On Designing for Modularity

Cristina Cerschi Seceleanu
Tiberiu Seceleanu
email: ccerschi@abo.fi,
tiberiu.seceleanu@utu.fi

Turku Centre for Computer Science

TUCS Technical Reports

No 534, October 2003

On Designing for Modularity

Cristina Cerschi Seceleanu

Åbo Akademi University, Department of Computer Science and
Turku Centre for Computer Science (TUCS),
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland

Tiberiu Seceleanu

University of Turku, Dept. of Applied Physics,
Lab. of Electronics and Information Technology,
FIN-20014 Turku, Finland

email: ccerschi@abo.fi, tiberiu.seceleanu@utu.fi

Turku Centre for Computer Science
TUCS Technical Report No 534
October 2003

ISBN 952-12-1177-6
ISSN 1239-1891

Abstract

Action systems, a state-based formalism for modeling reactive, distributed systems, provides a simple, yet powerful conceptual model for concurrency. Interleaved execution of actions gives the basic semantics to parallel execution of action systems. Trace refinement of action systems carried out within the higher-order logic framework of refinement calculus lets us derive a correct implementation of a given specification. We revisit the execution concept of action systems and extend it with a synchronization mechanism, within the traditional model. We define a new parallel composition operator and show by means of illustrative examples that our new approach improves the modularity of a reactive system design. We also prove that trace refinement of composed action systems executing in a synchronized environment adds no extra complexity to the equivalent process, in the classical model. Also, when implementing our view in the generalized action systems model for hybrid systems, we bring in the possibility of creating more concise, still correct specifications of hybrid / real-time systems, by combining synchronization and interleaving.

Keywords: Action systems, Trace refinement, Modular design, Concurrency

1 Introduction

The global behavior of a reactive system results from the cooperation of its components. Hence, a reactive system deals with issues like communication, composability, concurrency and preemption. Out of these, concurrency is related to the fundamental aspects of systems with *multiple*, *simultaneously active* computing agents that interact with one another. The complexity of such systems comes as an inherent byproduct, which leads further to problems concerning the correctness of the steps performed in the development flow. On one hand, component-based design is one of the solutions towards partially reducing the task of the designer of complex systems, while on the other hand, the employment of formal methods in system design tries to solve the aspects related to correctness.

In this study, we concentrate on issues regarding the design of reactive systems, be they software or hardware-targeted systems. A feasible design methodology requires the designer to compose the system from parallel concurrent components called *modules*. Such modules are modeled here by *action systems*. We approach aspects of concurrency and modular 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 [4] is a state-based formalism, relying on an extended version of Dijkstra's language of *guarded commands* [20]. Therefore, it behaves according to Dijkstra's guarded iteration statement on the state variables. Recently, it has been extended to *continuous action systems* [5] that provides a unified framework for handling both discrete and continuous behavior.

In addition to composability properties, an abstract specification of a module needs to be refined into concrete, implementable models. The higher-order logic refinement calculus theory [8, 28] can be used for reasoning about properties of action systems, and also for proving that the correctness of the specifications of abstract modules is preserved by their implementations.

Our favorite formalism uses a built-in interleaving semantics for handling concurrency. Parallel behavior is modeled by interleaved actions that can be executed in any order. Modules communicate via shared variables. Though versatile and general, this way of modeling large systems can sometimes have a negative impact on the *composability* of modules that interact concurrently. When plugging modules together, we have to specify additional details regarding the order in which they exchange information. This problem results in compromising the data abstraction of a module's interface. Moreover, the interleaving usually goes together with nondeterminism. 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, as stated for instance by Montanari and Rossi [29]. We will also illustrate these symptoms by examples.

Our wish is to provide an additional concurrency mechanism for action systems, namely *synchronization*, as a way to describe controllable behavior. Therefore, we define a new parallel composition operator. We show that the concepts that we introduce continue to rely on the strong mathematical background of action systems, while providing the designer with additional means for system development. We also extend the new method to continuous action systems, since concurrency, in all its flavors, plays a crucial role in real-time / hybrid system design.

Fortunately, 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* [7] can be used to ensure that the implementation is correct with respect to a specification that captures the system's reaction to all sets of inputs.

The rest of the paper is organized as follows. Firstly, in section 2, we give a short review of some of the related studies, after which action systems are briefly introduced in section 3. The sequential execution of action systems, the interleaving model for parallel composition and a simple example are outlined in section 4. In section 5, we introduce our synchronized model, define the new parallel operator and show how composed action systems can be trace refined in the new context. Further, we extend the model of synchronized parallel action systems to continuous action systems, in section 6. Two concrete design situations, a discrete and a hybrid one, are presented in section 7. In section 8, we discuss our contribution and show some comparison to other similar approaches. All proofs of theorems and lemmas are outlined in the Appendix.

2 Related Work

Numerous studies analyzed aspects of control and / or composability within different formal frameworks. For instance, Petri nets [31] offers the possibility of designing reactive / distributed systems by truly concurrent models. Alternatively, the approximation of concurrency by interleaving is used in most process algebras like CSP [23], CCS [27], as well as in input-output automata [26] and UNITY [18]. In these latter cases, problems may arise due to the interpretation of concurrency, the common denominator being the interleaved model of computation. The non-deterministic behavior induced by the interleaved model requires solutions for controlling the data flow. However, as we will show in the forthcoming sections, resolving control aspects reduces the design independence across the different levels of the design process. Several recent studies analyzed aspects of control and / or composability within different formal frameworks, all of them dealing with a certain interleaved environment.

