From Synchronous Programs to Symbolic Representations of Hybrid Systems

Kerstin Bauer and Klaus Schneider
Department of Computer Science
University of Kaiserslautern, Germany

ABSTRACT

In this paper, we present an extension of the synchronous language Quartz by new kinds of variables, actions and statements for modeling the interaction of synchronous systems with their continuous environment. We present an operational semantics of the obtained hybrid modeling language and moreover show how compilation algorithms that have been originally developed for synchronous languages can be extended to these hybrid programs. Thus, we can automatically translate the hybrid programs to compact symbolic representations of hybrid transition systems that can be immediately used for simulation and formal verification.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.1 [Theory of Computation]: Specifying and Verifying and Reasoning about Programs; F.3.2 [Theory of Computation]: Semantics of Programming Languages

General Terms
Languages, Verification.

Keywords
Hybrid Systems, Synchronous Languages, Symbolic Representation of Transition Systems

1. INTRODUCTION

Reactive systems [32] are systems that have an ongoing interaction with their environment in terms of a discrete sequence of reaction steps. In each reaction step, new inputs are read and corresponding outputs and next states of the system are computed. Since the environment may trigger new reaction steps, reactive systems are special real-time systems.

Synchronous languages [8, 9, 30] such as Esterel [10, 12], Lustre [31], and Quartz [48] have been developed to describe reactive systems. The operational semantics of these languages is defined by so-called micro and macro steps, where a macro step consists of finitely many micro steps whose maximal number is known at compile time. Macro steps correspond with reaction steps of the reactive system, and micro steps correspond with atomic actions like assignments of the program. Variables of a synchronous program are synchronously updated between macro steps, so that the execution of the micro steps of one macro step is done in the same variable environment of their macro step.

The distinction between micro and macro steps does not only lead to a convenient programming model for reactive systems that allows deterministic and efficient hardware and software syntheses as well as a simplified estimation of worst-case reaction times. It is also the key to a compositional formal semantics which is a necessary requirement for formal verification and provably correct synthesis procedures. Typically, the semantics are described by means of SOS (structural operational semantics [45]) transition rules that recursively follow the syntax of the program [11, 48].

Additionally, synchronous programs can be efficiently translated to transition systems that capture their entire behavior. This translation can be done in polynomial time even though the transition system may have exponentially (or even infinitely many) states. This is possible since symbolic representations of polynomial size are computed by these translations. Model-checking of temporal logic specifications [47] can be directly performed on these symbolic representations of the transition systems. Hence, synchronous languages perfectly lend themselves to the development of embedded reactive systems which are used in applications requiring a formal verification.

In contrast to the discrete reaction steps of an embedded reactive system, its environment often consists of continuous behaviors that are determined by the laws of physics. For this reason, the verification of properties that depend on the interaction with the continuous environment requires the consideration of hybrid systems (see e.g. [1, 2]). Since most verification problems are undecidable for hybrid systems [33, 37], a rich theory of algorithmic, approximative approaches has been developed over the years [5, 18, 28, 29, 40, 41, 46]. In contrast, only a few languages and tools to deal with non-trivial hybrid systems [19] are available. Moreover, the languages of most of these tools have either no formal semantics, are restricted to very special subclasses of hybrid automata, or approximate the continuous behaviors by discrete behaviors.

Furthermore, also the size of the discrete part of hybrid systems may challenge tools: even though hybrid systems typically consist of 'only' finitely many discrete states, the number of these discrete states can become too large to al-
low an explicit enumeration. For this reason, there are already some approaches that employ symbolic representations like [18, 24, 25, 29, 35, 38] to compute successor or predecessor states. For example, Collins' cylindrical algebraic decomposition [7, 23] for quantifier elimination can be used to symbolically compute successor or predecessor sets of states analogous to finite state systems [14, 17].

Symbolic representations of hybrid systems are quite new: despite some early approaches [38], up to now, there are no tools available that are entirely based on symbolic representations. For example, HyTech [36] as well as PHAVer [27] only represent the continuous states in a symbolic way, and therefore fail for large state spaces as reported in [24]. Only some recent approaches [18, 24, 25, 29] aim at using a symbolic representation for the entire behavior, but these descriptions require the user to manually describe a boolean formula for the transition relation, which is very inconvenient and error-prone.

Hence, there is still a need for powerful modeling languages having a precise formal semantics that allow the description of non-trivial discrete programs within their continuous environments. Due to the huge number of discrete states of such systems, symbolic representations are mandatory for verification, and clearly, the interaction between discrete and continuous variables must be precisely defined. Thus, it is natural to consider extensions of synchronous languages to model hybrid systems consisting of discrete and continuous behaviors.

In this paper, we propose an extension of our synchronous language Quartz for modeling hybrid systems. The benefit of this approach is that the descriptions of the hybrid systems are, compared to their symbolic representations, much more readable, so that abstractions and compositional reasoning becomes much simpler. Symbolic representations of the underlying transition systems like those used in [18, 24, 25, 29] can be obtained by first compiling the programs into guarded actions and then constructing a symbolic representation from the guarded actions.

The paper is organized as follows: After presenting related work in Section 2.1, the basics of the synchronous language Quartz are introduced in Section 2.2. In Section 3, we define the syntax and semantics of the hybrid extension of Quartz. In particular, we describe the semantics in terms of SOS rules that cover the discrete as well as the continuous behavior of the modeled hybrid system. The compilation to symbolic representations of transition systems is described in Section 4.

