Modular Design of Reactive Systems

Cristina Cerschi Seceleanu
Åbo Akademi and TUCS, Finland
ccerschi@abo.fi

Tiberiu Seceleanu
University of Turku, Finland
tiberiu.seceleanu@utu.fi

Abstract

We concentrate on two major aspects of reactive system design: behavior control and modularity. These are studied from a formal point of view, within the framework of action systems. The traditional interleaving paradigm is completed with a new barrier synchronization mechanism. This is achieved by introducing a new parallel composition operator, applicable to both discrete and hybrid models. While offering improvements with respect to control and modularity, the approach uses the correctness preserving mechanisms provided by the underlying reasoning environment.

Keywords: Reactive systems, Action systems, Modular design, Concurrency

1. Introduction

In this study, we tackle issues regarding the design of reactive systems, be they software or hardware-targeted systems, in the formal framework of action systems. We approach aspects of design from the perspective of the system-level integrator that has access to a library of predefined subsystems. His only task is to appropriately connect them in order to obtain the system functionality.

Action systems, introduced by Back and Kurki-Suonio [2] is a state-based formalism, relying on an extended version of Dijkstra’s language of guarded commands [10]. It uses an interleaving semantics for handling concurrency. Parallel behavior is modeled by interleaved actions that can be executed in any order. This approach goes together with behavioral nondeterminism, as observations of an interleaved model are sequential, therefore the updates of two systems executing in parallel may not be consistent over a set of executions [14].

We provide an additional concurrency mechanism for action systems, namely synchronization, as a way to describe controllable behavior of reactive systems. For this purpose, we define a new parallel composition operator, applicable to both discrete and hybrid models. The concepts that we introduce still rely on the rigorous techniques of action systems, while providing the designer with additional means of system development. Our goal is completed by showing that the new virtual execution environment also enhances the capabilities of our framework, for modular design. Components may be picked up from existing libraries and just plugged into the system representation. The traditional techniques of trace refinement [4] can be used to ensure that the implementation is correct with respect to a specification that captures the system’s global reaction to all sets of inputs.

2. Action Systems

An action system (AS, henceforward) is a collection of actions (guarded commands). An AS is built according to the following syntax [2]:

\[ A(z : T_0) \triangleq \begin{array}{l} \text{begin} \quad \text{var} \quad x : T_0 \cdot \text{Init} ; \quad \text{do} \ A_1[\ldots] A_n \quad \text{od} \end{array} \quad (1) \]

Here, \( A \) contains the declaration of local variables \( x \) and global variables \( z \), followed by an initialization statement \( \text{Init} \) and the actions \( A_1[\ldots] A_n \). The initialization statement assigns starting values to \( x \) and \( z \). We regard an action \( A_i \) as being of the form \( g_i \rightarrow S_i \). Thus, an action is enabled and its body \( S_i \) is executed, when the guard \( g_i \) evaluates to true.

An AS is viewed as part of a more complex structure, the rest of which communicates with the AS via shared (read and written) variables. We use the following notations: the set of state variables accessed by some action \( A \) is denoted \( vA \), and is composed of the read variable set of action \( A \), denoted \( rA \), and the write variable set of action \( A \), denoted \( wA \). We have that \( vA = rA \cup wA \). At the system level we have: the access set, \( vA \), split into the global read / write variables, denoted by \( grA/wA \) and the local read / write variables, denoted by \( lrA/lwA \). An action \( A \) is global, if \( grA \cap lwA \neq \emptyset \) or local, if \( wA \subseteq lwA \).

A statement \( S_i \) is defined by the following grammar:

\[ S_i :: \begin{array}{l} \text{skip} \quad \text{(stuttering, empty statement)} \ 
|x := e| \quad \text{((multiple) assignment)} \ 
|S_m ; \ldots ; S_n| \quad \text{(sequential composition)} \ 
g_m \rightarrow S_m \ldots \rightarrow g_n \rightarrow S_n \quad \text{(nondeterministic choice)} \ 
x := x'.Q \quad \text{(nondeterministic assignment)}\end{array} \]

Above, \( S_m, \ldots, S_n \) are statements, \( g_m, \ldots, g_n \) and \( Q \) are predicates (boolean conditions), \( x \) a variable or a list of...
variables, and $a$ an expression or a list of expressions. Actions can be much more general, but this simple syntax suffices for the purpose of this paper. A loop can be reduced to iterations [5]. Therefore, the while loop is written as while $g$ do $S$ od = do $g$ → $S$ od. In this paper, we do not consider nested loops.

