right. \\
\qquad \left.\left. (ok' \wedge \neg wait' \wedge tr'_t = tr_t)\right) \wedge (A \vee B)\right) \vee \\
\quad L\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \right.\right. \\
\qquad \left.\left. \left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right)\right) \wedge (A \vee B)\right) \vee \\
\quad L\left(\left(\begin{array}{l} (ok \wedge \neg wait \wedge Skip); \\ (ok' \wedge \neg wait' \wedge tr'_t = tr_t) \end{array}\right) \wedge (A \vee B)\right) \vee \\
\quad L\left(\left(\begin{array}{l} (ok \wedge \neg wait \wedge Skip); \\ \left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right) \end{array}\right) \wedge (A \vee B)\right)
\end{array}\right)
$$

$=$ [Property 4.1 L4]

$$
CSP2\left(\begin{array}{l}
L(A \wedge B \wedge Stop) \vee \\
\qquad \neg ok' \wedge L(A) \vee \\
\qquad \neg ok' \wedge L(B) \vee \\
\quad L\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \right.\right. \\
\qquad \left.\left. (ok' \wedge \neg wait' \wedge tr'_t = tr_t)\right) \wedge (A \vee B)\right) \vee \\
\quad L\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \right.\right. \\
\qquad \left.\left. \left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right)\right) \wedge (A \vee B)\right) \vee \\
\quad L\left(\left(\begin{array}{l} (ok \wedge \neg wait \wedge Skip); \\ (ok' \wedge \neg wait' \wedge tr'_t = tr_t) \end{array}\right) \wedge (A \vee B)\right) \vee \\
\quad L\left(\left(\begin{array}{l} (ok \wedge \neg wait \wedge Skip); \\ \left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right) \end{array}\right) \wedge (A \vee B)\right)
\end{array}\right)
$$

$\Rightarrow$ [Property 4.1 L3]

$$
CSP2\left(\begin{array}{l}
(L(A) \wedge L(B) \wedge L(Stop)) \vee \\
\qquad \neg ok' \wedge L(A) \vee \\
\qquad \neg ok' \wedge L(B) \vee \\
\left(L\left(\begin{array}{l} \left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \\ (ok' \wedge \neg wait' \wedge tr'_t = tr_t) \end{array}\right) \wedge L(A \vee B)\right) \vee \\
\left(L\left(\begin{array}{l} \left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \\ \left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right) \end{array}\right) \wedge L(A \vee B)\right) \vee \\
\left(L\left(\begin{array}{l} (ok \wedge \neg wait \wedge Skip); \\ (ok' \wedge \neg wait' \wedge tr'_t = tr_t) \end{array}\right) \wedge L(A \vee B)\right) \vee \\
\left(L\left(\begin{array}{l} (ok \wedge \neg wait \wedge Skip); \\ \left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right) \end{array}\right) \wedge L(A \vee B)\right)
\end{array}\right)
$$

$=$ [Property 4.1 L2 and L6]

$$
CSP2\left(\begin{array}{l}
(L(A) \wedge L(B) \wedge L(Stop)) \vee \\
\qquad \neg ok' \wedge L(A) \vee \\
\qquad \neg ok' \wedge L(B) \vee \\
\left(\left(\begin{array}{l} L\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \\ L(ok' \wedge \neg wait' \wedge tr'_t = tr_t) \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} L\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ A \wedge B \wedge ok' \wedge \\ wait' \wedge trace' = \langle\rangle \end{array}\right); \\ L\left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right) \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} L(ok \wedge \neg wait \wedge Skip); \\ L(ok' \wedge \neg wait' \wedge tr'_t = tr_t) \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} L(ok \wedge \neg wait \wedge Skip); \\ L\left(\begin{array}{l} ok' \wedge \\ fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle \end{array}\right) \end{array}\right) \wedge (L(A) \vee L(B))\right)
\end{array}\right)
$$

$=$ [Property 4.1 L4]

$$
CSP2\left(\begin{array}{l}
(L(A) \wedge L(B) \wedge L(Stop)) \vee \\
\qquad \neg ok' \wedge L(A) \vee \\
\qquad \neg ok' \wedge L(B) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge \\ L(A \wedge B \wedge trace' = \langle\rangle) \end{array}\right); \right.\right. \\
\qquad \left.\left. (ok' \wedge \neg wait' \wedge L(tr'_t = tr_t))\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge \\ L(A \wedge B \wedge trace' = \langle\rangle) \end{array}\right); \right.\right. \\
\qquad \left.\left. \left(\begin{array}{l} ok' \wedge \\ L(fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \end{array}\right)\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge L(Skip); \\ ok' \wedge \neg wait' \wedge L(tr'_t = tr_t) \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge L(Skip); \\ \left(\begin{array}{l} ok' \wedge \\ L(fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \end{array}\right) \end{array}\right) \wedge (L(A) \vee L(B))\right)
\end{array}\right)
$$

$\Rightarrow$ [Property 4.1 L3]

$$
CSP2\left(\begin{array}{l}
(L(A) \wedge L(B) \wedge L(Stop)) \vee \\
\qquad \neg ok' \wedge L(A) \vee \\
\qquad \neg ok' \wedge L(B) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge \\ L(A) \wedge L(B) \wedge L(trace' = \langle\rangle) \end{array}\right); \right.\right. \\
\qquad \left.\left. (ok' \wedge \neg wait' \wedge L(tr'_t = tr_t))\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge \\ L(A) \wedge L(B) \wedge L(trace' = \langle\rangle) \end{array}\right); \right.\right. \\
\qquad \left.\left. \left(\begin{array}{l} ok' \wedge \\ L(fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \end{array}\right)\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge L(Skip); \\ ok' \wedge \neg wait' \wedge L(tr'_t = tr_t) \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge L(Skip); \\ \left(\begin{array}{l} ok' \wedge \\ L(fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \end{array}\right) \end{array}\right) \wedge (L(A) \vee L(B))\right)
\end{array}\right)
$$

$=$ [Property 4.3 L3 and L2]

$$
CSP2\left(\begin{array}{l}
(L(A) \wedge L(B) \wedge Stop) \vee \\
\qquad \neg ok' \wedge L(A) \vee \\
\qquad \neg ok' \wedge L(B) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge \\ L(A) \wedge L(B) \wedge L(trace' = \langle\rangle) \end{array}\right); \right.\right. \\
\qquad \left.\left. (ok' \wedge \neg wait' \wedge L(tr'_t = tr_t))\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge \\ L(A) \wedge L(B) \wedge L(trace' = \langle\rangle) \end{array}\right); \right.\right. \\
\qquad \left.\left. \left(\begin{array}{l} ok' \wedge \\ L(fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \end{array}\right)\right) \wedge (L(A) \vee L(B))\right) \vee \\
\qquad \left(\left(\begin{array}{l} ok \wedge \neg wait \wedge Skip; \\ ok' \wedge \neg wait' \wedge L(tr'_t = tr_t) \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\left(\begin{array}{c} ok \wedge \neg wait \wedge Skip; \\ \left(\begin{array}{l} ok' \wedge \\ L(fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \end{array}\right) \end{array}\right)\right) \wedge (L(A) \vee L(B))\right)
\end{array}\right)
$$

$=$ [Property 4.1 L8 and L10]

$$
CSP2\left(\begin{array}{l}
(L(A) \wedge L(B) \wedge Stop) \vee \\
\qquad \neg ok' \wedge L(A) \vee \\
\qquad \neg ok' \wedge L(B) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge tr = tr' \wedge \\ L(A) \wedge L(B) \end{array}\right); \right.\right. \\
\qquad \left.\left. (ok' \wedge \neg wait' \wedge tr = tr' \wedge ref = ref')\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge tr = tr' \wedge \\ L(A) \wedge L(B) \end{array}\right); \right.\right. \\
\qquad \left.\left. \left(\begin{array}{l} ok' \wedge \\ L(fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \end{array}\right)\right) \wedge (L(A) \vee L(B))\right) \vee \\
\qquad \left(\left(\begin{array}{l} ok \wedge \neg wait \wedge Skip; \\ ok' \wedge \neg wait' \wedge tr = tr' \wedge ref = ref' \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\left(\begin{array}{c} ok \wedge \neg wait \wedge Skip; \\ \left(\begin{array}{l} ok' \wedge \\ L(fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \end{array}\right) \end{array}\right)\right) \wedge (L(A) \vee L(B))\right)
\end{array}\right)
$$

$=$ $[L(fst(head(dif(tr'_t, tr_t))) \neq \langle\rangle) \equiv tr \neq tr']$

$$
CSP2\left(\begin{array}{l}
(L(A) \wedge L(B) \wedge Stop) \vee \\
\qquad \neg ok' \wedge L(A) \vee \\
\qquad \neg ok' \wedge L(B) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge tr = tr' \wedge \\ L(A) \wedge L(B) \end{array}\right); \right.\right. \\
\qquad \left.\left. (ok' \wedge \neg wait' \wedge tr = tr' \wedge ref = ref') \right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge wait' \wedge tr = tr' \wedge \\ L(A) \wedge L(B) \end{array}\right); \right.\right. \\
\qquad \left.\left. (ok' \wedge tr \neq tr') \right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} (ok \wedge \neg wait \wedge Skip); \\ (ok' \wedge \neg wait' \wedge tr = tr' \wedge ref = ref') \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} (ok \wedge \neg wait \wedge Skip); \\ (ok' \wedge tr \neq tr') \end{array}\right) \wedge (L(A) \vee L(B))\right)
\end{array}\right)
$$

$=$ [Relational calculus]

$$
CSP2\left(\begin{array}{l}
(L(A) \wedge L(B) \wedge Stop) \vee \\
\qquad \neg ok' \wedge L(A) \vee \\
\qquad \neg ok' \wedge L(B) \vee \\
\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ L(A)[true, true/ok', wait'] \wedge \\ L(B)[true, true/ok', wait'] \wedge \\ ok' \wedge \neg wait' \wedge tr = tr' \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{c} ok \wedge \neg wait \wedge \\ L(A)[true, true/ok', wait'] \wedge \\ L(B)[true, true/ok', wait'] \wedge \\ ok' \wedge tr \neq tr' \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge \neg wait' \wedge tr = tr' \end{array}\right) \wedge (L(A) \vee L(B))\right) \vee \\
\left(\left(\begin{array}{l} ok \wedge \neg wait \wedge \\ ok' \wedge tr \neq tr' \end{array}\right) \wedge (L(A) \vee L(B))\right)
\end{array}\right)
$$