2. PRELIMINARIES

2.1 Modeling Hybrid Systems

Of course, many languages to describe hybrid systems have already been proposed for simulation and verification. A good survey has been made by the Columbus project [19, 20] covering languages and tools like Stateflow/Simulink, Modelica [42], SADIE, HyTech, Scicos, Charon [3], Checkmate [39], Massaccio [34], Shift [26], Hysdel [49], etc. in order to define an interchange format for hybrid systems [20, 44]. In our opinion, such a language has to fulfill the following requirements:

1. **Modeling**: The language must support the modeling of non-trivial discrete programs within their continuous environment. This excludes an explicit enumeration of the discrete state space and requires the usage of typical program statements and data types.
2. **Simulation**: The language must have a precise operational semantics that can be used for simulation.
3. **Verification**: Automatic translations to symbolic representations of the transition systems used in formal verification must be available.

We are not aware of a language that meets the above requirements in a satisfactory way: A major weakness of analog mixed signal (AMS) extensions of hardware description languages like VHDL-AMS and SystemC-AMS [43, 50] is the lack of a formal semantics [51]. The same is the case for Simulink/Stateflow, as well as for CheckMate since it is also based on Simulink models. HyVisual, Shift, HyTech, PHAVer, HSolver and d/dt explicitly enumerate the discrete locations of the modeled hybrid automaton and therefore do not allow the modeling of non-trivial discrete programs. Languages like Scade/Lustre [21] and Hysdel [49] only have a time-discrete semantics which is only an approximation of hybrid systems.

Charon has a formal semantics [3] whose discrete behaviors are described in terms of guarded actions. We also use guarded actions as intermediate result of our compilation. While Charon's guarded actions are interleaved, ours are synchronous guarded actions that are more related to the description language described in [4]. We believe that guarded actions do not provide enough readability compared to structured program statements. Nevertheless, guarded actions are a very good intermediate language where hybrid models can be translated to and where system descriptions for a model checker can be generated from.

In [22], the synchronous language Esterel has been extended to model timed automata in order to reason about real-time constraints. We have the same motivation for extending a synchronous language as [22], but our targets are hybrid systems that are more general than timed automata. A seminal approach for extending the synchronous language Quartz has already been presented in [6]. However, it was neither completed in its formal semantics nor in its compilation to symbolic representations. The extension presented here is different to [6], although it shares some common ideas.

2.2 The Synchronous Language Quartz

Quartz [48] is a synchronous language derived from Esterel [10, 13]. In the following, we only give a brief overview of Quartz, and refer to [48] for further details. Provided that \( S, S_1, S_2 \) are statements, \( \ell \) is a location variable, \( x \) is a variable, \( \sigma \) is a Boolean expression, and \( \rho \) is a type, then the following are statements (parts given in square brackets are optional):

- **nothing** (empty statement)
- \( x = \tau \) and \( \text{next}(x) = \tau \) (assignments)
- \( \text{assume}(\varphi) \) and \( \text{assert}(\varphi) \) (assumptions and assertions)
- \( \ell : \text{pause} \) (start/end of macro step)
- \( \text{if} (\sigma) S_1 \text{ else } S_2 \) (conditional)
- \( S_1; S_2 \) (sequences)
- \( \text{do } S \text{ while}(\sigma) \) (loops)
- \( S_1 || S_2 \) (synchronous concurrency)
- \([\text{weak}] [\text{immediate}] \text{ abort } S \text{ when}(\sigma)\)
- \([\text{weak}] [\text{immediate}] \text{ suspend } S \text{ when}(\sigma)\)
- \( \{\rho x; S\} \) (local variable \( x \) of type \( \rho \))
The \texttt{pause} statement defines a control flow location \( \ell \), which is a boolean variable that is true iff the control flow is currently at the statement \( \ell : \texttt{pause} \). Since all other statements are executed in zero time, the control flow can only rest at these positions in the program, and therefore the possible control flow states are the subsets of the set of locations.

There are two variants of assignments that both evaluate the right-hand side \( \tau \) in the current macro step (variable environment). While immediate assignments \( x = \tau \) immediately transfer the value of \( \tau \) to the left-hand side \( x \), delayed assignments \( \text{next}(x) = \tau \) transfer this value only in the following step.

If the value of a variable is not determined by assignments, a default value is computed according to the declaration of the variable. To this end, declarations consist of \textit{a storage class in addition to the type of a variable}. There are two storage classes, namely \texttt{mem} and \texttt{event} that choose the previous value (\texttt{mem} variables) or a default value (\texttt{event} variables) in case no assignment determines the value of a variable. Available data types are booleans, bivectors, signed/unsigned bounded/unbounded integers, arrays and tuples.

In addition to the statements known from other imperative languages (conditional, sequential composition and loops), Quartz offers synchronous concurrency \( S_1 \parallel S_2 \) and sophisticated preemption and suspension statements, as well as many more statements to allow comfortable descriptions of reactive systems (see [48] for the complete syntax and semantics).

