Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions

Jens Brandt, Mike Gemünde, Klaus Schneider, Sandeep Shukla, Jean-Pierre Talpin

Des Autom Embed Syst
DOI 10.1007/s10617-012-9087-9

Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions

Jens Brandt · Mike Gemünde · Klaus Schneider · Sandeep K. Shukla · Jean-Pierre Talpin

Received: 6 January 2012 / Accepted: 21 June 2012

Abstract For the design of embedded systems, many languages are in use, which are based on different models of computation such as event-, data-, and clock-driven paradigms as well as paradigms without a clear notion of time. Systems composed of such heterogeneous components are hard to analyze so that mainly co-simulation by coupling different simulators has been considered so-far. In this article, we propose clocked guarded actions as a unique intermediate representation that can be used as a common basis for simulation, analysis, and synthesis. We show how synchronous, (untimed) asynchronous, and polychronous languages can be translated to clocked guarded actions to demonstrate that our intermediate representation is powerful enough to capture rather different models of computation. Having a unique and composable intermediate representation of these components at hand allows one a simple composition of these components. Moreover, we show how clocked guarded actions can be used for verification by symbolic model checking and simulation by SystemC.

Keywords Models of computation · Co-simulation · Synchronous vs. asynchronous models · Guarded command language

1 Introduction

For the design of embedded systems, a plethora of languages based on different models of computation (MoC) [3, 29, 46, 53, 54] have been proposed over the years. For example,

Submitted to special issue on languages, models and model based design for embedded systems.

J. Brandt (✉) · M. Gemünde · K. Schneider
Department of Computer Science, University of Kaiserslautern, 67653 Kaiserslautern, Germany
e-mail: brandt@cs.uni-kl.de
url: http://es.cs.uni-kl.de

S.K. Shukla
Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA
url: http://www.fermat.ece.vt.edu

J.-P. Talpin
INRIA, Unité de Recherche Rennes-Bretagne-Atlantique, Rennes, France
url: http://www.irisa.fr/espresso

Published online: 06 July 2012

languages like Verilog [42], VHDL [44], and SystemC [43] are based on an event-driven paradigm [18], synchronous languages [4, 32] such as Esterel [7, 9, 11], Lustre [17, 34], Quartz [64], and some statechart variants are based on clock-driven paradigms, polychronous languages like Signal [2, 28, 49] are based on a declarative and non-deterministic paradigm using several clocks, data flow languages like CAL [25] are based on data-driven paradigms [50–52], and others like SHIM [24] or most multi-threaded languages are based on asynchronous threads with a rendezvous-style communication [38, 39].

Depending on a particular application domain such as digital signal processing or reactive controllers, depending on the design task such as modeling, simulation, analysis or synthesis, and depending on the synthesis target such as digital hardware circuits, multi-threaded software or software for heterogeneous MPSoCs, the one or the other language might be preferable. For this reason, many existing components are given in different languages using different MoCs. The co-simulation of such heterogeneous systems has been widely considered [10, 55, 59, 67, 69] and covers also languages with different MoCs [20]. Co-simulation is also used to create virtual prototypes that are required to achieve hard time-to-market constraints.

However, co-simulation alone is not sufficient for a seamless design flow. Formal verification and a common synthesis of the different components require an *integration* that has gained a lot of interest in recent years [26, 31, 36, 37, 48, 56, 61, 62, 68]. Moreover, having a common description for the different components at hand, one can easily combine the components, e.g. one can create new components by using existing ones in a hierarchical way as known in block/schematic-oriented languages such as Simulink or those used by many tools for digital hardware circuit design. This way, one is no longer restricted in a parallel composition of the heterogeneous components by using appropriate wrappers. Instead, one can create a hierarchy of modules that combine modules based on different MoCs. This allows one to establish a design flow using several steps of refinement where asynchronous descriptions can become synchronous by adding a schedule to clocks, and synchronous ones may become asynchronous again when one considers the actions scheduled to one clock tick. Finally, one can re-use existing backend tools for synthesis and verification and of course, simulation would be greatly simplified: Instead of coupling different simulators, a single one could consider the entire system in a way it will later on be verified and synthesized.

A classical solution used e.g. in most compilers 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 intermediate representation must be based on a common model of computation which covers all MoCs that should be integrated without complex translations. Such an intermediate representation has many advantages. It not only achieves the above mentioned integration, but it also allows designers to share the tool infrastructure: new input languages can be added by simply implementing a corresponding front-end while existing back-end tools can be re-used.

In this article, we propose *clocked guarded actions* as an intermediate representation to cover various MoCs used for the design of embedded 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 conditional term rewriting systems [21, 30, 47], guarded actions have been not only used in many specification and verification formalisms (e.g. Dijkstra's guarded commands [22], Unity [19], Murphi [23]) but they have also shown their power in hardware and software synthesis (e.g. Bluespec [1, 40] and Concurrent Action-Oriented Specifications [41]).

To demonstrate the power of our approach, we sketch a design flow based on our intermediate representation (see Fig. 1): In particular, we show how different languages based

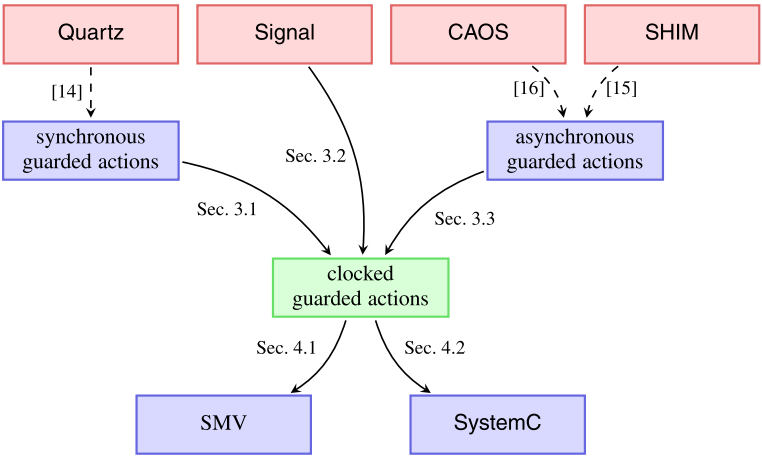


Fig. 1 Design flow for clocked guarded actions

on different MoCs can be translated to clocked guarded actions, and systems given as a set of clocked guarded actions can be used for formal verification by symbolic model checking using SMV and simulation using SystemC simulators.

This article is an extension of a previous paper [13], which introduced clocked guarded actions as a common intermediate representation. In this article, we extend our previous approach by introducing control-flow contexts, which allows us to nest components in an arbitrary behavioral hierarchy, i.e. we can *call* components from other components. Obviously, this has not only consequences for the intermediate representation itself but also related tasks: the translation to the intermediate format must be revised, and the procedure to link several components must be generalized.

The rest of this article is organized as follows: Sect. 2 defines the core of our approach, i.e., clocked guarded actions, components represented by a set of clocked guarded actions and an interface, and their meaning in terms of a denotational semantics. Section 3 shows how synchronous languages like Quartz [64], polychronous languages like Signal [2, 28, 49], and asynchronous languages like CAOS [66] can be translated to clocked guarded actions. Section 4 shows how clocked guarded actions can be used for verification and simulation. Section 5 illustrates our intermediate representation with the help of a case study. Finally, Sect. 6 considers some related work on the integration of components with different MoCs before we conclude the article with a short summary in Sect. 7.

2 Intermediate representation

This section presents our intermediate representation of components by clocked guarded actions. To this end, we define an extension of our Averest Intermediate Format AIF that is based on guarded actions without clocks and is used in our Averest framework[1] as intermediate representation for simulation, verification, analysis, transformations, and synthesis. Clocked guarded actions lead to AIF$^+$ which generalizes and extends the single-clocked synchronous intermediate format AIF [14] by clocks to cover polychronous and asynchronous systems.

[1] http://www.averest.org.

In the following, we only show fundamental design decisions of AIF$^+$. We start with some fundamental definitions in Sect. 2.1 before Sect. 2.2 describes the basic artifacts in our intermediate representation, namely components and their interfaces. Clocked guarded actions, which are the basis for the description of the behavior, are introduced in Sect. 2.3.

2.1 Foundations

Guarded commands or guarded actions are a well-established concept for the description of concurrent systems. As already stated in the introduction, they have already been used for many purposes. Usually, guarded actions are seen as an asynchronous model as follows: In the current state, the guards of all actions are evaluated and then an arbitrary subset of the enabled actions is selected for execution. The action to be executed may consist of several statements which are then executed in parallel. This kind of semantics defines an asynchronous system since the enabled actions may be executed, but have no need to be executed. As a result the execution is by definition in general non-deterministic.

Guarded actions have also been successfully used for synchronous languages [14, 33, 63]. In contrast to the traditional guarded actions as described above, synchronous guarded actions required that *all enabled actions are executed*. Thus, they define a deterministic model of computation.

Causality problems may occur in synchronous and asynchronous guarded actions if the actions are allowed to take immediate effects, i.e. if the executed action modify variables that are used in their guards. Causality analysis checks whether such cyclic dependencies can be constructively resolved which means in practice that the execution follows the data dependencies between the actions.

Clocked guarded actions (CGA), which we propose in this article, 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. The basis of the whole temporal model are so-called instants, i.e. the points of time where some event in the system occurs.

Definition 1 (Instants) Let $\mathcal{I}$ be the set of *instants* and let $\preceq \subseteq \mathcal{I} \times \mathcal{I}$ be a preorder on $\mathcal{I}$ such that for any two instants $I_1, I_2 \in \mathcal{E}$, we say $I_1 \preceq I_2$ iff I_1 *occurs before* I_2, or if both of them occur together. Let $\approx$ be the equivalence relation induced by $\preceq$: thus $I_1 \approx I_2$ iff $I_1 \preceq I_2 \wedge I_2 \preceq I_1$. We also define a precedence relation $\prec \subseteq \mathcal{I} \times \mathcal{I}$ on events such that $I_1 \prec I_2$ iff $I_1 \preceq I_2 \wedge \neg(I_1 \approx I_2)$.

The actions of programs are scheduled to a set of such instants (also called reactions or macro steps [35]). The actions that take place within an instant (sometimes called micro steps) are not explicitly ordered. Instead, micro steps are assumed to happen simultaneously, i.e. in the same variable environment. Hence, variables seem to be constant during the execution of the micro steps and only change synchronously at macro steps. From the semantical point of view, which postulates that a reaction is atomic, neither communication nor computation take time in this sense. In reality, all actions within an instant are executed according to their data dependencies to establish the illusion of zero-time computations.

With the help of this temporal framework, we define the behavior of the system. It is given by a set of signals $\mathcal{V}$, which reflect the state changes in the course of the execution.

Definition 2 (Data values and signals) Let $\mathcal{D}$ be the set of data values, i.e. all the values that program expressions may be evaluated to. A *signal* $x \in \mathcal{V}$ is a function that maps a totally ordered sequence of instants $C = \langle I_0, I_1, I_2, \ldots \rangle \subseteq \mathcal{I}$ with $I_i \prec I_{i+1}$ to the data values $\mathcal{D}$.

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.

Definition 3 (Clocks) For each signal x, let $\mathsf{Instants}(x) \subseteq \mathcal{I}$ be the domain of a signal x. This set gives rise to the *clock* $\widehat{x}$ of a signal x, which is the characteristic function of this set, i.e. it holds in instant $I \in \mathcal{I}$ iff $I \in \mathsf{Instants}(x)$. The signal x is said to be *present* in an instant iff $\widehat{x}$ holds, otherwise the signal x is *absent*. Furthermore, two signals x_1 and x_2 are *synchronous to each other* iff they have the same domain, i.e. $\mathsf{Instants}(x_1) = \mathsf{Instants}(x_2)$. The union (intersection) of two clocks C_1 and C_2 is the union (intersection) of the set of their instances.