Statements in AS are defined by the weakest precondition semantics, consistent with Dijkstra’s original semantics for the guarded commands languages [10]. For statement $S$ and postcondition $Q$, the formula $wp(S, Q)$, called the weakest precondition of $S$ with respect to $Q$, gives the largest set of initial states from which statement $S$ is guaranteed to terminate in a state satisfying $Q$. Here, we assume that all statements are conjunctive predicate transformers (functions from predicates to predicates), that is, $\forall p, q \cdot wp(S, (p \land q)) = wp(S, p) \land wp(S, q)$.

For statement $S_i$, $wp(S_i, \text{false})$ represents the set of initial states for which $S_i$ is guaranteed to establish any postcondition, that is, behaves miraculously. A statement is enabled only in those initial states in which it behaves non-miraculously. Therefore, the guard of $S_i$ is defined as $g_i \triangleq \neg wp(S_i, \text{false})$.

When at least one action is enabled in a given AS $A$, we say that $A$ is enabled. We obtain information on the enabledness of a system $A$, given by (1), by evaluating the predicate $ggA$: $ggA \triangleq \bigwedge_{k=1}^{N} gh_k$.

### 3. The Traditional Model of Parallel Execution

Parallel execution of AS is modeled by interleaving actions [2]. The execution of an AS assumes that there exists a virtual external entity - the execution controller (controller in short) - which, at any moment, knows which actions are enabled. After the initialization, the controller non-deterministically selects for execution any of the enabled actions, after which the system moves to a new state. We call this operation an execution round. After this, the controller evaluates the new state, observes the enabled actions and starts another execution round. Termination is normal if no action of the composition is enabled anymore.

**Example.** We consider the task of modeling a digital filter [11]. Briefly, a filter is a module that takes as input a sequence of samples, performs certain operations on it and delivers as output a corresponding sequence of samples. The incoming sequence is described as $X[n]$, where $X$ is the input signal and $n$ identifies the sample position; a similar notation applies to the output signal $Y$, for which we have the samples $Y[n]$. The relation between the input and output is given by $Y[n] = \sum_{k=0}^{N-1} h[k] \times X[n-k]$, where the vector $h[0..N-1]$ contains the filter coefficients. Hence, apart from the incoming current sample of $x$, $N-1$ previous samples are stored in a buffer, and can be accessed by the filter.

![Figure 1. Simple filter representation.](image)

From the above informal description of the filter we can identify two submodules of such a device: the storing FIFO-buffer and the actual implementation of the filter. We model the signal source by system $S$, the buffer by system $B$, and the actual filter by system $F$ (Fig. 1 a)). The complete AS description is given as $P \triangleq S || B || F$ (Fig. 2).

![Figure 2. Filter model.](image)

Observe first that an interleaved execution of $P$ would not ensure that every signal emitted by $S$ is correspondingly received by $B$ and $F$: several executions of $S$ may be selected, before any of $B$ or $F$. Also, different values can be assigned to $Y$ for the same sample provided by $S$, depending on the order of selection for execution of $B$ and $F$. Both problems may be solved by specifying the order in which the submodules of $P$ should be executed. Hence, the systems should know about the status of the partners. This is achieved by introducing communication variables $reqs, reqF, acks, ackF$, and devising a communication protocol, such that the desired order is enforced. The systems $B$ and $F$ (on which we concentrate next) may be re-modeled as in Fig. 3, corresponding to Fig. 1 b), where the communication variables are shown as dotted lines.

Consider that, in the above example, $X$ is an audio signal and $F_1$ models a low-pass filter. The output of $F_1$ goes to the woofer speaker of one’s audio system. We would also like to have a high-pass filter, the output of which goes to the speakers of the corresponding audio system. We want to reuse the previously designed modules and then add one that detects the high frequencies of the input signal. The new module is modeled by the system $M_1$ - Fig. 4. In ord-
global variables of the system. In hardware, this translates into "more wires"; in software, this violates the principle of modularity, and the interleaved approaches are suitable for the two filters to read its data, before updating $Z$. Hence, we have to change the representation of $B_1$ (Fig. 4).