The structural operational semantics (SOS) of Quartz is described by two sets of rules: The \textit{reaction rules} determine the current values of local and output variables according to given values of input variables and the semantics of Quartz statements. To this end, one starts with a variable environment \( \mathcal{E} \) that maps input and state variables to known values, and local and output variables to an unknown value \( \bot \). Environments are endowed with an incarnation index \( h \) that is used to distinguish between overlapping scopes of local variables that coexist in the same macro step. We ignore incarnation levels in the following, since this is a subtle issue in the semantics of synchronous languages. The reader may view the incarnation levels as part of the variable environment \( \mathcal{E} \). In general, reaction rules have the following form:

\[
(\mathcal{E}, h, S) \xrightarrow{\alpha} (h', D_{\text{must}}, D_{\text{can}}, t_{\text{must}}, t_{\text{can}}),
\]

where \( D_{\text{must}} \) and \( D_{\text{can}} \) are the actions that must be and can be executed in the current environment and where the boolean flags \( t_{\text{must}} \) and \( t_{\text{can}} \) denote whether the execution must be or can be instantaneous\(^1\), respectively. By the optimistic and pessimistic estimations, it is clear that the conditions \( D_{\text{must}} \subseteq D_{\text{can}} \) and \( t_{\text{must}} \rightarrow t_{\text{can}} \) hold.

Given an incomplete variable environment \( \mathcal{E} \) and a statement \( S \), the reaction rules determine the sets \( D_{\text{must}} \) and \( D_{\text{can}} \). If an action \( x = \tau \) is added to \( D_{\text{must}} \), the value of the variable \( x \) becomes known as the result of the evaluation of \( \tau \). Moreover, if all actions writing to a variable \( x \) are removed from \( D_{\text{can}} \), the value of \( x \) is determined by the reaction-to-acceptance: memorized variables maintain their previous value while event variables are reset to a constant default value.

Having updated an environment, the reaction rules are once more applied until no more updates are possible. If we finally have \( D_{\text{must}} = D_{\text{can}} \) and \( t_{\text{must}} = t_{\text{can}} \), then the program is causally correct, otherwise it is rejected due to causality problems. Hence, for causally correct programs (and only these are of interest), the reaction rules determine a complete environment \( \mathcal{E} \) that maps not only all input, local, output, and state variables to corresponding values.

The second kind of SOS rules are called \textit{transition rules}. Their major task is to determine the movement of the control flow from the current to the next macro step. Transition rules also determine the delayed assignments that have to be executed in the next macro step and are of the following form

\[
(\mathcal{E}, h, S) \rightarrow (h', S', D, t)
\]

with the following meaning:

- \( (\mathcal{E}, h) \) is the complete environment of the current macro step as determined by the reaction rules.
- \( h' \) is the updated incarnation level function that is obtained by executing local variable declarations in \( S \).
- \( S' \) is the residual statement that has to be executed in the next micro or macro step (depending on the flag \( t \)).
- \( D \) is a set of pairs \( (\tau, v) \) where \( \tau \) is a variable and \( v \) is a value consistent with the type of \( \tau \). We must assign \( v \) to \( \tau \) before starting the next macro step, as the pair \( (\tau, v) \) stems from a delayed assignment \( \text{next}(\tau) = \tau \).
- Finally, the boolean value \( t \) denotes whether the execution described by the SOS transition rule is instantaneous. Hence, the SOS transition rule describes a micro step if \( t \) is true, and a macro step otherwise.

Reaction rules and transition rules are defined for all statements. Both rule sets together describe the operational semantics of the synchronous programs.

Our Averest system\(^2\) provides algorithms that translate a synchronous program to a set of guarded actions [48], i.e., pairs \( (\gamma, \alpha) \) consisting of a trigger condition \( \gamma \) and an action \( \alpha \). Actions are thereby assignments \( x = \tau \) and \( \text{next}(x) = \tau \), assumptions \( \text{assume}(\phi) \), or assertions \( \text{assert}(\phi) \). The meaning of a guarded action is obvious: in every macro step, all actions are executed whose guards are true. Thus, it is straightforward to construct a symbolic representation of the transition relation in terms of the guarded actions (see [48]).

3. HYBRID QUARTZ

In this section, we present the syntax and semantics of our extension of the synchronous language Quartz. Clearly, we have to introduce continuous variables and continuous transitions of these variables within a macro step. The overall idea is explained best by considering the execution of a hybrid macro step: First, inputs are read and the reaction rules are used to execute the micro steps of the current macro step. This completes the variable environment of the discrete variables so that the transition rules are executed next to determine the next control state and the values that have to be assigned due to delayed assignments at the beginning of the next macro step. Then, a new phase of the macro step is added that consists of the continuous behavior of the continuous variables. During this phase, physical time will increase and the behavior of the continuous variables is determined by the specified differential equations. The end of the continuous phase is determined by special release conditions that are derived from new statements explained below. As soon as one of these release conditions becomes true, the continuous phase and thus, also the

---

\(^1\)The execution of a statement is instantaneous iff it is executed in zero time.

\(^2\)http://www.averest.org
macro step, terminates. Then, the updates due to the already determined delayed actions and the discrete state transition takes place and a new macro step can be invoked depending on the termination of the program.

