Integrating System Descriptions by Clocked Guarded Actions

Jens Brandt*, Mike Gemünde*, Klaus Schneider*, Sandeep K. Shukla† and Jean-Pierre Talpin‡
* *Department of Computer Science, University of Kaiserslautern, http://es.cs.uni-kl.de*
† *Electrical and Computer Engineering, Virginia Tech, http://www.fermat.ece.vt.edu*
‡ *INRIA, Unité de Recherche Rennes-Bretagne-Atlantique, http://www.irisa.fr/espresso*

***Abstract*—For the description of reactive systems, there is a large number of languages and formalisms, and depending on a particular application or design phase, one of them may be better suited than another one. In the design of reactive systems, their integration is a key issue for a system-wide simulation, analysis and verification.**

In this paper, we propose *clocked guarded actions* as a unified intermediate representation for several synchronous and asynchronous models in the design flow, which does not only provide an integrated description but it also allows designers to share common components of the tool infrastructure. Furthermore, we sketch how various models can be translated to this intermediate format, and how the intermediate format can be used for verification (by symbolic model checking) and integrated simulation (by SystemC).

I. Introduction

For the description of reactive systems, a large number of languages and formalisms have been proposed over the years, ranging from discrete-event models such as SystemC [23, 29], synchronous languages [3, 18] such as Esterel [6], Quartz [34] or Lustre, polychronous specifications such as SIGNAL [16] to untimed formalisms such as SHIM [14] or static dataflow [26, 27]. All these languages focus on different aspects of the system, and they differ in their abstraction levels and underlying models of computation (MoC) [28]. Depending on a particular application or design phase, one MoC may be better suited than another one. Due to this, reactive systems are often based on a heterogeneous model using different MoCs, and their *integration* [1, 17, 24, 28] has gained much interest in recent years.

The integration of different MoCs is a key issue in the design of reactive systems for several reasons. First, it is essential to obtain an integrated model, which covers all the parts relying on different MoCs, so that the composed system can be simulated and system-wide properties can be analyzed. Second, creating the tool infrastructure for all different MoCs is a difficult and time-consuming task that requires experts in the source languages, program analysis, and target architectures.

A classical solution is to use a common intermediate representation, which bridges the gap between powerful programming languages with complex semantics and the low-level description of the target code. This integration must be based on a common meta-MoC, in which all integrated MoCs are implemented. Such an intermediate representation has many advantages. It does not only provide an integrated description but it also allows designers to share common components of the tool infrastructure: new input languages can be added by simply implementing the front-end; additional back-ends support new target architectures.

In this paper, we propose *clocked guarded actions* as such an intermediate representation for various MoCs used in the domain of reactive systems. This representation is in the spirit of guarded commands, a well-established concept for the description of concurrent systems. With a theoretical background in term rewriting systems, they have been not only used in many specification and verification formalisms (e. g. Dijkstra's guarded commands [12], Unity [11], Murφ [13]) but they have also shown their power in hardware and software synthesis (e. g. Concurrent Action-Oriented Specifications [22]).

There are several contributions in this paper. The first one is the definition of an intermediate representation based on clocked guarded actions. In order to support our approach, we sketch a design flow based on it (see Figure 1), which leads to the other contributions: We show how models can be translated to clocked guarded actions, and how they can be used for symbolic model checking by SMV and simulation in SystemC.

Our integration significantly differs from previous approaches. Whereas the tagged signal model [1, 28] only gave a general framework for comparing different MoCs, environments such as Ptolemy [15], ForSyDe [24], HetSC [21], SystemC-H [30] or SystemMoC [20] embed various different MoCs in their host languages (i. e. Java, Haskell or SystemC). Thereby, all models get an operational semantics by these languages, which allows an integrated simulation. However, analysis and hardware-software (co)synthesis may become harder since the internal representation primarily aims at an efficient or/and flexible simulation.

More related are the approaches that establish a common intermediate format in the context of synchronous languages, in particular the multi-clock declarative code DC+ [33]. This format adds clock hierarchies to traditional DC [19], the privileged interchange format for hardware circuit synthesis, symbolic verification and optimization. The format reflects declarative or data flow synchronous programs like Lustre, as well as the equational representations of imperative synchronous programs. The underlying idea is the definition of flows by equations governed by clock hierarchies. From the overall idea, the intermediate representation in this paper is very similar but it has a broader scope: it does not only aim at just integrating synchronous models but also

