Extending Synchronous Languages for Generating Abstract Real-Time Models

G. Logothetis and K. Schneider
University of Karlsruhe
Institute for Computer Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid)
P.O. Box 6980, 76128 Karlsruhe, Germany
email: {logo,schneide}@informatik.uni-karlsruhe.de
http://goethe.ira.uka.de/fmg

Abstract

We present an extension of synchronous programming languages that can be used to declare program locations irrelevant for verification. An efficient algorithm is proposed to generate from the output of the usual compilation an abstract real-time model by ignoring the irrelevant states, while retaining the quantitative information. Our technique directly generates a single real-time transition system, thus overcoming the known problem of composing several real-time models. A major application of this approach is the verification of real-time properties by symbolic model checking.

1. Introduction

Designing real-time systems is a relatively error-prone task, especially when the systems consist of several processes, which is usually the case. Decreasing time-to-market and the overall design costs make it necessary to check as early as possible in the design work flow whether the desired specifications are met.

For real-time systems, the task is to verify that certain actions are executed within some strict deadlines or that they will start only after some point of time. For this purpose, several approaches to the verification of real-time systems have been developed [1, 18, 6, 3, 11, 23, 20] that are based on different formalisms for describing finite state transition systems endowed with some notion of time.

In general, all verification tools that automatically traverse finite state spaces suffer from the enormous number of states. To overcome this problem, methods to abstract from irrelevant details have been developed [8, 22], which are closely related to abstract interpretation [10] of programs. Such abstractions are used for the proof of qualitative temporal properties, like safety properties meaning that a property always holds, or liveness properties meaning that a property will hold at least once.

First approaches to abstraction techniques for the verification of real-time properties have been developed in [30, 12, 20]. In [20], a real-time extension of CTL was introduced, that allows abstractions without loss of quantitative information. The key to this approach is to interpret timed transitions in such a way that only information about the source and target state is given. In particular, no information is given about the system's state during a transition (see definition 1).

Hence, many approaches to verify real-time properties of formal real-time models already exist. However, far less is known about translating programs to real-time models necessary for later verification. Tools like [7, 3, 11, 23] all read special formats that describe the formal models more or less directly. For an application of these tools in an industrial design flow, it is therefore important to develop appropriate front-ends that compile system descriptions of early design phases to real-time models which can then be formally verified. Of course, if the complete realization is not yet known (in early design phases), one can not argue about physical time, since this depends on the hardware chosen for the realization.

Nevertheless, it is possible to reason about *time at a logical level* as done by synchronous languages like Esterel [4, 15]. The basic paradigm of these languages is the *perfect synchrony*, which follows from the fact that most of the statements are executed as 'microsteps' in zero time. Consumption of time must be explicitly programmed with special statements like Esterel's **pause** statement that consumes a logical unit of time. Nowadays, synchronous languages are used in many industrial applications [15, 16, 29], where often real-time embedded systems have to be implemented. The semantics of synchronous languages lends itself well for formal verification, and there already exist tools [17, 19, 25, 14] for the verification of qualitative temporal properties.

In this paper, we introduce an extension of Esterel-like languages together with its translation to abstract real-time models (in the sense of [20]). Our extension allows the

programmer to declare irrelevant program locations. After the usual compilation of the program into a transition system, certain states are thus to be ignored. For this purpose, we present an efficient algorithm that uses symbolic[1] techniques to generate a *timed Kripke structure (TKS)* [20, 21] as a real-time model from the output of the compiler. The algorithms presented in [20, 21] are then used to check quantitative temporal properties of the generated TKS.

Our goal is to show how abstractions can be incorporated in synchronous programs to obtain abstract real-time models that retain the quantitative temporal information. In particular, the programmer can generate abstract real-time models without having the need of special knowledge about verification techniques. A well-known problem of all approaches based on abstraction techniques is that the chosen abstraction might be too coarse. In this case, our technique is able to detect the problematic program locations.

It is important to note that our method allows the generation of real-time models without having the need of parallel composition, which is one of the most important drawbacks of other approaches: Most verification procedures based on timed automata require the construction of a so-called region graph to reduce the infinite state space of timed automata to a finite state problem. However, the construction of the region graph is very expensive [2], and becomes more complex for timed automata that are obtained by parallel composition. Another approach to real-time temporal model checking which was introduced in [23] requires to model the system as a parallel composition of several timed Kripke structures. Again, the problem is thereby that the composition of these models is a complex operation [23].

There is not much related work. In [28], an extension of Esterel has been presented that focuses on the runtime verification of the perfect synchrony. However, it is assumed that the compiler preserves the ordering of the microstep statements, and therefore the approach is restricted to special compilation techniques like those proposed in [13]. A more powerful approach has recently been presented in [5]. There, Esterel programs are endowed with pragmas that contain the quantitative temporal information. This has no effect on code generation, but allows the generation of appropriate models (timed automata [1]) for verification of temporal properties. In contrast to our approach, [5] does not support abstractions. The major difference to our approach is however that [5] requires a low-level worst-case execution time (WCET) analysis in advance to obtain the real-time constraints. In contrast, the real-time constraints of our models are obtained by combining transitions while removing irrelevant states. Hence, our approach only refers to the system description in an early design phase, while [5] additionally needs architecture-dependent runtime data, and therefore [5] can only be applied in late design phases.

[1]The notion 'symbolic' is used here in the sense of 'symbolic model checking' which means that we represent transition relations and state spaces implicitly by means of propositional formulas.