The execution of a hybrid Quartz program is therefore still divided into discrete macro steps that refer to physical points of time \( t_0, t_1, \ldots \in \mathbb{R} \). The immediate and delayed assignments are executed at these points of time, and their execution still does not consume time in the programming model. In the same way, the enabled assertion and assumption conditions only refer to these discrete points of time and not to the points of time in between which belong to the continuous phases. In analogy to hybrid automata, we say that a program makes a discrete transition at a point of time \( t_i \), and it makes continuous transitions in between these points of time (see below for a more precise definition).

In order to describe our extension of Quartz to deal with hybrid systems, we describe the syntax of our new constructs in Section 3.1 and define their semantics in Section 3.2.

### 3.1 Syntax

#### 3.1.1 Continuous Variables

Recall that discrete variables in Quartz are declared with either the storage class \texttt{mem} or \texttt{event} which determines their reaction to absence. Our generalization to hybrid systems adds a third storage class \texttt{cont} to introduce continuous variables that may change their values during the continuous phase of a macro step.

In the current version, we demand that continuous variables must have the data type \texttt{real}, so that a continuous variable \( x \) is always declared by \texttt{cont real x}. In the following, we distinguish between continuous variables having the mentioned declaration, and discrete variables (having another declaration).

Continuous variables \( x \) can occur everywhere where discrete variables may occur, i.e., as left hand sides of immediate (\( x = \tau \)) and delayed assignments (\( \text{next}(x) = \tau \)) as well as in all control-flow conditions that occur in the program (including assertions and assumptions).

#### 3.1.2 Continuous Actions

The 'discrete' actions (assignments \( x = \tau \), \( \text{next}(x) = \tau \) and assertions/assumptions \( \text{assume}(\varphi) \) and \( \text{assert}(\varphi) \)) only refer to the discrete points of time \( t_i \) on the execution trace. For the new continuous phase of a macro step, we add the following new 'continuous actions' that are evaluated in the continuous phase, i.e., at the points of time \( t \in (t_i, t_{i+1}) \) between two macro steps:

- \( x \leftarrow \tau \) is a direct definition of the continuous flow of \( x \) which essentially corresponds to an immediate assignment \( x = \tau \) that is evaluated in the continuous phase.
- \( \text{der}(x) \leftarrow \tau \) is an indirect definition of the continuous flow of \( x \) which defines the continuous behavior of \( x \) during the continuous phase via its derivation on physical time.
- The following new continuous assertions\(^4\) impose specifications that must be satisfied for certain points of time in the continuous phase of a macro step:
  - \( \text{constrainS}(\varphi) \) demands that \( \varphi \) holds at the Start of the continuous phase
  - \( \text{constrainM}(\varphi) \) demands that \( \varphi \) holds at all intermediate (Middle) points of the continuous phase
  - \( \text{constrainE}(\varphi) \) demands that \( \varphi \) holds at the End of the continuous phase

Macros are given for any combination of \( S, M, \) and \( E \) (in this order), e.g., \( \text{constrainSME}(\varphi) \) demands that \( \varphi \) holds for all points of time of the continuous phase.

#### 3.1.3 Continuous Statements

While discrete actions are statements on their own, the new continuous actions \( \alpha_i \) may only occur in new flow-statements of the following form:

\[
(\ell, \ell') : \text{flow} \{\alpha_1; \ldots; \alpha_n\} \text{ until}(\sigma).
\]

The intuitive meaning of such a statement is as follows: If the control flow is entering the statement, the current macro step will be terminated, and the control flow will resume the execution in the next macro step either from location \( \ell \) or \( \ell' \). Before reaching one of these locations, the continuous phase of the current macro step is determined by (1) the continuous actions \( \alpha_1, \ldots, \alpha_n \) and (2) the release condition \( \sigma \). The actions \( \alpha_i \) are enabled during the continuous phase, and the physical point of time \( t_{i+1} \) where the macro step terminates is determined as the first point of time where a release condition \( \sigma \) holds. Note that several flow statements may run in parallel, but that every thread may execute at most one flow statement (since we do not allow that flow statements are nested in preemption statements that could make them schizophrenic). If the condition \( \sigma \) holds at the point of time \( t_{i+1} \) where the continuous phase ends, the location \( \ell \) is reached, otherwise location \( \ell' \) is reached.

If the execution resumes in the next macro step from location \( \ell \), the statement behaves as nothing, since its release condition became true in the previous continuous phase. Otherwise, the execution resumes in the next macro step from location \( \ell' \), which means that the flow statement behaves as if it would have been started in that macro step: the actions \( \alpha_i \) are enabled in the continuous phase of the next macro step, and the end of that continuous phase be be determined by \( \sigma \) (if that is the first release condition that becomes true).

Note that the semantics of our flow statement refers to the typical definition of the parallel composition of hybrid automata [1, 15, 16]. A thread, i.e., a flow statement, proceeds with its continuous phase if it has been interrupted by another flow statement whose release condition became true first.

In order to avoid write conflicts, causality problems and schizophrenia problems, we impose the following restrictions:

- Flow statements must not occur in abortion or suspension statements to avoid schizophrenia. As flow statements are therefore never aborted, the execution of their threads is never instantaneous, and thus flow statements will not overlap with themselves within a macro step. Hence, we have no schizophrenia problems with flow statements.
- It is not allowed that more than one flow assignment \( x \leftarrow \tau \) or \( \text{der}(x) \leftarrow \tau \) writing to the same variable \( x \) is enabled in the same continuous phase. On the other hand, if there is no assignment for a variable \( x \), the default assignment \( \text{der}(x) \leftarrow 0 \) is used as default action that maintains the value of the continuous variable \( x \).