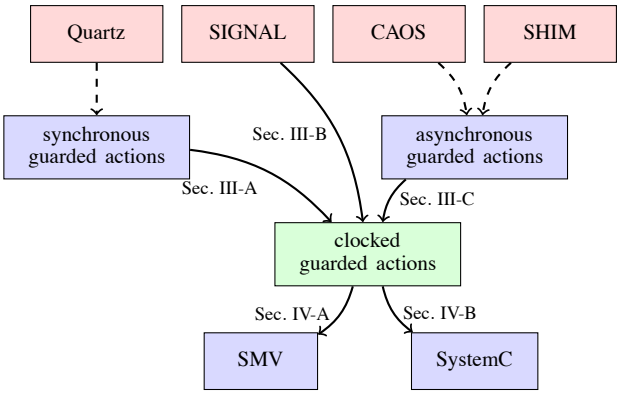


Figure 1. Design Flow for Clocked Guarded Actions

asynchronous ones, such as CAOS [9] or SHIM [14]. To this end, we have to generalize DC+, with respect to the definitions of flows and clocks (covered by the clocked guarded actions) and with respect to the semantic properties they need to fulfill.

The rest of this paper is structured as follows: Section II presents clocked guarded actions and details about our intermediate representation. The following sections focus on the design flow (see Figure 1): Section III sketches how various languages can be translated to them, while Section IV shows how they can be used for verification and simulation. Finally, Section V concludes with a short summary.

II. Intermediate Representation

A. Clocked Guarded Actions

Guarded commands or guarded actions are a well-established concept for the description of concurrent systems. As already stated in the introduction, they have been used in many domains. Basically, they consist of a guard γ and actions $\mathcal{A}$, i.e. they have the form $\langle \gamma \Rightarrow \mathcal{A} \rangle$, where $\mathcal{A}$ usually modifies the variables over which the guards are defined. The intuition behind guarded actions is that the body $\mathcal{A}$ is executed if its guard evaluates to true in the current state.

Usually guarded actions are seen as an asynchronous model. In the current state, the guards of all actions are checked and subsequently, any subset of the activated actions is selected for execution. Inside the body, there are multiple statements which are considered to execute in parallel.

However, guarded actions have also been successfully used in the domain of synchronous languages [8, 18, 19]. The variant used there differs from the traditional behavior: the synchronous model of computation does not only fire all activated actions, but the execution of the actions is postulated to happen simultaneously to the evaluation of the guards so that there may be interdependencies among guards and actions. In practice, this means that the execution follows the data dependencies between the actions.

Clocked guarded actions (CGA), which we propose in this paper, provide a basis to integrate both variants. As the name suggests, they are defined over a set of explicitly declared *clocks* $\mathcal{C}$, which define logical timescales the system uses to synchronize its computations so that asynchrony and synchrony in the system can be precisely described. Technically, they are signals of Boolean type which mark the instants. The data flow is based on a set of explicitly declared variables $\mathcal{V}$. Each variable $x \in \mathcal{V}$ is related to a clock $c \in \mathcal{C}$, which will be referred to as its clock $\widehat{x}$ in the following. In designs, the first instant usually differs from all other ones since additional behavior for initialization must be done. In the following presentation, we use the expression $\mathsf{init}(C)$ for a clock C that exactly holds the first time C ticks.

Variable values are determined by assignments, whereas clocks are only described by assumptions: every execution that complies to them is considered to be a valid one. In our intermediate representation clocked guarded actions have one of the following forms:

$$\begin{array}{lllll} (a) & \gamma & \Rightarrow & x = \tau \\ (b) & \gamma & \Rightarrow & \texttt{next}(x) = \tau \\ (c) & \gamma & \Rightarrow & \texttt{assume}(\sigma) \end{array}$$

The first action (a) is an *immediate assignment*, which changes the value of the variable x in the given instant to the value of the expression τ. It implicitly imposes the constraint $\gamma \rightarrow \widehat{x}$: the clock of x must hold whenever x is assigned. The delayed assignment (b) evaluates the expression τ in the given instant but changes the value of the variable x the next time clock $\widehat{x}$ ticks. Thus, no additional constraint is imposed by a delayed assignment since the instant where the variable is updated is still defined by its clock. Finally, the assumption (c) determines a Boolean condition which has to hold at the given instant.

Clocked guarded actions do not only fix the values of the signals but they also express causality. Whereas the immediate assignment $\langle \gamma \Rightarrow x = \tau \rangle$ implies a data dependency from τ to x, the corresponding assumption $\langle \gamma \Rightarrow \texttt{assume}(x = \tau)$ is undirected in that sense.

B. Additional Semantic Properties

The semantics of clock guarded actions is a simple model, which logically describes the behavior. In order to fit all considered input models, it does not require that the clocked guarded actions fulfill further conditions, e.g. that they are deterministic, constructive or clock consistent. However, as the semantics of some languages are inherently linked to some of these conditions, we have to consider them in the intermediate representation as well. For instance, if we optimize a Quartz program (which is considered to be constructive), we must ensure that this property is maintained by transformations. Otherwise, we would not be able to cover the full semantics of the input models.