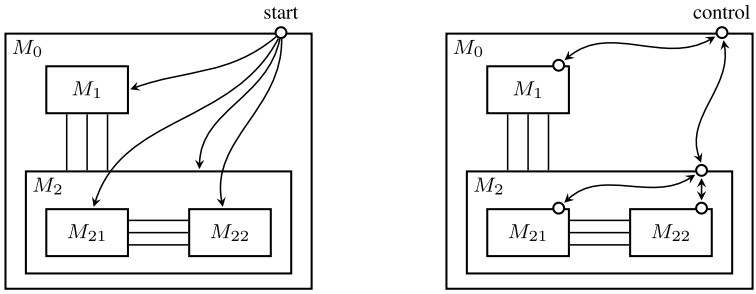
Clocks are fundamental elements of all synchronous languages, and they have a similar role in our approach. They define the possibly infinite set of instants at which the signal communicates a data value. As we have the underlying notion of perfectly synchronous instants [4], it is always possible to detect the absence of a signal (i.e. to be sure that an event for this signal does not arrive in the current instant) and based on this knowledge, initiate an action (the so-called reaction to absence).

2.2 Interfaces

The purpose of the proposed intermediate representation is to describe heterogeneous systems, i.e. systems that consist of components based on different MoCs. Thus, the basic structural unit of AIF$^+$ is a *component*. Components are hierarchically organized in a tree structure, i.e. a component can aggregate a set of child components. Thus, in a system a component is either the top component (communicating with the environment) or it has a unique parent component.

In order to cooperate, components need to exchange information. To this end, each component has an *interface*, which defines how it interacts with its parent module. In AIF$^+$, the interface consists of a list of signals $X = \langle x_1, \ldots, x_n \rangle$ exposed to the parent module (or the environment in the case of the top module). As described in Sect. 2.1, a signal does not only consist of a sequence of values but its domain also fixes the instants in which the data is transmitted. Thus, the components at either end of the interface know when to exchange information. In order to define the direction of the communication, the *information flow* of each signal in the interface is explicitly declared. *Inputs* can be only read by a component, *outputs* only written, and *inouts* are shared between components.

For common data-flow models, this type of interface would be sufficient. The modules are usually hierarchically organized, but this is only used for structuring the system. Finally, all modules functionally run in parallel from the start of the system forevermore, and there are no means to start or stop a module during the execution. Figure 2(a) illustrates this. When the whole system (module M_0 in the figure) is started, all its submodules are started, and they run in parallel. In this case, the *start* functionality is only given implicitly and not addressed in the system model. However, since we want to integrate control-flow oriented models (such as hierarchical statecharts or imperative synchronous languages) as first-class citizens in our intermediate representation, we have to address starting, suspending and stopping a module. This is done by an additional control-flow context in the interface of an AIF$^+$ module, e.g.



(a) structural hierarchy *(b) behavioral hierarchy*

Fig. 2 Structural vs. behavioral hierarchy

activation or preemption conditions given by the surrounding component are passed to the child component and status information (e.g. about its activation or termination) are passed to the calling module in return. This situation is illustrated in Fig. 2(b). In contrast to the structural approach, each module has a control interface, which does not only allow to start it but also to control its execution. In turn, the module itself provides some status information to the parent module e.g. it notifies the outer module whether it is still running or whether it terminates. The path of the control signals follows the hierarchy of the modules, and a module controls its submodules. Thus, in the example, M_2 can start and abort its submodule M_{21}. In this way, a module is e.g. able to execute its submodules in a sequence which is pretty common in control-flow oriented languages. The first submodule is started and not until it terminates the second one is started (and so on).

To this end, we implicitly extend the interface by eight additional signals of Boolean type, which indicate the control-flow context of a component. This is in the spirit of the compilation of imperative synchronous languages, which use similar parameters for of their compilation [8, 14, 64]. Each component is given the following five inputs.

- $\mathsf{strt}(M)$ is the start or reset signal, which holds iff M should be (re)started in the given instant. With the help of this signal, the internal state of the component is initialized and its behavior is (re)started. Please not that the data variables appearing in the interface are not reset since they are shared with the surrounding component. Not also that a module may only be started if it is currently inactive or it currently terminates.
- $\mathsf{prmt}(M)$ is a preemption signal for the initial instant. This signal can be used together with $\mathsf{strt}(M)$ to immediately abort the behavior of a component so that only a the initialization is executed in a weakly preempted start.
- $\mathsf{abrt}(M)$ is a signal that holds if an already running component M should be aborted in the given instant. As the compilation of imperative synchronous programs shows, we need to distinguish $\mathsf{prmt}(M)$ and $\mathsf{abrt}(M)$: the first one aborts a starting component, while the latter one aborts an already active one.
- $\mathsf{susp}(M)$ similarly describes the suspension context: if this signal holds in an instant, M will be suspended. Suspension means that the current state is sustained as long as this control signal is set and the execution resumes from that state as soon as the suspension is over.
- $\mathsf{strg}(M)$ is an enable condition. The signal is used to prevent that the component sets its outputs. With its help several mutually exclusive components may drive the same signals depending on control-flow context.

$\mathsf{strt}(M)$ and $\mathsf{prmt}(M)$ control the initial step of a component, while the other inputs are only read by an already running component. All the control inputs are mostly orthogonal, i.e. they can be set arbitrarily. The only potential conflict is a simultaneous set of $\mathsf{abrt}(M)$ and $\mathsf{susp}(M)$. If both are set for a component M, $\mathsf{susp}(M)$ has a higher priority, i.e. then the suspension takes place. Combinations of the other signals can be used to control a specific control-flow behavior: e.g. $\mathsf{abrt}(M) \wedge \mathsf{strg}(M)$ means that the module is strongly aborted (the macro-step is abort before executing its actions), while $\mathsf{abrt}(M) \wedge \neg\mathsf{strg}(M)$ aborts it after the execution of its actions. To illustrate the interaction between control inputs for the initial step and the rest, consider another example: $\mathsf{strt}(M) \wedge \mathsf{abrt}(M)$ means that a currently running module should be aborted and immediately restarted i the current step.

While the above list shows the additional input signals, each component also provides additional output signals for its control flow:

- $\mathsf{inst}(M)$ is a signal that holds if the execution of M will immediately terminate if it would now be started (M is often said to be instantaneous).
- $\mathsf{insd}(M)$ holds at some point of time if the control flow is currently at some location inside M, i.e., if M is active.
- $\mathsf{term}(M)$ is signal that holds if the control flow is currently somewhere inside M and wants to leave M voluntarily.

Similar to the data signals in the interface, we need to determine a clock for the additional control-flow signals so that this information can be aligned to the data streams. AIF^+ implicitly assumes that all control-flow signals share the same clock and that this clock is the union of the clocks of the data signals of the interface. Therefore, the intermediate format does not store this information in its data structure.

This choice for the definition of the clock of the control-flow signals is motivated as follows: First, all control signals should have the same clock as they all refer to the same control-flow. Obviously, this clock should consists of the instants when control moves through the program. Second, as component-based design always aims at proving a well-defined functionality while abstracting from internal design decisions, the control-flow signals should be at least on the level of the data signals. Otherwise, internal intermediate computations may become visible to the environment, which violates the common principle of information hiding. We also get the advantage that components can be only preempted at well-defined instants (and not during internal computations), which avoids inconsistent system configurations. Finally, the control-flow should not be unnecessarily coarse-grained. These three considerations lead to our choice that the control-flow signals have the same clock as the union of the data-flow signals of the interface.

2.3 Behavior

As the name suggests, guarded actions basically consist of a guard γ and an action A, i.e. they have the form $\langle \gamma \Rightarrow A \rangle$. The intuition behind a guarded action is that the body A is executed if its guard evaluates to true in the current instant.

The guard γ is a Boolean expression over the program variables and their clocks. There are no syntactical restrictions for the guards. As they must be evaluated in all instants, our semantic model ensures that all variables can be evaluated in all instants of a program execution (see below).

In our intermediate representation, an action A can be either an assignment or a constraint: assignments are an operational description, which completely fix the value of the written signal, while constraints may leave several possibilities: every behavior that complies to them is considered to be a valid one.

In our intermediate representation clocked guarded actions have one of the following forms:

$$\text{(a)} \quad \gamma \Rightarrow x = \tau$$
$$\text{(b)} \quad \gamma \Rightarrow \texttt{next}(x) = \tau$$
$$\text{(c)} \quad \gamma \Rightarrow \texttt{assume}(\sigma)$$
$$\text{(d)} \quad \gamma \Rightarrow \texttt{assert}(\sigma)$$

The first action (a) is an *immediate assignment*, which set the signal x at 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 defined by the next tick of its clock. Assumptions (c) and assertions (d) define constraints. They determine a Boolean condition which has to hold whenever the guard γ is true. The difference between them is that assumptions are guaranteed by the programmer, whereas assertions represent verification tasks, which have to be addressed in the following design flow.

In designs, the first instant usually differs from all other ones since additional behavior for initialization must be done. In the following, we use the expression $\mathsf{init}(C)$ for any clock C, which exactly holds the first time C ticks.

Furthermore, there are generally several guarded actions that write a variable x. For each variable, the behavior part of our intermediate representation also defines a *default assignment* if no action determines its value in the current step. For a variable x, this is the case iff the guards of all immediate assignments to x are false in the current step and the guards of all delayed assignments to x have also been false in the preceding step. In this case, it takes the default value according to its intended storage mode: *event* variables $eventVars \subseteq \mathcal{V}$ are reset (like wires in hardware circuits), while *memorized* variables $memVars \subseteq \mathcal{V}$ store their previous values (like registers in hardware circuits). In general, this default assignment can be given by

$$\text{(e)} \quad \texttt{default}(x) = \tau$$

The intended meaning is that x will be given the value x in the *next* step if there is no delayed action in the current step and if there is no immediate action in the next step that write to x. Thus, the expression τ is 0 (or `false`) for event variables, and it is x for memorized variables.

Before we give a formal denotational semantics of the guarded actions, we first explain the evaluation of an expression τ at instant I, which will be denoted by $[\![\tau]\!]_I$ in the following. In contrast to some synchronous languages [4, 32], we can evaluate a variable even if 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 has held the last time.[2] Thus, in AIF$^+$ a clock ticks corresponds to a potential value change. Based on this evaluation of expressions we can define a formal denotational semantics.

[2]This is somehow similar as reading the value `?x` of Signal `x` in Esterel [7], or implicitly using the `cell` operator in Signal [27].

Definition 4 (Consistency of guarded actions) A guarded action A is defined to be *consistent* (written $\mathsf{Consistent}(A)$) w.r.t. a set of instants $\mathcal{I}$ of a program execution as follows:

- $\mathsf{Consistent}(\gamma \Rightarrow x = \tau) := \forall I \in \mathcal{I}.\ [\![\gamma]\!]_I \rightarrow ([\![\widehat{x}]\!]_I \wedge [\![x]\!]_I = [\![\tau]\!]_I)$
- $\mathsf{Consistent}(\gamma \Rightarrow \mathtt{next}(x) = \tau) := \forall I_1 \prec I_2 \in \mathcal{I}.\ [\![\gamma]\!]_{I_1} \rightarrow ([\![\widehat{x}]\!]_{I_2} \wedge [\![x]\!]_{I_2} = [\![\tau]\!]_{I_1})$ where I_2 such that $\nexists I' \in \mathcal{I}.\ (I_1 \prec I' \prec I_2) \wedge [\![\widehat{x}]\!]_{I'}$
- $\mathsf{Consistent}(\gamma \Rightarrow \mathtt{assume}(\sigma)) := \forall I \in \mathcal{I}.\ [\![\gamma]\!]_I \rightarrow [\![\sigma]\!]_I$
- $\mathsf{Consistent}(\gamma \Rightarrow \mathtt{assert}(\sigma)) := \forall I \in \mathcal{I}.\ [\![\gamma]\!]_I \rightarrow [\![\sigma]\!]_I$
- $\mathsf{Consistent}(\mathtt{default}(x) = \tau) := \forall I_1 \prec I_2 \in \mathcal{I}.\ \xi \rightarrow ([\![\widehat{x}]\!]_{I_2} \wedge [\![x]\!]_{I_2} = [\![\tau]\!]_{I_1})$ where I_2 such that $\nexists I' \in \mathcal{I}.\ (I_1 \prec I' \prec I_2) \wedge [\![\widehat{x}]\!]_{I'}$ and $\xi = \left(\left[\!\left[\bigwedge_{(\gamma \Rightarrow x = \tau)} \neg\gamma \right]\!\right]_{I_2} \wedge \left[\!\left[\bigwedge_{(\gamma \Rightarrow \mathtt{next}(x) = \tau)} \neg\gamma \right]\!\right]_{I_1} \right)$

A set of guarded actions $\mathcal{A}$ is consistent if all its elements are consistent:

- $\mathsf{Consistent}(\mathcal{A}) := \forall A \in \mathcal{A}.\ \mathsf{Consistent}(A)$

We are convinced that the representation of the behavior by clocked guarded actions is exactly at *the right level of abstraction* for an intermediate code format, since guarded actions provide a good balance between (1) removal of complexity from the source code level and (2) the independence of a specific synthesis target. (This will be illustrated by Sect. 4, which gives straightforward translations targeting symbolic model checking and simulation.) On the one hand, problems such as schizophrenia [8, 57, 65] and the semantics of complex control flow statements like preemption statements can be completely solved during the translation to guarded actions, so that subsequent analysis, optimization and synthesis become much simpler. On the other hand, despite their very simple structure, efficient translation to both software and hardware is efficiently possible from guarded actions.