\(^4\)In contrast to floating point numbers, the data type \texttt{real} denotes the real numbers with arbitrary precision.

\(^5\)In contrast to the invariants of hybrid automata, these assertions only generate specifications to be verified.
In each macro step, enabled actions $x \leftarrow \tau$ must not have cyclic dependencies in order to guarantee computability (we currently do not apply a causality analysis in the continuous phase which could be done in analogy to the causality analysis in the discrete phase).

Each system of ordinary differential equations (ODEs) defined by the enabled flow statements must be symbolically solvable.

### 3.2 Semantics

In this section, we define the formal semantics of our new constructs. To this end, we have to introduce for the continuous phase a new variable environment $C$ that maps continuous variables $x$ to functions $C(x) : \mathbb{R} \rightarrow \mathbb{R}$ and discrete variables $x$ to constant functions $C(x) : \mathbb{R} \rightarrow \rho$, where $\rho$ is determined by the type of $x$. Clearly, $C$ must be chosen such that equations $x = \tau$ and $\text{der}(x) = \tau$ that correspond to flow assignments $x \leftarrow \tau$ and $\text{der}(x) \leftarrow \tau$ as well as all continuous assertions $\text{constrainS}(\varphi)$, $\text{constrainN}(\varphi)$, and $\text{constrainE}(\varphi)$ are satisfied during the continuous phase of the macro step.

To this end, we define a satisfaction relation $C, E, h, \Delta \models \sigma$, where $C$ is the above mentioned continuous variable assignment, $E$ and $h$ denote the discrete variable environment, $\Delta$ denotes an interval of real numbers, and $\sigma$ is either a boolean condition, a flow assignment or a continuous constraint. Intuitively, $C, E, h, \Delta \models \sigma$ means that $\sigma$ holds at all points of time $\Delta$ with the variable environments $C$ and $(E, h)$. Recall that the discrete environment $(E, h)$ maps both continuous and discrete variables to corresponding values, and that the continuous environment $C$ maps both continuous and discrete variables to functions of type $C(x) : \mathbb{R} \rightarrow \rho$ (where discrete variables are constant functions).

Based on the variable environment $C$, we define an evaluation function $[\tau]_{C,t}$ that determines the value of the expression $\tau$ under the continuous variable environment $C$ at the point of time $t \in \mathbb{R}$. Clearly, for variables $x$, we have $[x]_{C,t} := (C(x))(t)$, and for the allowed operators like arithmetic operations $+, -, \times, \div$ or $\sin, \cos$, etc. the definitions are as expected (due to lack of space, we omit the full definition).

Using the evaluation function $[\tau]_{C,t}$, we can then define the semantics of flow assignments $x \leftarrow \tau$ and $\text{der}(x) \leftarrow \tau$ in that we demand that the equations $x = \tau$ and $\dot{x} = \tau$ must hold invariantly during the specified interval $\Delta$. The semantics of the continuous assertions is also trivial as well as the boolean operators as shown in Figure 1.

The definition of the SOS rules of the flow statement are shown in Figure 2. The SOS reaction rule simply formalizes that the flow statement does not execute discrete actions and that its execution is never instantaneous. In addition to the so-far used transition rules (see Section 2.2), the new rules use as further parameters the continuous environment $C$ as well as the amount of physical time $\delta$ that is postulated to be required by the continuous phase of the macro step. The transition rules of all statements other than the flow statement ignore these additional parameters.

The second rule in Figure 2 describes the case where the release condition $\sigma$ of the flow statement is satisfied at the end $\delta$ of the continuous phase $\Delta := [0, \delta]$. In this case, the flow statement terminates, the control flow reaches location $\ell$, and the residual statement remaining for the next macro step is nothing. In the third rule, the release condition $\sigma$ is not yet true, so that the release condition of another flow statement interrupted the continuous phase of the considered flow statement. In this case, the residual statement is the complete flow statement, so that its behavior is repeated in the next macro step. However, note that the initial values of the variables might have changed in the next macro step.

Using the formal semantics as given in Figure 1 and Figure 2, we conclude this section by the description of the overall operational semantics of a macro step which is illustrated in Figure 3 by a two-dimensional notion of ‘time’, where physical time (= continuous transition) flows from left to right and logical time (= discrete transition) flows from the bottom to the top. We assume that the $n$-th macro step takes place at a physical time $t_n \in \mathbb{R}$. Then, in the programmer’s view, the
delayed assignments of macro step \( n - 1 \) and the immediate assignments of macro step \( n \) determine the values of all variables at point \( P^n \) in Figure 3 that refers to the physical point of time \( t_n \). Note that this does not consume physical time and that this is done by the reaction rules with a fixpoint iteration. Note further that the flow statement does not contribute to the this fixpoint iteration (except for guaranteeing that it is not instantaneous) since flow statements do not execute discrete actions.