Hence, our intermediate representation does not only contain a set of clocked guarded actions $\mathcal{G}$ but also a set of semantic properties $\mathcal{P}$ so that the pair $(\mathcal{G}, \mathcal{P})$ precisely describes the original model. In order to capture Quartz,

Lustre or SIGNAL, we are interested in the following properties related to the data-flow and the clocks.

Causality: For systems based on the synchronous abstraction causality is an important property. Obviously, due to the interrelation of clocks and variables, guards and actions in the intermediate representation depend on each other. These dependencies may create causal cycles, which make it impossible to constructively determine the behavior of the system. Causality analysis is well studied in the context of synchronous systems [35, 36], and for our clocked guarded actions, we handle it similarly: we call a system *constructive* (or causally correct) if we can infer clocks and values incrementally only relying on already known values, i.e. we do not need to guess for a valid execution.

Clock Consistency: For multi-clock systems, clock consistency is an important property, which ensures that the clock definitions are satisfiable. In that context, endochrony [2, 16, 31] is another important property. It ensures that the system can compute the clocks internally in each instant only from the values. These systems are based on a linear model of time and therefore, they only need a single clock (commonly called the master trigger) which drives the execution.

Other languages may come up with additional semantic constraints, and our intermediate representation is made extensible to integrate them.

C. Structure of AIF$^+$

Clocked guarded actions and the additional semantic properties are the basis of our intermediate format AIF$^+$, which we will briefly sketch in this section. AIF$^+$ extends the single-clocked synchronous intermediate format AIF [8], which was primarily developed as an interchange format between the tools of the Averest system. AIF$^+$ and AIF share many parts so that existing tool infrastructure can be reused. This is not only true for the conceptual design decisions, where the same set of program expressions and the same type system are used, but also for the technical ones, where the same XML-based concrete syntax is used for common parts.

In the following, we only give a brief overview of the structure, which is sufficient to understand the rest of the paper. In detail, AIF$^+$ consist of the following parts:

- a *name* of the described module,
- an *interface*, which contains a list of input and output variable declarations and additional clocks,
- *local declarations*, which contain additional variables and clocks used by the actions but not declared in the interface,
- an *abbreviation* table, which stores a list of common subexpressions (experience has shown that especially Boolean expressions in the guards are often shared, and that the size can be significantly reduced for practical examples),
- the *behavior* itself, which is described by a set of clocked guarded actions (over the variables and clocks declared above and the given abbreviations), and
- *absence reactions*, which define the values of variables if no guarded action explicitly sets them (e.g. event variables may be reset to a default value, while memorized variables keep their old values)
- *specifications*, which contain the semantic properties described in the previous section and other verification goals, either implicitly derived from the program (arithmetic overflow checks) or explicitly given by the developer.

III. Translating to Clocked Guarded Actions

In this section, we show how various models can be compiled to clocked guarded actions. For some of them, we make use of existing translations (as cited in the subsections) which transform systems to some kind of guarded actions. Hence, the starting point are models which have a similar syntax (see Figure 1). Hence, we only need a few adaptations to achieve the intended integration, which lets us focus on the core: the mapping of clocks and synchronizations.

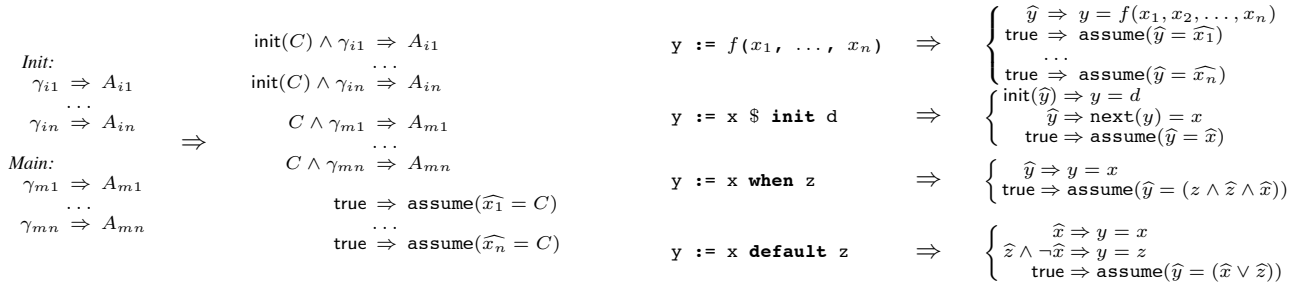
The rest of this section describes the foundations of different classes of systems which we will consider in this paper, namely single-clocked synchronous programs in Section III-A, polychronous specifications in Section III-B and finally concurrent action-oriented specifications in Section III-C. For each of them, we briefly describe their semantics before we show how they can be represented by clocked guarded actions as introduced in the previous section.