$=$ [Predicate calculus]

$$
\begin{array}{l}
CSP2 \left(\begin{array}{l}
(L(A) \land L(B) \land Stop) \lor \\
\qquad (\neg ok' \land (L(A) \lor L(B))) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} ok \land \neg wait \land \\ L(A)[true, true/ok', wait'] \land \\ L(B)[true, true/ok', wait'] \land \\ ok' \land \neg wait' \land tr = tr' \end{array} \right) \lor \\
\left(\begin{array}{l} ok \land \neg wait \land \\ L(A)[true, true/ok', wait'] \land \\ L(B)[true, true/ok', wait'] \land \\ ok' \land tr \neq tr' \end{array} \right) \lor \\
\left(\begin{array}{l} ok \land \neg wait \land \\ ok' \land \neg wait' \land tr = tr' \end{array} \right) \lor \\
\left(\begin{array}{l} ok \land \neg wait \land \\ ok' \land tr \neq tr' \end{array} \right)
\end{array} \right) \land (L(A) \lor L(B))
\end{array} \right) \\
= \qquad \text{[Predicate calculus]} \\
CSP2 \left(\begin{array}{l}
(L(A) \land L(B) \land Stop) \lor \\
\qquad (\neg ok' \land (L(A) \lor L(B))) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l} ok \land \neg wait \land \\ ok' \land \neg wait' \land tr = tr' \end{array} \right) \lor \\
\left(\begin{array}{l} ok \land \neg wait \land \\ ok' \land tr \neq tr' \end{array} \right)
\end{array} \right) \land (L(A) \lor L(B))
\end{array} \right) \\
= \qquad \text{[Predicate calculus]} \\
CSP2 \left(\begin{array}{l}
(L(A) \land L(B) \land Stop) \lor \\
\qquad \neg ok' \lor \\
\left(\begin{array}{l} ok \land \neg wait \land \\ ok' \land \neg wait' \land tr = tr' \end{array} \right) \lor \\
\qquad \left(\begin{array}{l} ok \land \neg wait \land \\ ok' \land tr \neq tr' \end{array} \right)
\end{array} \land (L(A) \lor L(B)) \right) \\
= \qquad \text{[Derivation of } \neg Stop \text{ in Section 3.5.8]} \\
CSP2 \left(\begin{array}{l} (L(A) \land L(B) \land Stop) \lor \\ ((L(A) \lor L(B)) \land \neg Stop) \end{array} \right) \\
= \qquad [3.5.25] \\
[\![L(A) \,\Box\, L(B)]\!] \qquad \Box
\end{array}
$$

In a similar manner we can show that $L([\![A \,\Box\, B]\!]_{time}) \Leftarrow [\![L(A) \,\Box\, L(B)]\!]$

L11. $L([\![A;\ B]\!]_{time}) = [\![L(A);\ L(B)]\!]$
Proof:
Based on property 4.1 L6 $\Box$

L12. $L([\![A \,[\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]\, B]\!]_{time}) = L([\![A]\!]_{time}) \,[\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B \,]\!]\, L([\![B]\!]_{time})$
Proof:

$$
\begin{array}{ll}
& L([\![A \,[\![\, s_A \mid \{\!|\, cs \,|\!\} \mid s_B \,]\!]\, B]\!]_{time}) \\
= & [3.5.39] \\
& L(A \parallel_{TM(cs,s_A,s_B)} B) \\
= & [3.5.32] \\
& L(((A;\ U0) \| (B;\ U1));\ TM(cs, s_A, s_B)) \\
= & [3.5.31] \\
& L(((A;\ U0) \wedge (B;\ U1));\ TM(cs, s_A, s_B)) \\
= & [\text{Property 4.1 L6}] \\
& L((A;\ U0) \wedge (B;\ U1));\ L(TM(cs, s_A, s_B))
\end{array}
$$

We divide the proof above in two parts. The first part of the proof involves showing that $L(TM(cs, s_A, s_B)) = M(cs, s_A, s_B)$, where M is the merge function of *Circus* parallel operator. For the next proof we use an alternative definition for $TSync$

$$
t_c\, TSync t_a, t_b, cs \Leftrightarrow \begin{array}{l}
\exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet \\
\left(\begin{array}{l}
t_a = \langle (t_1, r_1) \rangle \wedge \\
t_b = \langle (t_2, r_2) \rangle \wedge \\
t_c = \langle (t_3, r_3) \rangle \wedge \\
t_3 \in Sync(t_1, t_2, cs) \wedge \\
r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs)
\end{array}\right) \vee \\
\left(\begin{array}{l}
t_a = \langle (t_1, r_1) \rangle \frown S_1 \wedge \\
t_b = \langle (t_2, r_2) \rangle \frown S_2 \wedge \\
t_c = \langle (t_3, r_3) \rangle \frown S_3 \wedge \\
t_3 \in Sync(t_1, t_2, cs) \wedge \\
r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs) \wedge \\
S_3\, TSync S_1, S_2, cs
\end{array}\right)
\end{array}
\qquad \text{(E.3.0)}
$$

$$
\begin{array}{ll}
& L(TM(cs, s_A, s_B)) \\
= & [3.5.40] \\
& L\left(\begin{array}{c}
ok' = (0.ok \wedge 1.ok) \wedge \\
wait' = (0.wait \vee 1.wait) \wedge \\
dif(tr_t', tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right) \wedge \\
state' = (0.state - s_B) \oplus (1.state - s_A)
\end{array}\right) \\
= & [4.1\ \text{L4}]
\end{array}
$$

$$\left(\begin{array}{c} ok' = (0.ok \land 1.ok) \land \\ wait' = (0.wait \lor 1.wait) \land \\ state' = (0.state - s_B) \oplus (1.state - s_A) \land \\ L\left(dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right)\right) \end{array}\right)$$

For to conclude the above proof we need to show that

$$L\left(dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right)\right)$$
$$= \left(\begin{array}{l} tr' - tr \in Sync 0.tr, 1.tr, cs \land \\ ref' = ((0.ref \cup 1.ref) \cap cs) \cup ((0.ref \cap 1.ref) \setminus cs) \end{array}\right)$$

$$L\left(dif(tr'_t, tr_t) \in TSync \left(\begin{array}{c} dif(0.tr_t, tr_t), \\ dif(1.tr_t, tr_t), \\ cs \end{array}\right)\right)$$
$$= \qquad [12]$$
$$L\left(\begin{array}{l} \exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet \\ \left(\begin{array}{l} dif(0.tr_t, tr_t) = \langle (t_1, r_1) \rangle \land \\ dif(1.tr_t, tr_t) = \langle (t_2, r_2) \rangle \land \\ dif(tr'_t, tr_t) = \langle (t_3, r_3) \rangle \land \\ t_3 \in Sync(t_1, t_2, cs) \land \\ r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs) \end{array}\right) \lor \\ \left(\begin{array}{l} dif(0.tr_t, tr_t) = \langle (t_1, r_1) \rangle \frown S_1 \land \\ dif(1.tr_t, tr_t) = \langle (t_2, r_2) \rangle \frown S_2 \land \\ dif(tr'_t, tr_t) = \langle (t_3, r_3) \rangle \frown S_3 \land \\ t_3 \in Sync(t_1, t_2, cs) \land \\ r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs) \land \\ S_3\, TSync S_1, S_2, cs \end{array}\right) \end{array}\right)$$
$$= \qquad [3.4.7]$$

$$
\begin{array}{ll}
 & \left(\begin{array}{l}
\exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet \\
\left(\begin{array}{l}
t_1 = (fst(hd(0.tr_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_1 = snd(hd(0.tr_t - front(tr_t))) \land \\
t_2 = (fst(hd(1.tr_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_2 = snd(hd(1.tr_t - front(tr_t))) \land \\
t_3 = (fst(hd(tr'_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_3 = snd(hd(tr'_t - front(tr_t))) \land \\
t_3 \in Sync(t_1, t_2, cs) \land \\
r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs)
\end{array}\right) \\
\lor \\
\left(\begin{array}{l}
t_1 = (fst(hd(0.tr_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_1 = snd(hd(0.tr_t - front(tr_t))) \land \\
t_2 = (fst(hd(1.tr_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_2 = snd(hd(1.tr_t - front(tr_t))) \land \\
t_3 = (fst(hd(tr'_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_3 = snd(hd(tr'_t - front(tr_t))) \land \\
S_1 = tail(0.tr_t - front(tr)) \land \\
S_2 = tail(1.tr_t - front(tr)) \land \\
S_3 = tail(tr'_t - front(tr)) \land \\
S_3\, TSync S_1, S_2, cs \land t_3 \in Sync(t_1, t_2, cs) \land \\
r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs) \land \\
S_3\, TSync S_1, S_2, cs
\end{array}\right)
\end{array}\right) \\
L & \\
= & \text{[Predicate calculus]} \\
 & \left(\begin{array}{l}
\exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet \\
\left(\begin{array}{l}
t_1 = (fst(hd(0.tr_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_1 = snd(hd(0.tr_t - front(tr_t))) \land \\
t_2 = (fst(hd(1.tr_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_2 = snd(hd(1.tr_t - front(tr_t))) \land \\
t_3 = (fst(hd(tr'_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_3 = snd(hd(tr'_t - front(tr_t))) \land \\
t_3 \in Sync(t_1, t_2, cs) \land \\
r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs)
\end{array}\right)
\end{array}\right) \\
L & \\
= & \text{[4.2.1]}
\end{array}
$$

$$
\begin{array}{l}
\exists\, tr_t, tr'_t \bullet \\
\exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet \\
\left(\begin{array}{l}
t_1 = (fst(hd(0.tr_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_1 = snd(hd(0.tr_t - front(tr_t))) \land \\
t_2 = (fst(hd(1.tr_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_2 = snd(hd(1.tr_t - front(tr_t))) \land \\
t_3 = (fst(hd(tr'_t - front(tr_t))) - fst(last(tr_t))) \land \\
r_3 = snd(hd(tr'_t - front(tr_t))) \land \\
t_3 \in Sync(t_1, t_2, cs) \land \\
r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs)
\end{array}\right) \land \\
tr = Flat(tr_t) \land tr' = Flat(tr'_t) \land \\
ref = snd(last(tr_t)) \land ref' = snd(last(tr'_t))
\end{array}
$$

Next wee use structural induction over time traces. Because time traces are none empty sequences we take the following to be the base step of the structural induction prove.

$$
\begin{array}{l}
tr_t = \langle (tr, ref) \rangle \\
tr'_t = \langle (tr', ref') \rangle \\
0.tr_t = \langle (0.tr, 0.ref) \rangle \\
1.tr_t = \langle (1.tr, 1.ref) \rangle
\end{array}
$$