The outline of the paper is as follows: In the next section, we explain the basics of synchronous languages with our Esterel variant Quartz [24, 25, 26, 27]. Using our Quartz compiler, we can generate C-programs, circuit netlists, and finite-state transition systems that are used for verification. In section 3, we explain the basics of timed Kripke structures as introduced in [20, 21]. Section 4 contains the main contribution of the paper: *By extending the language* Quartz*, it is possible to declare irrelevant program locations. After the usual compilation of the program into a transition system, an efficient algorithm is proposed to generate an abstract real-time model by ignoring the irrelevant states, while retaining the quantitative information. Our technique directly generates a single real-time transition system, thus overcoming the known problem of composing several real-time models. A major application of this approach is the verification of real-time properties [20, 21].* It is important to note that all algorithms presented here use symbolic techniques to efficiently manipulate large finite state transition systems.

2. The Synchronous Language Quartz

Quartz [25, 26, 24, 27] is a variant of Esterel [4, 15] that differs from Esterel in some minor points. The semantics of Quartz has been defined in [25] and a hardware synthesis for compilation has been presented in [24]. An extension of the latter including schizophrenia problems has been given in [27]. A complete reference is given in [26]. In the following, we briefly describe the basics of Quartz and Esterel; for more details on Quartz, the reader is referred to [25, 26, 24, 27], for more details on Esterel, the reader should consult the Esterel primer [4], which is an excellent introduction to synchronous programming.

In synchronous languages, time is modeled by the natural numbers $\mathbb{N}$, so that the semantics of an expression is a function of type $\mathbb{N} \to \alpha$ for some type α. Quartz distinguishes between two kinds of variables, namely *event variables* and *state variables*. The semantics of an event variable is a function of type $\mathbb{N} \to \mathbb{B}$, while the semantics of a state variable may have the more general type $\mathbb{N} \to \alpha$. The main difference is however the data flow: the value of a state variable y is 'sticky', i.e. if no data operation has been applied to y, then its value does not change. On the other hand, the value of an event variable x is not stored: at the next step, the value of x would be reset to 0 (we denote Boolean values as 1 and 0), if it is not explicitly made 1 at the considered point of time. Hence, the value of an event variable x is 1 at a point of time if and only if there is at least one thread that emits x at this point of time.

Event variables are made present with the **emit** state-

ment, while state variables are manipulated with assignments. Of course, any event or state variable may also be an input variable, so that their values are determined by the environment only. Emissions and assignments are all data manipulating statements. The execution of these statements, as well as the execution of most other statements does not consume time (in the programmer's view). A complete list of all basic statements of Quartz is given below, where S, S_1, and S_2 are also basic statements of Quartz, ℓ is a location variable, x is an event variable, y is a state variable, and σ is a Boolean expression:

- **nothing** (empty statement)
- **emit** x and **emit delayed** x (emissions)
- $y := \tau$ and $y :=$ **delayed** τ (assignments)
- ℓ : **pause** (consumption of time)
- **if** σ **then** S_1 **else** S_2 **end** (conditional)
- $S_1; S_2$ (sequential composition)
- $S_1 \parallel S_2$ (synchronous parallel composition)
- $S_1 \parallel\!\!\parallel S_2$ (asynchronous parallel composition)
- **choose** $S_1 \,[]\, S_2$ **end** (nondeterministic choice)
- **do** S **while** σ (iteration)
- **suspend** S **when** σ (suspension)
- **weak suspend** S **when** σ (weak suspension)
- **abort** S **when** σ (abortion)
- **weak abort** S **when** σ (weak abortion)
- **local** x **in** S **end** (local event variable)
- **local** $y : \alpha$ **in** S **end** (local state variable)
- **now** σ (instantaneous assertion)
- **during** S **holds** σ (invariant assertion)

In general, a statement S may be started at a certain point of time t_1, and may terminate at time $t_2 \geq t_1$, but it may also never terminate. If S immediately terminates when it is started ($t_2 = t_1$), it is called *instantaneous*, otherwise we say that the execution of S takes time, or simply that S *consumes time*. Whether a statement is instantaneous or not may depend on input or local variables. There is only one basic statement that consumes time, namely the **pause** statement. In other words, the **pause** statements were the only statements where the control flow may rest. For this reason, we endow **pause** statements with unique location variables ℓ. These labels are used in [25, 26, 24, 27] as state variables to encode the control flow automaton.

A detailed explanation of the semantics of Quartz is given in [25, 26, 27]. The control flow of a statement S has been defined by the control flow predicates [25, 26] $\mathsf{in}(S)$, $\mathsf{inst}(S)$, $\mathsf{enter}(S)$, $\mathsf{term}(S)$, and $\mathsf{move}(S)$, and the data flow of S has been defined by the set of guarded commands $\mathsf{gcmd}(\varphi, S)$:

$\mathsf{in}(S)$ is the disjunction of the **pause** labels occurring in S. Therefore, $\mathsf{in}(S)$ holds at some point of time iff at this point of time, the control flow is at some location inside S.

$\mathsf{inst}(S)$ holds iff the control flow can not stay in S when S would now be started. This means that the execution of S would be instantaneous at this point of time.

$\mathsf{enter}(S)$ describes where the control flow will be at the next point of time, when S would now be started.

$\mathsf{term}(S)$ describes all conditions where the control flow is currently somewhere inside S and wants to leave S. Note however, that the control flow might still be in S at the next point of time since S may be entered at the same time, for example, by a surrounding loop statement.

$\mathsf{move}(S)$ describes all internal moves, i.e., all possible transitions from somewhere inside S to another location inside S.

