Towards a Time Model for *Circus*

Adnan Sherif and He Jifeng *

The United Nations University,
International Institute for Software Technology,
Casa Silva Mendes, Est. do Engenheiro Trigo No. 4,
P.O Box 3058, Macau
Tel.:+853 - 712.930, Fax: +853 - 712.940
ams@iist.unu.edu, hjf@iist.unu.edu

Abstract. In this work we propose a time model for *Circus*. The model is an extension to the model proposed by the unifying theories of programming and used by *Circus*. We take a subset of *Circus* and study its semantics in the new model. We create an abstraction function that maps the timed model to the original model. The main objective of this mapping is to create a relation between the two models. This allows the exploration of some properties of the timed semantics in the untimed model. We study a toy example to illustrate the use of this mapping.

1 Introduction

Real time systems have always been a strong candidate for formal development methods. This fact rises due to the complexity and, usually, critical nature of these systems. The development of formal specification languages, and the adaptation of existing languages with time expressing capacity, was and still is a challenging task. Many languages such as DC [1], and RTL [4] are based on temporal logic and are powerful for expressing timed functionalities. Timed CSP [8] is an extension to the well known CSP [9]. Reed and Roscoe have developed several semantic models for the language [8].

Lately the combination of different languages and techniques have been adapted to obtain formalisms that can be used in a wider range of applications. *Circus* is a combination of CSP and Z [12]; it includes specification statements found in Morgan's refinement calculus [7] and Dijkstra's language of guarded commands [2]. *Circus* has a well-defined syntax and a formal semantics [15,13] based on the unified theories of programming [3]. Case studies using the language are explored in [14] to show its power of expression. A development method for *Circus* using refinement is described in [10].

This paper aims to provide a model which is enriched with time in a conservative manner to the untimed *Circus* . We add some time operators to the language as well. A mapping between the two models is created with the objective of studying the properties of timed programs in the untimed model.

* On leave from East China Normal University, Shanghai. The work is partly supported by research grant 02104 of MoE P. R. of China

C. George and H. Miao (Eds.): ICFEM 2002, LNCS 2495, pp. 613–624, 2002.

To define this model we adopt a simple language (CT^*). $CT^*$is a subset of *Circus*. For simplicity, we only consider actions, guarded commands and assignment from the original language. The fact that we are using a subset of the original language has no effect over the model, as the other constructs of the language are abstractions and declarations that have little effect over the model.

In the next section we give an informal introduction to the syntax and semantics of the language CT^*. In Section 3, we present the semantic model and give the formal semantics of CT^*. A relation between the timed model and the original model is explored in Section 4. We study a simple example and explore some properties in Section 5, and conclude in Section 6 with a discussion on future work.

2 CT^*: Informal Description

A CT^*program is formed by actions, commands and channel communication events. Figure 1 presents the BNF description of the syntax of CT^*.

$$
\begin{array}{lll}
\mathsf{Action} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathit{Wait}\ \mathsf{t} \\
 & & \mid \mathsf{CommunicationAction} \mid \mathsf{b}\ \&\ \mathsf{Action} \\
 & & \mid \mathsf{Action};\ \mathsf{Action} \mid \mathsf{Action} \,\Box\, \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
 & & \mid \mathsf{Action} \,[\![\, \mathsf{CS} \,]\!]\, \mathsf{Action} \mid \mathsf{Action} \setminus \mathsf{CS} \mid \mathsf{Command} \\
 & & \mid \mathsf{Action} \overset{\mathsf{t}}{\triangleright} \mathsf{Action} \mid \mu\, \mathsf{N} \bullet \mathsf{Action} \\
\mathsf{Communication} & ::= & \mathsf{N}\ \mathsf{CParameter}^* \\
\mathsf{CParameter} & ::= & ?\,\mathsf{N} \mid !\,\mathsf{e} \mid .\,\mathsf{e} \\
\mathsf{Command} & ::= & \mathsf{N}^+ := \mathsf{e} \mid \mathsf{Action} \lhd b \rhd \mathsf{Action}
\end{array}
$$

Fig. 1. CT^* syntax

In the syntax above, e stands for any expression, t stands for a positive integer time expression, N any valid name, N^+ a list of names and CS stands for a set of channel names.