By substituting in the last equation above

$$
\begin{array}{l}
\exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet \\
\left(\begin{array}{l}
t_1 = (fst(hd(\langle (0.tr, 0.ref) \rangle - front(\langle (tr, ref) \rangle))) - fst(last(\langle (tr, ref) \rangle))) \land \\
r_1 = snd(hd(\langle (0.tr, 0.ref) \rangle - front(\langle (tr, ref) \rangle))) \land \\
t_2 = (fst(hd(\langle (1.tr, 1.ref) \rangle - front(\langle (tr, ref) \rangle))) - fst(last(\langle (tr, ref) \rangle))) \land \\
r_2 = snd(hd(\langle (1.tr, 1.ref) \rangle - front(\langle (tr, ref) \rangle))) \land \\
t_3 = (fst(hd(\langle (tr', ref') \rangle - front(\langle (tr, ref) \rangle))) - fst(last(\langle (tr, ref) \rangle))) \land \\
r_3 = snd(hd(\langle (tr', ref') \rangle - front(\langle (tr, ref) \rangle))) \land \\
t_3 \in Sync(t_1, t_2, cs) \land \\
r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs)
\end{array}\right) \land \\
tr = Flat(\langle (tr, ref) \rangle) \land tr' = Flat(\langle (tr', ref') \rangle) \land \\
ref = snd(last(\langle (tr, ref) \rangle)) \land ref' = snd(last(\langle (tr', ref') \rangle))
\end{array}
$$

$=$ [Sequence operation]

$$
\begin{array}{l}
\exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet \\
\left(\begin{array}{l}
t_1 = (0.tr - tr) \land r_1 = 0.ref \land \\
t_2 = (1.tr - tr) \land r_2 = 1.ref \land \\
t_3 = (tr' - tr) \land r_3 = ref' \land \\
t_3 \in Sync(t_1, t_2, cs) \land \\
r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs)
\end{array}\right) \land \\
tr = tr \land tr' = tr' \land \\
ref = ref \land ref' = ref'
\end{array}
$$

$=$ [Predicate calculus and substitution]

$$(tr' - tr) \in Sync((0.tr - tr), (1.tr - tr), cs) \land$$
$$ref' = ((0.ref \cup 1.ref) \cap cs) \cup ((0.ref \cap 1.ref) \setminus cs)$$

□

Lets consider that the above equation is valid for any arbitrary time trace S, this is the induction hypothesis. We need to prove that it is also valid for the following:

$$tr_t = S \frown \langle (tr, ref) \rangle$$
$$tr'_t = S \frown \langle (tr', ref') \rangle$$
$$0.tr_t = S \frown \langle (0.tr, 0.ref) \rangle$$
$$1.tr_t = S \frown \langle (1.tr, 1.ref) \rangle$$

By substituting in the last equation above

$$\exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet$$
$$\left(\begin{array}{l} t_1 = (fst(hd(S \frown \langle (0.tr, 0.ref) \rangle \\ -front(S \frown \langle (tr, ref) \rangle))) - fst(last(S \frown \langle (tr, ref) \rangle))) \land \\ r_1 = snd(hd(S \frown \langle (0.tr, 0.ref) \rangle - front(S \frown \langle (tr, ref) \rangle))) \land \\ t_2 = (fst(hd(S \frown \langle (1.tr, 1.ref) \rangle - \\ front(S \frown \langle (tr, ref) \rangle))) - fst(last(S \frown \langle (tr, ref) \rangle))) \land \\ r_2 = snd(hd(S \frown \langle (1.tr, 1.ref) \rangle - front(S \frown \langle (tr, ref) \rangle))) \land \\ t_3 = (fst(hd(S \frown \langle (tr', ref') \rangle - \\ front(S \frown \langle (tr, ref) \rangle))) - fst(last(S \frown \langle (tr, ref) \rangle))) \land \\ r_3 = snd(hd(S \frown \langle (tr', ref') \rangle - front(S \frown \langle (tr, ref) \rangle))) \land \\ t_3 \in Sync(t_1, t_2, cs) \land \\ r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs) \end{array} \right) \land$$
$$tr = Flat(S \frown \langle (tr, ref) \rangle) \land tr' = Flat(S \frown \langle (tr', ref') \rangle) \land$$
$$ref = snd(last(S \frown \langle (tr, ref) \rangle)) \land ref' = snd(last(S \frown \langle (tr', ref') \rangle))$$
$$=$$ [Sequence operation]
$$\exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet$$
$$\left(\begin{array}{l} t_1 = (fst(hd(S \frown \langle (0.tr, 0.ref) \rangle - S)) - fst(\langle (tr, ref) \rangle)) \land \\ r_1 = snd(hd(S \frown \langle (0.tr, 0.ref) \rangle - S)) \land \\ t_2 = (fst(hd(S \frown \langle (1.tr, 1.ref) \rangle - S)) - fst(\langle (tr, ref) \rangle)) \land \\ r_2 = snd(hd(S \frown \langle (1.tr, 1.ref) \rangle - S)) \land \\ t_3 = (fst(hd(S \frown \langle (tr', ref') \rangle - S)) - fst(\langle (tr, ref) \rangle)) \land \\ r_3 = snd(hd(S \frown \langle (tr', ref') \rangle - S)) \land \\ t_3 \in Sync(t_1, t_2, cs) \land \\ r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs) \end{array} \right) \land$$
$$tr = Flat(S) \frown Flat(\langle (tr, ref) \rangle) \land tr' = Flat(S) \frown Flat(\langle (tr', ref') \rangle) \land$$
$$ref = snd(\langle (tr, ref) \rangle) \land ref' = snd(\langle (tr', ref') \rangle)$$
$$=$$ [Sequence operation]

$$
\begin{array}{l}
\exists\, t_1, t_2, t_3, r_1, r_2, r_3 \bullet \\
\left(\begin{array}{l}
t_1 = (0.tr - tr) \land r_1 = 0.ref \land \\
t_2 = (1.tr - tr) \land r_2 = 1.ref \land \\
t_3 = (tr' - tr) \land r_3 = ref' \land \\
t_3 \in Sync(t_1, t_2, cs) \land \\
r_3 = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs)
\end{array} \right) \land \\
tr = Flat(S) \frown tr \land tr' = Flat(S) \frown tr' \land \\
ref = ref \land ref' = ref' \\
= \qquad \text{[Predicate calculus, substitution and } Flat(S) = \langle\rangle] \\
(tr' - tr) \in Sync((0.tr - tr), (1.tr - tr), cs) \land \\
ref' = ((0.ref \cup 1.ref) \cap cs) \cup ((0.ref \cap 1.ref) \setminus cs) \qquad \Box
\end{array}
$$

Next we need to prove that $L((A;\ U0) \land (B;\ U1)) \Rightarrow (L(A);\ U0 \land L(B);\ U1)$. From the definition of $U0$ and $U1$ are simple renaming functions, they take each input variable, example tr and rename it to $0.tr$ or $1.tr$ respectively. Therefore the functions $U0$ and $U1$ have no semantic effect and then $(L(A);\ U0) = L(A;\ U0)$

L13. $L([\![A \setminus cs]\!]_{time}) = L([\![A]\!]_{time}) \setminus cs$
Proof:

$$
\begin{array}{l}
L(A \setminus cs) \\
= \qquad \text{[3.5.55]} \\
L\left(\begin{array}{l} R_t\left(\begin{array}{l} \exists\, s_t \bullet A[s_t/tr'_t] \land \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array} \right); \\ Skip \end{array} \right) \\
= \qquad \text{[Property 4.1 L3]} \\
L\left(R_t\left(\begin{array}{l} \exists\, s_t \bullet A[s_t/tr'_t] \land \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array} \right) \right); \\
L(Skip) \\
= \qquad \text{[Property 4.3 L1]} \\
L\left(R_t\left(\begin{array}{l} \exists\, s_t \bullet A[s_t/tr'_t] \land \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array} \right) \right); \\
Skip \\
= \qquad \text{[Property 4.2 L1,L2 and L3]} \\
R\left(L\left(\begin{array}{l} \exists\, s_t \bullet A[s_t/tr'_t] \land \\ dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs) \end{array} \right) \right); \\
Skip \\
= \qquad \text{[A.3]}
\end{array}
$$

$$
R\left(\begin{array}{l} L\left(\begin{array}{l} \exists\, s_t \bullet A[s_t/tr'_t] \land \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ \left(\begin{array}{l} fst(dif(tr'_t, tr_t)(i)) = fst(dif(s_t, tr_t)(i)) \downarrow cs \land \\ snd(dif(s_t, tr_t)(i)) = (snd(dif(tr'_t, tr_t)(i)) \cup (Events - cs)) \land \\ \#dif(s_t, tr_t) = \#dif(s_t, tr_t)(i) \end{array}\right) \end{array}\right) \\ Skip \end{array}\right);
$$

$=$ [4.2.1]

$$
R\left(\begin{array}{l} \exists\, tr_t, tr'_t, s_t \bullet A[s_t/tr'_t] \land \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ \left(\begin{array}{l} fst(dif(tr'_t, tr_t)(i)) = fst(dif(s_t, tr_t)(i)) \downarrow cs \land \\ snd(dif(s_t, tr_t)(i)) = (snd(dif(tr'_t, tr_t)(i)) \cup (Events - cs)) \land \\ \#dif(s_t, tr_t) = \#dif(s_t, tr_t)(i) \end{array}\right) \land \\ tr = Flat(tr_t) \land tr' = Flat(tr'_t) \land \\ ref = snd(last(tr_t)) \land ref' = snd(last(tr'_t)) \\ Skip \end{array}\right);
$$

$=$ [predicate calculus]

$$
R\left(\begin{array}{l} \exists\, tr_t, tr'_t, s_t \bullet A[s_t/tr'_t] \land \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ fst(dif(tr'_t, tr_t)(i)) = fst(dif(s_t, tr_t)(i)) \downarrow cs \land \\ \forall\, i : 1..\#dif(s_t, tr_t) \bullet \\ snd(dif(s_t, tr_t)(i)) = (snd(dif(tr'_t, tr_t)(i)) \cup (Events - cs)) \land \\ \#dif(s_t, tr_t) = \#dif(s_t, tr_t)(i) \land \\ tr = Flat(tr_t) \land tr' = Flat(tr'_t) \land \\ ref = snd(last(tr_t)) \land ref' = snd(last(tr'_t)) \\ Skip \end{array}\right);
$$

$=$ [Property 3.3 L3,L6 and L7]

$$
R\left(\begin{array}{l} \exists\, tr_t, tr'_t, s_t \bullet A[s_t/tr'_t] \land \\ Flat(tr'_t) - Flat(tr_t) = (Flat(s) - Flat(tr_t)) \downarrow cs \land \\ \forall\, i : \#tr_t..\#s_t \bullet snd(s_t(i)) = (snd(tr'_t(i)) \cup (Events - cs)) \land \\ \#dif(s_t, tr_t) = \#dif(s_t, tr_t)(i) \land \\ tr = Flat(tr_t) \land tr' = Flat(tr'_t) \land \\ ref = snd(last(tr_t)) \land ref' = snd(last(tr'_t)) \\ Skip \end{array}\right);
$$