$$M_1((\text{req}, \text{ack}, \text{ack}_M), \text{ack}_M) \equiv \text{Boo}(X, Z(0, N - 2), W)$$

$$\begin{align*}
&\begin{array}{c}
\text{Fig. 4. Communicating models.}
\end{array}
\end{align*}$$

Discussion. In order to reduce the implicit nondeterministic behavior of the interleaving model of execution, one may introduce control channels, for ensuring that data emitted by one source is not missed by any of the intended targets, or that data is processed in a correct manner. Still, an observer of the composed system $P_2 \equiv B_2 || M_1$ (the listener, in the example) has access to both output sequences, $Y(n)$ and $W(n)$. Depending on the execution order of $F_1$ and $M_1$, until the listener observes the new output ($Y(n+1), W(n+1)$), it will also observe the intermediate state, either ($Y(n), W(n+1))$ or ($Y(n+1), W(n)$)), which is also an incorrect aspect of the design. A solution is provided, again, by the introduction of new communication channels, between $F_1$ and $M_1$, on one side, and the observer, on the other. What happens if multiple, different observers become necessary in the design?

Any extension / reduction of the design elements requires an internal change of the involved subsystems. This destroys the idea of a modular design flow and the reuse of components. We may assign meanings like "data valid", "operation finished", etc., to the signals of the communication channels, thus the interleaved approaches are suitable for asynchronous designs. Unfortunately, these signals are global variables of the system. In hardware, this translates into “more wires”; in software, this violates the principle of information hiding.

4. Synchronized Parallel Environments

We want to build an environment in which the response of the system is a collection of the individual component reactions to the input stimuli. The solution that we propose requires that the subsystems synchronize when the global variables of the compound system are updated. Thus, we extend the execution round to an execution cycle, defined by the activities carried out between two global states: it is a sequence of rounds in which each participating AS updates the local variables, as necessary, followed by a last round, in which, simultaneously, all the global variables are updated, accordingly.

From the controller’s point of view, we can imagine the following scenario. It selects for execution an enabled action from one component AS. If the action updates global variables, the system is marked as "executed" and no other action can be selected from that system. However, the other participants, and the external observers, do not see the changes yet. Another action is then selected, from an "unexecuted" AS. The process continues until all the components are marked “executed”, signaling the end of a cycle.

Definition 1 Consider the action system $A$.

$$A(z; T_z) \equiv \begin{array}{c}
\begin{array}{c}
\text{begin var } z : T_z \bullet \text{Init;} \text{do } g_1 \rightarrow L \mid g_S \rightarrow \text{Sod end}
\end{array}
\end{array}$$

We say that $A$ is a proper action system if:

1. $g_1 \subseteq w_S$ meaning that $S$ is a global action of $A$.
2. $w_L \subseteq w_A$ meaning that $L$ is a local action of $A$.
3. $wp(\text{do } g_L \rightarrow L \mid \text{do } g_S \rightarrow S) \equiv \text{true}$ meaning that the execution of $L$, taken separately, terminates, leaving $S$ enabled.

Definition 2 Let us consider n proper action systems ($k = 1 \ldots n$):

$$A_k(z_k) \equiv \begin{array}{c}
\begin{array}{c}
\text{begin var } z_k : T_{z_k} \bullet \text{Init;} \text{do } g_{z_k} \rightarrow L_k \mid g_S \rightarrow S_k \text{ od end,}
\end{array}
\end{array}$$

for which we also have that $\forall j, k \in [1,n], j \neq k, ((g_k \cap g_{z_k} = \emptyset) \lor (\forall z_k x_k = 0))$. Their synchronized parallel composition is a new action system $P = A_1 \ldots A_n$, given by:

$$\begin{array}{c}
\begin{array}{c}
\text{Fig. 4. The systems } M_1 \text{ and } B_2.
\end{array}
\end{array}$$

$$P(z) \equiv \begin{array}{c}
\begin{array}{c}
\text{begin var } z : T_z \bullet \text{sr}()[1,n] : \text{Boo}; \text{run } : \text{Nat } \bullet \text{Init;} \text{do } g_{sr} \rightarrow (\text{run } = 0 \lor -\text{sr}[1]) \rightarrow \text{sr}[1] : \text{true}; \text{run } = 1
\end{array}
\end{array}$$

The operator "\#" (‘sharp’) is called the synchronized parallel operator. The set $\mathbb{Z}$ of global variables of $P$ is, initially, the union of the global variable sets of all individual systems, $z = \bigcup_k z_k$, without duplicates. It may be possible
that communication between several submodules of \( \mathcal{P} \) (the composing systems \( \mathcal{A}_i \)) should not be disclosed at the interface of \( \mathcal{P} \). Therefore, the variables modeling such channels will be hidden within the system \( \mathcal{P} \). They will not appear in \( z \).