$\mathsf{gcmd}(\varphi, S)$ is a set of pairs of the form $(\gamma, \mathcal{C})$, where $\mathcal{C}$ is a data manipulating statement, i.e., either an emission or an assignment. The meaning of $(\gamma, \mathcal{C})$ is that $\mathcal{C}$ is immediately executed whenever the guard γ holds.

Using the above control flow predicates, one can define a finite-state transition system that defines the control flow of a statement [25, 26]. The data flow is determined by the guarded commands $\mathsf{gcmd}(\varphi, S)$ that appear as conditional emissions and assignments on the transitions of the control flow transition system. In case that only finite data types were used, it is possible to translate a program to a classical finite state (Mealy) automaton.

Based on the presented basic statements, one can define a couple of several macro statements whose semantics is then simply given by the macro expansion. The most popular ones (most of them are used by the Esterel language) are the following:

- **while** σ **do** S **end** $:\equiv$ $\left(\begin{array}{l}\textbf{if } \sigma \textbf{ then}\\ \quad \textbf{do } S \textbf{ while } \sigma\\ \textbf{else nothing end}\end{array}\right)$
- ℓ : **halt** $:\equiv$ **do** ℓ : **pause while** 1
- **loop** S **end** $:\equiv$ **while** 1 **do** S **end**
- ℓ : **loop** S **each** σ
  $:\equiv$ **loop abort** $S; \ell$: **halt when** σ **end**
- ℓ_0 : **every** σ ℓ_1 : **do** S **end**
  $:\equiv \ell_0$: **await** $\sigma; \ell_1$: **loop** S **each** σ
- ℓ : **sustain** $x :\equiv$ **do emit** $x; \ell$: **pause while** 1
- ℓ : **await** $\sigma :\equiv$ **do** ℓ : **pause while** $\neg\sigma$
  $:\equiv$ **abort** ℓ : **halt when** σ
- ℓ : **await immediate** σ
  $:\equiv$ **while** $\neg\sigma$ **do** ℓ : **pause end**
  $:\equiv$ **abort** ℓ : **halt when immediate** σ

3. Timed Kripke Structures

We consider systems modeled as timed Kripke structures[2] over some set of variables $\mathcal{V}$. These timed Kripke structures are formally defined as follows:

Definition 1 (Timed Kripke Structures (TKS)) *A timed Kripke structure over the finite set of variables $\mathcal{V}$ is a tuple $(\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, such that $\mathcal{S}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, and $\mathcal{R} \subseteq \mathcal{S} \times \mathbb{N} \times \mathcal{S}$ is a finite set of transitions. For any state $s \in \mathcal{S}$, the set $\mathcal{L}(s) \subseteq \mathcal{V}$ is the set of variables that hold on s. We furthermore demand that for any $(s, t, s') \in \mathcal{R}$, we have $t > 0$ and that for any $s \in \mathcal{S}$, there must be a $t \in \mathbb{N}$ and a $s' \in \mathcal{S}$ such that $(s, t, s') \in \mathcal{R}$ holds.*

Timed Kripke structures may be pictorially drawn as given in Figure 1, where initial states are drawn with double lines. It is possible to consider certain infinite sets of transitions; we will see this in more detail in section 4.3. Roughly speaking, we could allow labels with linear constraints, as e.g. $\{2n + 3m + 5 \mid n, m \in \mathbb{N}\}$ or $\{n \in \mathbb{N} \mid n > 10\}$. The reason is that these labeled transitions can be replaced by finitely many states including some cycles. On the other hand, labeling transitions with intervals $[a, b]$ of time is not an extension of the model: It is easily seen that our TKSs subsume these models, since we can add for any $t \in [a, b]$ a new transition between the considered two states.

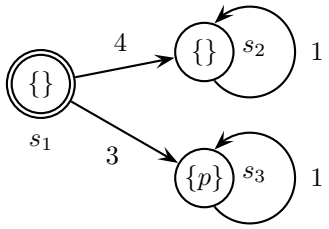

Figure 1. A Timed Kripke Structure

It is crucial to understand what is modeled by a TKS: A transition from state s_1 to state s_2 with label $t \in \mathbb{N}$ means that at any time t_0, where we are in state s_1, we can perform an atomic action that requires t units of time. The action terminates at time $t_0 + t$, where we are in state s_2. *There is no information about the intermediate points of time* $\{k \in \mathbb{N} \mid t_0 < k < t_0 + t\}$.

Traditional Kripke structures without labels on their transitions are special cases of TKSs where all labels are 1. Clearly, these labels can then be omitted. In the following, we call such special cases of TKSs unit-delay structures (UDSs).

[2]Other authors use different names for timed transition systems like timed transitions graphs [6] or timed temporal structures in [23]. Following the CTL notations, we prefer the name timed Kripke structures.

4. Compiling Quartz Programs to TKSs

We now present the main contribution of the paper: In the first section, we present our extension to the language Quartz, which makes it possible to declare irrelevant program locations. After the usual compilation of the program $\mathcal{P}$ into a UDS $\mathcal{K}_\mathcal{U}$, an efficient algorithm is proposed in section 4.2 to generate a TKS $\mathcal{K}_\mathcal{R}$ by ignoring the irrelevant states, while retaining the quantitative information. In section 4.3, we then present the relationship between the TKS $\mathcal{K}_\mathcal{R}$ and the corresponding UDS $\mathcal{K}_\mathcal{U}$.