$=$ [Predicate calculus and substitution]

$$
R\left(\begin{array}{l} \exists\, tr_t, tr'_t, s_t \bullet A[s_t/tr'_t] \land \\ tr = Flat(tr_t) \land tr' = Flat(tr'_t) \land \\ ref = snd(last(tr_t)) \land ref' = snd(last(tr'_t)) \land \\ tr' - tr = (Flat(s) - tr) \downarrow cs \land \\ snd(last(s_t)) = (snd(last(tr'_t)) \cup (Events - cs)) \land \\ \forall\, i : \#tr_t..\#s_t - 1 \bullet snd(s_t(i)) = (snd(tr'_t(i)) \cup (Events - cs)) \land \\ \#dif(s_t, tr_t) = \#dif(s_t, tr_t)(i) \\ Skip \end{array}\right);
$$

$$
\begin{array}{ll}
= & \text{[Substitution]} \\
& R\left(\begin{array}{l}
\exists\, tr_t, tr'_t, s_t \bullet A[s_t/tr'_t] \land \\
tr = Flat(tr_t) \land tr' = Flat(tr'_t) \land \\
ref = snd(last(tr_t)) \land ref' = snd(last(tr'_t) \land \\
tr' - tr = (Flat(s) - tr) \downarrow cs \land \\
snd(last(s_t)) = (ref' \cup (Events - cs)) \land \\
\forall\, i : \#tr_t..\#s_t - 1 \bullet snd(s_t(i)) = (snd(tr'_t(i)) \cup (Events - cs)) \land \\
\#dif(s_t, tr_t) = \#dif(s_t, tr_t)(i)
\end{array}\right); \\
& Skip \\
= & \text{[4.2.1]} \\
& R\left(\begin{array}{l}
\exists\, s_t \bullet L(A)[s_t/tr'_t] \land \\
tr' - tr = (Flat(s) - tr) \downarrow cs \land \\
snd(last(s_t)) = (ref' \cup (Events - cs)) \land \\
\forall\, i : \#tr_t..\#s_t - 1 \bullet snd(s_t(i)) = (snd(tr'_t(i)) \cup (Events - cs)) \land \\
\#dif(s_t, tr_t) = \#dif(s_t, tr_t)(i)
\end{array}\right); \\
& Skip \\
= & \text{[Rename } s_t \text{ to } s \text{ and } r \text{ such that } s = Flat(s) \text{ and } r = snd(last(s_t))\text{]} \\
& R\left(\begin{array}{l}
\exists\, s_t \bullet L(A)[s, r/tr', ref'] \land \\
tr' - tr = (s - tr) \downarrow cs \land \\
r = (ref' \cup (Events - cs)) \land \\
\forall\, i : \#tr_t..\#s_t - 1 \bullet snd(s_t(i)) = (snd(tr'_t(i)) \cup (Events - cs)) \land \\
\#dif(s_t, tr_t) = \#dif(s_t, tr_t)(i)
\end{array}\right); \\
& Skip \\
= & \text{[refusals are irrelevant]} \\
& R\left(\begin{array}{l}
\exists\, s_t \bullet L(A)[s, r/tr', ref'] \land \\
tr' - tr = (s - tr) \downarrow cs \land \\
r = (ref' \cup (Events - cs)) \land
\end{array}\right); \\
& Skip \\
= & \text{[3.5.53]} \\
& L(A) \setminus cs \qquad \square
\end{array}
$$

L14. $L([\![\, \mu\, X \bullet A(X) \,]\!]_{time}) = \mu\, X' \bullet L(A(X'))$
Proof:

$$
\begin{array}{ll}
& \mu\, X \bullet L(A(X)) \\
= & \text{[UTP Exercise 2.6.4 (1)]} \\
& L(\mu\, X \bullet A(L(X))) \\
= & \text{[Rename } X \text{ to } X' \text{ such that } L(X) = X'\text{]} \\
& L(\mu\, X' \bullet A(X')) \,\square
\end{array}
$$

APPENDIX F

PROOFS FOR CHAPTER 5

In this appendix we list the proofs of some properties explored in chapter 5.

F.1 EQUIVALENCE OF CHOICE OPERATORS

In Section 5.2.1 we give a new definition for the external choice operator using parallel composition. In this section we present the proof that the two definitions are equivalent.
Proof: Let A and B be two healthy actions, then we intend to show that both definitions are equivalent. Lets consider the new definition for external choice and show in 3.5.26.

$$
\begin{aligned}
& A \,\Box\, B \\
&= && [5.2.4] \\
& CSP2\left(\; CSP1\left(\; \left(\begin{array}{c} (A \|_{CM} B) \vee \\ ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array}\right) \right) \right) \\
&= && [3.5.32] \\
& CSP2\left(\; CSP1\left(\; \left(\begin{array}{c} ((A;\ U0) \| (B;\ U1));\ CM \vee \\ ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array}\right) \right) \right) \\
&= && [3.5.33 \text{ and } 3.5.34]
\end{aligned}
$$

$$
CSP2\left(CSP1\left(\begin{array}{l}
\left(\begin{array}{l} ok' \land wait \land (tr' = tr) \land \\ (wait' = wait) \land (state' = state) \land \\ (ref' = ref) \end{array}\right) \lor \\
\left(\left(\begin{array}{l}
\left(\begin{array}{l} A; \\ \mathbf{var}\ 0.ok, 0.wait; \\ \mathbf{var}\ 0.tr, 0.ref, 0.state; \\ \left(\begin{array}{l} 0.ok = ok \land \\ 0.wait = wait \land \\ 0.tr = tr \land \\ 0.ref = ref \land \\ 0.state = state \end{array}\right); \\ \mathbf{end}\ ok, wait, tr; \\ \mathbf{end}\ ref, state \end{array}\right) \land \\
\left(\begin{array}{l} B; \\ \mathbf{var}\ 1.ok, 1.wait; \\ \mathbf{var}\ 1.tr, 1.ref, 1.state; \\ \left(\begin{array}{l} 1.ok = ok \land \\ 1.wait = wait \land \\ 1.tr = tr \land \\ 1.ref = ref \land \\ 1.state = state \end{array}\right); \\ \mathbf{end}\ ok, wait, tr; \\ \mathbf{end}\ ref, state \end{array}\right)
\end{array}\right); CM; \\
\left(\begin{array}{l} \mathbf{end}\ 0.ok, 1.ok;\ \mathbf{end}\ 0.wait, 1.wait; \\ \mathbf{end}\ 0.tr, 1.tr;\ \mathbf{end}\ 0.ref, 1.ref; \\ \mathbf{end}\ 0.state, 1.state \end{array}\right)
\right)
\end{array}\right)\right)
$$

$=$ [A and B do not use indexed vararariables]

$$
CSP2\left(\begin{array}{l} CSP1\left(\begin{array}{l}
\left(\begin{array}{l} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array}\right) \vee \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\textbf{var } 0.ok, 0.wait; \\
\textbf{var } 0.tr, 0.ref, 0.state; \\
A; \\
\left(\begin{array}{l} 0.ok = ok \wedge \\ 0.wait = wait \wedge \\ 0.tr = tr \wedge \\ 0.ref = ref \wedge \\ 0.state = state \end{array}\right); \\
\textbf{end } ok, wait, tr; \\
\textbf{end } ref, state
\end{array}\right) \\
\left(\begin{array}{l}
\textbf{var } 1.ok, 1.wait; \\
\textbf{var } 1.tr, 1.ref, 1.state; \\
B; \\
\left(\begin{array}{l} 1.ok = ok \wedge \\ 1.wait = wait \wedge \\ 1.tr = tr \wedge \\ 1.ref = ref \wedge \\ 1.state = state \end{array}\right); \\
\textbf{end } ok, wait, tr; \\
\textbf{end } ref, state
\end{array}\right)
\end{array}\right) \wedge \\
\left(\begin{array}{l} \textbf{end } 0.ok, 1.ok; \ \textbf{end } 0.wait, 1.wait; \\ \textbf{end } 0.tr, 1.tr; \ \textbf{end } 0.ref, 1.ref; \\ \textbf{end } 0.state, 1.state \end{array}\right)
\end{array}\right); \ CM;
\end{array}\right)
$$

$$
= \qquad [3.5.21]
$$

$$
CSP2\left(CSP1\left(\begin{array}{l}
\left(\begin{array}{l}
ok' \land wait \land (tr' = tr) \land \\
(wait' = wait) \land (state' = state) \land \\
(ref' = ref)
\end{array}\right) \lor \\
\left(\left(\begin{array}{l}
\left(\begin{array}{l}
\mathbf{var}\ 0.ok, 0.wait; \\
\mathbf{var}\ 0.tr, 0.ref, 0.state; \\
\left(\begin{array}{l}
\exists\, ok_o, wait_o, tr_o, ref_o, state_o \bullet \\
\quad A\left[\begin{array}{l}
ok_o/ok', \\
wait_o/wait', \\
tr_o/tr', \\
ref_o/ref', \\
state_o/state
\end{array}\right] \land \\
\quad \left(\begin{array}{l}
0.ok = ok_o \land \\
0.wait = wait_o \land \\
0.tr = tr_o \land \\
0.ref = ref_o \land \\
0.state = state_o
\end{array}\right)
\end{array}\right) \\
\mathbf{end}\ ok, wait, tr; \\
\mathbf{end}\ ref, state
\end{array}\right) \land \\
\left(\begin{array}{l}
\mathbf{var}\ 1.ok, 1.wait; \\
\mathbf{var}\ 1.tr, 1.ref, 1.state; \\
\left(\begin{array}{l}
\exists\, ok_o, wait_o, tr_o, ref_o, state_o \bullet \\
\quad B\left[\begin{array}{l}
ok_o/ok', \\
wait_o/wait', \\
tr_o/tr', \\
ref_o/ref', \\
state_o/state
\end{array}\right] \land \\
\quad \left(\begin{array}{l}
1.ok = ok_o \land \\
1.wait = wait_o \land \\
1.tr = tr_o \land \\
1.ref = ref_o \land \\
1.state = state_o
\end{array}\right)
\end{array}\right) \\
\mathbf{end}\ ok, wait, tr; \\
\mathbf{end}\ ref, state
\end{array}\right)
\end{array}\right); \\
CM; \\
\left(\begin{array}{l}
\mathbf{end}\ 0.ok, 1.ok;\ \mathbf{end}\ 0.wait, 1.wait; \\
\mathbf{end}\ 0.tr, 1.tr;\ \mathbf{end}\ 0.ref, 1.ref; \\
\mathbf{end}\ 0.state, 1.state
\end{array}\right)
\end{array}\right)
\end{array}\right)\right)
$$