Further, the local variables \( x \) of the new action system \( \mathcal{P} \) are the union of the local variables \( x_k \), to which we add the hidden variables. We also add copies \( (wS_k/c) \) of the original write variables of each action body \( S_k \). They replace the original variables \( wS_k \), therefore we have \( S' = S_k[wS_k/c/wS_k] \). Finally, the list \( x \) is completed by adding the array \( \text{sel} \) and the execution indicator, \( \text{run} \). We have that \( g_{\text{def}} \equiv V_1^\ast (g_{L_2}^L \lor g_{S_2}^S) \).

The \textit{Init} statement is a sequential composition of the individual \textit{Init} statements to which we add the initialization of variables \( wS_k/c \), \( \text{sel} \) and \( \text{run} \), and the action \textit{Update} is a sequence of simple assignments:

\[
\begin{align*}
\text{Init} & \triangleq \text{Init}_1 \ldots \text{Init}_n; \text{wS}_1 := \ldots \text{wS}_n; \text{sel} := \text{false} \\
\text{Update} & \triangleq \text{wS}_1 := \text{wS}_1/c; \ldots \text{wS}_n := \text{wS}_n/c; \text{sel} := \text{false}
\end{align*}
\]

The definition of the \( \ast \) operator says that, whenever there is a change in the input, such a composition of \( \mathcal{A} \) reduces the number of subcomponents, and the result is a collection of individual systems. The system composition reacts only if at least one subcomponent is enabled \((\exists k \in [1..n] \bullet g_{L_k}^L \lor g_{S_k}^S \equiv \text{true})\). The variable \( \text{run} \) identifies the system that is selected for execution. The variable \( \text{sel} \) stores the information on which are the executing, or already executed systems. Whenever all its elements \((\text{sel} \equiv \text{sel}[1] \land \ldots \land \text{sel}[n])\) become true and \( \text{run} = 0 \), we have reached the end of an execution cycle.

**Theorem 1** Assume that the proper action systems \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) are of the form given by (2). Then, the synchronized parallel composition \( \hat{\mathcal{A}}_1 \parallel \hat{\mathcal{A}}_2 \) satisfies the following properties:

\[
\begin{align*}
(a) \quad & \hat{\mathcal{A}}_1 \parallel \hat{\mathcal{A}}_2 \text{ is a proper action system} \\
(b) \quad & \hat{\mathcal{A}}_1 \parallel \hat{\mathcal{A}}_2 = \hat{\mathcal{A}}_2 \parallel \hat{\mathcal{A}}_1 \\
\end{align*}
\]

**Design Implications.** We revisit briefly the example proposed in Section 3. Consider that instead of the parallel composition \( \mathcal{B} \parallel \mathcal{F} \), we write the description of our system as \( \mathcal{B} \parallel ^\ast \mathcal{F} \). It is easy to check that the components \( \mathcal{B}, \mathcal{F} \) are proper AS. Therefore, we do not have to add communication channels to any of the respective subsystems, which remain as described in Fig. 2. Also, then, the multiplicity of targets stops being an issue for the composition. We can introduce as many \( \mathcal{F} \)-like systems as required, without modifying \( \mathcal{B} \) in order to accommodate their presence. Additionally, an external observer will always observe only the state \((Y(n + 1), W(n + 1))\), regardless of the order in which the systems \( \mathcal{F} \) and the corresponding \( \mathcal{M} \) (\( \mathcal{M}_1 \) without the communication variables) are selected for execution.

5. **Design Process**

Crucial to a module-based design context is the possibility to separately analyze and, if necessary, improve the functionality of the subsystems, optimize them for a given technology, or map them to existent library elements. These actions may involve certain transformations of the initial representations, which have to be guaranteed correct, with respect to behavior. Within the refinement calculus, which is our reasoning environment, such correct transformations are ensured by refinement rules [1, 4].