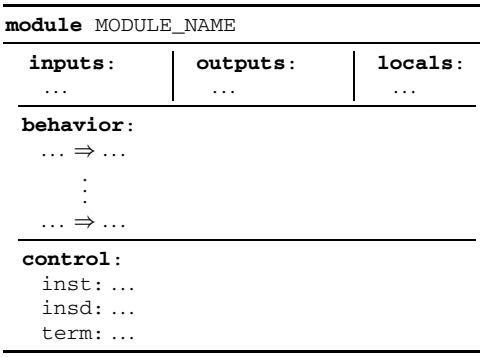
Guarded actions allow *many analyses and optimizations*. In particular, causality analysis can be effectively performed on guarded actions. If the causality analysis determined that a set of guarded actions does always have a dynamic schedule to compute the variables without first reading them in each macro step, then even an acyclic set of guarded actions can be determined. Other transformations on guarded actions are the grouping of guarded actions with regard to the variable they modify, which corresponds to the generation of static single-assignment form in the compilation of sequential languages. For synchronous languages this is often called an equational code generation, since for every variable, a single equation is generated.

2.4 Structure

Having explained the parts contained in the AIF$^+$ intermediate representation, we can now put all parts together to describe the whole structure. In the representation, we distinguish between *modules* and *systems*. Thereby, an AIF$^+$ module (keyword **`module`**) contains the control interface, which allows the composition with other modules. In contrast, an AIF$^+$ system (keyword **`system`**) is considered as the result of the composition of modules. The control interface is bound, and all module calls resolved, i.e. it is fully linked. The only control signal which is still used in a system is $\mathsf{strt}(\mathtt{M})$ to start the system.

The structure of the AIF$^+$ intermediate format is presented in Fig. 3. The data interface contains the declarations of inputs, outputs and local variables. The behavior is given by clocked guarded actions. The control interface is described in the last part of the structure. These are the conditions which are defined by the module. The control signals which come

Fig. 3 AIF$^+$ module structure



from the outside, are simply used as expressions in the behavior, i.e. the clocked guarded actions. For a system, the control part is simply omitted.

3 Translating to clocked guarded actions

In this section, we show how different 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 Fig. 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 article, namely single-clocked synchronous programs in Sect. 3.1, polychronous specifications in Sect. 3.2 and finally concurrent action-oriented specifications in Sect. 3.3. 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.

3.1 Synchronous programs

The synchronous *model of computation* [4, 32] 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 [5, 58, 64]; it also provides a very convenient programming model, which allows compilers to generate *deterministic* single-threaded code from multi-threaded synchronous programs [6].

In general, synchronous programs such as Esterel, Lustre or Quartz can be translated to synchronous guarded actions [14]. 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.

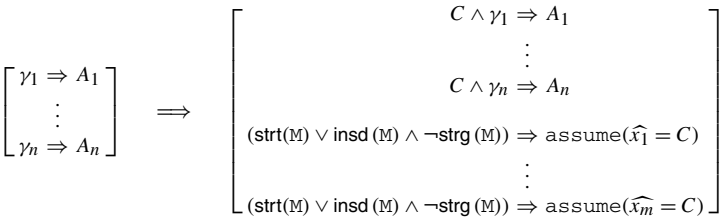


Fig. 4 Translating synchronous guarded actions of module `M` to clocked guarded actions

```
module Sequence
  (bool ?switch, int ?i,!o,!mode)
{
  abort {
    mode = 1;
    M1(i, o); // call module M1
  } when (switch);
  ℓ: pause;
  abort {
    mode = 2;
    M2(i, o); // call module M2
  } when (switch);
}
```

Fig. 5 Quartz example: `Sequence`

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 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 Fig. 4(a). A system in the AIF -format, which is based on synchronous guarded actions, contains a set of guarded actions which are executed in any instant. The guards of the guarded actions are strengthened by the clock C. Thus, they are now executed at any tick of C. The clock constraints ensure that all variables have the same clock. The clock constraints are just taken into account when the module is running which is the case when the module is started (strt(M)), or when the control flow is inside the module (insd (M)). In the second case, it also has to be ensured that the module is not aborted or suspended (¬strg (M)).

An AIF component already stores a control-flow context, which consists of the signals described in Sect. 2.2. Since all the variables of the interface have the same clock, its translation to AIF$^+$is trivial: the conditions are stored without any modification in AIF$^+$.

We will now give an example AIF$^+$ representation of an synchronous module. Since the representation in guarded actions is pretty simple to the representation with clocked guarded actions, we will omit this version. However, we will give the synchronous example in Quartz code which provides a more readable representation and the translation to guarded actions is given in [14] and also mentioned in Fig. 1. The behavior of the example Quartz module `Sequence` which is given in Fig. 5 is the following. It gets the input `i` and produces the output `o`. However, the real computation of `o` is done either by the submodule `M1` or `M2`. The module `Sequence` does *just* control which of these both submodules drive the output.

module Sequence

inputs:	**outputs**:	**locals**:
bool switch	**int** o, mode	**clock** C
int i		**label** ℓ

behavior:

$$\mathtt{C} \wedge \mathsf{strt}(\mathtt{Sequence}) \Rightarrow \mathtt{mode} = 1$$

$$\mathtt{C} \wedge \mathsf{strt}(\mathtt{Sequence}) \Rightarrow \begin{cases} \mathtt{M1(i,o)\{} \\ \quad \mathtt{prmt:}\ \mathsf{prmt}(\mathtt{Sequence}) \\ \quad \mathtt{abrt:}\ \mathsf{abrt}(\mathtt{Sequence}) \vee \mathtt{switch} \\ \quad \mathtt{susp:}\ \mathsf{susp}(\mathtt{Sequence}) \\ \quad \mathtt{strg:}\ \mathsf{strg}(\mathtt{Sequence}) \vee \mathtt{switch} \\ \mathtt{\}} \end{cases}$$

$$\mathtt{C} \wedge \mathsf{strt}(\mathtt{Sequence}) \wedge \mathsf{inst}(\mathtt{M1}) \Rightarrow \mathtt{next}(\ell) = \mathsf{true}$$

$$\mathtt{C} \wedge \ell \wedge \mathsf{susp}(\mathtt{Sequence}) \Rightarrow \mathtt{next}(\ell) = \mathsf{true}$$

$$\mathtt{C} \wedge \begin{pmatrix} \mathsf{term}(\mathtt{M1}) \wedge \\ \neg\mathsf{strg}(\mathtt{Sequence}) \wedge \\ \neg(\mathsf{susp}(\mathtt{Sequence}) \vee \\ \mathsf{abrt}(\mathtt{Sequence})) \end{pmatrix} \Rightarrow \mathtt{next}(\ell) = \mathsf{true}$$

$$\mathtt{C} \wedge \ell \wedge \neg\mathsf{strg}(\mathtt{Sequence}) \Rightarrow \mathtt{mode} = 2$$

$$\mathtt{C} \wedge \ell \wedge \neg\mathsf{strg}(\mathtt{Sequence}) \Rightarrow \begin{cases} \mathtt{M2(i,o)\{} \\ \quad \mathtt{prmt:}\ \mathsf{abrt}(\mathtt{Sequence}) \vee \mathsf{susp}(\mathtt{Sequence}) \\ \quad \mathtt{abrt:}\ \mathsf{abrt}(\mathtt{Sequence}) \vee \mathtt{switch} \\ \quad \mathtt{susp:}\ \mathsf{susp}(\mathtt{Sequence}) \\ \quad \mathtt{strg:}\ \mathsf{strg}(\mathtt{Sequence}) \vee \mathtt{switch} \\ \mathtt{\}} \end{cases}$$

$$\begin{pmatrix} \mathsf{strt}(\mathtt{Sequence}) \vee \\ \mathsf{insd}(\mathtt{Sequence}) \wedge \neg\mathsf{strg}(\mathtt{Sequence}) \end{pmatrix} \Rightarrow \mathtt{assume}(\widehat{\mathtt{switch}} = \mathtt{C})$$

$$\begin{pmatrix} \mathsf{strt}(\mathtt{Sequence}) \vee \\ \mathsf{insd}(\mathtt{Sequence}) \wedge \neg\mathsf{strg}(\mathtt{Sequence}) \end{pmatrix} \Rightarrow \mathtt{assume}(\widehat{\mathtt{i}} = \mathtt{C})$$

$$\begin{pmatrix} \mathsf{strt}(\mathtt{Sequence}) \vee \\ \mathsf{insd}(\mathtt{Sequence}) \wedge \neg\mathsf{strg}(\mathtt{Sequence}) \end{pmatrix} \Rightarrow \mathtt{assume}(\widehat{\mathtt{o}} = \mathtt{C})$$

$$\begin{pmatrix} \mathsf{strt}(\mathtt{Sequence}) \vee \\ \mathsf{insd}(\mathtt{Sequence}) \wedge \neg\mathsf{strg}(\mathtt{Sequence}) \end{pmatrix} \Rightarrow \mathtt{assume}(\widehat{\mathtt{mode}} = \mathtt{C})$$

$$\mathsf{true} \Rightarrow \mathtt{assume}(\widehat{\ell} = \mathtt{C})$$

control:
inst: false
insd: $\ell \vee \mathsf{insd}(\mathtt{M1}) \vee \mathsf{insd}(\mathtt{M2})$
term: $\ell \wedge \mathsf{inst}(\mathtt{M2}) \vee \mathsf{term}(\mathtt{M2}) \vee \mathtt{switch} \wedge \mathsf{insd}(\mathtt{M2})$

Fig. 6 AIF$^+$ of example: Sequence

First, the output is computed by the submodule M1 until it is finished or its execution is aborted by the signal switch. Then, M2 computes the output until it is finished or its execution is aborted by the signal switch. The resulting AIF$^+$ representation is given in Fig. 6. According to the translation described above, except for the clocks it is the same as the AIF representation. If the module is started, i.e. strt(Sequence), the assignment mode=1 is executed and also the module M1 is called. For calling the module, also the control interface has to be defined. In this case it is combined from the control interface of Sequence and the **abort** statement. Thus, e.g. the module M1 is aborted, when the internal abort condition (switch) holds or an abort condition is given from the outside (abrt(Sequence)). The module M2 is started, when the module M1 terminates or when it is aborted with the input switch.

We now introduce the first part of a *running example*, which we will use in the remainder of this paper. It shows how modules and processes written in different languages and based

```
module OuterQuartz
  (bool ?susp, int ?i, !o)
{
  suspend {
    InnerSignal(i, o);
  } when (susp);
}
```

```
module InnerQuartz
  (int ?y, !x)
{
  loop {
    x = y;
    pause;
    x = 2 * y;
    pause;
  }
}
```

Fig. 7 Running example: Quartz modules `OuterQuartz` and `InnerQuartz`

module OuterQuartz

inputs:	**outputs:**	**locals:**
bool susp	**int** o	**clock** C1
int i		

behavior:

$$\begin{aligned}
&\text{true} \Rightarrow \texttt{assume}(\hat{\texttt{i}} = \texttt{C1})\\
&\text{true} \Rightarrow \texttt{assume}(\widehat{\texttt{susp}} = \texttt{C1})\\
&\text{true} \Rightarrow \texttt{assume}(\hat{\texttt{o}} = \texttt{C1})\\
&\text{C} \wedge \text{strt}(\texttt{OuterQuartz}) \Rightarrow \left\{ \begin{array}{l} \texttt{InnerSignal(i,o)\{}\\ \quad \texttt{prmt:}\ \text{prmt}(\texttt{OuterQuartz})\\ \quad \texttt{abrt:}\ \text{abrt}(\texttt{OuterQuartz})\\ \quad \texttt{susp:}\ \text{susp}(\texttt{OuterQuartz}) \vee \texttt{susp}\\ \quad \texttt{strg:}\ \text{strg}(\texttt{OuterQuartz}) \vee \texttt{susp}\\ \texttt{\}} \end{array} \right.
\end{aligned}$$

control:

inst: inst(InnerSignal)
insd: insd(InnerSignal)
term: term(InnerSignal)

Fig. 8 AIF$^+$ of example: `OuterQuartz`

on different models can be combined on the basis of AIF$^+$. Thereby, it will show how a hierarchical structure as it is described in Sect. 2.2 can be established. In total, the running example consists of two Quartz modules, which are described here, and another module, which will be explained later.