4.1. Abstraction from Irrelevant Locations

Using the available algorithms for compiling Esterel and Quartz, one can compile every program into an equivalent sequential program. If only finite data types were used, then additionally hardware circuits and UDSs can be generated. This is sufficient for code generation and for the verification of temporal properties.

To enhance the efficiency of the verification, it is often advantageous to omit irrelevant details so that the generated formal models are as small as possible. In the opinion of the authors, the choice between relevant and irrelevant program locations must be left to the programmer. For this reason, we propose a new statement to explicitly mark these locations by emitting a special signal δ. To this end, we introduce a new macro statement of the form **abstract** S **end** that can be defined as follows:

$$\textbf{abstract}\ S\ \textbf{end} :\equiv \left(\begin{array}{l}\textbf{local}\ t\ \textbf{in}\\ \quad S; \textbf{emit}\ t\\ \quad \|\\ \quad \textbf{abort}\\ \quad\quad \textbf{loop}\\ \quad\quad\quad \ell : \textbf{pause};\\ \quad\quad\quad \textbf{emit}\ \delta\\ \quad\quad \textbf{end loop}\\ \quad \textbf{when immediate}\ t\\ \textbf{end local}\end{array}\right)$$

Hence, **abstract** S **end** behaves like S, but additionally emits the variable δ whenever the control flow moves inside S. The entering and termination transitions (from and to S) do not emit δ. The above definition has however the drawback that an additional **pause** statement and an additional local signal t are used. This additional overhead can be circumvented by the following alternative definitions:

- $\mathsf{in}\,(\textbf{abstract}\ S\ \textbf{end}) :\equiv \mathsf{in}\,(S)$
- $\mathsf{inst}\,(\textbf{abstract}\ S\ \textbf{end}) :\equiv \mathsf{inst}\,(S)$
- $\mathsf{enter}\,(\textbf{abstract}\ S\ \textbf{end}) :\equiv \mathsf{enter}\,(S)$
- $\mathsf{term}\,(\textbf{abstract}\ S\ \textbf{end}) :\equiv \mathsf{term}\,(S)$
- $\mathsf{move}\,(\textbf{abstract}\ S\ \textbf{end}) :\equiv \mathsf{move}\,(S)$

- $\mathsf{gcmd}\,(\varphi, \textbf{abstract}\ S\ \textbf{end})$
  $:\equiv \mathsf{gcmd}\,(\varphi, S) \cup \{(\mathsf{in}\,(S) \wedge \neg\mathsf{term}\,(S)\,, \textbf{emit}\ \delta)\}$

Hence, the control flows of **abstract** S **end** and S are the same, and the data flow differs only in that **abstract** S **end** additionally emits the variable δ whenever the control flow moves inside S. Using this direct definition via the semantics given in [25, 26] instead of the above macro expansion, we circumvent the use of the additional local variable t and the additional **pause** statement.

Using our new statement **abstract** S **end**, it is easily seen that the following statements were equivalent for any $n \in \mathbb{N}$ and any Boolean condition σ:

- **abstract** ℓ : **await** σ **end**
  $\equiv$
  do
  ℓ : **pause**;
  if $\neg\sigma$ **then emit** δ **end**
  while $\neg\sigma$

- **abstract** ℓ : **await** n **end**
  $\equiv$
  local c **in**
  $c := 0$;
  do
  $c :=$ **delayed** $c + 1$;
  ℓ : **pause**;
  if $c \neq n$ **then**
  emit δ
  end
  while $c \neq n$
  end local

The abstract **await** statements are important to model delays. As can be seen, these statements can be easily defined in terms of existing Esterel/Quartz statements so that existing tools can be used for their compilation. Hence, using the available compilers, we obtain a transition system where each irrelevant state is marked with the special variable δ. Note that **await** n will definitely terminate after n macro steps, while the termination of **await** σ is not guaranteed unless we could guarantee that σ will eventually hold. The termination of abstract statements must be considered in the generation of TKSs as described in the next section.

4.2. Generating TKSs

We now consider the generation of a TKS from a given Quartz program. For this purpose, we assume that we already have a function QuartzCompileUDS that computes an equivalent unit delay structure (UDS) $\mathcal{K}_\mathcal{U}$ of a given Quartz program $\mathcal{P}$. Such a function is essentially implemented by any compiler, like the one described in [24, 25, 26, 27]. To finally obtain an equivalent TKS $\mathcal{K}_\mathcal{R}$, it is therefore sufficient to be able to compute a corresponding TKS from a given UDS where certain states are marked to be irrelevant. In the following, these irrelevant states are labeled by the variable δ.

The overall idea is to replace finite paths $s_0 \to s_1 \to \ldots \to s_{n-1} \to s_n$ of states where $s_1, \ldots, s_{n-1}$ were labeled with δ, but s_0 and s_n were not labeled with δ, by single transitions $s_0 \xrightarrow{n} s_n$. This will generate a TKS, where the quantitative properties of the UDS are preserved. However, if a cycle of states labeled with δ is reachable, then the generation of the corresponding TKS transition can not be performed: If such a cycle is between two states s and s' that are not labeled with δ, then this means that there would exist infinitely many transitions in the TKS between these states (see Figure 2). Such cycles arise when too large parts of the program are embraced within an **abstract** statement.

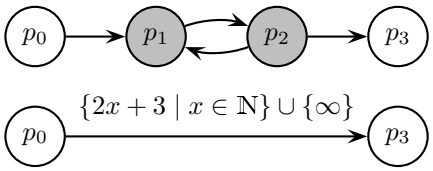

Figure 2. Impossibility of TKS Generation

For this reason, we have to check for reachable cycles of states that are labeled with δ. To be concise in the following, we call states labeled with δ simply 'δ-states', and cycles consisting of δ-states 'δ-cycles'.