A. Synchronous Programs

The synchronous *model of computation* [3, 18] assumes that the execution of programs consists of a totally ordered sequence of instants. In each of these instants, the system reads its inputs and computes and writes its outputs. In the single-clocked case, which we will consider in the following, all signals have a value in every instant. The introduction of this logical time scale is not only the key for a straightforward translation of synchronous programs to hardware circuits [4, 32, 34]; it also provides a very convenient programming model, which allows compilers to generate *deterministic* single-threaded code from multi-threaded synchronous programs [5].

In general, synchronous programs such as Esterel, Lustre or Quartz can be translated to synchronous guarded actions [8]. This translation extracts all actions (assignments, assumptions and assertions) of the program and computes for each of them a trigger condition from its context in the program. As already stated in the previous section, the *semantics of synchronous guarded actions* implements the synchronous model of computation and fires all activated actions simultaneously in each macro step. Synchronous guarded actions without causality problems are always deterministic since there is no choice due to the firing of all actions.

As expected, translating synchronous guarded actions to clocked guarded actions is straightforward. We only need to introduce a single clock C, which serves as the clock for



(a) Translating AIF to AIF$^+$ *(b) Translating Signal to AIF$^+$*

Figure 2. Translation from AIF and SIGNAL to AIF$^+$

all variables of the system. This clock C is then added as an additional clause to the guard of all actions. Additional clock constraints ensure that this clock is the clock of all variables: Whenever C holds, the original system performs a computation step. The general principle of the translation is shown in Figure 2 (a). A system in the AIF-format, which is based on synchronous guarded actions, contains a set of guarded actions which are executed initially at the first instant and guarded actions which are executed in any instant. The guards of the initial guarded actions are strengthened by $\mathsf{init}(C)$ and the other guards just by C. Thus, they are now executed at any tick of C. The clock constraints ensure that all variables have the same clock. All programs of this class should be endochronous and constructive, which is additionally stored in the intermediate representation.

B. Polychronous Specifications

Polychronous specifications [16, 25] as implemented by SIGNAL use several clocks, which means that signals do not need to be present in all instants. Second, in contrast to synchronous systems, polychronous models are not based on a linear model of time, so that the reactions of a polychronous system are partially ordered. Two instants are only compared on the time scale if both contain events on a shared signal x.

Polychronous specifications are usually considered to be *relational* and not *functional*: even in the presence of the same input values, various temporal alignments, which comply to the clock constraints, may lead to different output values. Hence, polychronous models are generally nondeterministic and should be seen as specifications, which describe a set of acceptable implementations. State-of-the art tools check for determinism before synthesis.

SIGNAL programs consist of a composition of several processes, where each process is either given by a set of equations or a composition of other processes. Each processes has an input interface consisting of input signals, an output interface consisting of output signals and several possible internal signals. The equations can be built from one of the following primitive operators:

- *Function:* A function `y := f(x1, ..., xn)` determined the output y by applying the given function f to the input values. Additionally, this process requires that all inputs and the output have the same clock.
- *Delay:* The delay operator `y := x $ init d` has exactly one input `x` and one output `y`. Each time a new incoming value arrives, it outputs the previously stored value and stores the new one. Intially, the buffer simply returns the given value `d`. By definition, the input and the output have the same clock, i.e. $\widehat{x} = \widehat{y}$.
- *When:* The downsample operator `y := x when z` has two inputs, `x` of arbitrary type and `z` of Boolean type, and one output port `y`. Each time a new `x` arrives, it checks whether there is an input at `z`. If there is one and if it is `true`, a new output event with the value of `x` is emitted for `y`. In all other cases, no event will be produced.
- *Default:* The merge operator `y := x default z` has two input ports `x` and `z` and a single output port `y`. Each time inputs arrive at `x` and `z`, they will be forwarded to `y`. If there are events present at both ports in a particular instant, `x` will be forwarded, and `z` will be discarded.

Programs may contain more clock constraints to restrict the behaviors of clocks. They are very general: for example, a clock can be declared to be a subclock of another one $\widehat{x} \rightarrow \widehat{y}$.

Polychronous SIGNAL programs can be structurally translated to clocked guarded actions by translating each operator separately as shown in Figure 2 (b):

- *Function:* The function operator is applied to the inputs and produces the output value. All variables are forced to have the same clock for a function application.
- *Delay:* The translation of the delay operator is split up in two cases. (1) The first value that is produced by this operator when its clock initially ticks is the value that is given by the constant `d`. (2) In all other steps the value of `x` of the last tick is used. Therefore, we model this behavior by transferring the value of `x` to the following step in every tick. The constraint ensures that both variables have the same clock.
- *When:* The sample operator **when** transfers the value of `x` to `y` whenever it is needed. The clock constraint

ensures that $\widehat{y}$ only holds when both inputs are present and `z` holds.

- *Default:* The **`default`** operator merges two signals with priority for the first one. Therefore, if the first input is present, it is passed to `x`. If it is not present, but the second one is, the second one is passed through. The clock constraint ensures that $\widehat{y}$ only holds when at least one of the inputs is present.