$=$ [indexed variables not free]

$$CSP2\left(CSP1\left(\begin{array}{l}\left(\begin{array}{l}ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref)\end{array}\right) \vee \\ \left(\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\exists\, ok_o, wait_o, tr_o, ref_o, state_o \bullet \\ A\left[\begin{array}{l}ok_o, wait_o, tr_o, ref_o, state_o / \\ ok', wait', tr', ref', state'\end{array}\right] \wedge \\ \left(\begin{array}{l}0.ok = ok_o \wedge 0.wait = wait_o \wedge \\ 0.tr = tr_o \wedge 0.ref = ref_o \wedge \\ 0.state = state_o\end{array}\right)\end{array}\right) \\ \mathbf{end}\ ok, wait, tr; \\ \mathbf{end}\ ref, state\end{array}\right) \wedge \\ \left(\begin{array}{l}\left(\begin{array}{l}\exists\, ok_o, wait_o, tr_o, ref_o, state_o \bullet \\ B\left[\begin{array}{l}ok_o, wait_o, tr_o, ref_o, state_o / \\ ok', wait', tr', ref', state'\end{array}\right] \wedge \\ \left(\begin{array}{l}1.ok = ok_o \wedge 1.wait = wait_o \wedge \\ 1.tr = tr_o \wedge 1.ref = ref_o \wedge \\ 1.state = state_o\end{array}\right)\end{array}\right) \\ \mathbf{end}\ ok, wait, tr; \\ \mathbf{end}\ ref, state\end{array}\right)\end{array}\right); \\ CM; \\ \left(\begin{array}{l}\mathbf{end}\ 0.ok, 1.ok;\ \mathbf{end}\ 0.wait, 1.wait; \\ \mathbf{end}\ 0.tr, 1.tr;\ \mathbf{end}\ 0.ref, 1.ref; \\ \mathbf{end}\ 0.state, 1.state\end{array}\right)\end{array}\right)\end{array}\right)\right)$$

$$= \qquad [\text{susbtitution}]$$

$$CSP2\left(CSP1\left(\begin{array}{l}\left(\begin{array}{l}ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref)\end{array}\right) \vee \\ \left(\begin{array}{l}\left(\begin{array}{l}A\left[\begin{array}{l}0.ok, 0.wait, 0.tr, 0.ref, 0.state / \\ ok', wait', tr', ref', state'\end{array}\right] \wedge \\ B\left[\begin{array}{l}1.ok, 1.wait, 1.tr, 1.ref, 1.state / \\ ok', wait', tr', ref', state'\end{array}\right]\end{array}\right); \\ \mathbf{end}\ ok, wait, tr; \\ \mathbf{end}\ ref, state; \\ CM;\ \left(\begin{array}{l}\mathbf{end}\ 0.ok, 1.ok;\ \mathbf{end}\ 0.wait, 1.wait; \\ \mathbf{end}\ 0.tr, 1.tr;\ \mathbf{end}\ 0.ref, 1.ref; \\ \mathbf{end}\ 0.state, 1.state\end{array}\right)\end{array}\right)\end{array}\right)\right)$$

$$= \qquad [\text{unmarked variables free}]$$

$$
CSP2\left(\begin{array}{l} CSP1\left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array}\right) \vee \\ \left(\begin{array}{l} \left(\begin{array}{l} A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\ B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \end{array}\right); \\ CM; \left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok;\ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr;\ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array}\right) \end{array}\right) \end{array}\right)
$$

$=$ [Associativity of sequential comp. 5.2.4,5.2.4]

$$
CSP2\left(\begin{array}{l} CSP1\left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array}\right) \vee \\ \left(\begin{array}{l} \left(\begin{array}{l} A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\ B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\ \left(\begin{array}{c} Diverge \vee \\ WaitokNoEvent \vee \\ DoAnyEvent \vee \\ Terminate \end{array}\right) \end{array}\right); \\ \left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok;\ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr;\ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array}\right) \end{array}\right) \end{array}\right)
$$

$=$ [Predicate calculus]

$$
\begin{array}{l}
CSP2 \left(\begin{array}{l} CSP1 \left(\begin{array}{l}
\left(\begin{array}{l} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array} \right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} A \left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ B \left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \end{array} \right) \vee \\ Diverge \end{array} \right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} A \left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ B \left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \end{array} \right) \vee \\ WaitokNoEvent \end{array} \right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} A \left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ B \left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \end{array} \right) \vee \\ DoAnyEvent \end{array} \right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} A \left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ B \left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \end{array} \right) \\ Terminate \end{array} \right) \\
\left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok;\ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr;\ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array} \right)
\end{array} \right) \end{array} \right) \\
= \qquad [5.2.4]
\end{array}
$$

$$
CSP2\left(CSP1\left(\begin{array}{l}
\left(\begin{array}{l} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array}\right) \vee \\
\left(\begin{array}{l}
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
\left(\begin{array}{l} ok \wedge \neg ok' \wedge \\
\left(\begin{array}{c} 0.ok = ok' \wedge 0.wait = wait' \wedge 0.tr = tr' \\ 0.ref = ref' \wedge 0.state = state' \end{array}\right) \vee \\
\left(\begin{array}{c} 1.ok = ok' \wedge 1.wait = wait' \wedge 1.tr = tr' \\ 1.ref = ref' \wedge 1.state = state' \end{array}\right)
\end{array}\right)
\end{array}\right) \vee \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
WaitokNoEvent
\end{array}\right) \vee \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
DoAnyEvent
\end{array}\right) \vee \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
Terminate
\end{array}\right)
\end{array}\right) ; \\
\left(\begin{array}{l} \mathbf{end}\ 0.ok, 1.ok;\ \mathbf{end}\ 0.wait, 1.wait; \\ \mathbf{end}\ 0.tr, 1.tr;\ \mathbf{end}\ 0.ref, 1.ref; \\ \mathbf{end}\ 0.state, 1.state \end{array}\right)
\end{array}\right)\right)
$$

$=$ [Predicate calculus]

$$
CSP2\left(CSP1\left(\begin{array}{l}
\left(\begin{array}{l}
ok' \land wait \land (tr' = tr) \land \\
(wait' = wait) \land (state' = state) \land \\
(ref' = ref)
\end{array}\right) \lor \\
\left(\begin{array}{l}
\left(\begin{array}{l}
ok \land \neg ok' \land \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
\left(\begin{array}{c} 0.ok = ok' \land 0.wait = wait' \land 0.tr = tr' \\ 0.ref = ref' \land 0.state = state' \end{array}\right)
\end{array}\right) \lor \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
\left(\begin{array}{c} 1.ok = ok' \land 1.wait = wait' \land 1.tr = tr' \\ 1.ref = ref' \land 1.state = state' \end{array}\right)
\end{array}\right)
\end{array}\right) \lor \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
WaitokNoEvent
\end{array}\right) \lor \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
DoAnyEvent
\end{array}\right) \lor \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \land \\
Terminate
\end{array}\right)
\end{array}\right) ; \\
\left(\begin{array}{l}
\mathbf{end}\ 0.ok, 1.ok;\ \mathbf{end}\ 0.wait, 1.wait; \\
\mathbf{end}\ 0.tr, 1.tr;\ \mathbf{end}\ 0.ref, 1.ref; \\
\mathbf{end}\ 0.state, 1.state
\end{array}\right)
\end{array}\right)\right)
$$

$$
= \qquad \text{[Susbtitution]}
$$

$$
CSP2\left(CSP1\left(\begin{array}{l}
\left(\begin{array}{l}
ok' \wedge wait \wedge (tr' = tr) \wedge \\
(wait' = wait) \wedge (state' = state) \wedge \\
(ref' = ref)
\end{array}\right) \vee \\
\left(\begin{array}{l}
\left(\begin{array}{l}
ok \wedge \neg ok' \wedge \\
\left(\begin{array}{l}
A \wedge \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right]
\end{array}\right) \vee \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
B
\end{array}\right)
\end{array}\right) \vee \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
WaitokNoEvent
\end{array}\right) \vee \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
DoAnyEvent
\end{array}\right) \vee \\
\left(\begin{array}{l}
A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\
Terminate
\end{array}\right)
\end{array}\right) ; \\
\left(\begin{array}{l}
\mathbf{end}\ 0.ok, 1.ok;\ \mathbf{end}\ 0.wait, 1.wait; \\
\mathbf{end}\ 0.tr, 1.tr;\ \mathbf{end}\ 0.ref, 1.ref; \\
\mathbf{end}\ 0.state, 1.state
\end{array}\right)
\end{array}\right)\right)
$$

$=$ [sequential composition distributes over $\vee$]

$$
\begin{array}{l}
CSP2 \left(\begin{array}{l}
CSP1 \left(\begin{array}{l}
\left(\begin{array}{l} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array} \right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} ok \wedge \neg ok' \wedge \\ \left(\begin{array}{l} A \wedge \\ B \left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state / \\ ok', wait', tr', ref', state' \end{array} \right] \end{array} \right) \vee \\ \left(\begin{array}{l} A \left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ B \end{array} \right) \end{array} \right) ; \\ \left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok; \ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr; \ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array} \right) \end{array} \right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} A \left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ B \left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ WaitokNoEvent \end{array} \right) ; \\ \left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok; \ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr; \ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array} \right) \end{array} \right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} A \left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ B \left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ DoAnyEvent \end{array} \right) ; \\ \left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok; \ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr; \ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array} \right) \end{array} \right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} A \left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ B \left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state / \\ ok', wait', tr', ref', state' \end{array} \right] \wedge \\ Terminate \end{array} \right) ; \\ \left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok; \ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr; \ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array} \right) \end{array} \right) \end{array} \right) \\
= \qquad \text{[property of } \textbf{end} \text{]}
\end{array}
$$

$$
CSP2\left(\begin{array}{l} CSP1\left(\begin{array}{l}
\left(\begin{array}{c} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array}\right) \vee \\
(ok \wedge \neg ok' \wedge (A \vee B)) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\ B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\ WaitokNoEvent \end{array}\right); \\ \left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok;\ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr;\ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array}\right) \end{array}\right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\ B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\ DoAnyEvent \end{array}\right); \\ \left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok;\ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr;\ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array}\right) \end{array}\right) \vee \\
\left(\begin{array}{l} \left(\begin{array}{l} A\left[\begin{array}{l} 0.ok, 0.wait, 0.tr, 0.ref, 0.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\ B\left[\begin{array}{l} 1.ok, 1.wait, 1.tr, 1.ref, 1.state/ \\ ok', wait', tr', ref', state' \end{array}\right] \wedge \\ Terminate \end{array}\right); \\ \left(\begin{array}{l} \textbf{end}\ 0.ok, 1.ok;\ \textbf{end}\ 0.wait, 1.wait; \\ \textbf{end}\ 0.tr, 1.tr;\ \textbf{end}\ 0.ref, 1.ref; \\ \textbf{end}\ 0.state, 1.state \end{array}\right) \end{array}\right) \end{array}\right)
$$

$=$ [5.2.4,5.2.4, 5.2.3 and applying the last two steps]

$$
CSP2\left(\begin{array}{l} CSP1\left(\begin{array}{c}
\left(\begin{array}{c} ok' \wedge wait \wedge (tr' = tr) \wedge \\ (wait' = wait) \wedge (state' = state) \wedge \\ (ref' = ref) \end{array}\right) \vee \\
(ok' \wedge \neg wait \wedge tr = tr' \wedge wait' \wedge (A \wedge B)) \vee \\
(ok \wedge (A \vee B) \wedge \neg ok') \vee \\
(ok \wedge (A \vee B) \wedge (tr' \neq tr)) \vee \\
(ok \wedge (A \vee B) \wedge \neg wait')
\end{array}\right)\end{array}\right)
$$

$=$ [3.5.26]

$A \,\Box\, B$ $\Box$

From the above derivation we can clearly conclude that the two definitions for external choice are equivalent.