Furthermore, all right hand side expressions of delayed assignments enabled at macro step \( n \) are evaluated and the residual statement to be executed in the next macro step is determined using the transition rules. The transition rules also invoke the continuous phase of the macro step in case a flow statement is executed. In this case, a duration \( \delta := t_{n+1} - t_n \) must be 'guessed' so that one of the two rules of Figure 2 can be applied to the enabled flow statements: To this end, the discrete variables maintain their values between \( t_n \) and \( t_{n+1} \), while the continuous variables may change in \([t_n, t_{n+1}]\) according to the enabled flow assignments \( \text{der}(x) \leftarrow \tau \) and \( x \leftarrow \tau \). Finally, the continuous assertions must be satisfied during the continuous phase. The continuous environment \( C \) is thereby determined by (1) the current values of the variables given by \( E, h \) at point \( P^n \) in Figure 3, and (2) the solution of the system of ordinary differential equations (ODEs) generated by the enabled flow assignments.

To summarize, a macro step consists of the following steps:

1. perform delayed actions of the previous macro step
2. read new values of all input variables to determine an incomplete environment \( E \)
3. complete \( E \) by means of the reaction rules for local and output variables and the reactions to absence
4. compute delayed actions as well as the residual statement for next macro step by the transition rules
5. determine \( C \) by solving the system of active ODEs using \( E \) for initial values of the ODEs
6. determine the duration \( \delta \) of the continuous phase of the macro step by considering the release conditions \( \sigma \) of the enabled flow statements as well as the continuous environment \( C \)

7. repeat these steps until the program terminates (as seen by the transition rules)

Steps 1.-4. are identical to the discrete version of Quartz with the only exception, that default values for the reaction to absence of continuous variables \( x \) are not given by the previous value \( E(x) \) but by the previous value \( C(x)(\delta) \). Step 5 may involve the use of a symbolic solver for ODEs (but of course, simulators may also approximate solutions numerically). Step 6 is the only non-constructive step since it is rarely possible to determine \( \delta \) symbolically. This problem can be reduced to find roots of continuous functions, which is undecidable in general. In practice, this step has to be done either numerically (when doing simulation) or by a formal verification.

Note, however, that this problem is not a consequence of our modeling framework, but is inherent in any tool for modeling hybrid systems that does not restrict its continuous dynamics to decidable classes.

4. GENERATING TRANSITION SYSTEMS

4.1 Compilation to Guarded actions

As already stated in Section 2.2, there are algorithms for compiling synchronous programs to symbolic representations of their underlying transition systems. The common basis for the algorithms used in our Averest system is an intermediate translation to guarded actions, i.e. pairs \((\gamma, \alpha)\) consisting of a trigger condition \( \gamma \) and an action \( \alpha \). The guarded actions generated from the new flow statements contain actions that are flow assignments \( x \leftarrow \tau \), \( \text{der}(x) \leftarrow \tau \), continuous assertions \( \text{constraint}(x), \text{constraint}(c), \text{constraint}(\varphi) \) or release conditions \( \text{release}(\sigma) \). The latter are obtained from the condition \( \sigma \) of the flow statements, and the former are directly obtained from the body actions \( \alpha \) of the flow statements.

Note that the guards of the guarded actions listed below may contain subformulas of the form \( \text{cont}_{\alpha}(\sigma) \). These subformulas denote that the boolean expression \( \sigma \) must be evaluated at the end of the continuous phase, i.e., with the variable environment \( C(\cdot)(\delta) \), whereas all other subformulas are evaluated with the variable environment \( E \) of the discrete phase. Using this distinction allows us to generate a single boolean formula for the transition relation that is satisfied whenever the considered variable assignments describe a transition of the represented transition system.

The compilation of synchronous programs into guarded actions is divided into two parts [48]: CompSurface computes the guarded actions that are executed when the control flow enters the statement, and CompDepth computes the guarded actions that are executed when the control flow resumes the execution from some location inside the statement. Due to lack of space, we do not list the code that has to be added to these functions to handle flow statements, but we list the relevant information required for this purpose: First, it is trivial that flow statements neither invoke module calls nor do they trigger discrete actions. They also do not contain local variable declarations. Moreover, flow statements must not occur in preemption statements, so that an error is reported whenever a surrounding preemption context is seen.
Let us abbreviate \( S \coloneqq (\ell, \ell'):\text{flow}\{\alpha_1; \ldots; \alpha_n\}\text{ until}(\sigma) \) for the following. The control flow of \( S \) is determined by the following control flow predicates [48]:

- \( S \) is never instantaneous, since the control flow will either reach location \( \ell \) or \( \ell' \) where the macro steps end.
- The control flow is inside \( S \) iff the condition \( \ell \lor \ell' \) holds.
- \( S \) terminates iff \( \ell \land \ell' \land \neg \rho \) holds.

These conditions are returned by the functions CompSurface and CompDepth, respectively, and are used to generate the guards of the guarded actions. Besides these conditions, we also have to generate guarded actions from the flow statement \( S \):

- If CompSurface is called with start condition \( \text{strt} \), it generates the following guarded actions:
  \[
  \begin{align*}
  &\text{strt}, \alpha_1, \ldots, \text{strt}, \alpha_n \\
  &\text{strt} \land \rho \land \text{next}(\ell) = \text{true} \\
  &\text{strt} \land \neg \rho \land \text{next}(\ell') = \text{true}
  \end{align*}
  \]
- Finally, CompDepth generates the same guarded actions where \( \text{strt} \) is replaced with \( \ell' \):
  \[
  \begin{align*}
  &\ell', \alpha_1, \ldots, \ell', \alpha_n \\
  &\ell' \land \rho \land \text{next}(\ell) = \text{true} \\
  &\ell' \land \neg \rho \land \text{next}(\ell') = \text{true}
  \end{align*}
  \]

As an example, consider Figure 4. From the program shown in (a), we compute the guarded actions shown in (b) (we have already combined the surface and depth parts, which is done by the final linking step [48]).