Additional clock constraints ϕ should hold in every instant. Thus, we guard them by true and generate the following guarded action $\langle \mathsf{true} \Rightarrow \phi \rangle$.

Note that there is an elementary difference between AIF$^+$ and Signal (and thereby also DC+). In SIGNAL, it is not possible to read a variable when its clock does not hold. Thus, the clock of a signal does identify when there is a value and when not. In AIF$^+$, every variable can be read in every instant and the variable will have the value that was assigned to it when its clock holds the last time. Thus, in AIF$^+$ the clock means a potential value change. Nevertheless, the translation of Signal to AIF$^+$ works because it ensures that every variable is read or set if and only if its clock holds. This can be easily checked in Figure 2 (b). However, interaction with other computational models is done in the model of AIF$^+$ by allowing to read the variable in every instant.

C. Concurrent Action-Oriented Specifications

Concurrent Action-Oriented Specifications (CAOS) aim at describing the data-flow of systems (in particular hardware circuits) without fixing its timing but only its causalities. Thereby, developers can defer the difficult task of global scheduling and coordination to the compiler.

Instead of their full syntax and semantics, we take simple asynchronous guarded actions in the form of rules and methods. This corresponds to the intermediate representation a CAOS compiler obtains after dismantling the source program. The behavior is described by a set of *rules*, which are guarded atomic actions of the form:

$$\textbf{rule}\ r_i\ \textbf{when}(\gamma_{r_i})\ B_i$$

Thereby, γ_{r_i} is the guard and B_i the body of rule r_i. CAOS provides two kinds of assignments: while wire assignments are immediately visible, register assignments are committed with the current state update. Hence, the body of a rule B_i is a set of synchronous guarded actions of the form $\langle \gamma \Rightarrow x = \tau \rangle$ (for an immediate assignment) or $\langle \gamma \Rightarrow \mathtt{next}(x) = \tau \rangle$ (for a delayed assignment) as known from synchronous programs (see Section III-A).

For the interaction with the environment, the target model makes use of so-called methods, which are parameterized rules. In addition to the local variables, the actions of a method have access to the variables specified in its parameter list, which may contain inputs and outputs.

$$\textbf{method}\ m_i(p_{i1}, p_{i2}, \ldots)\ \textbf{when}(\gamma_{m_i})\ B_1$$

As already described in Section II, the semantics of the asynchronous guarded actions is as follows: first the guards

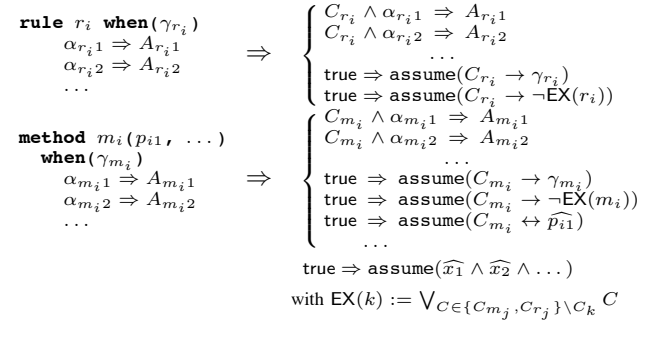


Figure 3. Translation from CAOS to AIF$^+$

of all actions are evaluated with respect to the current state, then an arbitrary activated one is chosen and its body is executed. Inside the body, there are multiple synchronous actions, which are considered to execute in parallel. Hence, let q_0 be the initial state of the system, and $q \xrightarrow{S} q'$ indicate that action S transforms the system in state q to state q'. Then, a run of a model is a sequence of system states $\langle q_0, q_1 \ldots \rangle$ where $q_i \xrightarrow{S_x} q_{i+1}$ and $\mathtt{when}(\gamma_x)\ \mathcal{C}_x$ is an arbitrary action which is activated in state q_i, i.e. $q_i(\gamma_x) =$ true. Obviously, the system description is nondeterministic: even in the presence of the same inputs, which lead to the same activation of guards, the system can produce different outputs by choosing different activated actions. Models consisting of asynchronous guarded actions are generally intended to be specifications, which describe a set of acceptable implementations.