The Quartz modules `OuterQuartz` and `InnerQuartz` are shown in Fig. 7. `OuterQuartz` is the outermost module of the hierarchy and forms the interface to the environment. It takes two inputs `susp` and `i` and produces an output `o`. Inside, it calls a module named `InnerSignal`, which will be described later. `OuterQuartz` just calls the sub-module and provides a suspension context based on the input `susp`. `InnerSignal` by itself will make use of the second Quartz module `InnerQuartz`. It takes one input `y` and produces one output `x`, which is set either to `x = y;` or `x = 2 * y;`.

The AIF$^+$ descriptions of both modules are shown in Figs. 8 and 9. Thereby, `OuterQuartz` synchronizes the clocks of the inputs and outputs and gives the interface signals to `InnerSignal`. The suspension context is strengthened by the input `susp`. `InnerQuartz` does not call a submodule, but it shows the usage of the interface signals for its own behavior.

module InnerQuartz

inputs:	**outputs:**	**locals:**
int y	**int** x	**clock** C2 **label** ℓ_1, ℓ_2

behavior:

$$\text{true} \Rightarrow \texttt{assume}(\widehat{\texttt{y}} = \texttt{C2})$$
$$\text{true} \Rightarrow \texttt{assume}(\widehat{\texttt{x}} = \texttt{C2})$$
$$\text{true} \Rightarrow \texttt{assume}(\widehat{\ell_1} = \texttt{C2})$$
$$\text{true} \Rightarrow \texttt{assume}(\widehat{\ell_2} = \texttt{C2})$$
$$\texttt{C2} \wedge \begin{pmatrix} \text{strt}(\texttt{InnerQuartz}) \vee \\ \ell_2 \wedge \neg\text{strg}(\texttt{InnerQuartz}) \end{pmatrix} \Rightarrow \texttt{x} = \texttt{y}$$
$$\texttt{C2} \wedge (\ell_1 \wedge \neg\text{strg}(\texttt{InnerQuartz})) \Rightarrow \texttt{x} = 2 * \texttt{y}$$
$$\texttt{C2} \wedge \begin{pmatrix} \text{strt}(\texttt{InnerQuartz}) \wedge \neg\text{prmt}(\texttt{InnerQuartz}) \vee \\ \ell_1 \wedge \text{susp}(\texttt{InnerQuartz}) \vee \\ \ell_2 \wedge \neg\text{strg}(\texttt{InnerQuartz}) \wedge \\ \neg(\text{susp}(\texttt{InnerQuartz}) \vee \text{abrt}(\texttt{InnerQuartz})) \end{pmatrix} \Rightarrow \texttt{next}(\ell_1) = \text{true}$$
$$\texttt{C2} \wedge \begin{pmatrix} \ell_2 \wedge \text{susp}(\texttt{InnerQuartz}) \vee \\ \ell_1 \wedge \neg\text{strg}(\texttt{InnerQuartz}) \wedge \\ \neg(\text{susp}(\texttt{InnerQuartz}) \vee \text{abrt}(\texttt{InnerQuartz})) \end{pmatrix} \Rightarrow \texttt{next}(\ell_2) = \text{true}$$

control:
inst: false
insd: $\ell_0 \vee \ell_1$
term: false

Fig. 9 AIF$^+$ of example: InnerQuartz

3.2 Polychronous programs

Polychronous specifications [27, 28, 49] as implemented by Signal use several clocks, which means that signals do not need to be present in all instants. Furthermore, 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 [27, 28].

Signal programs consist 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 $\texttt{y} := f(x_1, \ldots, x_n)!$ determines 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. Initially, the buffer simply returns the given value d. By definition, the input and the output have the same clock, i.e. $\widehat{x} = \widehat{y}$.

$$\texttt{y := f(x1, ..., xn)} \Rightarrow \left[\begin{array}{r} \widehat{y} \wedge \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow y = f(x_1, \ldots, x_n) \\ \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow \texttt{assume}(\widehat{y} = \widehat{x_1}) \\ \ldots \\ \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow \texttt{assume}(\widehat{y} = \widehat{x_n}) \end{array}\right]$$

$$\texttt{y := x \$ init d} \Rightarrow \left[\begin{array}{r} \widehat{y} \wedge \mathsf{strt}(\mathtt{M}) \Rightarrow y = d \\ \widehat{y} \wedge \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M}) \Rightarrow y = y' \\ \widehat{y} \Rightarrow \texttt{next}(y') = x \\ \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow \texttt{assume}(\widehat{y} = \widehat{x}) \\ \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow \texttt{assume}(\widehat{y} = \widehat{y'}) \end{array}\right]$$

$$\texttt{y := x when z} \Rightarrow \left[\begin{array}{r} \widehat{y} \wedge \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow y = x \\ \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow \texttt{assume}(\widehat{y} = (z \wedge \widehat{z} \wedge \widehat{x})) \end{array}\right]$$

$$\texttt{y := x default z} \Rightarrow \left[\begin{array}{r} \widehat{x} \wedge \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow y = x \\ \widehat{z} \wedge \neg\widehat{x} \wedge \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow y = z \\ \begin{pmatrix}\mathsf{strt}(\mathtt{M}) \vee \\ \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})\end{pmatrix} \Rightarrow \texttt{assume}(\widehat{y} = (\widehat{x} \vee \widehat{z})) \end{array}\right]$$

Fig. 10 Translation from Signal to clocked guarded actions

– *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. Figure 10 shows the translation where the condition:

$$\mathsf{strt}(\mathtt{M}) \vee \mathsf{insd}(\mathtt{M}) \wedge \neg \mathsf{strg}(\mathtt{M})$$

indicates that the module is currently executed and is added to the guards.

– *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 delay operator violates the rule to add the above condition to all guards, because its behavior is split into two cases. (1) The first value that is produced by this operator when the process is started 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 of the additional signal y'. The constraint ensures that all 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 when the process is running. 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, in its context the clock of a signal identifies when there is a value and when not. As explained in Sect. 2.3, 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 has held 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 Fig. 10(b). However, interaction with other computational models is done in the model of AIF$^+$ by allowing to read the variable in every instant.

The translation above uses the control signal $\mathsf{insd}(\mathtt{M})$ which is usually defined by the process itself but not yet explained for Signal processes. In order to fully translate Signal programs to AIF$^+$, we have to define a rudimentary control-flow. We introduce a local Boolean event variable ℓ, which simply models the activation of the component: if ℓ is set, the control-flow is currently inside the component and all the equations of the data-flow description are activated; if ℓ is not set, the component is inactive. Its value is set by the following guarded actions:

- $\widehat{\ell} \wedge \mathsf{strt}(M) \wedge \neg\mathsf{prmt}(M) \Rightarrow \mathtt{next}(\ell) = \mathsf{true}$
- $\widehat{\ell} \wedge \ell \wedge \neg\mathsf{abrt}(M) \Rightarrow \mathtt{next}(\ell) = \mathsf{true}$

The first action considers the activation of the component. If it is started and not immediately preempted, the component becomes active and therefore ℓ true. The second action models a running component: it will remain active unless an abortion takes place (suspension does not have an effect on ℓ). We use this flag to provide the necessary control-flow information to its parent component:

- $\mathsf{inst}(M) = \mathsf{false}$
- $\mathsf{insd}(M) = \ell$
- $\mathsf{term}(M) = \mathsf{false}$

As the component is considered to run the given SINGAL equations, it is never instantaneous. We modeled its activation by ℓ so that we can use it for the $\mathsf{insd}(M)$ signal. Finally, it can only terminate if an explicit abortion is triggered from the output.

Based on the control-flow context the data-flow of the component can then be guarded: each guarded action $\gamma \Rightarrow A$ is only executed if the module is started or if it is running and the data-flow enabled, i.e. its guard is strengthened to $\gamma \wedge (\mathsf{strt}(M) \vee \ell \wedge \mathsf{strg}(M))$.

A first example illustrates the translation of a Signal specification to an AIF$^+$ system. The Signal process Counter is shown in Fig. 11. The intention of the process is that for each input value n, the output values $n, n-1, \ldots, 0$ are produced. To this end, the local signal c stores the last value of the produced output, whereas o is produced by subtraction of 1 from c. However, when a new value for the input n arrives, the output is updated by this value. The clock constraint `n ^= (when (c = 0))` ensures that a new input is only allowed

Fig. 11 Signal example: Counter

```
process Counter =
(? integer n;
 ! integer o;)
(| c := o $ init 0
 | o := n default (c-1)
 | n ^= (when (c=0))
 |)
where
    integer c;
end;
```

n = [2, ⊡, ⊡, 1, ⊡]
c = [0, 2, 1, 0, 1]
o = [2, 1, 0, 1, 0]
Trace 1

n = [2, ⊡, 1, ⊡]
c = [0, 2, 1, 1]
o = [2, 1, 1, 0]
Trace 2

Fig. 12 AIF$^+$ of example: Counter

system Counter

inputs:	**outputs:**	**locals:**
int n	**int** o	**int** c, c'
		label ℓ

behavior:

$$\widehat{c} \wedge \text{strt(Counter)} \Rightarrow c = 0$$
$$\widehat{c} \wedge \ell \Rightarrow c = c'$$
$$\widehat{c'} \wedge (\text{strt(Counter)} \vee \ell) \Rightarrow \text{next}(c') = o$$
$$(\text{strt(Counter)} \vee \ell) \Rightarrow \text{assume}(\widehat{c} = \widehat{o})$$
$$(\text{strt(Counter)} \vee \ell) \Rightarrow \text{assume}(\widehat{c} = \widehat{c'})$$
$$\widehat{n} \wedge (\text{strt(Counter)} \vee \ell) \Rightarrow o = n$$
$$\widehat{c} \wedge \neg\widehat{n} (\text{strt(Counter)} \vee \ell) \Rightarrow o = c$$
$$(\text{strt(Counter)} \vee \ell) \Rightarrow \text{assume}(\widehat{o} = \widehat{x} \vee \widehat{z})$$
$$(\text{strt(Counter)} \vee \ell) \Rightarrow \text{assume}(\widehat{n} = (\widehat{c} \wedge c == 0))$$
$$\widehat{\ell} \wedge \text{strt(Counter)} \Rightarrow \text{next}(\ell) = \text{true}$$
$$\widehat{\ell} \wedge \ell \Rightarrow \text{next}(\ell) = \text{true}$$
$$\text{true} \Rightarrow \text{assume}(\widehat{\ell} = \widehat{\text{strt(Counter)}})$$

to arrive when the local signal c reaches 0. On the right side of the figure, two sample traces for this example are shown. Thereby, ⊡ indicates the absence of a signal, i.e. it is not present in the instant. The first trace is a valid one and shows the desired behavior. First, 2 arrives as input and the output produces sequently the values 2, 1, 0. After that, the local signal c is 0 and a new input is allowed to arrive. The second trace is an invalid one, because the second input value of n arrives too early and thus, the clock constraint is not fulfilled by this execution. Note, that without the given clock constraint both traces are valid but the constraint selects just the first one to be valid. The translation to an AIF$^+$ system of this example is shown in Fig. 12. Each given Signal equation is translated. The additional label ℓ stores the activation state of the process.