A CT^*program is formed from one single action. An action can be a basic action, or a combination of one or more actions.

Skip, is a basic action that terminates immediately. *Stop* represents an abnormal termination which simply puts a program in an ever waiting state. *Chaos* is the worst action, nothing can be said about its behavior.

The action (*Wait* t) will be held for an amount of time determined by the positive integer expression t before terminating normally. Guarded actions (b & Action) are proceeded by a predicate which has to be *true* for the action to take place; otherwise the guarded action cannot be executed and the resulting behavior is similar to the action *Stop*.

An internal choice ($\mathsf{Action} \sqcap \mathsf{Action}$) selects one of the two actions in a non-deterministic manner, whereas the external choice ($\mathsf{Action} \Box \mathsf{Action}$) waits for any of the two actions to interact with the environment. The first action that shows an interaction with the environment (either by synchronizing on an event or terminating) is the resulting action.

The sequential composition of two actions ($\mathsf{Action}; \mathsf{Action}$) will result in a new action that will behave as the first action followed immediately by the second action.

An action can be prefixed with a communication event (input or output) which will take place before the action starts. The action waits for the other actions that need to synchronize on the channel before the communication can take place. The parallel composition of two actions ($\mathsf{Action} \llbracket \mathsf{CS} \rrbracket \mathsf{Action}$) involves a set ($\mathsf{CS}$) containing the events they need to synchronize on. A hiding operation also takes a set of events (CS). The set is to be excluded from the resulting observation; hidden events can no longer be seen by other actions.

The timeout construct ($\mathsf{Action} \rhd^{\mathsf{t}} \mathsf{Action}$) takes a positive integer value as the length of the timeout. The timeout operator acts as a time guarded choice. If the first action performs an observable event or terminates before the specified time elapses, it is chosen. Otherwise, the first action will be suspended and the only possible observations are those produced by the second action.

Assignment is a command; it simply assigns a value to a variable 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 given value. The conditional command ($\mathsf{Action} \lhd b \rhd \mathsf{Action}$) associates two actions with a boolean expression b. If the expression evaluates to *true* then the first action is chosen, otherwise the second action is chosen.

3 The Semantic Model

The first question that had to be answered is: what model of time we would like to have, discrete time or continuous time? On one hand the second seems to be more appropriate for it is powerful to express time in both forms, and for the nature of time in the real world to be continuous. On the other hand, it cannot be implemented by a computer system. Unlike a continuous model, a discrete model is implementable, and therefore the untimed refinement rules can be extended in a more natural way. Following the main objective of making the model conservative we make a choice for the discrete model.

Similar approaches such as those in [6] and [5] use Extended Duration Calculus (EDC) to add continuous time to the language semantics. Both works, show clearly the elegance and powerful expression capacity of the EDC formulas. But 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 laborious task.

A reactive system behavior can be studied with two observations. The initial observation shows the state of the environment before the program starts, and

the second observations shows the state of the environment at the moment the program reaches a stable state. A stable state is either a termination state or a none termination state in which the program has no interaction with the environment [3]. The final observation registers the interaction of the program with the 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 approach we continue with the same pair of observations, at the initial and end of the program. But we enrich the observations on the interaction with environment, by adding time information. The interaction with the environment is recorded as a sequence of tuples, each element of the sequence denoting the observations over a single time unit. 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, ok' are boolean variables. When ok is true, it states that the program started and ok' indicates that the program is in a stable state.

$$ok, ok' : Boolean$$

$wait, wait'$ boolean variables. When $wait$ is true the program starts in an intermediate state. When $wait'$ is true the program has not terminated; when it is false, it indicates a final observation.

$$wait, wait' : Boolean$$

$state$, $state'$ A mapping from variable names to values. This mapping associates each user variable in the program to a value.