The translation of CAOS to AIF$^+$ is illustrated in Figure 3. In order to model the nondeterminism, a clock C_r for each rule r and a clock C_m for each method m is introduced. A tick of such a clock models an execution of the rule or method. First, the rules and the methods are translated on their own as shown in the figure. The guard of each action of a rule r is strengthened by the clock C_r that is associated with the rule. Thus, all actions of the rule are just executed when the clock ticks and the clock constraint $C_r \rightarrow \gamma_r$ ensures that the clock can only tick when the rule is enabled, i.e. γ_r holds. The reference semantics requires that at most one rule is executed at once. The second clock constraint for a rule ensures that its clock can only tick when no clock of an other rule or method does. Methods are translated accordingly, but the input and output variables of a method have a different clock than all internal variables: they only change their value when the method is executed. This restriction is added by clock constraints for the clocks of the variables. In this way new parameters can only be given if and only if the method is executed. After translating the rules and methods, the clocks for the local variables need to be fixed. The clock constraint ensures that the clock of all local variables (identified with $x_1, x_2, \ldots$) ticks at each instant. This is because the semantics of register assignments in CAOS require that the changes are visible right after the

execution of the rule or method, thus for the next execution instant. This concludes the translation of CAOS to clocked guarded actions.

IV. Translating From Clocked Guarded Actions

From our intermediate representation of guarded actions, many synthesis targets can be thought of. In the following, we sketch the translation to two exemplary targets, a symbolic transition system, which is suitable for formal verification of program properties by symbolic model checking, and the translation to SystemC code, which can be used for an integrated simulation of the system. Similar to the previous section, we adapt previous work [7, 34] for synchronous languages and extend it by multiple clocks.

A. Symbolic Model Checking

For symbolic model checking, the system generally needs to be represented by a transition system. This basically consists of a triple $(\mathcal{S}, \mathcal{I}, \mathcal{T})$ with set of states $\mathcal{S}$, initial states $\mathcal{I} \subseteq \mathcal{S}$ and a transition relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$. Each state s is a mapping from variables to values, i.e. s assigns to each variable a value of its domain. As we aim for a symbolic description, we describe the initial states and the transition relation by propositional formulas $\Phi_{\mathcal{I}}$ and $\Phi_{\mathcal{T}}$, which are their characteristic functions.

For the presentation of the translation, assume that our intermediate representation contains immediate and delayed actions for each variable x of the following form

$$\begin{array}{llll} (\gamma_1, \mathtt{x} = \tau_1), & \ldots, & (\gamma_p, \mathtt{x} = \tau_p) \\ (\chi_1, \mathtt{next(x)} = \pi_1), & \ldots, & (\chi_q, \mathtt{next(x)} = \pi_q) \end{array}$$

Figure 4 sketches the translation of the immediate and delayed actions writing variable x to clauses used for the description of a symbolic transition system.

As one might expect first, the construction of a transition system is not straightforward. Since delayed actions generally predetermine a new value for the next point of time $\widehat{x}$ is present while other actions still read its current value. To circumvent this problem, we introduce an auxiliary variable x' called the *the carrier of* x to capture delayed assignments at the previous point of time [34].

Before considering the constraints for x', let us consider the invariant for x (Invar_x): clearly, we have to demand that x equals to τ_i whenever the guard γ_i of an immediate assignment $x = \tau_i$ holds. In case no guard of an immediate assignment to x holds, we have to distinguish whether x is expected to tick or not: if this is the case, its reaction to absence determines the value, which is covered by the equations for the carrier variable x'. If x is absent, it just keeps its old value so that other actions can still read it - which is covered by the clause Trans_x.

The meaning of x' is as follows: x' captures all of the delayed assignments $\mathsf{next}(x) = \pi_j$ to x, that is whenever $\mathsf{next}(x) = \pi_j$ is executed, we evaluate the right hand side π_j at the current point of time and assign this value to x' (not yet to x) at the next point of time. Hence, x' is determined by the delayed assignments to x. This leaves open what the initial value of x' should be, so we additionally define the initial value of x' as the default value of x.

$$\begin{array}{rcl} \mathsf{Invar}_x & :\equiv & \left(\begin{array}{l} \bigwedge_{j=1}^{p} (\gamma_j \rightarrow x = \tau_j) \wedge \\ \\ \left(\bigwedge_{j=1}^{p} \neg\gamma_j \right) \wedge \widehat{x} \rightarrow x = x' \end{array} \right) \\ \\ \mathsf{Init}_{x'} & :\equiv & \mathsf{Default}(x) \\ \\ \mathsf{Trans}_x & :\equiv & \neg\mathsf{next}(\widehat{x}) \rightarrow \mathsf{next}(x) = x \\ \\ \mathsf{Trans}_{x'} & :\equiv & \left(\begin{array}{l} \bigwedge_{j=1}^{q} (\chi_j \rightarrow \mathsf{next}(x') = \pi_j) \wedge \\ \\ \left(\bigwedge_{j=1}^{q} \neg\chi_j \right) \wedge \mathsf{next}(\widehat{x}) \rightarrow \mathsf{next}(x') = x \\ \\ \left(\bigwedge_{j=1}^{q} \neg\chi_j \right) \wedge \neg\mathsf{next}(\widehat{x}) \rightarrow \mathsf{next}(x') = x' \end{array} \right) \end{array}$$

Figure 4. Transition Relation for x

By this definition of the initial value of x', the initial value of x is correct. In later macro steps, if one of the immediate assignments to x is enabled, then this assignment determines the value of x at this point of time as given by the invariant equation for x. Otherwise, a delayed assignment $\mathsf{next}(x) = \pi_j$ may have been executed at some previous point of time. If so, then x' has now the value that has been obtained by evaluating π_j at the previous point of time, and the invariant equation takes this value via x'.