Having generated guarded actions, we are in principle at the level of the languages used in [3, 4]. For visualizing the encoded transition systems, it is moreover possible to generate extended finite state machines (EFSM) that correspond to hybrid automata. In EFSMs, the control flow is enumerated explicitly while the data flow is kept in a symbolic way. Part (c) in Figure 4 shows the EFSM obtained from the guarded actions shown in part (b) of the same figure. Circles denote control flow states, rectangles denote the discrete transition, and diamonds contain the continuous transition.

### 4.2 Symbolic Description

In this section, we consider the construction of a symbolic transition relation for the underlying state transition system encoded by the guarded actions. To this end, we consider the guarded actions for a variable \( x \):

- \( (\alpha_1, x = \tau_1), \ldots, (\alpha_n, x = \tau_n) \)
- \( (\beta_1, \text{next}(x) = \pi_1), \ldots, (\beta_n, \text{next}(x) = \pi_n) \)
- \( (\gamma_1, x < - \tau_1), \ldots, (\gamma_i, x < - \tau_i) \)
- \( (\delta_1, \text{der}(x) < 0), \ldots, (\delta_k, \text{der}(x) < 0) \)

We define the discrete \( R_d(x) \) and continuous behaviors \( R_c(x) \) of \( x \) as follows:

- \( R_d(x) \) := \[
  \begin{align*}
  &\left( \bigwedge_{i=1}^n \alpha_i \rightarrow x = \tau_i \right) \land \\
  &\left( \bigwedge_{i=1}^n \beta_i \rightarrow \text{next}(x) = \pi_i \right) \land \\
  &\left( \bigwedge_{i=1}^n \gamma_i \rightarrow x < - \tau_i \right) \land \\
  &\left( \bigwedge_{i=1}^n \delta_i \rightarrow \text{der}(x) < 0 \right)
  \end{align*}
  \]

- \( R_c(x) \) := \[
  \begin{align*}
  &\left( \bigwedge_{i=1}^n \gamma_i \rightarrow x = \tau_i \right) \land \\
  &\left( \bigwedge_{i=1}^n \delta_i \rightarrow \text{der}(x) = 0.0 \right)
  \end{align*}
  \]

This means that the actions are executed whenever their guards hold, and that whenever no guard holds, then a default value is chosen. In case of the discrete behavior \( R_d(x) \), the default value \( \text{default}(x) \) is chosen according to the declared storage class, and in case of the continuous behavior \( R_c(x) \) the derivation is zero so that \( x \) is a constant function. We also define \( R_c(x) \) for discrete variables where \( R_c(x) \) simplifies to \( R_c(x) = \text{der}(x) = 0.0 \).

Moreover, we have to consider the release conditions that determine when a continuous phase must end. To this end, assume that we obtain the guarded actions \( (e_1, \text{release}(\sigma_1)), \ldots, (e_k, \text{release}(\sigma_i)) \). Then, we first construct the formula

\[
R_{rel} \coloneqq \bigwedge_{i=1}^k e_i \rightarrow \neg \sigma_i
\]

Having constructed this formula \( R_{rel} \) and \( R_d(x) \) and \( R_c(x) \) for every variable \( x \in V \), we can combine them to obtain the entire transition relation \( R \):

\[
R \coloneqq \bigwedge_{x \in V} R_d(x) \land \text{cont}_d(R_c(x)) \land \text{cont}_c(R_{rel})
\]

Similar to the operator \( \text{cont}_d \) appearing in the guards \( \gamma_i \) of location variables, we use another operator \( \text{cont}_c \) demanding that its argument formula must hold during the continuous phase. Note that the subformulas used in the transition relation \( R \) refer to different points of time along a transition: The operator \( \text{next-ref} \) refers to the variable values at the target state, \( \text{cont}_d(\varphi) \) demands that \( \varphi \) holds at the endpoint \( \delta \) of the continuous phase, and \( \text{cont}_c(\varphi) \) demands that \( \varphi \) holds at all points of time during the continuous phase \([0, \delta]\). Subformulas of \( R \) that are not the scope of one of these operators refer to the variable environment of the source state. All variable environments to satisfy \( R \) encode therefore a valid transition.
5. ILLUSTRATING EXAMPLE

We conclude the section with an illustrating example. The following program shows a bouncing ball whose velocity is reduced by half whenever it bounces on the floor. Initially, the ball starts at height \(h=10.0\) with velocity \(v=0.0\), and no bounces appeared so far \((n=0)\).