A first attempt is to directly compute δ-cycles, and to check if one of them is reachable. However, this is not necessary. It is sufficient to know if there is a reachable δ-cycle, but we have no need to compute such a δ-cycle.

Given a transition relation $\mathcal{U}$ of a UDS $\mathcal{K}_\mathcal{U}$, the function Deadends computes the set of states where *all* paths starting in these states are finite. The computation starts with those states that have no successors, and adds in step $k+1$ those predecessors that only have successors with finite paths of length $\leq k$ (thus obtaining the set of states with finite paths of length $\leq k+1$). At the end, the result is the set of states where all paths leaving these states are finite. The function Reach clearly computes the set of states that are reachable from the states $\mathcal{S}_\varphi$ by the transitions of $\mathcal{U}$.

Consider now the overall translation of a Quartz program $\mathcal{P}$ to an equivalent TKS, as given in Figure 3. We first compute the corresponding UDS $\mathcal{U}$ and their initial states $\mathcal{I}$ with the function QuartzCompileUDS as given in [24, 25, 26, 27]. Our next task is to check whether there is a reachable δ-cycle. For this reason, we restrict the transitions to δ-states, thus obtaining $\mathcal{U}_\delta$, and compute the set of states $\mathcal{S}_{fin}$ that only have finite paths in $\mathcal{U}_\delta$. If $\mathcal{S}_{fin} = \mathcal{S}_\delta$ holds, then we have no δ-cycles at all, since all δ-states have only finite paths. In this case, we can compute the desired TKS by calling $\mathsf{Chronos}(\mathcal{U}, \mathcal{S}_\delta)$. We will discuss that function below.

function Deadends($\mathcal{U}$)
 $\mathcal{S}_0 := \{s \in \mathcal{S} \mid \neg\exists s' \in \mathcal{S}.(s,s') \in \mathcal{U}\}$;
 repeat
  $\mathcal{S}_1 := \mathcal{S}_0$;
  $\mathcal{S}_0 := \mathcal{S}_0 \cup \{s \in \mathcal{S}_0 \mid \forall (s,s') \in \mathcal{U}.s' \in \mathcal{S}_1\}$;
 until $\mathcal{S}_0 = \mathcal{S}_1$;
 return $\mathcal{S}_0$;
end function

function Reach($\mathcal{U}, \mathcal{S}_\varphi$)
 $\mathcal{S}_{reach} := \mathcal{S}_\varphi$
 repeat
  $\mathcal{S}_{old} := \mathcal{S}_{reach}$;
  $\mathcal{S}_{next} := \{s' \in \mathcal{S} \mid (s,s') \in \mathcal{U} \wedge s \in \mathcal{S}_{reach}\}$;
  $\mathcal{S}_{reach} := \mathcal{S}_{reach} \cup \mathcal{S}_{next}$;
 until $\mathcal{S}_{reach} = \mathcal{S}_{old}$;
 return $\mathcal{S}_{reach}$;
end function

function Chronos($\mathcal{U}, \mathcal{S}_\delta$)
 $\mathcal{R} := \{(s,1,s') \mid (s,s') \in \mathcal{U} \wedge \{s,s'\} \cap \mathcal{S}_\delta = \{\}\}$;
 $\mathcal{U}_{out} := \{(s,s') \in \mathcal{U} \mid s \in \mathcal{S}_\delta \wedge s' \notin \mathcal{S}_\delta\}$;
 $\tau := 1$;
 repeat
  $\tau := \tau + 1$;
  $\mathcal{U}_0 := \{(s,s') \mid \exists s_1.(s,s_1) \in \mathcal{U} \wedge (s_1,s') \in \mathcal{U}_{out}\}$;
  $\mathcal{R} := \mathcal{R} \cup \{(s,\tau,s') \mid (s,s') \in \mathcal{U}_0 \wedge s \notin \mathcal{S}_\delta\}$;
  $\mathcal{U}_{out} := \{(s,s') \in \mathcal{U}_0 \mid s \in \mathcal{S}_\delta\}$;
 until $\mathcal{U}_{out} = \{\}$;
 return $\mathcal{R}$;
end function

function QuartzCompileTKS($\mathcal{P}$)
 $(\mathcal{I}, \mathcal{S}, \mathcal{U}) :=$ QuartzCompileUDS($\mathcal{P}$);
 $\mathcal{S}_\delta := \{s \in \mathcal{S} \mid \delta \in \mathcal{L}(s)\}$;
 $\mathcal{U}_\delta := \{(s,s') \in \mathcal{U} \mid s,s' \in \mathcal{S}_\delta\}$;
 $\mathcal{S}_{fin} :=$ Deadends($\mathcal{U}_\delta$);
 if $\mathcal{S}_{fin} = \mathcal{S}_\delta$ **then**
  return Chronos($\mathcal{U}, \mathcal{S}_\delta$)
 else
  $\mathcal{S}_{reach} :=$ Reach($\mathcal{U}, \mathcal{I}$);
  $\mathcal{S}_{\delta c} := \mathcal{S}_{reach} \cap (\mathcal{S}_\delta \setminus \mathcal{S}_{fin})$;
  if $\mathcal{S}_{\delta c} = \{\}$ **then**
   $\mathcal{U}_{reach} := \{(s,s') \in \mathcal{U} \mid s \in \mathcal{S}_{reach}\}$;
   return Chronos($\mathcal{U}_{reach}, \mathcal{S}_\delta$)
  else
   raise exception *AbstractionTooCoarse*($\mathcal{S}_{\delta c}$)
  end
 end;
end function