**Invariants.** A predicate \( I(\varphi A) \) is an invariant of the action \( \varphi A \equiv g \rightarrow S \), if it holds prior to and after the execution of \( A \). We then say that \( I \) is preserved by \( A \), that is, \( g \land I \Rightarrow \text{wp}(SI) \). At the system level, a predicate \( I(\varphi A) \) is an invariant of the AS \( \mathcal{A} \), given by (1), if it is established by \( \text{Init} \), that is, \( \text{true} \Rightarrow \text{wp}(\text{Init}, I) \), and also if it is preserved by each action \( A_i \).

**Definition 3** A predicate \( I \) is a proper invariant of a proper AS \( \mathcal{A} \), if \( \forall \varphi \not\in wS \cdot g_S \equiv \{I(wS/wS) \equiv I(wS/wS)\} \).

The above definition says that, following the execution of the global action \( g_S \rightarrow S \), the computed value of a proper invariant \( I \) depends on the variables in \( wS \), only.

**Refinement of actions and AS.** An action \( \mathcal{A} \) is (algorithmically) refined by the action \( C \), written \( \mathcal{A} \subseteq C \), if, whenever \( A \) establishes a certain postcondition, so does \( C \) [1]. Moreover, let \( \hat{R}(a, c, z) \) (simply written as \( R \)) be a boolean abstraction relation, which links the abstract local variables \( a \) to the concrete local variables \( c \). Additionally, let \( I \) be an invariant of the action \( C \). Then, action \( A \) is data refined by action \( C \) using the relation \( R \) and the invariant \( I \), that is, \( A \subseteq_{R,I} C \), if

\[
\forall Q \cdot R \land I \land \text{wp}(A, Q) \Rightarrow \text{wp}(C, 3a \cdot R \land I \land Q),
\]

where \( Q \) is a predicate on the variables \( a, z \), and \((\exists a, R \land I) \) is a predicate on \( a, c, z \).

**Lemma 1** Given the proper action systems

\[
\begin{align*}
\mathcal{A}(z, a) & \triangleq \begin{array}{c}
\text{begin} \ a \ • \ z, a := a_0, z_0; \\
d \ g^L_0 \rightarrow L_A \ g^S_0 \rightarrow S_A \ \text{od end}
\end{array} \\
\mathcal{C}(z) & \triangleq \begin{array}{c}
\text{begin} \ c \ • \ c, z := c_0, z_0; \\
d \ g^L_0 \rightarrow L' \ g^S_0 \rightarrow S' \ g_X \rightarrow X \ \text{od end}
\end{array}
\end{align*}
\]

let \( R \) be an abstraction relation and \( I \) a proper invariant of \( C \). The system \( C \) refines the system \( A, A \subseteq_{R,I} C \), if:

1. Initialization: \( R(a_0, c_0, z_0, z_0) \land I(c_0, z_0) \equiv \text{true} \)
2. Main actions: \( (g^L_0 \rightarrow L_A \leq_{R,I} g^L_{L'} \rightarrow L'_A) \land (g^S_0 \rightarrow S_A \leq_{R,I} g^S_{S'} \rightarrow S'_A) \)
3. Auxiliary action: \( \text{skip} \leq_{R,I} S_X \rightarrow X \)
4. Continuation condition: \( R \land I \land (g^L_0 \lor g^S_0) \Rightarrow g^L_X \lor g^S_X \lor g_X \)
5. Properness: \( R \land I \Rightarrow \text{wp}(d, g_X \rightarrow X \mid g^L_X \rightarrow L'_A \ \text{od end}, \neg(g_X \lor g^L_X \lor g^S_X)) \)

The first four requirements of the lemma are adaptations of the original ones, given in [6]. The fifth strengthens the original request by specifying that the new group of local
actions, $g_X \rightarrow X \parallel g^C \rightarrow L'_A$ must terminate and establish the necessary conditions for the (possibly) new global action $g^C \rightarrow S'_A$ to execute.

**Refinement Example.** Let us consider a hardware implementation for the filter introduced in section 3. A direct mapping of the filter functionality on hardware elements is represented in Fig. 5 a). Characteristic to this implementation of system $F$ is the parallel processing and the large area occupied by the hardware elements. A functionally equivalent implementation (Fig. 5 b)) results from a serial representation of the filtering device, requiring a reduced silicon area. We transform the original system $F$ into $F_S$ (Fig. 6).