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

The dashed variable represents the state of the program variables at the final observation.

tr, tr' A sequence of observations on the program interaction with its environment. tr records the observations that occurred before the program starts, and tr' records the final observation. Each element of the sequence represents an observation over one time unit. Each observation element is composed of a tuple, where the first element of the tuple is the sequence of events that occurred during the time unit, and the second one is the associated set of refusals at the end of the same time unit.

$$tr, tr' : \text{seq}(\text{seq}\, Event \times \mathbb{P}\, Event)$$

The type *Event* represents all the possible events of a program. We also define a relation between two timed traces. We define a relation *Expands* as follows

$$Expands(tr, tr') \mathrel{\widehat{=}} (Front(tr) \preceq tr') \wedge (first(Last(tr)) \preceq first(tr'(\#\ tr)))$$

Given two timed traces, we state that the second expands the first if the initial part of the first timed trace is a subsequence of the second timed trace, and the untimed traces registered at the last time unit of the first timed trace is a subsequence of the traces registered at the same time in the second timed trace.

trace′ A sequence of events that occurred since the last observation. In this observation we are interested in recording only the events without time.

$$trace' : \operatorname{seq} Event$$
$$trace' = Flat(tr') - Flat(tr)$$

where

$$Flat : \operatorname{seq}(\operatorname{seq} Event \times \mathbb{P}\, Event) \to \operatorname{seq} Event$$
$$Flat(<>) = <>$$
$$Flat(< (el, ref) > \frown S) = el \frown Flat(S)$$

A single observation is given by the combination of the above variables. We will define our programs as predicates over the observation variables. We define a condition R that needs to be satisfied by all observations.

$$R(P) \mathrel{\widehat{=}} P \wedge Expands(tr, tr')$$

The condition states that for all valid observations the final values of the timed trace (tr') are always an expansion of the initial timed traces (tr). We use the term $[\![P]\!]_{time}$ to stand for the timed semantic of a program P

3.1 Basic Actions

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

$$[\![Skip]\!]_{time} \mathrel{\widehat{=}} (ok' \wedge \neg wait' \wedge tr' = tr \wedge state' = state) \tag{1}$$

On the other hand the semantics of the action *Stop* is given as a predicate that waits for ever. Notice that *Stop* permits time to pass, but it does not interact with the environment ($trace' =<>$).

$$[\![Stop]\!]_{time} \mathrel{\widehat{=}} (ok' \wedge wait' \wedge trace' =<>) \tag{2}$$

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.

$$[\![Chaos]\!]_{time} \mathrel{\widehat{=}} R(true) \tag{3}$$

The assignment assigns a value to a variable in the current state. If the variable does not exist it will be added, otherwise its value will be overwritten. The assignment operation is instantaneous and does not consume time.

$$[\![x := e]\!]_{time} \mathrel{\widehat{=}} (ok' \wedge \neg wait' \wedge tr' = tr \wedge state' = state \oplus \{x \mapsto e\}) \tag{4}$$

Wait: The only possible behavior for this action is to wait for the specified number of time units to pass before terminating immediately.

$$\begin{aligned}[\![Wait\ d]\!]_{time} \widehat{=} &((ok' \wedge wait' \wedge (\#tr' - \#tr) < d) \vee \\ &(ok' \wedge \neg wait' \wedge (\#tr' - \#tr) = d)) \wedge \\ &(trace' = <>)\end{aligned} \tag{5}$$

Communication: An action can engage in a communication if all the other actions involved in the same communication are ready to do so. We model this with the help of two predicates. $wait\_com(c)$ models the waiting state of an action to communicate on channel c. The only possible observation is that the communication channel cannot appear in the refusal set during the observation period. $term\_com(m)$ represents the act of the communication itself. It states clearly that the communication does not take any time ($\#tr' = \#tr$), but the event appears in the traces of the observation.