For the additional assumptions, the translation is straightforward. Assume that the intermediate representation contains the following set of additional assumptions

$$(\delta_1, \mathtt{assume}(\sigma_1)), \quad \ldots, \quad (\delta_r, \mathtt{assume}(\sigma_r))$$

They are translated to the following clause:

$$\mathsf{Assume} \quad :\equiv \quad \textstyle\bigwedge_{i=1,\ldots,r} \delta_r \rightarrow \sigma$$

The final result is then the conjunction of the clauses of all writable variables $\mathcal{V}_W$ together with the additional assumptions, i.e.

$$\begin{array}{rcl} \Phi_{\mathcal{I}} & = & \mathsf{Assume} \wedge \bigwedge_{\mathcal{V}_W} (\mathsf{Init}_{x'} \wedge \mathsf{Invar}_x) \\ \Phi_{\mathcal{T}} & = & \mathsf{Assume} \wedge \bigwedge_{\mathcal{V}_W} (\mathsf{Trans}_x \wedge \mathsf{Trans}_{x'} \wedge \mathsf{Invar}_x) \end{array}$$

B. SystemC Simulation

The simulation semantics of SystemC is based on the discrete-event model of computation [10], where reactions of the system are triggered by events. All threads that are sensitive to a specified set of events are activated and produce new events during their execution. Updates of variables are not immediately visible, but become visible in the next delta cycle.

We start the translation by the definition of a global clock that ticks in each instant and drives all the computation. Thus, we require that the processed model is tagged to be *endochronous*. In SystemC, this clock is implemented by a single `sc_clock` at the uppermost level, and all other components are connected to this clock. Hence, the translations of the macro steps of the synchronous program in SystemC are triggered by this clock, while the micro steps are triggered by signal changes in the delta cycles. For this reason, input and output variables of the synchronous program are mapped to input signals (`sc_in`) and output signals (`sc_out`) of SystemC of the corresponding type.

Additionally, we declare signals for all other clocks of the system. They are inputs since the clock constraints (as given by `assume`) do not give an operational description of the clocks, but can be only checked in the system. The clock calculus for SIGNAL [16] or scheduler creation for CAOS [9] aim at creating exactly these schedulers which give an operational description of the clocks. Although not covered in the following, their result can be linked to the system description so that clocks are driven by the system itself.

The translation of the synchronous guarded actions to SystemC processes is however not as simple as one might expect. The basic idea is to map guarded actions to methods which are sensitive to the read variables so that the guarded action is re-evaluated each time one of the variables it depends on changes. For a *constructive* model it is guaranteed that the simulation does not hang up in delta-cycles.

The translation to SystemC must tackle the following two problems: (1) As SystemC does not allow a signal to have multiple drivers, all immediate and delayed actions must be grouped by their target variables. (2) The SystemC simulation semantics can lead to spurious updates of variables (in the AIF$^+$ context), since threads are always triggered if some variables in the sensitivity list have been updated - even if they are changed once more in later delta cycles. As actions might be spuriously activated, it must be ensured that at least one action is activated in each instant, which sets the final value. Both problems are handled in a similar way as the translation to the transition system presented in the previous section: we create an additional variable `nxt_x` for each variable x to record values from their delayed assignments, and group all actions in the same way as for the transition system.

With these considerations, the translation of the immediate guarded actions $\langle \gamma \Rightarrow x = \tau_i \rangle$ is straightforward: We translate each group of actions into an asynchronous thread in SystemC, which is sensitive to all signals read by these actions (variables appearing in the guards γ_i or in the right-hand sides τ_i). Thereby, all actions are implemented by an `if`-block except for the last one, which handles the case that no action fires. Since the immediate actions should become immediately visible, the new value can be immediately written to the variable with the help of a call to `x.write(...)`. Analogously, the evaluations of the guard γ_i and the right-hand side of the assignment τ_i make use of the `read` methods of the other signals. The left-hand side of Figure 5 shows the general structure of such a thread and shows a small example.

Delayed actions $\langle \gamma \Rightarrow \texttt{next}(x) = \pi_j \rangle$ are handled differently: While the right-hand side is immediately evaluated, the assignment should only be visible in the following macro step and not yet in the current one. Hence, they do not take part in the fixpoint iteration. Therefore, we write their result to `nxt_x`. The transfer to the actual x is done in a *clocked thread* which is triggered by the clock of x. Thereby, signals changed by the delayed actions do not affect the current fixpoint iteration and vice versa.