F.2 PROOF OF THEOREM 5.1

First lets consider the base cases except for assignment that has been considered in Chapter 5. **Case:** *Skip*

$$\Phi(Skip)\ \mathbf{par}\ Timers(k,n) \equiv Skip$$

$\Phi(Skip)\ \mathbf{par}\ Timers(k,n)$
$=$ [5.2.1]
$Skip\ \mathbf{par}\ Timers(k,n)$
$=$ [5.2.3]
$(Skip;\ Terminate(n) \llbracket \{\} \mid \{\!|\ TSet\ |\!\} \mid \{\} \rrbracket\ Timers(k,n)) \setminus TSet$
$=$ [*Skip* left identity of sequential composition]
$(Terminate(n) \llbracket \{\} \mid \{\!|\ TSet\ |\!\} \mid \{\} \rrbracket\ Timers(k,n)) \setminus TSet$
$=$ [Lemma 5.1]
$Skip$ □

Case: *Stop*

$$\Phi(Stop)\ \mathbf{par}\ Timers(k,n) \equiv Stop$$

$\Phi(Stop)\ \mathbf{par}\ Timers(k,n)$
$=$ [5.2.2]
$\Phi(Stop)\ \mathbf{par}\ Timers(k,n)$
$=$ [5.2.3]
$(Stop;\ Terminate(n) \llbracket \{\} \mid \{\!|\ TSet\ |\!\} \mid \{\} \rrbracket\ Timers(k,n)) \setminus TSet$
$=$ [Property of *Stop*]
$(Stop \llbracket \{\} \mid \{\!|\ TSet\ |\!\} \mid \{\} \rrbracket\ Timers(k,n)) \setminus TSet$
$=$ [5.2.2]

$$\left(\begin{array}{c} Stop \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ \Vert_{i=1}^{n}\ (Timer(i, delay(i))) \end{array} \right) \setminus TSet$$

$=$ [5.2.2]

$$
\left(\begin{array}{c} Stop \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ \Big|\!\Big|\!\Big|_{i=1}^{n}\ \mu\, X \bullet \left(\begin{array}{l} setup.i.delay(i) \rightarrow \\ \left(\begin{array}{l} (halt.i \rightarrow X) \\ \Box \left(\begin{array}{l} Wait\ \ \mathsf{delay(i)}; \\ \left(\begin{array}{l} (out.i \rightarrow X) \\ \Box \\ (terminate.i \rightarrow Skip) \end{array}\right) \end{array}\right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array}\right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array}\right) \end{array}\right) \setminus TSet
$$

$=$ [Step law and 5.2.2]

$$
\left(\begin{array}{c} Stop \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ \Big|\!\Big|\!\Big|_{i=1}^{n}\ \left(\begin{array}{l} setup.i.delay(i) \rightarrow \\ \left(\begin{array}{l} (halt.i \rightarrow (Timer(i, delay(i)))) \\ \Box \left(\begin{array}{l} Wait\ \ \mathsf{delay(i)}; \\ \left(\begin{array}{l} (out.i \rightarrow (Timer(i, delay(i)))) \\ \Box \\ (terminate.i \rightarrow Skip) \end{array}\right) \end{array}\right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array}\right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array}\right) \end{array}\right) \setminus TSet
$$

$=$ $[terminate.i \in TSet \wedge setup.i.delay(i) \in TSet]$

$Stop$ $\Box$

Case: *Chaos*

$$\Phi(Chaos)\ \mathbf{par}\ Timers(k,n) \equiv Stop$$

$\Phi(Chaos)\ \mathbf{par}\ Timers(k,n)$

$=$ [5.2.3]

$Chaos\ \mathbf{par}\ Timers(k,n)$

$=$ [5.2.3]

$$
\left(\begin{array}{c} (Chaos;\ Terminate(n)) \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ Timers(k,n) \end{array}\right) \setminus TSet
$$

$=$ [*Chaos* is left zero of CSP sequential compositon]

$$
\left(\begin{array}{c} Chaos \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ Timers(k,n) \end{array}\right) \setminus TSet
$$

$=$ [Property 3.12 L4]

$Chaos \setminus TSet$

$=$ [hide]
$Chaos$ $\Box$

Next we assume that for any actions A and B then $\Phi(A)$ **par** $Timers(k,n) \equiv A$ and $\Phi(B)$ **par** $Timers(j,m) \equiv B$ such that the timer index intervals (k,n) and (j,m) are disjoint. This is the hypothesis of the structural induction proof.

Case: Guarded action

$$\Phi(b\&A) \textbf{ par } Timers(k,n) \equiv b\&A$$

$\Phi(b\&A)$ **par** $Timers(k,n)$
$=$ [5.2.5]
$b\&\Phi(A)$ **par** $Timers(k,n)$
$=$ [b free in $Timers(k,n)$]
$b\&(\Phi(A)$ **par** $Timers(k,n))$
$=$ [from the hypothesis]
$b\&A$ $\Box$

Case: Communication

$$\Phi(c \to A) \textbf{ par } Timers(k,n) \equiv c \to A$$

$\Phi(c \to A)$ **par** $Timers(k,n)$
$=$ [5.2.7]
$(c \to \Phi(A))$ **par** $Timers(k,n)$
$=$ [$c \notin TSet$ and property 3.12 L14]
$c \to (\Phi(A)$ **par** $Timers(k,n))$
$=$ [from the hypothesis]
$c \to A$ $\Box$

Case: Recursion

$$\Phi(\mu\, X \bullet A(X)) \textbf{ par } Timers(k,n) \equiv \mu\, X \bullet A(X)$$

$\Phi(\mu\, X \bullet A(X))$ **par** $Timers(k,n)$
$=$ [5.2.11]
$(\mu\, X \bullet \Phi(A(\Phi(X))))$ **par** $Timers(k,n)$
$=$ [5.2.12]
$\mu\, X \bullet (\Phi(A)(X)$ **par** $Timers(k,n))$
$=$ [from the hypothesis]

$= \mu\, X \bullet A(X)$ $\square$

Case: Conditional choice

$$\Phi(A \lhd b \rhd B) \textbf{ par } (Timers(k,n) \,|||\, Timers(j,m)) \equiv (A \lhd b \rhd B)$$

$\Phi(A \lhd b \rhd B) \textbf{ par } (Timers(k,n) \,|||\, Timers(j,m))$

$=$ [5.2.8]

$(\Phi(A) \lhd b \rhd \Phi(B)) \textbf{ par } (Timers(k,n) \,|||\, Timers(j,m))$

$=$ [5.2.3]

$$\left(\begin{array}{c} ((\Phi(A) \lhd b \rhd \Phi(B));\ (Terminate(k,n) \,|||\, Terminate(j,m))) \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array} \right) \setminus TSet$$

$=$ [Property 3.6 L5]

$$\left(\begin{array}{c} \left(\begin{array}{c} (\Phi(A);\ (Terminate(k,n) \,|||\, Terminate(j,m))) \\ \lhd\ b\ \rhd \\ (\Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m))) \end{array} \right) \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array} \right) \setminus TSet$$

$=$ [Property 3.12 L5]

$$\left(\begin{array}{c} \left(\begin{array}{c} (\Phi(A);\ (Terminate(k,n) \,|||\, Terminate(j,m))) \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array} \right) \\ \lhd\ b\ \rhd \\ \left(\begin{array}{c} (\Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m))) \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array} \right) \end{array} \right) \setminus TSet$$

$=$ [Property 3.14 L9]

$$\left(\begin{array}{c} \left(\begin{array}{c} (\Phi(A);\ (Terminate(k,n) \,|||\, Terminate(j,m))) \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array} \right) \setminus TSet \\ \lhd\ b\ \rhd \\ \left(\begin{array}{c} (\Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m))) \\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array} \right) \setminus TSet \end{array} \right)$$

$=$ [5.2.3]

$$\begin{array}{c} (\Phi(A) \textbf{ par } (Timers(k,n) \,|||\, Timers(j,m))) \\ \lhd\ b\ \rhd \\ (\Phi(B) \textbf{ par } (Timers(k,n) \,|||\, Timers(j,m))) \end{array}$$

$=$ [hipothises]

$A \lhd b \rhd B$ $\square$

Case: Sequential composition

$$\Phi(A;\ B)\ \mathbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m)) \equiv (A;\ B)$$

$\Phi(A;\ B)\ \mathbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m))$

$=$ [5.2.9]

$(\Phi(A);\ \Phi(B))\ \mathbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m))$

$=$ [5.2.3]

$$\left(\begin{array}{c}((\Phi(A);\ \Phi(B));\ (Terminate(k,n) \mid\mid\mid Terminate(j,m)))\\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!]\\ (Timers(k,n) \mid\mid\mid Timers(j,m))\end{array}\right) \setminus TSet$$

$=$ [The timers of A and B are disjoint]

$$\left(\begin{array}{c}\left(\begin{array}{c}(\Phi(A);\ (Terminate(k,n) \mid\mid\mid Terminate(j,m)))\\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!]\\ (Timers(k,n) \mid\mid\mid Timers(j,m))\end{array}\right) \setminus TSet\\ ;\\ \left(\begin{array}{c}(\Phi(B);\ (Terminate(k,n) \mid\mid\mid Terminate(j,m)))\\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!]\\ (Timers(k,n) \mid\mid\mid Timers(j,m))\end{array}\right) \setminus TSet\end{array}\right)$$