\[
\text{module bouncingball(cont real h,v,start, int n) \{}
\begin{align*}
h &= \text{start}; \\
v &= 0.0; \\
n &= 0; \\
\text{loop}() \text{: flow} \{
\begin{align*}
\text{der}(h) &\leftarrow v; \\
\text{der}(v) &\leftarrow -9.81; \\
\text{until}(h<0.0 \text{ and } v<0.0) \\
\text{next}(n) &= n+1; \\
\text{next}(v) &= v \cdot -0.5; \\
\text{bounce} &= \text{pause};
\end{align*}
\}
\}
\]
\]
\[
\text{The guarded actions that are obtained from this program are as follows: The set of guarded actions that have to be executed at the initial point of time are:}
\begin{align*}
1. \text{true } &\Rightarrow h = 10.0 \\
2. \text{true } &\Rightarrow v = 0.0 \\
3. \text{true } &\Rightarrow n = 0 \\
4. \text{true } &\Rightarrow \text{der}(h) \leftarrow v; \\
5. \text{true } &\Rightarrow \text{der}(v) \leftarrow -9.81; \\
6. \text{true } &\Rightarrow \text{release}(h<0.0 \text{ and } v<0.0); \\
7. \text{cont}_w(h<0.0 \text{ and } v<0.0) &\Rightarrow \text{next}(\text{Init}) = \text{true}; \\
8. \neg\text{cont}_w(h<0.0 \text{ and } v<0.0) &\Rightarrow \text{next}(\text{Init}) = \text{true};
\end{align*}
\]
After the initial point of time, the following guarded actions must be repeatedly executed:
\begin{align*}
1. (\text{Init } \lor \text{bounce }) \land \text{cont}_w(h<0.0 \text{ and } v<0.0) &\Rightarrow \text{next}(w) = \text{true} \\
2. (w' \lor \text{Init } \lor \text{bounce }) \land \neg\text{cont}_w(h<0.0 \text{ and } v<0.0) &\Rightarrow \text{next}(w') = \text{true} \\
3. w &= \text{next(bounce)} = \text{true} \\
4. \text{Init } \lor \text{bounce } \lor w' &\Rightarrow \text{der}(h) \leftarrow v \\
5. \text{Init } \lor \text{bounce } \lor w' &\Rightarrow \text{der}(v) \leftarrow -9.81 \\
6. w &= \text{next}(n) = n+1 \\
7. w &= \text{next}(v) = v \cdot -0.5 \\
8. \text{Init } \lor \text{bounce } \lor w' &\Rightarrow \text{release}(h<0.0 \land v<0.0)
\end{align*}
As explained in the previous section, we could now compute the symbolic representation of the transition relation of these guarded actions. Instead of listing this transition relation, we can use the symbolic representation of the transition relation of these guarded actions.

The EFSM representation has the drawback that it might suffer from a state space explosion since even the number of control flow states might grow exponentially with the size of the program. However, in practice, this does usually not happen (while the state space explosion of the data flow is typical), and therefore the EFSM computation is often feasible.

The EFSM representation has the important advantage that one can solve the different ODEs state by state to obtain another EFSM that contains no more derivation flows. In case of the EFSM of the bouncing ball, the continuous flow assignments \(\text{der}(h) \leftarrow v\) and \(\text{der}(v) \leftarrow -9.81\) can be replaced with the direct flow assignments \(v \leftarrow -9.81 \cdot t + v(0)\) and \(h \leftarrow -9.81 \cdot 0.5 \cdot t^2 + v(0) \cdot t + h(0)\). As explained in the previous section, this greatly simplifies the construction of the transition relation.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an extension of the synchronous language Quartz that allows us to describe hybrid systems in terms of convenient program descriptions. We presented new kinds of variables, actions and statements to deal with the continuous dynamics of hybrid systems, and defined their operational semantics in terms of SOS rules. Furthermore, we showed, how the hybrid Quartz programs can be automatically translated to symbolic representations of hybrid systems, which can be directly used for symbolic analysis like model checking. This symbolic representation will be the basis for our future work on the verification of hybrid systems.

References


As explained in the previous section, we could now compute the symbolic representation of the transition relation of these guarded actions. Instead of listing this transition relation, we consider the parallel composition of two bouncing-balls given in the following module in order to visualize the functionality of the flow statement.

```plaintext
module twoballs(cont real h1,h2,v1,v2,start1,start2)

  start1 = 10.0; start2 = 12.0;
  int n1,n2;
  Init: pause;
  { b = bouncingball(h1,v1,start1,n1);
    } ||
    { b = bouncingball(h2,v2,start2,n2);
      } ||
      { loop
        bounce = n1+n2;
        l:pause;
      }

end
```

The EFSM of `twoballs` is shown in Figure 5. The EFSM representation has the drawback that it might suffer from a state space explosion since even the number of control flow states might grow exponentially with the size of the program. However, in practice, this does usually not happen (while the state space explosion of the data flow is typical), and therefore the EFSM computation is often feasible.

The EFSM representation has the important advantage that one can solve the different ODEs state by state to obtain another EFSM that contains no more derivation flows. In case of the EFSM of the bouncing ball, the continuous flow assignments

\[
\text{der}(h) \leftarrow v \\
\text{der}(v) \leftarrow -9.81
\]

that appear in two boxes, can be replaced with

\[
\text{v} \leftarrow -9.81 \times t + v(0) \\
\text{h} \leftarrow - 9.81 \times 0.5 \times t^2 + v(0) \times t + h(0)
\]

As explained in the previous section, this greatly simplifies the construction of the transition relation.

### Abbreviations

\[
(h1<0.0 \text{ and } v1 \leq 0.0) := \sigma_1 \\
(h2<0.0 \text{ and } v2 \leq 0.0) := \sigma_2
\]

![Figure 5: EFSM of `twoballs`]

50