$$wait\_com(c) \widehat{=} ok' \wedge wait' \wedge possible(tr, tr', c) \wedge trace' = <>$$
$$term\_com(m) \widehat{=} ok' \wedge \neg wait' \wedge trace' = < m > \wedge \#tr' = \#tr$$

Where, $possible(tr, tr', c)$ returns true if the channel c is not contained in the refusal set of all the observations recorded from tr to tr'. The following is a definition of the function

$$possible(tr, tr', c) \widehat{=} \forall\, i : \#tr..\#tr' \bullet c \notin second(tr'(i))$$

The semantics of the output communication is given bellow. The definition describes two states for the communication semantics. The first is that the communication observation is in a waiting state. The second describes the case of a communication waiting for a time period before terminating with the event registered in the traces.

$$\begin{aligned}[\![c!e]\!]_{time} \widehat{=} &wait\_com(c) \vee \\ &(wait\_com(c) \circ (term\_com(c.e) \wedge state' = state))\end{aligned} \tag{6}$$

We use ($\circ$) to stand for observation concatenation. It is defined as follows

$$A \circ B \widehat{=} \exists\, \bar{o} \bullet A(\bar{v}, \bar{o}) \wedge B(\bar{o}, \bar{v}')$$

Where $\bar{v}, \bar{v}'$ represent the vector of observation variables. We use the term $A(\bar{v}, \bar{v}')$ to denote the predicate that satisfies the vector of observation variables $\bar{v}$ as input and the vector $\bar{v}'$ as output. The concatenation states that there exists a vector of observation variables $\bar{o}$ such that this vector is the output of the first predicate and satisfies the input of the second predicate. The concatenation operator satisfies the following properties

$$(A \vee B) \circ C = (A \circ C) \vee (B \circ C)$$
$$(A \circ B) \circ C = A \circ (B \circ C)$$

We can define the input operation in a similar manner. The main difference is that when the communication takes place, the value transmitted on the channel is assigned to the variable used in the input command.

$$\begin{aligned}[\![c?x]\!]_{time} \widehat{=} &wait\_com(c) \vee \\ &(wait\_com(c) \circ (term\_com(c.e) \wedge state' = state \oplus [x \mapsto e]))\end{aligned} \tag{7}$$

The semantics of the communication prefix can be given with the help of the sequential composition. The action *comm* is either an input or output operation.

$$[\![commAction]\!]_{time} \mathrel{\widehat{=}} [\![comm;\ Action]\!]_{time} \tag{8}$$

Conditional Choice: We use the conditional choice operator exactly as defined in the unifying theories of programming [3].

3.2 Sequential Composition

The sequential composition has three possible states. The first shows that if the first action diverges then so does the sequential composition. The second state shows that the initial action is in a waiting state and therefore the following action can not start. The alternative behavior would be for the first action to terminate and the second to start immediately after.

$$\begin{aligned}[\![A;\ B]\!]_{time} \mathrel{\widehat{=}} &([\![A]\!]_{time}[false/ok']) \vee \\ &([\![A]\!]_{time} \wedge wait') \vee \\ &(([\![A]\!]_{time}[true, false/ok', wait']) \circ [\![B]\!]_{time})\end{aligned} \tag{9}$$

3.3 Guarded Action

A guarded action has a predicate p which needs to be satisfied before the action can take place. If the predicate is *false* the only possible behavior of the resulting action is to wait for ever. But if the predicate evaluates to *true* then the result will be any possible behavior of the guarded action.

$$[\![\mathsf{p}\ \&\ A]\!]_{time} \mathrel{\widehat{=}} ([\![A]\!]_{time} \lhd p \rhd [\![Stop]\!]_{time}) \tag{10}$$

3.4 External Choice

The external choice between two actions is determined by the environment. The composed system will behave as either one of the two actions which ever reacts first to the environment. This can be expressed as two possible behaviors, either the system is in a waiting state and only internal behavior can take place or the system reacts to the environment after waiting for an external event which satisfies either one of the component actions or both, and in this case the choice is non-deterministic.