The second Signal example InnerSignal, which is shown in Fig. 13 is part of the running example, and it illustrates the translation to an AIF$^+$ module. It takes the input i and produces the output o. The local signal s alternates the values sigtrue and false each time a new input i is given (its clock is set to be the same as the clock of i). The signal y holds the value of the input i whenever s is true, thus, for every second input. The Quartz module InnerQuartz is used to compute the value of x for every value of y. Finally, the output o is set to x if it is present, otherwise the given inputsi is passed through. In summary, every second input value of i is passed to InnerQuartz to compute the output o, all other inputs are directly passed to o. The resulting AIF$^+$ module of the process InnerSignal is shown in Fig. 14.

```
process InnerSignal =
(? integer i;
 ! integer o;)
(| s := (s $ init true) xor true
 | y := i when s
 | o := x default i
 | i ^= s
 | InnerQuartz(y, x)
 |)
where
  integer x, y; boolean s;
end;
```

Fig. 13 Running example: Signal process `InnerSignal`

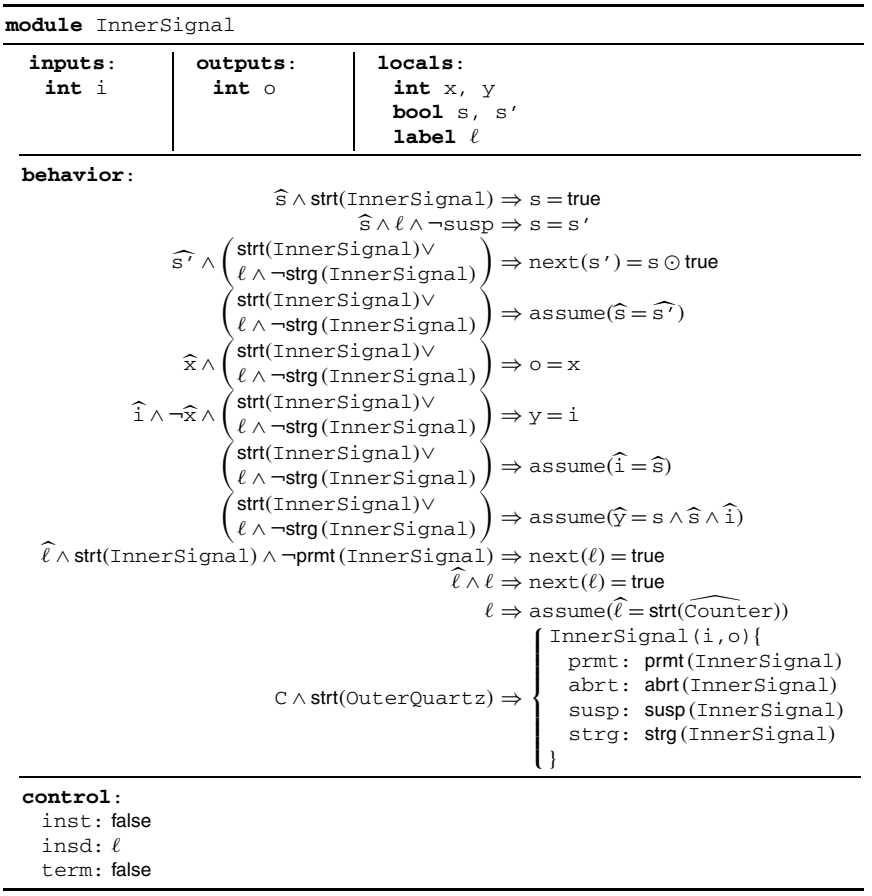


Fig. 14 AIF$^+$ of example: `InnerSignal`

3.3 Asynchronous (untimed) programs

The third class of programs we consider in this article are the asynchronous (untimed) programs, which do not have an underlying notion of synchrony. They focus on describing the causalities of actions, i.e. which event happens after an other one, thereby, deferring the difficult task of global scheduling and coordination to compilers or runtime environments.

In particular, this class includes languages such as Concurrent Action-Oriented Specifications [1, 40, 41] (CAOS), a language intended to describe the data-flow of hardware circuits, or SHIM [24], a multi-threaded language based on asynchronous threads with a rendezvous-style communication [38, 39].

Similar to the synchronous and polychronous programs before, we do not consider these languages as the starting point of our translation to AIF. Instead, we start with simple asynchronous guarded actions, which can be used as an intermediate representation of the compiler. The step from full CAOS to this intermediate representation is straightforward: it only consists of a simple dismantle step [16]. SHIM can be also efficiently translated to asynchronous guarded actions as described in [15].

In our approach, we take asynchronous guarded actions in the form of rules and methods [16]. Thereby, 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 Sect. 3.1).

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 Sect. 2, the semantics of the asynchronous guarded actions is as follows: first the guards 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. $q_i(\gamma_x) = \mathsf{true}$. 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 Fig. 15. 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

$$
\begin{array}{l}
\texttt{rule } r_i \texttt{ when}(\gamma_{r_i}) \\
\quad \alpha_{r_i 1} \Rightarrow A_{r_i 1} \\
\quad \alpha_{r_i 2} \Rightarrow A_{r_i 2} \\
\quad \ldots
\end{array}
\Rightarrow
\left\{
\begin{array}{l}
C_{r_i} \wedge \alpha_{r_i 1} \Rightarrow A_{r_i 1} \\
C_{r_i} \wedge \alpha_{r_i 2} \Rightarrow A_{r_i 2} \\
\qquad \ldots \\
\mathsf{true} \Rightarrow \texttt{assume}(C_{r_i} \rightarrow \gamma_{r_i}) \\
\mathsf{true} \Rightarrow \texttt{assume}(C_{r_i} \rightarrow \neg \bigvee_{C \in \{C_{m_j}, C_{r_j}\} \setminus C_{r_i}} C)
\end{array}
\right.
$$

$$
\begin{array}{l}
\texttt{method } m_i(p_{i1}, \ldots) \\
\quad \texttt{when}(\gamma_{m_i}) \\
\quad\quad \alpha_{m_i 1} \Rightarrow A_{m_i 1} \\
\quad\quad \alpha_{m_i 2} \Rightarrow A_{m_i 2} \\
\quad\quad \ldots
\end{array}
\Rightarrow
\left\{
\begin{array}{l}
C_{m_i} \wedge \alpha_{m_i 1} \Rightarrow A_{m_i 1} \\
C_{m_i} \wedge \alpha_{m_i 2} \Rightarrow A_{m_i 2} \\
\qquad \ldots \\
\mathsf{true} \Rightarrow \texttt{assume}(C_{m_i} \rightarrow \gamma_{m_i}) \\
\mathsf{true} \Rightarrow \texttt{assume}(C_{m_i} \rightarrow \neg \bigvee_{C \in \{C_{m_j}, C_{r_j}\} \setminus C_{m_i}} C) \\
\mathsf{true} \Rightarrow \texttt{assume}(C_{m_i} \leftrightarrow \widehat{p_{i1}}) \\
\qquad \ldots
\end{array}
\right.
$$

$$
\mathsf{true} \Rightarrow \texttt{assume}(\widehat{x_1} \wedge \widehat{x_2} \wedge \ldots)
$$

Fig. 15 Translation from CAOS to AIF$^+$

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.

For the control-flow context we follow the translation of Signal. As we have an underlying data-flow model again, we add the same rudimentary as described in Sect. 3.2 and guard all clocked guarded actions similarly. This concludes the translation of CAOS to AIF$^+$.

Now, we illustrate our approach by a CAOS example. The CAOS model is given in Fig. 16, and the description derived from our translation is given in Fig. 17. The system describes a token ring, where messages can be only exchanged between neighbors by a common single-place buffer. Communication is directed and its direction is static, i.e. each buffer is always the input of the following node and the output of the preceding node. In this example, we have three nodes connected to the ring, which all have the same behavior: if the output buffer is empty and the message in the input buffer is not for the node itself, the packet is forwarded. This part of the behavior is described by the rules `node1`, `node2`, and `node3`. Packets are inserted and removed from the ring by send and receive methods. For the sake of simplicity, this example contains these methods only for the first node. By firing a `send1`, a new packet is inserted into the ring, which can be only done if the current output buffer of the first node is empty. Packets can be received by a call to `receive1`, which requires that there is a packet waiting in the input buffer of the node. Obviously, all rules write to different buffers. The only resource conflict is between the forwarding of the first node and the introduction of a new packet, which both write to `31`. The translation is done according to the rules described above. The last constraint is interesting, which is due to the CAOS

```
module TokenRing {
  int buf_addr_12 = 0, buf_addr_23 = 0, buf_addr_31 = 0;
  int buf_data_12 = 0, buf_data_23 = 0, buf_data_31 = 0;

  rule node1
    when((buf_addr_31 != 1) & (buf_data_12 == 0)) {
      next(buf_addr_12) = buf_addr_31;
      next(buf_data_12) = buf_data_31;
      next(buf_addr_31) = 0;
      next(buf_data_31) = 0;
  }
  rule node2
    when((buf_addr_12 != 2) & (buf_data_23 == 0)) {
      next(buf_addr_23) = buf_addr_12;
      next(buf_data_23) = buf_data_12;
      next(buf_addr_12) = 0;
      next(buf_data_12) = 0;
  }
  rule node3
    when((buf_addr_23 != 3) & (buf_data_31 == 0)) {
      next(buf_addr_31) = buf_addr_23;
      next(buf_data_31) = buf_data_23;
      next(buf_addr_23) = 0;
      next(buf_data_23) = 0;
  }
  method send1(int ?a , int ?d)
    when (buf_data_31 == 0) {
      next(buf_addr_31) = a;
      next(buf_data_31) = d;
  }
  method receive3(int !d)
    when ((buf_data_23 != 0) & (buf_addr_23 == 3)) {
      next(buf_data_23) = 0;
      d = buf_data_23;
  }
}
```

Fig. 16 CAOS example: `TokenRing`

semantics. It forbids that all four nodes fire in parallel, since this cannot be represented by any sequential firing.

3.4 Composition

As already highlighted in Sect. 2.2, AIF$^+$ is not only an intermediate format for different languages but it also aims at composing modules obtained from different languages. To this end, there is an AIF$^+$ linker, which can substitute module calls by the appropriate instance and the connects the signals of the control-flow and the data-flow interfaces.

We illustrate the linking with the help of our running example. While the previous sections showed the individual modules `OuterQuartz`, `InnerSignal` and `InnerQuartz`, this section describes their composition. To avoid confusion, the variable names in the running example have been chosen that the usual renaming step is not necessary.

module TokenRing

inputs:	**outputs**:	**locals**:
int n	**int** n	**int** buf_addr_12, buf_addr_23, buf_addr_31 **int** buf_data_12, buf_data_23, buf_data_31 **clock** Cnode1, Cnode2, Cnode3 **clock** Csend1, Creceive3 **bool** ℓ

behavior:

*/*initialization */*

$$\text{init}(\widehat{\texttt{buf\_addr\_12}}) \Rightarrow \texttt{buf\_addr\_12} = 0$$
$$\vdots$$
$$\text{init}(\widehat{\texttt{buf\_data\_31}}) \Rightarrow \texttt{buf\_addr\_31} = 0$$

*/*rule: node1 */*

$$\left(\begin{array}{l}\texttt{Cnode1}\wedge\\ \texttt{(buf\_addr\_31 != 1)}\wedge\\ \texttt{(buf\_data\_12.data==0)}\end{array}\right) \Rightarrow \texttt{next(buf\_addr\_12)} = \texttt{buf\_addr\_31}$$
$$\left(\begin{array}{l}\texttt{Cnode1}\wedge\\ \texttt{(buf\_addr\_31 != 1)}\wedge\\ \texttt{(buf\_data\_12.data==0)}\end{array}\right) \Rightarrow \texttt{next(buf\_data\_12)} = \texttt{buf\_data\_31}$$
$$\left(\begin{array}{l}\texttt{Cnode1}\wedge\\ \texttt{(buf\_addr\_31 != 1)}\wedge\\ \texttt{(buf\_data\_12.data==0)}\end{array}\right) \Rightarrow \texttt{next(buf\_addr\_31)} = 0$$
$$\left(\begin{array}{l}\texttt{Cnode1}\wedge\\ \texttt{(buf\_addr\_31 != 1)}\wedge\\ \texttt{(buf\_data\_12.data==0)}\end{array}\right) \Rightarrow \texttt{next(buf\_data\_31)} = 0$$
$$\text{true} \Rightarrow \text{assume}\left(\texttt{Cnode1} \rightarrow \left(\begin{array}{l}\texttt{(buf\_addr\_31 != 1)}\wedge\\ \texttt{(buf\_data\_12.data==0)}\end{array}\right)\right)$$
$$\text{true} \Rightarrow \text{assume}\left(\texttt{Cnode1} \rightarrow \neg\left(\begin{array}{l}\texttt{Cnode2}\wedge\\ \texttt{Cnode3}\wedge\\ \texttt{Csend1}\wedge\\ \texttt{Creceive3}\end{array}\right)\right)$$