Figure 3. Algorithms for Generation of TKSs

On the other hand, if $\mathcal{S}_{fin} \neq \mathcal{S}_\delta$ holds, then the δ-states of $\mathcal{S}_\delta \setminus \mathcal{S}_{fin}$ occur on at least one δ-cycle. There is still a chance to generate a TKS, namely if none of these states is reachable. For this reason, we next compute the set of reachable states $\mathcal{S}_{reach}$, and check if a δ-cycle is included in $\mathcal{S}_{reach}$ (note that $\mathcal{S}_{\delta c}$ is the set of reachable δ-cycles). If no δ-cycle is reachable, then we can generate the TKS by calling Chronos with the transitions restricted to reachable states.

Finally, if there is a reachable δ-cycle, then the construction of some TKS transitions is not possible (cf. Figure 2), and therefore an exception is raised where the reachable δ-cycles are returned. With this information, the programmer can identify the program locations that lead to the too coarse abstraction. In this case, there are two possible solutions: either the programmer proceeds with the verification at the UDS level or the programmer has to weaken the abstraction. We consider the first case in the next section in more detail. In the second case, one must consider that the specifications must also be adapted.

Now, consider how Chronos works. Recall, that we may assume that the transition relation $\mathcal{U}$ does not contain any δ-cycle, when Chronos is called. We first compute the transitions between states s and s' that both are not labeled with δ. These transitions are labeled with time duration 1. To compute the further transitions, we have to compute the transitions $\mathcal{U}_{out}$ that leave a δ-sequence. In the following loop, we have as an invariant that the timed transitions with duration $< \tau$ have already been computed, and that $\mathcal{U}_{out}$ contains transitions (s,s') such that s' is not a δ-state, and that there will be a timed transition leading to s' with duration $\geq \tau$.

To compute the timed transitions with duration $\tau+1$, we consider the δ-states s that are connected by a $\mathcal{U}$-transition to a state s_1, such that $(s_1,s') \in \mathcal{U}_{out}$ holds. Then, the transition (s,τ,s') is added to $\mathcal{R}$. If, on the other hand, s is labeled with δ, then this transition will be extended in a later loop iteration until no δ-state remains.

It is easily seen that the loop in Chronos will be repeated at most d_{len} times, where d_{len} is the maximal length of a finite sequence of δ-states. An example for the execution of Chronos is given in Figure 4. The different loop iterations for the UDS given in the upper half of Figure 4 are as follows:

- $\mathcal{U}_{out}^{(1)} = \{(s_1,s_0),(s_5,s_3),(s_5,s_6),(s_7,s_8),(s_{10},s_6)\}$
- $\mathcal{R}^{(1)} = \{(s_3,1,s_3),(s_8,1,s_9),(s_9,1,s_9),(s_0,1,s_3)\}$
- $\mathcal{U}_{out}^{(2)} = \{(s_2,s_0),(s_4,s_3),(s_4,s_6),$
  $(s_6,s_6),(s_3,s_8),(s_9,s_6),(s_6,s_3)\}$
- $\mathcal{R}^{(2)} = \{(s_6,2,s_6),(s_3,2,s_8),(s_9,2,s_6),(s_6,2,s_3)\}$
- $\mathcal{U}_{out}^{(3)} = \{(s_6,s_0),(s_3,s_6),(s_3,s_3)\}$
- $\mathcal{R}^{(3)} = \{(s_6,3,s_0),(s_3,3,s_6),(s_3,3,s_3)\}$

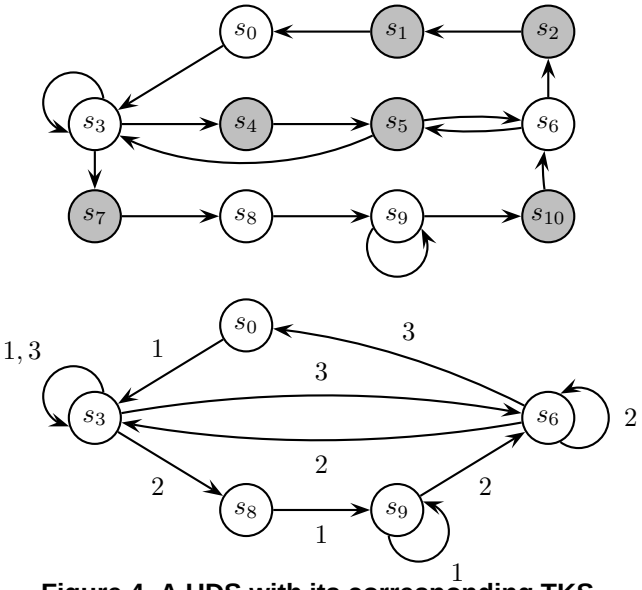

Figure 4. A UDS with its corresponding TKS

At the end, we obtain the TKS given in the lower part of Figure 4.

4.3. Verifying Real-Time Properties for Programs

Recall that our task is to check for a given program $\mathcal{P}$ whether a temporal property Φ holds. To describe real-time temporal properties, we use the CTL real-time extension JCTL [20] that is defined on TKSs.