$$\begin{aligned}[\![A \Box B]\!]_{time} \mathrel{\widehat{=}} &([\![A]\!]_{time} \wedge [\![B]\!]_{time} \wedge wait' \wedge trace' = <>) \vee \\ &(([\![A]\!]_{time} \wedge [\![B]\!]_{time} \wedge ok' \wedge wait' \wedge state' = state \wedge \\ &trace' = <>) \vee [\![Skip]\!]_{time}) \circ (([\![A]\!]_{time} \vee [\![B]\!]_{time}) \wedge \\ &(\neg wait' \vee (\neg(tr \preceq tr') \wedge trace' \neq <>)))\end{aligned} \tag{11}$$

The internal choice is specified just as in [3].

3.5 Recursion

To define recursion we need to define an ordering operator. An action A is as good as action B in the sense that it will meet all the operations and satisfy all the specifications satisfied by B. This relation is denoted by $A \sqsupseteq B$.

$$A \sqsupseteq B \mathrel{\widehat{=}} [[\![A]\!]_{time} \Rightarrow [\![B]\!]_{time}] \tag{12}$$

An action A is equal to an action B if

$$A = B \mathrel{\widehat{=}} A \sqsupseteq B \wedge B \sqsupseteq A \tag{13}$$

We notice that the set of observations in our model form a complete lattice with respect to the relation $\sqsupseteq$, having $[\![Chaos]\!]_{time}$ as its bottom element, $\sqcap$ as the *greatest lower bound*. So we can define the semantics of recursion as the *weakest fixed point* [3].

$$\mu\, X \bullet F(X) \mathrel{\widehat{=}} \sqcap \{X \mid X \Rightarrow F(X)\} \tag{14}$$

3.6 Timeout

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

$$[\![\mathsf{A} \overset{d}{\rhd} \mathsf{B}]\!]_{time} \mathrel{\widehat{=}} [\![(A \,\Box\, (Wait\ d;\ intB)) \setminus \{int\}]\!]_{time} \tag{15}$$

The event *int* is taken to be an event that is not used by A and B or another event can be used. The main objective of adding this event is to trigger the external choice and force it to select the second option. It can only do the first option if the action A engages in a communication or terminates before the wait period d elapses. The event *int* is hidden from the rest of the environment.

For details of the semantic of the other operators of the language and the properties please refer to [11].

4 Linking *Circus* Models

Our proposed model is different from other approaches, we are interested in adding time information to the semantics of the language, but we will also like to preserve the untimed semantics of our programs in the time model. To show the relation between the two models we create a function L that given a set of timed observations related to a CT^* program P, the function returns the equivalent observation in the original model without time information. This function is defined as follows

$$\begin{aligned} L([\![P]\!]_{time}) \mathrel{\widehat{=}} \exists\, obs \bullet [\![P]\!]_{time} \wedge trace' = Flat(tr') - Flat(tr) \wedge \\ ref' = second(Last(tr')) \end{aligned} \tag{16}$$

The function L maps the timed semantics of a program P to the untimed semantics of the *Circus* program. This is done by applying the *Flat* function to

the timed traces to obtain the original model traces. A projection on the second element of the last entry in tr', results in the refusal set of the original model.

We also introduce a function R as an inverse function for L. This function takes as input a set of untimed observations and adds arbitrary time information.

$$R([\![P]\!]) \mathrel{\widehat=} \bigvee \{ [\![Q]\!]_{time} \mid L([\![Q]\!]_{time}) \sqsupseteq [\![P]\!] \} \tag{17}$$

We observe that the functions L and R form a *Galois connection* [3]. If we apply the mapping R to the result of applying the mapping L to a timed specification S, we will get a larger set of observation. The time information contained in S is lost and the result is a weaker specification.

$$S \sqsupseteq R(L(S))$$

This permits us to explore some properties of the timed language. Let us consider the following theorem.