*/*rule: node2 */*

$$\ldots$$

*/*method: send1 */*

$$\texttt{Csend1} \wedge \texttt{(buf\_data\_31 == 0)} \Rightarrow \texttt{next(buf\_addr\_31)} = \texttt{a}$$
$$\texttt{Csend1} \wedge \texttt{(buf\_data\_31 == 0)} \Rightarrow \texttt{next(buf\_data\_31)} = \texttt{d}$$
$$\text{true} \Rightarrow \text{assume}(\texttt{Csend1} \rightarrow \texttt{(buf\_data\_31 == 0)})$$
$$\text{true} \Rightarrow \text{assume}\left(\texttt{Csend1} \rightarrow \neg\left(\begin{array}{l}\texttt{Cnode1}\wedge\\ \texttt{Cnode2}\wedge\\ \texttt{Cnode3}\wedge\\ \texttt{Creceive3}\end{array}\right)\right)$$
$$\text{true} \Rightarrow \text{assume}(\texttt{Csend1} \leftrightarrow \widehat{\texttt{a}})$$
$$\text{true} \Rightarrow \text{assume}(\texttt{Csend1} \leftrightarrow \widehat{\texttt{d}})$$

*/*method: receive3 */*

$$\ldots$$

*/*overall constraint */*

$$\text{true} \Rightarrow \text{assume}\left(\left(\begin{array}{l}\widehat{\texttt{buf\_addr\_12}} \wedge \widehat{\texttt{buf\_data\_12}}\wedge\\ \widehat{\texttt{buf\_addr\_23}} \wedge \widehat{\texttt{buf\_data\_23}}\wedge\\ \widehat{\texttt{buf\_addr\_31}} \wedge \widehat{\texttt{buf\_data\_31}}\end{array}\right)\right)$$
$$\begin{array}{r}\text{strt}(\texttt{TokenRing})\wedge\\ \neg\text{prmt}(\texttt{TokenRing})\end{array} \Rightarrow \texttt{next}(\ell) = \text{true}$$
$$\begin{array}{r}\ell\wedge\\ \neg\text{abrt}(\texttt{TokenRing})\end{array} \Rightarrow \texttt{next}(\ell) = \text{true}$$

control:
inst: false
insd: ℓ
term: false

Fig. 17 AIF$^+$ of example: TokenRing

system OuterQuartz

inputs:	**outputs**:	**locals**:
bool susp	**int** o	**int** x, y
int i		**bool** s, s'
		clock C1, C2
		label ℓ, ℓ_1, ℓ_2

behavior:

/* *OuterQuartz* */

$$\begin{aligned}
\text{true} &\Rightarrow \texttt{assume}(\widehat{\texttt{i}} = \texttt{C1})\\
\text{true} &\Rightarrow \texttt{assume}(\widehat{\texttt{susp}} = \texttt{C1})\\
\text{true} &\Rightarrow \texttt{assume}(\widehat{\texttt{o}} = \texttt{C1})
\end{aligned}$$

/* *Signal* */

$$\begin{aligned}
\widehat{\texttt{s}} \wedge \text{strt}(\texttt{OuterQuartz}) &\Rightarrow \texttt{s} = \text{true}\\
\widehat{\texttt{s}} \wedge \ell \wedge \neg\texttt{susp} &\Rightarrow \texttt{s} = \texttt{s'}\\
\widehat{\texttt{s'}} \wedge \begin{pmatrix}\text{strt}(\texttt{OuterQuartz}) \vee\\ \ell \wedge \neg\texttt{susp}\end{pmatrix} &\Rightarrow \texttt{next}(\texttt{s'}) = \texttt{s} \odot \text{true}\\
\begin{pmatrix}\text{strt}(\texttt{OuterQuartz}) \vee\\ \ell \wedge \neg\texttt{susp}\end{pmatrix} &\Rightarrow \texttt{assume}(\widehat{\texttt{s}} = \widehat{\texttt{s'}})\\
\widehat{\texttt{x}} \wedge \begin{pmatrix}\text{strt}(\texttt{OuterQuartz}) \vee\\ \ell \wedge \neg\texttt{susp}\end{pmatrix} &\Rightarrow \texttt{o} = \texttt{x}\\
\widehat{\texttt{i}} \wedge \neg\widehat{\texttt{x}} \wedge \begin{pmatrix}\text{strt}(\texttt{OuterQuartz}) \vee\\ \ell \wedge \neg\texttt{susp}\end{pmatrix} &\Rightarrow \texttt{y} = \texttt{i}\\
\begin{pmatrix}\text{strt}(\texttt{OuterQuartz}) \vee\\ \ell \wedge \neg\texttt{susp}\end{pmatrix} &\Rightarrow \texttt{assume}(\widehat{\texttt{i}} = \widehat{\texttt{s}})\\
\begin{pmatrix}\text{strt}(\texttt{OuterQuartz}) \vee\\ \ell \wedge \neg\texttt{susp}\end{pmatrix} &\Rightarrow \texttt{assume}(\widehat{\texttt{y}} = \texttt{s} \wedge \widehat{\texttt{s}} \wedge \widehat{\texttt{i}})\\
\widehat{\ell} \wedge \text{strt}(\texttt{OuterQuartz}) &\Rightarrow \texttt{next}(\ell) = \text{true}\\
\widehat{\ell} \wedge \ell &\Rightarrow \texttt{next}(\ell) = \text{true}\\
\ell &\Rightarrow \texttt{assume}(\widehat{\ell} = \widehat{\text{strt}(\texttt{OuterQuartz})})
\end{aligned}$$

/* *InnerQuartz* */

$$\begin{aligned}
\texttt{C2} \wedge \begin{pmatrix}\text{strt}(\texttt{OuterQuartz}) \vee\\ \ell_2 \wedge \neg\texttt{susp}\end{pmatrix} &\Rightarrow \texttt{x} = \texttt{y}\\
\texttt{C2} \wedge (\ell_1 \wedge \neg\texttt{susp}) &\Rightarrow \texttt{x} = 2 * \texttt{y}\\
\texttt{C2} \wedge \begin{pmatrix}\text{strt}(\texttt{OuterQuartz}) \vee\\ \ell_2 \wedge \neg\texttt{susp}\end{pmatrix} &\Rightarrow \texttt{next}(\ell_1) = \text{true}\\
\texttt{C2} \wedge (\ell_1 \wedge \neg\texttt{susp}) &\Rightarrow \texttt{next}(\ell_2) = \text{true}\\
\text{true} &\Rightarrow \texttt{assume}(\widehat{\texttt{x}} = \texttt{C2})\\
\text{true} &\Rightarrow \texttt{assume}(\widehat{\texttt{x}} = \texttt{C2})\\
\text{true} &\Rightarrow \texttt{assume}(\widehat{\ell_1} = \texttt{C2})\\
\text{true} &\Rightarrow \texttt{assume}(\widehat{\ell_2} = \texttt{C2})
\end{aligned}$$

Fig. 18 AIF$^+$ of example: OuterQuartz

The resulting AIF$^+$ system after linking the modules is shown in Fig. 18. The only control signal that remained in the behavior is strt(OuterQuartz), which is still needed to start the module. All other control signals have been bound by the linker. The example also illustrates that many guards of the intermediate representation, which look complicated due to the control-flow interface signals, have become very simple after inserting the actual control-flow context in the course of linking.

The clock of the outermost module `OuterQuartz` is twice as often present as the clock of inner module `InnerQuartz` due to the sampling within the SignalC part. Note that also the other direction would be possible if oversampling is used in the SignalC process (like it is shown in the example in Fig. 12).

4 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 [12, 64] for synchronous languages and extend it by multiple clocks.

This section should serve two purposes: first, it illustrates the usage of clocked guarded actions in design flows and shows how modeled systems can be translated to formats processed by existing tools. Second, as the presented translations are very efficient, it also supports our argument that the representation of the behavior by clocked guarded actions is at an appropriate level of abstraction, providing a good balance between (1) removal of complexity from the source code level and (2) the independence of a specific synthesis target.

4.1 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

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

Figure 19 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 [64].

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 default reaction determines the value, which is covered by the equations for the carrier variable x'. If x is not set by any action, it just keeps its old value so that other actions can still read it—which is covered by the clause Trans_x.

$$\mathsf{Invar}_x :\equiv \left(\begin{array}{l} \bigwedge_{j=1}^{p} (\gamma_j \to x = \tau_j \wedge \widehat{x}) \wedge \\ \\ \left(\bigwedge_{j=1}^{p} \neg\gamma_j \right) \wedge \widehat{x} \to x = x' \end{array} \right)$$

$$\mathsf{Init}_{x'} :\equiv \mathsf{Default}(x)$$

$$\mathsf{Trans}_x :\equiv \neg\mathsf{next}(\widehat{x}) \to \mathsf{next}(x) = x$$

$$\mathsf{Trans}_{x'} :\equiv \left(\begin{array}{l} \bigwedge_{j=1}^{q} (\chi_j \to \mathsf{next}(x') = \pi_j) \wedge \\ \\ \left(\bigwedge_{j=1}^{q} \neg\chi_j \right) \wedge \mathsf{next}(\widehat{x}) \to \mathsf{next}(x') = x \\ \\ \left(\bigwedge_{j=1}^{q} \neg\chi_j \right) \wedge \neg\mathsf{next}(\widehat{x}) \to \mathsf{next}(x') = x' \end{array} \right)$$

Fig. 19 Transition relation for x

The meaning of x' is as follows: x' captures all of the delayed assignments $\mathtt{next}(x) = \pi_j$ to x, that is whenever $\mathtt{next}(x) = \pi_j$ 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.

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 $\mathtt{next}(x) = \pi_j$ 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

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

They are translated to the following clause:

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

```
...
DEFINE  _grd18 := C2 & (_strt_OuterQuartz | ell2 & !susp)
DEFINE  _grd19 := C2 & (ell1 & !susp)
...
INVAR   _grd18 -> (x = y) & _clk_x
INVAR   _grd19 -> (x = 2*y) & _clk_x
INVAR   !_grd18 & !grd_19 & _clk_x -> x = _carrier_x
TRANS   next(_clk_x) -> next(_carrier_x) = x
TRANS   !next(_clk_x) -> next(_carrier_x) = _carrier_x

INVAR   _clk_ell1 -> ell1 = _carrier_ell1 & _clk_ell1
TRANS   _grd18 -> next(_carrier_ell1) = TRUE
TRANS   !_grd18 & next(_clk_x) -> next(_carrier_ell1) = FALSE
TRANS   !_grd18 & !next(_clk_x) -> next(_carrier_ell1) = _carrier_ell1
...

SPEC    AG AF (_clk_x)
SPEC    AG ((x=y) | (x=2*y))
SPEC    AG (_clk_susp & susp) -> !_clk_x
...
```

Fig. 20 Model checking AIF$^+$ with SMV

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

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

This general format can be transformed by a straightforward syntactic translation to input which can be used by a model checker such as SMV. Figure 20 shows a part of the code for our running example `OuterQuartz` (Fig. 8), which describes the transitions for the variables `x` and ℓ_1. In the SMV file, we first define all the guards in order to share them among many clauses in the rest of the description. The next part is a simple mapping of the clauses as described in Fig. 19. Thereby, `_clk_x` represents $\widehat{x}$ and `_carrier_x` the variable x'. Finally, some specifications can be given, which are verified by SMV. In our example given in Fig. 20, we check three properties: whether the clock of variable x is always live, whether x is always equal or the double of y, and whether a suspension really deactivates the emission of x.

From the theoretical point of view, the translation to transition systems does not have any limitations. All features of the AIF$^+$ system are translated to a symbolic transition system (including assignments, assumptions and assertions). Practically, state-space explosion is always an issue, which might make model checking unusable for large systems.