Using traditional compilers, it is possible to compute for a given program $\mathcal{P}$ the corresponding UDS $\mathcal{K}_\mathcal{U}$, so that temporal properties can be checked for the program. The algorithm given in the previous section allows furthermore to compute a TKS $\mathcal{K}_\mathcal{R}$ for $\mathcal{P}$ to increase the efficiency of the verification. Clearly, the structures $\mathcal{K}_\mathcal{U}$ and $\mathcal{K}_\mathcal{R}$ are different and therefore satisfy different formulas. In this section, we explain the relationship between the verification at the TKS and the UDS level. For this purpose, we define a function Θ_δ that computes for a JCTL formula Φ a corresponding JCTL formula $\Theta_\delta(\Phi)$ that holds on the UDS $\mathcal{K}_\mathcal{U}$ iff Φ holds on the TKS $\mathcal{K}_\mathcal{R}$.

Definition 2 *Given a* JCTL *formula* Φ *and a variable* δ*, we define a corresponding* JCTL *formula* $\Theta_\delta(\Phi)$ *as follows*[3]*:*

- $\Theta_\delta(x) :\equiv x$ *for variables* x
- $\Theta_\delta(\neg\varphi) :\equiv \neg\Theta_\delta(\varphi)$
- $\Theta_\delta(\varphi \wedge \psi) :\equiv \Theta_\delta(\varphi) \wedge \Theta_\delta(\psi)$
- $\Theta_\delta(\varphi \vee \psi) :\equiv \Theta_\delta(\varphi) \vee \Theta_\delta(\psi)$
- $\Theta_\delta(\mathsf{EX}^\kappa\varphi) :\equiv \mathsf{E}[\Theta_\delta(\varphi)\ \underline{\mathsf{XW}}^\kappa\ (\neg\delta)]$
- $\Theta_\delta(\mathsf{AX}^\kappa\varphi) :\equiv \mathsf{A}[\Theta_\delta(\varphi)\ \underline{\mathsf{XW}}^\kappa\ (\neg\delta)]$
- $\Theta_\delta(\mathsf{E}[\varphi\ \underline{\mathsf{U}}^\kappa\ \psi]) :\equiv \mathsf{E}[(\delta \vee \Theta_\delta(\varphi))\ \underline{\mathsf{U}}^\kappa\ (\neg\delta \wedge \Theta_\delta(\psi))]$
- $\Theta_\delta(\mathsf{E}[\varphi\ \mathsf{U}^\kappa\ \psi]) :\equiv \mathsf{E}[(\delta \vee \Theta_\delta(\varphi))\ \mathsf{U}^\kappa\ (\neg\delta \wedge \Theta_\delta(\psi))]$
- $\Theta_\delta(\mathsf{A}[\varphi\ \underline{\mathsf{U}}^\kappa\ \psi]) :\equiv \mathsf{A}[(\delta \vee \Theta_\delta(\varphi))\ \underline{\mathsf{U}}^\kappa\ (\neg\delta \wedge \Theta_\delta(\psi))]$
- $\Theta_\delta(\mathsf{A}[\varphi\ \mathsf{U}^\kappa\ \psi]) :\equiv \mathsf{A}[(\delta \vee \Theta_\delta(\varphi))\ \mathsf{U}^\kappa\ (\neg\delta \wedge \Theta_\delta(\psi))]$

$\Theta_\delta(\Phi)$ is of length $O(|\Phi|)$, i.e., there is only a linear blow-up. The above definition is used to transform a given JCTL Φ to another corresponding JCTL formula $\Theta_\delta(\Phi)$ such that only states not labeled with δ are considered for evaluation of $\Theta_\delta(\Phi)$. If no state is labeled with δ, then it is easily seen that Φ and $\Theta_\delta(\Phi)$ were equivalent. The following theorem, which is one of the main results of this paper, reveals the entire relationship between Φ and $\Theta_\delta(\Phi)$:

Theorem 1 (Relationship between UDS and TKS) *Given a UDS* $\mathcal{K}_\mathcal{U}$ *such that the corresponding TKS* $\mathcal{K}_\mathcal{R}$ *exists, then we have for any* JCTL *formula* Φ *and any state* s *of* $\mathcal{K}_\mathcal{R}$ *the following relationship:*

$$\mathcal{K}_\mathcal{U}, s \models \Theta_\delta(\Phi) \quad \textit{iff} \quad \mathcal{K}_\mathcal{R}, s \models \Phi$$

The theorem is easily proved by an induction on the structure of Φ. The essential property for the induction steps is thereby that (by construction of $\mathcal{K}_\mathcal{R}$) a transition $s \xrightarrow{t} s'$ in $\mathcal{K}_\mathcal{R}$ corresponds to a sequence of transitions $s \to s_1 \to \ldots \to s_{n-1} \to s'$ of states where $s_1, \ldots, s_{n-1}$ are labeled with δ (of course, s and s' are not labeled with δ, since they belong to $\mathcal{K}_\mathcal{R}$).

The above theorem precisely states that if an equivalent TKS $\mathcal{K}_\mathcal{R}$ exists, then we have the choice between checking $\mathcal{K}_\mathcal{U}, s \models \Theta_\delta(\Phi)$ or $\mathcal{K}_\mathcal{R}, s \models \Phi$. Both model checking problems are equivalent to each other.

5. Conclusions and Future Work

We propose a special statement for synchronous programming languages to declare program locations that are irrelevant for verification. After the usual compilation of the program into a transition system, an efficient algorithm is proposed to generate a TKS by ignoring the irrelevant states, while retaining the quantitative information. Our technique directly generates a single real-time transition system, thus overcoming the known problem of composing several real-time models.