Theorem 1. *A specification S is time insensitive if it satisfies the following equation*

$$R(L(S)) = S \tag{18}$$

The above theorem states that by applying the conjunction of the mapping functions to a specification S we obtain the same specification. Then the time information in the original specification is irrelevant to the behavior of the system. An example of such case is the action *Skip*. By applying the function L to the semantics of the constructs of CT^* we can obtain the equivalent *Circus* semantics.

$$L([\![x := e]\!]_{time}) = [\![x := e]\!] \tag{19}$$
$$L([\![Skip]\!]_{time}) = [\![Skip]\!] \tag{20}$$
$$L([\![Stop]\!]_{time}) = [\![Stop]\!] \tag{21}$$
$$L([\![Chaos]\!]_{time}) = [\![Chaos]\!] \tag{22}$$
$$L([\![comm]\!]_{time}) = [\![comm]\!] \tag{23}$$
$$L([\![Wait\ d]\!]_{time}) = [\![Stop]\!] \lor [\![Skip]\!] \tag{24}$$
$$L([\![A \lhd b \rhd B]\!]_{time}) \sqsupseteq L([\![A]\!]_{time}) \lhd b \rhd L([\![B]\!]_{time}) \tag{25}$$
$$L([\![p\&\ A]\!]_{time}) \sqsupseteq p\&\ L([\![A]\!]_{time}) \tag{26}$$
$$L([\![A \sqcap B]\!]_{time}) \sqsupseteq L([\![A]\!]_{time}) \sqcap L([\![B]\!]_{time}) \tag{27}$$
$$L([\![A \mathbin{\Box} B]\!]_{time}) \sqsupseteq L([\![A]\!]_{time}) \mathbin{\Box} L([\![B]\!]_{time}) \tag{28}$$
$$L([\![A;\ B]\!]_{time}) \sqsupseteq L([\![A]\!]_{time});\ L([\![B]\!]_{time}) \tag{29}$$
$$L([\![A \mid\mid cs \mid\mid B]\!]_{time}) \sqsupseteq L([\![A]\!]_{time}) \mid\mid cs \mid\mid L([\![B]\!]_{time}) \tag{30}$$
$$L([\![A \backslash cs]\!]_{time}) \sqsupseteq L([\![A]\!]_{time}) \backslash cs \tag{31}$$
$$L([\![\mu\, X \bullet A(X)]\!]_{time}) \sqsupseteq \mu\, X' \bullet L([\![A]\!]_{time})(X') \tag{32}$$
$$L([\![A \stackrel{d}{\rhd} B]\!]_{time}) = L([\![(A \mathbin{\Box} (Wait\ d;\ intB)) \backslash \{int\}]\!]_{time}) \tag{33}$$

The relation between the models will permit us to explore properties of programs 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 programs with time information some properties can still be investigated in the untimed model. In the next section we explore this topic in more detail with the aid of an example.

5 Example

A one place buffer, takes as input a value from the input channel, stores this value in a local internal variable and then offers to communicate the same value on the output channel. The buffer has a main safety requirement. The buffer can not lose data, i.e. it should not allow the data to be over written by new input before an output is issued. The simple buffer is given in the following CT^* actions

$$Buffer \mathrel{\widehat{=}} in?xBuffer_{<x>}$$
$$Buffer_{<x>} \mathrel{\widehat{=}} out!xBuffer$$

To add some timing constraints we state that the input and output operations have a duration of 3 time units. We will add a waiting state at the end of each operation. The communication does not consume time. We can change the buffer example as follows

$$TBuffer \mathrel{\widehat{=}} in?xWait\ 3;\ TBuffer_{<x>}$$
$$TBuffer_{<x>} \mathrel{\widehat{=}} out!xWait\ 3;\ TBuffer$$

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*.

$$S(trace) \mathrel{\widehat{=}} Front(trace) \upharpoonright \{in\} \leq trace \upharpoonright \{out\} \wedge$$
$$trace \upharpoonright \{out\} \leq trace \upharpoonright \{in\}$$