V. Summary

In this paper, we proposed clocked guarded actions as a common intermediate representation in the design flow for various languages targeting reactive systems. On the one hand they have a simple structure and semantics, and on the other hand, they provide enough information for later synthesis. Furthermore, we showed for several languages how they can be translated into the intermediate representation, and we showed how they can serve as a starting point for a symbolic model checker and a SystemC simulation. Future work now targets other synthesis targets, in particular concurrent or distributed software implementations.

References

[1] A. Benveniste, B. Caillaud, L.P. Carloni, and A.L. Sangiovanni-Vincentelli. Tag machines. In W. Wolf, editor, *Embedded Software (EMSOFT)*, pages 255–263, Jersey City, New Jersey, USA, 2005. ACM.

[2] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[4] G. Berry. A hardware implementation of pure Esterel. In *Formal Methods in VLSI Design*, Miami, Florida, USA, 1991.

[5] G. Berry. Synchronous languages for hardware and software reactive systems. In C. Delgado Kloos and E. Cerny, editors, *Computer Hardware Description Languages and Their Applications (CHDL)*, Toledo, Spain, 1997. Chapman & Hall.

[6] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[7] J. Brandt, M. Gemünde, and K. Schneider. From synchronous guarded actions to SystemC. In M. Dietrich, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 187–196, Dresden, Germany, 2010. Fraunhofer Verlag.

```
void Module::compute_x () {
  while (true) {
    if (γ1)
      x.write (τ1);
    else if (γ2)
      x.write (τ2);
      ...
    else
      x.write (τn);
    wait ();
  }
}
```

```
void Module::compute_delayed_x () {
  while (true) {
    if (ξ1)
      nxt_x.write (π1);
    else if (ξ2)
      nxt_x.write (π2);
      ...
    else
      nxt_x.write (πn);
    wait ();
  }
}
```

```
void Module:: transfer_delayed () {
  /* transfer value for x */
  x.write (nxt_x);
  wait ();
}
```

Figure 5. Translation of Immediate and Delayed Actions

[8] J. Brandt and K. Schneider. Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, March 2011.

[9] J. Brandt, K. Schneider, and S.K. Shukla. Translating concurrent action oriented specifications to synchronous guarded actions. In J. Lee and B.R. Childers, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 47–56, Stockholm, Sweden, 2010. ACM.

[10] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition, 2008.

[11] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, USA, May 1989.

[12] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM (CACM)*, 18(8):453–457, 1975.

[13] D.L. Dill. The Murphi verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, New Jersey, USA, 1996. Springer.

[14] S.A. Edwards and O. Tardieu. SHIM: a deterministic model for heterogeneous embedded systems. In W. Wolf, editor, *Embedded Software (EMSOFT)*, pages 264–272, Jersey City, New Jersey, USA, 2005. ACM.

[15] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.

[16] A. Gamatié, T. Gautier, P. Le Guernic, and J.P. Talpin. Polychronous design of embedded real-time applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2), April 2007.

[17] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 18(6):742–760, June 1999.

[18] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[19] N. Halbwachs. *The Declarative Code DC, version 1.2a*, April 1998.

[20] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich. A SystemC-based design methodology for digital signal processing systems. *EURASIP Journal on Embedded Systems*, 2007:ID 47580, 2007.

[21] F. Herrera and E. Villar. A framework for heterogeneous specification and design of electronic embedded systems in SystemC. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3), 2007.

[22] J.C. Hoe and Arvind. Operation-centric hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 23(9):1277–1288, September 2004.

[23] IEEE Computer Society. *IEEE Standard SystemC Language Reference Manual*. New York, USA, December 2005. IEEE Std. 1666-2005.

[24] A. Jantsch. *Modeling Embedded Systems and SoCs*. Morgan Kaufmann, 2004.

[25] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[26] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[27] E.A. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[28] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 17(12):1217–1229, December 1998.

[29] Open SystemC Initiative. *SystemC Version 2.1 User's Guide*, 2005.

[30] H.D. Patel, S.K. Shukla, and R.A. Bergamaschi. Heterogeneous behavioral hierarchy extensions for SystemC. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 26(4):765–780, April 2007.

[31] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Application of Concurrency to System Design (ACSD)*, pages 67–76, Hamilton, Ontario, Canada, 2004. IEEE Computer Society.

[32] F. Rocheteau and N. Halbwachs. Pollux, a Lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Algorithms and Parallel VLSI Architectures II*, Bonas, France, 1991.

[33] ESPRIT Project: Safety Critical Embedded Systems (SACRES). *The Declarative Code DC+, version 1.4*, November 1997.

[34] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[35] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington, DC, USA, 2004. ACM.

[36] T.R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design Automation Conference (EDAC)*, pages 328–333, Paris, France, 1996. IEEE Computer Society.