Current research of compiling synchronous programs considers how the UDS can be computed in a modular way, exploiting the modular structure of the given program. Analogously, future directions of this work may consider how the TKS could be computed in a modular way[4].

[3] The semantics is given in [20], except for $\mathsf{E}[\varphi\,\underline{\mathsf{XW}}^\kappa\,\psi]$. $\mathsf{E}[\varphi\,\underline{\mathsf{XW}}^\kappa\,\psi]$ holds in a state s iff there is a path starting in s such that at some position (different from the starting one) of the path ψ holds, and at the first such position (different from the starting one) φ holds, and the time required to reach this first position satisifies the time constraint κ.

[4] Given arbitrary statements $\mathcal{P}$ and $\mathcal{Q}$ with their UDSs $\mathcal{K}_\mathcal{P}$ and $\mathcal{K}_\mathcal{Q}$, the equation $\mathcal{K}_{\mathcal{P}\|\mathcal{Q}} = \mathcal{K}_\mathcal{P} \times \mathcal{K}_\mathcal{Q}$ is in general not valid. It is also not valid in general for the corresponding TKSs.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model Checking in Dense Real-time. Technical report, Stanford University, University of Crete, 1991.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes In Computer Science, pages 431–434. Springer-Verlag, March 1996.

[4] G. Berry. The Esterel v5_91 language primer. http://www.esterel.org, June 2000.

[5] V. Bertin, M. Poize, J. Pulou, and J. Sifakis. Towards validated real-time software. In *Euromicro Conference on Real Time Systems*, pages 157–164, Stockholm, 2000.

[6] S. Campos and E. Clarke. Real-Time Symbolic Model Checking for Discrete Time Models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*, AMAST Series in Computing. World Scientific Press, AMAST Series in Computing, May 1994.

[7] S. Campos, E. Clarke, and M. Minea. The Verus tool: A quantitative approach to the formal verification of real-time systems. In O. Grumberg, editor, *Conference on Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 452–455. Springer Verlag, June 1997.

[8] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and systems*, 16(5):1512–1542, September 1994.

[9] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. In *Conference on Computer Aided Verification (CAV)*, Paris, France, 2001.

[10] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

[11] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III*, volume 1066 of *LNCS*. Springer, 1996.

[12] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1998.

[13] S. Edwards. Compiling Esterel into sequential code. In *Design Automation Conference (DAC)*, pages 322–327, Los Angeles, California, June 2000.

[14] S. Edwards, T. Ma, and R. Damiano. Using a hardware model checker to verify software. In *International Conference on ASIC (ASICON)*, Shanghai, China, 2001.

[15] Esterel. Website. http://www.esterel.org.

[16] Esterel-Technology. Website. http://www.esterel-technologies.com.

[17] N. Halbwachs, J.-C. Fernandez, and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language Lustre. In *Symposium on Lucid and Intensional Programming (ISLIP)*, Quebec, Canada, April 1993.

[18] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 394–406, Santa-Cruz, California, June 1992. IEEE Computer Society Press.

[19] L. J. Jagadeesan, C. Puchol, and J. E. V. Olnhausen. Safety property verification of Esterel programs and applications to telecommunications software. In P. Wolper, editor, *Conference on Computer Aided Verification (CAV)*, volume 939 of *LNCS*, pages 127–140, Liege, Belgium, July 1995. Springer Verlag.

[20] G. Logothetis and K. Schneider. A new approach to the specification and verification of real-time systems. In *Euromicro Conference on Real-Time Systems*, pages 171–180, Delft, The Netherlands, June 2001. IEEE Computer Society. http://goethe.ira.uka.de/fmg/ps/LoSc01.ps.gz.

[21] G. Logothetis and K. Schneider. Symbolic model checking of real-time systems. In *International Symposium on Temporal Representation and Reasoning*, pages 214–223, Cividale del Friuli, Italy, June 2001. IEEE/ACM. http://goethe.ira.uka.de/fmg/ps/LoSc01a.ps.gz.

[22] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, February 1995.

[23] J. Ruf and T. Kropf. Using MTBDDs for composition and model checking of real-time systems. In *International Conference on Formal Methods in Computer Aided Design (FMCAD)*, volume 1166 of *LNCS*. Springer, November 1998.

[24] K. Schneider. A verified hardware synthesis for Esterel. In F. Rammig, editor, *International IFIP Workshop on Distributed and Parallel Embedded Systems*, pages 205–214, Schloß Ehringerfeld, Germany, 2000. Kluwer Academic Publishers.

[25] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *International Conference on Application of Concurrency to System Design (ICACSD 2001)*, pages 143–156, Newcastle upon Tyne, UK, June 2001. IEEE Computer Society Press. http://goethe.ira.uka.de/fmg/ps/Schn01a.ps.gz.

[26] K. Schneider. Formal reasoning about synchronous programming languages. Internal report 2001-15, University of Karlsruhe, December 2001. http://goethe.ira.uka.de/fmg/ps/Schn01c.ps.gz.

[27] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Atlanta, USA, November 2001. ACM. http://goethe.ira.uka.de/fmg/ps/ScWe01.ps.gz.

[28] R. Shyamasundar and J. Aghav. Realizing real-time systems from synchronous language specifications. In *Real Time Systems Symposium, Work in Progress Session*, Orlando, Florida, USA, November 2000. IEEE.

[29] Telelogic. Website. http://www.telelogic.com.

[30] S. Tripakis and S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In R. Alur and T. A. Henzinger, editors, *Conference on Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 232–243, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.