4.2 SystemC simulation

The simulation semantics of SystemC is based on the discrete-event model of computation [18], 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.

```
void Module
  ::compute_x()
{
  while(true) {
    if(_clk_x.read() && γ1)
      x.write(τ1);
      ...
    else if(_clk_x.read() && γn)
      x.write(τn);
    else if(_clk_x.read())
      x.write(_carrier_x.read());
    wait();
  }
}
```

```
void Module
  ::compute_delayed_x()
{
  while(true) {
    if(ξ1)
      _carrier_x.write(π1);
      ...
    else if(ξn)
      _carrier_x.write(πn);
    else
      _carrier_x.write(x.read());
    wait();
  }
}
```

Fig. 21 SystemC: translation of immediate and delayed actions

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 *endochronous* [27, 28], i.e. there is a signal which is present in all instants of the behavior and from which all other signals can be determined. 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 [2, 27, 28] or scheduler creation for CAOS [16] 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 `_carrier_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 action $\langle \gamma \Rightarrow x = \tau_i \rangle$ is straightforward: We translate each group of actions into an asynchronous thread in

```
void OuterQuartz
  ::compute_x()
{
  while(true) {
    if(_grd18)
      x.write(y.read());
    else if(_grd19)
      x.write(2*y.read());
    else if(_clk_x.read())
      x.write(_carrier_x.read());
    wait();
  }
}
```

```
void OuterQuartz
  ::compute_delayed_x()
{
  while(true) {
    _carrier_x.write(x.read());
    wait();
  }
}

void OuterQuartz
  ::compute_delayed_ell1()
{
  while(true) {
    if(_grd18)
      ell1.write(true);
    else
      ell1.write(false);
    wait();
  }
}
```

Fig. 22 SystemC code for `OuterQuartz`

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 Fig. 21 shows the general structure of such a thread.

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 `_carrier_x` in a *clocked thread*, which is triggered by the master trigger Thereby, signals changed by the delayed actions do not affect the current fixpoint iteration and vice versa.

Consider once again our running example given in Fig. 8. The procedure describe above can be used to generate cycle-accurate SystemC code, which can be used for a detailed simulation of the given system. Each simulation step of the SystemC kernel corresponds to an instant of our system.

Figure 22 gives the SystemC code for the part that simulates the variables `x`. Similar to the SMV translation, we abbreviate guards for reuse in different SystemC processes. Then, the translation of the immediate actions writing `x` is straightforward; they correspond to the first two cases in method `OuterQuartz::compute_x()`. The last case is responsible for setting `x` to its previous value if neither of the two immediate actions fires. As already stated above, we need to do this explicitly. To this end, the previous value of variable `x` is always stored in `_carrier_x`. For variables, which are only set by delayed actions, we can simplify the general scheme of Fig. 21. In this case, we can combine the two threads, as the clocked thread `compute_delayed_ell1` in Fig. 22 illustrates: we only need a single variable, which is set by this thread.

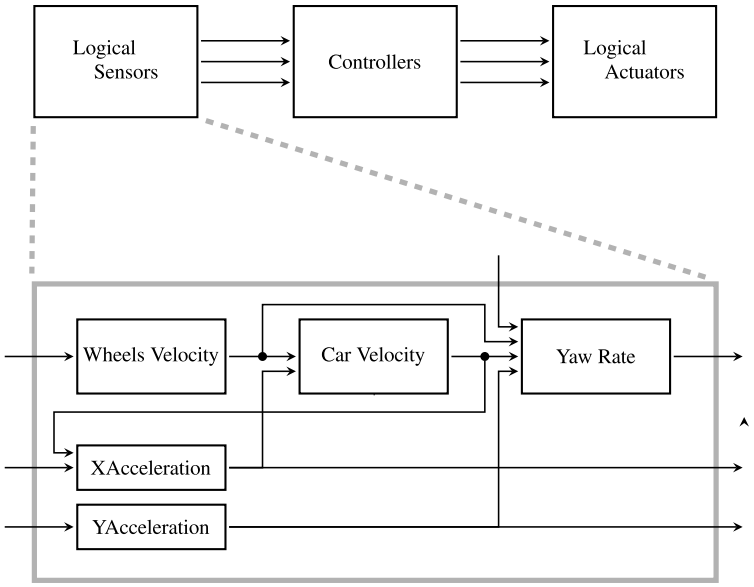


Fig. 23 Structure of VehicleStabilitySystem and LogicalSensors

5 Case study

In order to evaluate the modeling capabilities of our intermediate representation we used it for a case study taken from the automotive domain. The considered system describes a vehicle stability system, which controls the steering, traction and yaw rate of a car. Structurally, it consists of three components (see Fig. 23): the first one (LogicalSensors) encapsulates the sensors and sensor fusion, the second one (Controllers) contains the actual control software, and the last one (LogicalActuators) is responsible for the actuators.

In order to demonstrate the usability of our intermediate representation, we modeled the components of the vehicle stability system in different languages. The apparent data-flow layer is described by a Signal specification, which is a good choice: equations are good to describe pure data-flow, and the polychronous style does not constrain the rates of independent sensors, which allows us to instantiate concrete sensors at the end.

In contrast, the control-flow parts are very hard to encode in a data-flow language such as Signal. Therefore, we wrote Quartz programs for them. The rich set of precise control-flow statements makes it possible to write concise and readable models. In our example, the control-flow parts are on two different layers: below and on top of the data flow. Thus, we have a *hierarchy of different descriptions*: Quartz programs used by Signal data-flow specifications, which again are managed by mode automata described in Quartz.

To see the control-flow layer below of the data flow, consider the steering control. It is responsible to limit the steering movements at high speeds so that instable situations of the vehicle are avoided. It has three states: after the initialization, which sets up the parameters and checks all required parts, it can be activated and deactivated–depending on the speed of the vehicle. This part of the behavior is perfectly modeled by the following Quartz skeleton:

```
/* initialize device */
...
```

```
suspend {
  loop {
    /* control steering */
    ...
  }
} when(speed < SPEED_THRESHOLD);
```

The module has the input `steerIn` and the output `steerOut`, which represent the uncontrolled and controlled values of the steering angle. As they are only emitted in certain instants (which defines their clocks), the internal steps of the steering control are completely invisible to the outside. For example, during the initialization phase, no `steerOut` is computed, which hides this phase. To the rest of the system, the module is just a component that reads a steering input and writes a steering output each step.

To see the other control-flow layer on top of the data flow, consider the first component, which is responsible for sensor fusion. It must be very adaptive due to many reasons: e.g. a changing environment requires different strategies or transient hardware failures make sensor data unreliable. Therefore, on top of the data-flow, there is a control-flow part, which is responsible for switching between various modes of the system. For example, the component for the yaw rate determination is able to return data in many different ways: if the dedicated sensor fails, the lateral acceleration or the steering angle and the velocity of the rear wheels can be used to estimate the data. Similarly, the velocity can be determined from the acceleration and vice versa. As all possibilities cannot be combined arbitrarily (assume that speed and acceleration cannot be measured, then they cannot be determined from each other), the control is *on top of* the individual components.

In the actuator part of the system, we control several dual servo motors (our vehicle has an electric drive), which are given the number of revolutions per minute. If the control fails for some reason, we have to halt them so that the car stops safely. However, transient failures are quite common, we tolerate a short time (in which the old values are sustained) before we disrupt the drive. This adaptive behavior is modeled by the following Quartz fragment:

```
await(!error);
loop {
  abort {
    loop {
      /* normal mode */
      ...
    }
  } when(error);
  abort {
    /* sustain mode */
    loop {
      /* off mode */
      ...
    }
  } when(!error);
}
```

In this context, the control-flow interface (see end of Sect. 3.2) plays an important role. It is responsible that only the actions of the current mode are activated. All others are not active

since the control-flow label representing the mode are inactive or a preemption have been invoked (indicated by $\mathsf{prmt}(M)$).

Recall that preemption interrupts the execution of modules only at well-defined points. If a particular mode is aborted, it is made synchronous to the signals visible at the interface (and not between internal steps). In the vehicle stability system, this has the consequence that a mode change does not take place in the middle of a computation. For example, the steering controller will not be interrupted during initialization since these internal steps are invisible to the rest of the system.

This case study supports our claim that the proposed model is suitable to represent heterogeneous systems hierarchically composed of different descriptions. After modeling all parts and compiling them to our intermediate representation, we get the desired integrated model, which can be subsequently used in the design flow.

6 Related work

Our integration significantly differs from previous approaches: Whereas the tagged signal model [3, 29, 53, 54] only defined a general framework for comparing different MoCs, frameworks such as Ptolemy [26], ForSyDe [45], HetSC [37], SystemC-H [56] or SystemMoC [36] embed various different MoCs in a host language like Java, Haskell or SystemC. Most of the approaches support a composition of components based on different MoCs but typically only offer simulation of the combined systems.

For example, *Ptolemy* endows components which follow a particular MoC with a so-called *domain director*, which schedules the actors in the model and their actions according to the general rules of the MoC. When modules of different MoCs are composed, each one runs a director of its own, and the scheduling of actions by the actors of the submodules is orchestrated by another director which coordinates between the directors at the individual MoC level. The whole system model is therefore endowed by a hierarchy of directors, which has a certain overhead but allows an orthogonal use of MoCs. Determinism is often a burden of the writer of the model, as the director only schedules actors and their certain predesignated functions (such as *pre-fire*, *fire*, *post-fire* etc.).

For an integrated simulation, all the approaches mentioned above rely on the operational semantics of their host languages. System models are executable by construction, which is also their main purpose. However, formal verification and hardware-software (co)synthesis may become harder since the internal representation primarily aims at an efficient or/and flexible simulation. For example, synthesis from a Ptolemy model must either synthesize the directors together with the models, or has to provide a run-time system for the synthesized code.

In addition to frameworks to integrate components with different MoCs, there are also related approaches for intermediate representations. In particular, the multi-clock declarative code DC+ [60] adds clock hierarchies to traditional DC [33], the privileged interchange format of synchronous languages 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. Clocked guarded actions as used in this article share the same general idea, but have a more general scope: we are not only interested in integrating synchronous models but also asynchronous ones, such as CAOS [16] or SHIM [24]. 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.

Furthermore, AIF$^+$ was designed to support a better compositionality. As already shown in Sect. 2.2, the interfaces of AIF$^+$ make it possible to have a complete behavioral hierarchy (Fig. 2(b)), i.e. components can be called at arbitrary (control-flow) contexts in other components. In contrast, DC+ only supports the following two variants of composition.

– Basically, systems are a hierarchically organized as nodes [60], which are components with a pure data interface. They can be arbitrarily nested in DC+, which corresponds to our structural hierarchy (Fig. 2(a)).
– Another way to integrate behavior from other components are functions and procedures (basically functions with several results). Functions can be used in expressions (since they have a single return value), and procedures as actions. Hence, both of them may be generally influenced by a surrounding control-flow context (e.g. action in the scope of an abort or suspend statement). However, DC+ requires that the interfaces of functions and procedures are monochronous [60], i.e. all the input and output parameters must have the same clock.

Thus, it is impossible to implement the proposed behavioral hierarchy (Fig. 2(b)) in DC+. For imperative programs, essential information such as the control-flow context, local variable context [14] is missing. Only with these features, it is possible to write a system, which consists of a Quartz components that calls a Signal component, which itself instantiates a Quartz program. For such a system, we expect that an abortion/suspension of the outer Esterel component is forwarded through the Signal specification, which itself also become inactive until the outer Esterel component resumes the execution again.

7 Summary

In this article, we proposed clocked guarded actions as a common intermediate representation of components that were written in languages based on different models of computation. Using this common intermediate representation we are able to create new components that consist of other components so that new ways to combine heterogeneous components are achieved. Moreover, the common intermediate representation is the basis for a seamless design flow that covers simulation, verification and other kinds of analyses, as well as synthesis to different targets. In addition to the definition of a precise intermediate language based on clocked guarded actions with its formal semantics, we also described the translation of components that were written in synchronous, polychronous or asynchronous languages to clocked guarded actions. Moreover, we showed how formal verification by means of symbolic model checking and a combined simulation based on a translation to SystemC can be easily achieved with our intermediate representation.