**Figure 5. Implementation of the filter.**

From a system level point of view, we should check that the invariant $I$ (Fig. 6) is a proper invariant of $F$ (we consider a synchronized perspective on the system composition), we will immediately obtain that $S \models B \models F \models F_S$ (notice that $F_S$ is a proper AS). Besides this, a previous addition of module $M$ would not change the refinement, and we could have $S \models B \models F \models M \models F_S \models M$.

6. **Continuous Action Systems**

A continuous action system (CAS) [3] is of the form

$$C(z) \triangleq \begin{cases} \begin{array}{l}
\text{begin } x : \text{Real} \rightarrow T \text{ Init;}
\text{do } g_1 \rightarrow S_1; \ldots; g_m \rightarrow S_m \text{ od end}
\end{array} \end{cases}$$

Here, $\text{Real}$ stands for the non-negative reals, and models the time domain.

The execution of a CAS uses an implicit variable $\text{now}$, showing the present time. The actions may refer the value of $\text{now}$, but they can not change it. After the initialization, the system will start evolving, with time (measured by $\text{now}$) moving forward continuously. The execution resembles closely that of an ordinary AS, with the difference that, after the changes stipulated by $S_i$ have been done, the system evolves to the next time instance when one of the actions is enabled. We write $x : e$ rather than $x := e$, to emphasize that only the future behavior of the variables $x$ is changed. We explain the meaning of $C$ by translating it into an ordinary (discrete) AS, $\tilde{C}$:

$$\tilde{C}(z) \triangleq \begin{cases} \begin{array}{l}
\text{begin } x : \text{Real} \rightarrow T \text{ Init;}
\text{do } g_1 \rightarrow S_1; \ldots; g_m \rightarrow S_m \text{ od end}
\end{array} \end{cases}$$

The additional terms of $I$ help us make the connection between the copies of the variable $i$ and the respective original variables, at the moment when the action $\text{Update}$ is executed.

**Corollary 1** Consider the proper action systems $A_k$ as in Definition 2, and an abstraction relation $R_j$. The systems $A_k$ preserve the proper invariants $I_k$, respectively. Then

$$A_k \models B \models A_k \models \ldots \models A_k \models \ldots \mathbf{R} \models \ldots \mathbf{R} \models \ldots$$

The interpretation of Corollary 1 is that each component of a synchronized parallel composition may be refined in isolation, without knowledge about the invariants of the other components. Moreover, individual properties (as expressed by each $I_k$) are preserved through refinement. The module designer is responsible with improving the performance of the modules, and this is transparent for the integrator designer.

A conclusion similar to ours is reached in [7] for the parallel composition of AS. However, this is achieved provided that the invariants of all the subsystems are known, and a noninterference relation between them proves to hold.

**Refinement Example.** If we check $F \models F_S$ in the context of Lemma 1, (we consider a synchronized perspective on the system composition), we will immediately obtain that $S \models B \models F \models F_S$ (notice that $F_S$ is a proper AS). Besides this, a previous addition of module $M$ would not change the refinement, and we could have $S \models B \models F \models M \models F_S \models M$.
In \( \bar{A} \), the variable \( \text{now} \) is declared, initialized and updated explicitly. It models the starting time and the succeeding moments when some action is enabled. The value of a variable \( v \) or of an expression \( e \) at a given moment of time \( t \) is identified by \( v.t \) or \( e.t \), respectively. Their values at the current moment are consequently given by \( v.\text{now} \) and \( e.\text{now} \). The value of \( \text{now} \) is updated by the statement \( N \). The function next gives a moment of time when at least one action is enabled. If no action will ever be enabled, then the second branch of the definition will be followed, and \( \text{now} \) will denote the moment of time when the last discrete action was executed, the system terminating with the last assigned values for the variables.

The parallel composition of several CAS is defined using the same method as for composing ordinary AS. One needs to combine the component CAS before translating them into the corresponding discrete AS, to ensure that the composed system uses a unique variable \( \text{now} \).