$=$ [5.2.3]

$$\left(\begin{array}{c}(\Phi(A)\ \mathbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m)))\\ ;\\ (\Phi(B)\ \mathbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m)))\end{array}\right)$$

$=$ [Hipothises]

$A;\ B$ □

Case: Internal choice

$$\Phi(A \sqcap B)\ \mathbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m)) \equiv (A \sqcap B)$$

$\Phi(A \sqcap B)\ \mathbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m))$

$=$ [5.2.10]

$(\Phi(A) \sqcap \Phi(B))\ \mathbf{par}\ (Timers(k,n) \mid\mid\mid Timers(j,m))$

$=$ [5.2.3]

$$\left(\begin{array}{c}((\Phi(A) \sqcap \Phi(B));\ (Terminate(k,n) \mid\mid\mid Terminate(j,m)))\\ [\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!]\\ (Timers(k,n) \mid\mid\mid Timers(j,m))\end{array}\right) \setminus TSet$$

$=$ [Property 3.9 L5]

$$
\left(\begin{array}{c}
\left(\begin{array}{c}
(\Phi(A);\ (Terminate(k,n) \ |||\ Terminate(j,m))) \\
\sqcap \\
(\Phi(B);\ (Terminate(k,n) \ |||\ Terminate(j,m)))
\end{array}\right) \\
[\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet
$$

$=$ [Property 3.5.9 L6]

$$
\left(\begin{array}{c}
\left(\begin{array}{c}
(\Phi(A);\ (Terminate(k,n) \ |||\ Terminate(j,m))) \\
[\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \\
\sqcap \\
\left(\begin{array}{c}
(\Phi(B);\ (Terminate(k,n) \ |||\ Terminate(j,m))) \\
[\![\{\} \mid \{\!|\ TSet\ |\!\} \mid \{\}]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right)
\end{array}\right) \setminus TSet
$$

$=$ [5.2.3]

$$
\left(\begin{array}{c}
(\Phi(A)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m))) \\
\sqcap \\
(\Phi(B)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m)))
\end{array}\right) \setminus TSet
$$

$=$ [Hipothesis]

$A \sqcap B$ $\square$

Case: External choice

$$\Phi(A \,\Box\, B)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m)) \equiv (A \,\Box\, B)$$

First lets consider the case that A and B have no timer events as their initial events.

$\Phi(A \,\Box\, B)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m))$

$=$ [Law 5.2.1]

$\Phi(A) \boxtimes \Phi(B)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m))$

$=$ [Hypotheses and Law 5.2.8]

$\Phi(A) \,\Box\, \Phi(B)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m))$

$=$ [5.2.3]

$$
\left(\begin{array}{l}
(\Phi(A) \,\Box\, \Phi(B));\ (Terminate(k,n) \ |||\ Terminate(j,m)) \\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet
$$

$=$ [Property 3.10 L6]

$$
\left(\begin{array}{l}
\left(\begin{array}{c}
(\Phi(A);\ (Terminate(k,n) \ |||\ Terminate(j,m))) \\
\Box \\
(\Phi(B);\ (Terminate(k,n) \ |||\ Terminate(j,m)))
\end{array}\right) \\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet
$$

$=$ [Hypotheses and property 3.12 L10]

$$
\left(\begin{array}{l}
\left(\begin{array}{l}
(\Phi(A);\ (Terminate(k,n) \ |||\ Terminate(j,m))) \\
[\![s_A \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
(\Phi(B);\ (Terminate(k,n) \ |||\ Terminate(j,m))) \\
[\![s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right)
\end{array}\right) \setminus TSet
$$

$=$ [Property hiding L13]

$$
\left(\begin{array}{l}
\left(\begin{array}{l}
(\Phi(A);\ (Terminate(k,n) \ |||\ Terminate(j,m))) \\
[\![s_A \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet \\
\Box \\
\left(\begin{array}{l}
(\Phi(B);\ (Terminate(k,n) \ |||\ Terminate(j,m))) \\
[\![s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet
\end{array}\right)
$$

$=$ [5.2.3]

$$
\left(\begin{array}{l}
(\Phi(A)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m)) \\
\Box \\
(\Phi(B)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right)
$$

$=$ [Hypotheses]

$A \Box B$ $\Box$

Next we consider the case that A starts with a time action $Waitn$ and B starts with any none timer event c.

$\Phi((Wait\ \ \mathsf{n};\ A) \Box (c \to B))\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m))$

$=$ [Law 5.2.1]

$\Phi(Wait\ \ \mathsf{n};\ A) \boxtimes \Phi(c \to B)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m))$

$=$ [5.2.9 and 5.2.7]

$(\Phi(Wait\ \ \mathsf{n});\ \Phi(A)) \boxtimes c \to \Phi(B)\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m))$

$=$ [5.2.4,5.2.7 and 5.2.1]

$((setup.k.d \to out.k.d \to Skip);\ \Phi(A)) \boxtimes (c \to \Phi(B))\ \mathbf{par}\ (Timers(k,n) \ |||\ Timers(j,m))$

$=$ [5.2.3]

$$
\left(\begin{array}{l}
((setup.k.d \to out.k.d \to Skip);\ \Phi(A)) \boxtimes (c \to \Phi(B)); \\
(Terminate(k,n) \ |||\ Terminate(j,m)) \\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet
$$

$=$ [5.2.4]

$$
\left(\begin{array}{l}
\quad setup.k.d \rightarrow (((out.k.d \rightarrow Skip);\ \Phi(A)) \boxtimes (c \rightarrow \Phi(B)));\\
\quad (Terminate(k,n) \ |||\ Terminate(j,m))\\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!]\\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet
$$

$=$ [5.2.5]

$$
\left(\begin{array}{l}
\quad setup.k.d \rightarrow \left(\begin{array}{l}(out.k.d \rightarrow (\Phi(A) \boxtimes (c \rightarrow \Phi(B))))\\ \Box\\ (c \rightarrow halt.k \rightarrow \Phi(B))\end{array}\right);\\
\quad (Terminate(k,n) \ |||\ Terminate(j,m))\\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!]\\
(Timers(k,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet
$$

$=$ [Property of $|||$ and indexing]

$$
\left(\begin{array}{l}
\quad setup.k.d \rightarrow \left(\begin{array}{l}(out.k.d \rightarrow (\Phi(A) \boxtimes (c \rightarrow \Phi(B))))\\ \Box\\ (c \rightarrow halt.k \rightarrow \Phi(B))\end{array}\right);\\
\quad (Terminate(k,n) \ |||\ Terminate(j,m))\\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!]\\
Timer(k,d) \ |||\ (Timers(k+1,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet
$$

$=$ [5.2.2]

$$
\left(\begin{array}{l}
\quad setup.k.d \rightarrow \left(\begin{array}{l}(out.k.d \rightarrow (\Phi(A) \boxtimes (c \rightarrow \Phi(B))))\\ \Box\\ (c \rightarrow halt.k \rightarrow \Phi(B))\end{array}\right);\\
\quad (Terminate(k,n) \ |||\ Terminate(j,m))\\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!]\\
\left(\begin{array}{l}
\quad \mu\, X \bullet \left(\begin{array}{l}setup.k?d \rightarrow \left(\begin{array}{l}(halt.k \rightarrow X)\\ \Box\ Wait\ \ \mathsf{d};\ \left(\begin{array}{l}(out.k.d \rightarrow X)\\ \Box\\ (terminate.k \rightarrow Skip)\end{array}\right)\\ \Box\ (terminate.k \rightarrow Skip)\end{array}\right)\\ \Box\ (terminate.k \rightarrow Skip)\end{array}\right)\\
|||\ (Timers(k+1,n) \ |||\ Timers(j,m))
\end{array}\right) \setminus TSet
\end{array}\right)
$$

$=$ [Step law and 5.2.2]

$$
\left(\begin{array}{l}
\quad setup.k.d \rightarrow \left(\begin{array}{l}(out.k.d \rightarrow (\Phi(A) \boxtimes (c \rightarrow \Phi(B))))\\ \Box\\ (c \rightarrow halt.k \rightarrow \Phi(B))\end{array}\right);\\
\quad (Terminate(k,n) \;|||\; Terminate(j,m))\\
[\![s_A \cup s_B \mid \{\!|\; TSet \;|\!\} \mid]\!]\\
\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}setup.k?d \rightarrow \left(\begin{array}{l}(halt.k \rightarrow Timer(k,d))\\ \Box\ Wait\ \ \mathsf{d};\ \left(\begin{array}{l}(out.k.d \rightarrow Timer(k,d))\\ \Box\\ (terminate.k \rightarrow Skip)\end{array}\right)\\ \Box\ (terminate.k \rightarrow Skip)\end{array}\right)\end{array}\right)\\ \quad \Box\ (terminate.k \rightarrow Skip)\end{array}\right)\\ |||\ (Timers(k+1,n) \;|||\; Timers(j,m))\end{array}\right) \setminus TSet
\end{array}\right)
$$

$=$ [Property 3.12 L11]

$$
\left(\begin{array}{l}
setup.k.d \rightarrow\\
\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}(out.k.d \rightarrow (\Phi(A) \boxtimes (c \rightarrow \Phi(B))))\\ \Box\\ (c \rightarrow halt.k \rightarrow \Phi(B))\end{array}\right);\\ \quad (Terminate(k,n) \;|||\; Terminate(j,m))\\ [\![s_A \cup s_B \mid \{\!|\; TSet \;|\!\} \mid]\!]\\ \left(\begin{array}{l}\left(\begin{array}{l}(halt.k \rightarrow Timer(k,d))\\ \Box\ Wait\ \ \mathsf{d};\ \left(\begin{array}{l}(out.k.d \rightarrow Timer(k,d))\\ \Box\\ (terminate.k \rightarrow Skip)\end{array}\right)\\ \Box\ (terminate.k \rightarrow Skip)\end{array}\right)\\ |||\ (Timers(k+1,n) \;|||\; Timers(j,m))\end{array}\right)\end{array}\right) \setminus TSet
\end{array}\right)
$$