References

1. Arvind (2003) Bluespec: a language for hardware design, simulation, synthesis and verification invited talk. In: Formal methods and models for codesign (MEMOCODE), Mont Saint-Michel, France. IEEE Comput Soc, New York, pp 249–254
2. Benveniste A, Bournai P, Gautier T, Le Guernic P (1985) SIGNAL: A data flow oriented language for signal processing. Research report 378, INRIA, Rennes, France
3. Benveniste A, Caillaud B, Carloni LP, Sangiovanni-Vincentelli AL (2005) Tag machines. In: Wolf W (ed) Embedded software (EMSOFT), Jersey City, New Jersey, USA. ACM, New York, pp 255–263
4. Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages twelve years later. Proc IEEE 91(1):64–83

5. Berry G (1991) A hardware implementation of pure Esterel. In: Formal methods in VLSI design, Miami, Florida, USA
6. Berry G (1997) Synchronous languages for hardware and software reactive systems. In: Delgado Kloos C, Cerny E (eds) International conference on hardware description languages and their applications (CHDL), Toledo, Spaine. Chapman & Hall, London, p 3
7. Berry G (1998) The foundations of Esterel. In: Plotkin G, Stirling C, Tofte M (eds) Proof, language and interaction: essays in honour of Robin Milner. MIT Press, Cambridge
8. Berry G (1999) The constructive semantics of pure Esterel
9. Berry G, Cosserat L (1985) The Esterel synchronous programming language and its mathematical semantics. In: Brookes SD, Roscoe AW, Winskel G (eds) Seminar on concurrency (CONCUR), Pittsburgh, Pennsylvania, USA. LNCS, vol 197. Springer, Berlin, pp 389–448
10. Bombana M, Bruschi F (2003) SystemC-VHDL co-simulation and synthesis in the HW domain. In: Design, automation and test in Europe (DATE), Munich, Germany. IEEE Comput Soc, New York, pp 20101–20105
11. Boussinot F, de Simone R (1991) The Esterel language. Proc IEEE 79(9):1293–1304
12. Brandt J, Gemünde M, Schneider K (2010) From synchronous guarded actions to SystemC. In: Dietrich M (ed) Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Dresden, Germany, pp 187–196. Fraunhofer Verlag
13. Brandt J, Gemünde M, Schneider K, Shukla S, Talpin J-P (2011) Integrating system descriptions by clocked guarded actions. In: Morawiec A, Hinderscheit J, Ghenassia O (eds) Forum on specification and design languages (FDL), Oldenburg, Germany. IEEE Comput Soc, New York, pp 1–8
14. Brandt J, Schneider K (2011) Separate translation of synchronous programs to guarded actions. Internal report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany
15. Brandt J, Schneider K, Edwards SA (2012) Translating SHIM to guarded actions. Technical report 387/12. University of Kaiserslautern, Kaiserslautern, Germany
16. Brandt J, Schneider K, Shukla SK (2010) Translating concurrent action oriented specifications to synchronous guarded actions. In: Lee J, Childers BR (eds) Languages, compilers, and tools for embedded systems (LCTES), Stockholm, Sweden. ACM, New York, pp 47–56
17. Caspi P, Halbwachs N, Pilaud D, Plaice JA (1987) LUSTRE: A declarative language for programming synchronous systems. In: Principles of programming languages (POPL), Munich, Germany. ACM, New York, pp 178–188
18. Cassandras CG, Lafortune S (2008) Introduction to discrete event systems, 2nd edn. Springer, Berlin
19. Chandy KM, Misra J (1989) Parallel program design. Addison-Wesley, Austin
20. Chatelain A, Mathys Y, Placido G, La Rosa A, Lavagno L (2001) High-level architectural co-simulation using Esterel and C. In: Hardware-software codesign (CODES), Copenhagen, Denmark
21. Dershowitz N, Okada M, Sivakumar G (1988) Canonical conditional rewrite systems. In: Lusk EL, Overbeek RA (eds) Conference on automated deduction (CADE), Argonne, Illinois, USA. LNCS, vol 310. Springer, Berlin, pp 538–549
22. Dijkstra EW (1975) Guarded commands, nondeterminacy and formal derivation of programs. Commun ACM 18(8):453–457
23. Dill DL (1996) The Murphi verification system. In: Alur R, Henzinger TA (eds) Computer aided verification (CAV), New Brunswick, New Jersey, USA. LNCS, vol 1102. Springer, Berlin, pp 390–393
24. Edwards SA, Tardieu O (2005) SHIM: a deterministic model for heterogeneous embedded systems. In: Wolf W (ed) Embedded software (EMSOFT), Jersey City, New Jersey, USA. ACM, New York, pp 264–272
25. Eker J, Janneck JW (2002) CAL actor language—language report. EECS Department, University of California at Berkeley
26. Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y (2003) Taming heterogeneity—the Ptolemy approach. Proc IEEE 91(1):127–144
27. Gamatie A (2010) Designing embedded systems with the SIGNAL programming language. Springer, Berlin
28. Gamatié A, Gautier T, Le Guernic P, Talpin JP (2007) Polychronous design of embedded real-time applications. ACM Transactions on Software Engineering and Methodology (TOSEM) 16(2)
29. Girault A, Lee B, Lee EA (1999) Hierarchical finite state machines with multiple concurrency models. IEEE Trans Comput-Aided Des Integr Circuits Syst 18(6):742–760
30. Gramlich B, Wirth C (1996) Confluence of terminating conditional rewrite systems revisited. In: Ganzinger H (ed) Rewriting techniques and applications (RTA), New Brunswick, New Jersey, USA. LNCS, vol 1103. Springer, Berlin, pp 245–259
31. Gössler G, Sangiovanni-Vincentelli A (2002) Compositional modeling in Metropolis. In: Sangiovanni-Vincentelli AL, Sifakis J (eds) Embedded software (EMSOFT), Grenoble, France. LNCS, vol 2491. Springer, Berlin, pp 93–107

32. Halbwachs N (1993) Synchronous programming of reactive systems. Kluwer, Dordrecht
33. Halbwachs N (1998) The declarative code DC, version 1.2a
34. Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous dataflow programming language LUSTRE. Proc IEEE 79(9):1305–1320
35. Harel D, Naamad A (1996) The STATEMATE semantics of Statecharts. ACM Trans Softw Eng Methodol 5(4):293–333
36. Haubelt C, Falk J, Keinert J, Schlichter T, Streubühr M, Deyhle A, Hadert A, Teich J (2007) A SystemC-based design methodology for digital signal processing systems. EURASIP J Embed Syst 2007(1):15
37. Herrera F, Villar E (2007) A framework for heterogeneous specification and design of electronic embedded systems in SystemC. ACM Trans Des Autom Electron Syst 12(3):1–31
38. Hoare CAR (1978) Communicating sequential processes. Commun ACM 21(8):666–677
39. Hoare CAR (1983) Communicating sequential processes. Commun ACM 26(1):100–106
40. Hoe JC, Arvind (1999) Hardware synthesis from term rewriting systems. Technical report CSG-MEMO 421-a, Computer Science and Artificial Intelligence Laboratory. Cambridge, Massachusetts, USA
41. Hoe JC, Arvind (2004) Operation-centric hardware description and synthesis. IEEE Trans Comput-Aided Des Integr Circuits Syst 23(9):1277–1288
42. IEEE (2005) IEEE standard hardware description language based on the Verilog hardware description language. In: IEEE Std, pp 1394–2005
43. IEEE (2005) IEEE standard SystemC language reference manual. In: IEEE Std, New York, New York, USA, December, pp 1666–2005
44. IEEE (2008) IEEE standard VHDL language reference manual. In: IEEE Std, pp 1076–2008
45. Jantsch A (2004) Modeling embedded systems and SoCs. Morgan Kaufmann, Los Altos
46. Jantsch A (2006) Models of computation for networks on chip. In: Application of concurrency to system design (ACSD), Turku, Finland. IEEE Comput Soc, New York, pp 165–178
47. Kaplan S (1984) Conditional rewrite rules. Theor Comput Sci 33:175–193
48. Koo TJ, Liebman J, Ma C, Horowitz B, Sangiovanni-Vincentelli A, Sastry S (2002) Platform-based embedded software design for multi-vehicle multi-modal systems. In: Sangiovanni-Vincentelli AL, Sifakis J (eds) Embedded software (EMSOFT), Grenoble, France. LNCS, vol 2491. Springer, Berlin, pp 32–45
49. Le Guernic P, Gauthier T, Le Borgne M, Le Maire C (1991) Programming real-time applications with SIGNAL. Proc IEEE 79(9):1321–1336
50. Lee EA, Messerschmitt DG (1987) Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans Comput 36(1):24–35
51. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
52. Lee EA, Parks T (1995) Dataflow process networks. Proc IEEE 83(5):773–801
53. Lee EA, Sangiovanni-Vincentelli A (1996) Comparing models of computation. In: International conference on computer-aided design (ICCAD), San Jose, California, USA. ACM/IEEE Comput Soc, New York, pp 234–241
54. Lee EA, Sangiovanni-Vincentelli A (1998) A framework for comparing models of computation. IEEE Trans Comput-Aided Des Integr Circuits Syst 17(12):1217–1229
55. Passerone C, Lavagno L, Chiodo M, Sangiovanni-Vincentelli AL (1997) Fast Hardware/Software co-simulation for virtual prototyping and trade-off analysis. In: Design automation conference (DAC), Anaheim, California, USA. ACM, New York, pp 389–394
56. Patel HD, Shukla SK, Bergamaschi RA (2007) Heterogeneous behavioral hierarchy extensions for SystemC. IEEE Trans Comput-Aided Des Integr Circuits Syst 26(4):765–780
57. Poigné A, Holenderski L (1995) Boolean automata for implementing pure Esterel. In: Arbeitspapiere, vol 964. GMD, Sankt Augustin, Germany
58. Rocheteau F, Halbwachs N (1992) Implementing reactive programs on circuits: a hardware implementation of LUSTRE. In: de Bakker JW, Huizing C, de Roever WP, Rozenberg G (eds) Real-time: theory in practice, Mook, The Netherlands. LNCS, vol 600. Springer, Berlin, pp 195–208
59. Rowson JA (1994) Hardware/Software co-simulation. In: Design automation conference (DAC), San Diego, California, USA. ACM, Berlin, pp 439–440
60. ESPRIT Project (1997) Safety critical embedded systems (SACRES). The declarative code DC+, version 1.4, November
61. Sander I, Jantsch A, Lu Z (2003) Development and application of design transformations in ForSyDe. In: Design, automation and test in Europe (DATE), Munich, Germany. IEEE Comput Soc, New York, pp 10364–10369
62. Sangiovanni-Vincentelli AL, Carloni LP, De Bernardinis F, Sgroi M (2004) Benefits and challenges for platform-based design. In: Malik S, Fix L, Kahng AB (eds) Design automation conference (DAC), San Diego, California, USA. ACM, New York, pp 409–414
63. Schneider K (2001) Embedding imperative synchronous languages in interactive theorem provers. In: Application of concurrency to system design (ACSD), Newcastle Upon Tyne, England, UK. IEEE Comput Soc, New York, pp 143–154

64. Schneider K (2009) The synchronous programming language Quartz. Internal report 375. Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December
65. Schneider K, Brandt J, Schuele T (2006) A verified compiler for synchronous programs with local declarations. Electron Notes Theor Comput Sci 153(4):71–97
66. Singh G, Shukla SK (2007) Algorithms for low power hardware synthesis from concurrent action oriented specifications CAOS. Int J Embed Syst 3(1/2):83–92
67. Todesco ARW, Meng THY (1996) Symphony: a simulation backplane for parallel mixed-mode cosimulation of VLSI systems. In: Design automation conference (DAC), Las Vegas, Nevada, USA. ACM, New York, pp 149–154
68. Zebelein C, Falk J, Haubelt C, Teich J (2008) Classification of general data flow actors into known models of computation. In: Formal methods and models for codesign (MEMOCODE), Anaheim, California, USA. IEEE Comput Soc, New York, pp 119–128
69. Zivojnovic V, Meyr H (1996) Compiled HW/SW co-simulation. In: Design automation conference (DAC), Las Vegas, Nevada, USA. ACM, New York, pp 690–695.