We would like to check if our timed buffer still meets the safety requirement. Because the specification uses the untimed 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 satisfies the specification. We define a relation sat_T to state that a timed program P satisfies an untimed specification S that only uses the traces.

$$P\ sat_T\ S \mathrel{\widehat{=}} L(P) \Rightarrow S$$

We need to prove that

$$L([\![TBuffer]\!]_{time}) \Rightarrow S$$

Where

$$L([\![TBuffer]\!]_{time})$$
$$\sqsupseteq$$
$$[\![in?x(Stop \sqcap Skip);\ out?x(Stop \sqcap Skip)]\!];\ L(TBuffer)$$

Because the program is a guarded recursive action. We can use the following

$$F(X) \sqsupseteq S \text{ iff } F(S) \sqsupseteq S$$

and

$$(\llbracket in?x(Stop \sqcap Skip);\ out?x(Stop \sqcap Skip) \rrbracket;\ S) \sqsupseteq S$$
$$implies$$
$$L(TBuffer) \sqsupseteq S$$

From the semantic definition of communication and the definition of sequential composition

$$\begin{aligned} \llbracket in?x(Stop \sqcap Skip);\ out?x(Stop \sqcap Skip) \rrbracket;\ S \Rightarrow & (trace = <>) \vee \\ & (trace = < in >) \vee \\ & (trace < in\ out > \frown t'' \wedge \\ & S(t'') \end{aligned}$$

Therefore

$$L(\llbracket TBuffer \rrbracket_{time})\ sat_T\ S$$

□

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 the deadlock state into the program. Therefore deadlock free property can not be explored with this abstraction function. A more suitable abstraction would be, to substitute $Wait\ d$ with a *Skip*. We are currently exploring this type of abstractions.

6 Conclusions

In this paper we presented a model for adding time to *Circus* . The new model is an extension to the original untimed model. We also show that the semantics of a program that has no time information is the same in both models. We have created a mapping from one model to the other, this mapping forms a Galois connection between the two models. Therefore we explored the possibility of reasoning on some properties of the system in one model, given a program semantics in the other model.

As future work, we are studying other mappings and the possible properties to be explored by these mappings. Therefore different mappings to the untimed model can be used according to the type of property to be explored. *Circus* has a semantic model implemented in Z, we are also interested in extending this semantics model in Z with time, we would like to use a tool such as ZEVES to study the model. We are also interested in rewriting the case studies of the steam boiler presented by Jim Woodcock and Ana Cavalcanti in [14]. The validation of the refinement laws introduced in [10] in the timed model is in our scope as well.

The model we presented uses discrete time. Requirements for control systems are usually expressed in continuous time. A mapping between the continuous time models and our implementation model is an interesting aspect to be explored. This mapping can be used to validate the timing requirements.

References

1. Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A Calculus of Duration. *Information Processing Letters.*, 40:269–276, 1991.
2. E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8), 1975.
3. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming.* Prentice-Hall Series in Computer Science, 1998.
4. 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.
5. He Jifeng and Victor Verbovskiy. Integrating CSP and DC. R 248, International Institute for Software Technology, The United Nation University, P.O. Box 3058, Macau, January 2002.
6. Li Li and He 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.
7. C. Morgan. *Programming from Specifications.* Series in Computer Science. Prentice-Hall International, 2nd edition, 1994.
8. 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.
9. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall International, 1998.
10. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in Circus. To appear in proceedings of FME2002, 2002.
11. Adnan Sherif and He Jifeng. Towards a Time Model for Circus. R 257, International Institute for Software Technology, The United Nation University, P.O. Box 3058, Macau, July 2002.
12. M. Spivey. *The Z Notation.* Prentice-Hall International, 2nd edition, 1992.
13. J. C. P. Woodcock and A. L. C. Cavalcanti. Circus: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, July 2001.
14. J. C. P. Woodcock and A. L. C. Cavalcanti. The steam boiler in a unified theory of Z and CSP. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, 2001.
15. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of Circus - a Concurrent Language for Refinement. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*. Springer, January 2002.