**Synchronized Parallel CAS.** The synchronized parallel composition of CAS resembles the corresponding discrete case introduced in section 4. The semantics of a proper CAS \( A(z) \) is given by the discrete translation:

\[
A(z) \triangleq \begin{cases}
\text{begin} \ \text{var} \ z : \text{Real} \rightarrow T \bullet \text{now} := 0; \text{Init} ; N ; \text{od end}
\end{cases}
\]

Further, consider \( n \) proper CAS as being \( (k = 1 \ldots n) \):

\[
A_k(z_k) \triangleq \begin{cases}
\text{begin} \ \text{var} \ z_k : \text{Real} \rightarrow T \bullet \text{Init}_k ;
\text{do} \ g_{k, \text{now}} : T \rightarrow \text{Real} \ ; \ g_{k, \text{now}} : \text{Real} \rightarrow S_k ; \text{od end}
\end{cases}
\]

Their synchronized parallel composition is a new CAS, \( P = A_1 \parallel \ldots \parallel A_n \). Its semantics is given following the lines of Definition 2, as illustrated in Fig. 7. The variables of \( P \) are functions from Real_\( \bullet \) to some type \( T \), and the variables \( \text{sel}[1..n] \) and \( \text{run} \) are also written as functions from time to types Bool and Nat, respectively. The action guards of the component systems are evaluated at time point \( \text{now} \). The time is not advanced before all CAS components have updated their global variables, indicated by \( \text{sel} \land \text{run} = 0 \equiv \text{false} \). Also observe that, if no component system is supposed to react to a specific input situation, the composition is disabled (\( \text{gg} \equiv \text{false} \)). The theoretical results obtained for discrete synchronized AS apply to synchronized CAS, also, due to the discrete AS representation of the latter.

**Example - Hybrid system analysis.** Let us consider an abstract model of a simple heating-cooling hybrid control system, which keeps the temperature inside a place where some thermic processes happen, between a minimum and a maximum value. The system is equipped with a controller that either increases the temperature (modeled by \( \theta \)) until it reaches the maximum allowed value (\( \theta_M \)) or decreases \( \theta \), until it reaches a minimum value (\( \theta_m \)). These processes develop at speeds \( v_h \), for heating, and \( v_c \), for cooling. There also exists a counter (variable \( \text{counter} \) in the model) that records the number of times when \( \theta = \theta_M \). When \( \text{counter} = 9 \), the system sets the boolean function \( \text{beep} \) to true, and then it stops. Even if the system is simple enough to be designed as a monolith, we would rather design it modularly, to create the premises for further extensions, which may require the addition of other modules. We use the subsystems \( S_1 \) (the heating-cooling system) and \( S_2 \) (the counter) (Fig. 8).

![Figure 7. The system \( P \).](image)

![Figure 8. The systems \( S_1 \) and \( S_2 \).](image)

**Interleaved model.** The parallel composition of the CAS \( S_1 \) and \( S_2 \) gives a new CAS, \( S = S_1 \parallel S_2 \), with an implicit, unique variable \( \text{now} \). Following the interleaved execution model, at some moment, both first actions of \( S_1 \) and \( S_2 \) will be simultaneously enabled (when \( \theta = \theta_M \)). However, only one of them is selected by the controller. If the chosen action is the one of \( S_1 \), the action with the same guard in \( S_2 \) becomes disabled, since the temperature is decreased. Thus, the counter misses to record the respective event of \( \theta = \theta_M \), therefore presenting a wrong output.

**Synchronized model.** We now compose \( S_1 \) and \( S_2 \) by using our newly defined operator “\( \parallel \)" (the components are proper CAS). As a result, we get the CAS \( S_{\text{new}} = S_1 \parallel S_2 \). Then, we translate \( S_{\text{new}} \) into an ordinary AS, \( S_{\text{new}} \) (Fig. 9), with explicit time, by applying the definition given in Fig. 7.

If we repeat the scenario described above, when \( \theta = \theta_M \), the semantics of \( S_{\text{new}} \) does not let time progress unless all the global variables are updated. Therefore, both enabled actions are executed at the same moment of time, and, in
Parallel composition lets us reuse the already existing components. In [7], Back and von Wright established conditions that is the composition of the individual reactions of the integrated components. The conclusion, however, supports our point of view: interleaving “blurs” the distinction among the components. Bornot and Sifakis [9] analyze compositions of timed systems expressed as communicating processes. The authors strive for maximal progress, ensured in our case by the synchronized semantics.

By providing the new virtual execution environment, we have tackled two important problems of reactive system design: behavior control and modularity. The essential result of the study is mentioned by Corollary 1. Based on this, we can say that the system level integrator and the module designers gain an increased independency with respect to each other during the design process. We believe that our achievement of using maximal synchronization to increase the modular design capabilities of the AS framework is a contribution that could be easily adapted to other similar formal environments.

References