$=$ [Property 3.12 L13 and L14]

$$
\left(\begin{array}{l}
setup.k.d \rightarrow \\
\left(\begin{array}{l}
c \rightarrow \\
\left(\begin{array}{l}
halt.k \rightarrow \Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m)) \\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(halt.k \rightarrow Timer(k,d)) \\
\Box\ Wait\ \ \mathsf{d};\ \left(\begin{array}{l}
(out.k.d \rightarrow Timer(k,d)) \\
\Box \\
(terminate.k \rightarrow Skip)
\end{array}\right) \\
\Box\ (terminate.k \rightarrow Skip)
\end{array}\right) \\
|||\ (Timers(k+1,n) \,|||\, Timers(j,m))
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\Box \\
Wait\ \ \mathsf{d};\ \left(\begin{array}{l}
\left(\begin{array}{l}
(out.k.d \rightarrow (\Phi(A) \boxtimes (c \rightarrow \Phi(B)))) \\
\Box \\
(c \rightarrow halt.k \rightarrow \Phi(B))
\end{array}\right); \\
(Terminate(k,n) \,|||\, Terminate(j,m)) \\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(out.k.d \rightarrow Timer(k,d)) \\
\Box \\
(terminate.k \rightarrow Skip)
\end{array}\right)
\end{array}\right) \\
|||\ (Timers(k+1,n) \,|||\, Timers(j,m))
\end{array}\right)
\end{array}\right) \setminus TSet
$$

$=$ [Property 3.12 L11]

$$
\left(\begin{array}{l}
setup.k.d \rightarrow \\
\left(\begin{array}{l}
c \rightarrow halt.k \rightarrow \\
\left(\begin{array}{l}
\Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m)) \\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
Timer(k,d) \,|||\, (Timers(k+1,n) \,|||\, Timers(j,m))
\end{array}\right)
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
Wait\ \ \mathsf{d};\ out.k.d \rightarrow \\
\left(\begin{array}{l}
\Phi(A) \boxtimes (c \rightarrow \Phi(B)); \\
(Terminate(k,n) \,|||\, Terminate(j,m)) \,[\![\, s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
Timer(k,d) \,|||\, (Timers(k+1,n) \,|||\, Timers(j,m))
\end{array}\right)
\end{array}\right)
\end{array}\right) \setminus TSet
$$

$=$ [Property of indexing and interleaving]

$$
\left(\begin{array}{l}
setup.k.d \rightarrow \\
\left(\begin{array}{l}
c \rightarrow halt.k \rightarrow \\
\left(\begin{array}{l}
\Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m)) \\
[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
(Timers(k,n) \,|||\, Timers(j,m))
\end{array}\right)
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
Wait\ \ \mathsf{d};\ out.k.d \rightarrow \\
\left(\begin{array}{l}
\Phi(A) \boxtimes (c \rightarrow \Phi(B)); \\
(Terminate(k,n) \,|||\, Terminate(j,m)) \,[\![\, s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\
(Timers(k,n) \,|||\, Timers(j,m))
\end{array}\right)
\end{array}\right)
\end{array}\right) \setminus TSet
$$

$=$ [Property 3.14 L5]

$$
\left(\begin{array}{l}
\left(\begin{array}{l} c \to halt.k \to \\ \left(\begin{array}{l} \Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array}\right) \end{array}\right) \\
\Box \\
\left(\begin{array}{l} Wait\ \ \mathsf{d};\ out.k.d \to \\ \left(\begin{array}{l} \Phi(A) \boxtimes (c \to \Phi(B)); \\ (Terminate(k,n) \,|||\, Terminate(j,m)) \,[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array}\right) \end{array}\right)
\end{array}\right) \setminus TSet
$$

$=$ [Property 3.14 L13]

$$
\left(\begin{array}{l} c \to halt.k \to \\ \left(\begin{array}{l} \Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array}\right) \end{array}\right) \setminus TSet
$$

$\Box$

$$
\left(\begin{array}{l} Wait\ \ \mathsf{d};\ out.k.d \to \\ \left(\begin{array}{l} \Phi(A) \boxtimes (c \to \Phi(B)); \\ (Terminate(k,n) \,|||\, Terminate(j,m)) \,[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array}\right) \end{array}\right) \setminus TSet
$$

$=$ [Property 3.14 L4,L12 and L6]

$$
c \to \left(\begin{array}{l} halt.k \to \\ \left(\begin{array}{l} \Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array}\right) \end{array}\right) \setminus TSet
$$

$\Box$

$$
Wait\ \ \mathsf{d};\ \left(\begin{array}{l} out.k.d \to \\ \left(\begin{array}{l} \Phi(A) \boxtimes (c \to \Phi(B)); \\ (Terminate(k,n) \,|||\, Terminate(j,m)) \,[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array}\right) \end{array}\right) \setminus TSet
$$

$=$ [Property 3.14 L5]

$$
c \to \left(\begin{array}{l} \Phi(B);\ (Terminate(k,n) \,|||\, Terminate(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array}\right) \setminus TSet
$$

$\Box$

$$
Wait\ \ \mathsf{d};\ \left(\begin{array}{l} \Phi(A) \boxtimes (c \to \Phi(B)); \\ (Terminate(k,n) \,|||\, Terminate(j,m)) \,[\![s_A \cup s_B \mid \{\!|\ TSet\ |\!\} \mid]\!] \\ (Timers(k,n) \,|||\, Timers(j,m)) \end{array}\right) \setminus TSet
$$

$=$ [5.2.3]

$$
\begin{array}{l}
c \to \Phi(B)\mathbf{par}(\mathit{Timers}(k,n) \;|||\; \mathit{Timers}(j,m)) \\
\Box \\
\mathit{Wait}\ \ \mathsf{d};\ \left(\begin{array}{l} \Phi(A) \boxtimes (c \to \Phi(B)); \\ (\mathit{Terminate}(k,n) \;|||\; \mathit{Terminate}(j,m)) \;[\![\, s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \;|||\; \mathit{Timers}(j,m)) \end{array} \right) \setminus \mathit{TSet}
\end{array}
$$

$=$ [Hypotheses]

$$
\begin{array}{l}
(c \to B) \\
\Box \\
\mathit{Wait}\ \ \mathsf{d};\ \left(\begin{array}{l} \Phi(A) \boxtimes (c \to \Phi(B)); \\ (\mathit{Terminate}(k,n) \;|||\; \mathit{Terminate}(j,m)) \;[\![\, s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \;|||\; \mathit{Timers}(j,m)) \end{array} \right) \setminus \mathit{TSet}
\end{array}
$$

$=$ [Law 5.2.8]

$$
\begin{array}{l}
(c \to B) \\
\Box \\
\mathit{Wait}\ \ \mathsf{d};\ \left(\begin{array}{l} \Phi(A) \;\Box\; (c \to \Phi(B)); \\ (\mathit{Terminate}(k,n) \;|||\; \mathit{Terminate}(j,m)) \;[\![\, s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \;|||\; \mathit{Timers}(j,m)) \end{array} \right) \setminus \mathit{TSet}
\end{array}
$$

$=$ [Property 3.10 L6]

$$
\begin{array}{l}
(c \to B) \\
\Box \\
\mathit{Wait}\ \ \mathsf{d};\ \left(\begin{array}{l} \left(\begin{array}{l} \Phi(A);\ (\mathit{Terminate}(k,n) \;|||\; \mathit{Terminate}(j,m)) \\ \Box \\ (c \to \Phi(B));\ (\mathit{Terminate}(k,n) \;|||\; \mathit{Terminate}(j,m)) \end{array} \right) \\ [\![s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \;|||\; \mathit{Timers}(j,m)) \end{array} \right) \setminus \mathit{TSet}
\end{array}
$$

$=$ [Initial events of A and event c not in TSet]

$$
\begin{array}{l}
(c \to B) \\
\Box \\
\mathit{Wait}\ \ \mathsf{d};\ \left(\begin{array}{l} \left(\begin{array}{l} \Phi(A);\ (\mathit{Terminate}(k,n) \;|||\; \mathit{Terminate}(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \;|||\; \mathit{Timers}(j,m)) \end{array} \right) \Box \\ \left(\begin{array}{l} (c \to \Phi(B));\ (\mathit{Terminate}(k,n) \;|||\; \mathit{Terminate}(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \;|||\; \mathit{Timers}(j,m)) \end{array} \right) \end{array} \right) \setminus \mathit{TSet}
\end{array}
$$

$=$ [Property 3.12 L14]

$$
\begin{array}{l}
(c \to B) \\
\Box \\
\mathit{Wait}\ \ \mathsf{d};\ \left(\begin{array}{l} \left(\begin{array}{l} \Phi(A);\ (\mathit{Terminate}(k,n) \ |||\ \mathit{Terminate}(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \ |||\ \mathit{Timers}(j,m)) \end{array} \right) \Box \\ c \to \left(\begin{array}{l} \Phi(B);\ (\mathit{Terminate}(k,n) \ |||\ \mathit{Terminate}(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \ |||\ \mathit{Timers}(j,m)) \end{array} \right) \end{array} \right) \setminus \mathit{TSet} \\
= \qquad \text{[Property 3.14 L14]} \\
(c \to B) \\
\Box \\
\mathit{Wait}\ \ \mathsf{d};\ \left(\begin{array}{l} \left(\begin{array}{l} \Phi(A);\ (\mathit{Terminate}(k,n) \ |||\ \mathit{Terminate}(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \ |||\ \mathit{Timers}(j,m)) \end{array} \right) \setminus \mathit{TSet} \ \Box \\ c \to \left(\begin{array}{l} \Phi(B);\ (\mathit{Terminate}(k,n) \ |||\ \mathit{Terminate}(j,m)) \\ [\![s_A \cup s_B \mid \{\!|\ \mathit{TSet}\ |\!\} \mid]\!] \\ (\mathit{Timers}(k,n) \ |||\ \mathit{Timers}(j,m)) \end{array} \right) \setminus \mathit{TSet} \end{array} \right) \\
= \qquad \text{[Hypotheses]} \\
(c \to B) \\
\Box \\
\mathit{Wait}\ \ \mathsf{d};\ (A \ \Box\ c \to B) \\
= \qquad \text{[Property 3.10 L8]} \\
(c \to B) \\
\Box \\
((\mathit{Wait}\ \ \mathsf{d};\ A) \ \Box\ (\mathit{Wait}\ \ \mathsf{d};\ (c \to B))) \\
= \qquad \text{[Property 3.10 L4]} \\
(\mathit{Wait}\ \ \mathsf{d};\ A) \\
\Box \\
((c \to B) \ \Box\ (\mathit{Wait}\ \ \mathsf{d};\ (c \to B))) \\
= \qquad \text{[Property 3.10 L11]} \\
(\mathit{Wait}\ \ \mathsf{d};\ A) \ \Box\ (c \to B) \qquad \Box
\end{array}
$$

We do not consider the case that both A and B start with time action *Waitn* because this can be reduced to any of the above cases using the properties of *Wait* and external choice, see Chapter 3, Property 3.10 L7.

Case: Parallel composition

$$\Phi(A \,[\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]\, B) \ \textbf{par}\ (\mathit{Timers}(k,n) \ |||\ \mathit{Timers}(j,m)) \equiv (A \,[\![s_A \mid \{\!|\ cs\ |\!\} \mid s_B]\!]\, B)$$

The proof for the parallel composition can be realized in the same way as external choice.