Thus, the study presented by Treharne and Schneider [35] employs CSP processes in order to control B-machines, Butler [16] analyzed also the impact of a similar combination with regard to composition related issues. Here synchronization aspects appear when discussing compositionability in the presence of interaction between CSP processes.

Modularity, as a first concern, is analyzed by Bellegarde et all in [12], where the synchronized parallel composition is defined for event B [2] structures. Certain conditions are required to achieve modularity in the interleaved, nondeterministic environment of event B. Synchronization may be specified only for a subset of the events. Except for the synchronization idea, this approach has several common characteristics with the one taken by Back and von Wright [10], for action systems. They also analyze modularity, in terms of the traditional parallel composition. Also within the framework of action systems, Butler [15] employed CSP in order to address similar composability issues.

Synchronization, generally, is present in several forms in software design. Mechanisms such as semaphores, locks, monitors etc., are part of software development history as synchronization methods. They are becoming even more popular, with the growth of Java community. *Barriers* are also a common method to reach thread synchronization [22].

Parallel composition of hybrid systems has also been studied extensively. Besides the temporal logic of actions of Lamport [25], where barrier synchronization is specified as a way of applying *non-interleaving* to system design, we choose to mention here the approach proposed by Bornot and Sifakis [14], where timed systems are expressed as communicating processes. Their composition semantics guarantees by construction, the properties of time reactivity (absence of timelocks), and activity preservation (absence of local deadlocks). The authors strive for maximal progress, which in our case is ensured by the synchronized semantics and the variable hiding mechanism.

In our view on reactive systems design we subscribe to the idea that, while *internally*, components may exhibit complex, nondeterministic behaviors, for an external observer, only the *external* reaction is relevant [30]. Therefore, in the rest of the paper we show how we decrease the external nondeterminism by providing a barrier synchronization mechanism for action systems. This will also help us to improve modularity characteristics of the design process in our framework.

3 Action Systems

Back and Kurki-Suonio [4] introduced the action systems formalism, providing a framework for specifying and refining concurrent programs. An *action system* (AS, henceforward) is in general a collection of *actions* or guarded commands, which are executed one at a time.

An action system is built according to the following syntax:

$$\mathcal{A}(z : T_z) \triangleq \textsf{begin var } x : T_x \bullet Init\,;\ \textsf{do } A_1 [\!] \ldots [\!] A_n \textsf{ od end} \tag{1}$$

Here, $\mathcal{A}$ contains the declaration of local variables x (of type T_x), followed by an *initialization* statement $Init$ and the *actions* $A_1, \ldots, A_n$. Variables z (of type T_z) are *global* to the action system. The initialization statement assigns starting values to the global or local variables. After that, *enabled* actions are repeatedly chosen and executed. In this paper, we regard an action A_i as being of the form $g_i \rightarrow S_i$. An action is *enabled*, thus the *action body* S_i is executed, when the *guard* g_i evaluates to true. The chosen actions change the values of the variables in a way that is determined by the action body. Two or more actions can be enabled at the same time, in which case one of them is chosen for execution, in a demonically nondeterministic way. The statements inside $\mathcal{A}$ are iterated as long as the disjunction of the guards ($\bigvee_{i=1}^n g_i$) holds. Termination is normal if the exit condition ($\neg \bigvee_{i=1}^n g_i$) holds.

As an action system is not usually regarded in isolation, but rather as a part of a more complex system, the rest of the system (the *environment*) communicates with the action system via *shared* (read and written) variables. In the following, we assume and extend the notations defined in [6]. Thus, 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 $vA = rA \cup wA$. We can also build the same sets at the system level, considering the local / global partition of the variables. Thus, for a given action system $\mathcal{A}$, we have the access set, $v\mathcal{A}$, split into the global read / write variables, denoted by $gr\mathcal{A}/gw\mathcal{A}$ and the local read / write variables, denoted by $lr\mathcal{A}/lw\mathcal{A}$. We say that an action A of $\mathcal{A}$ is *global*, if $gw\mathcal{A} \cap wA \neq \emptyset$ or *local*, if $wA \subseteq lw\mathcal{A}$.

Two actions A_i and A_j are said to be *in conflict* with each other if they have a conflict on the commonly accessed variables. There are two kinds of conflicts: *write-write* conflict, if $wA_i \cap wA_j \neq \emptyset$, or *read-write* conflict[1], if $rA_i \cap wA_j \neq \emptyset$ or $wA_i \cap rA_j \neq \emptyset$. Actions that are not in conflict with each other are called *disjoint*.

[1] Read-write is called a conflict because it affects the parallel composition of AS, which are required to have non-interacting actions.

A statement S_i is defined by the following grammar:

$$\begin{array}{lll} S_i & ::= & abort & (abortion,\ nontermination) \\ & & \mid skip & (stuttering,\ empty\ statement) \\ & & \mid x := e & ((multiple)\ assignment) \\ & & \mid S_m\,;\ldots;S_n & (sequential\ composition) \\ & & \mid g_m \rightarrow S_m \,\|\, \ldots \,\|\, g_n \rightarrow S_n & (nondeterministic\ choice) \\ & & \mid x := x'.Q & (nondeterministic\ assignment) \end{array}$$

where $S_m, \ldots, S_n$ are conjunctive statements, $g_m, \ldots, g_n$ and Q are predicates (boolean conditions), x a variable or a list of variables, and e 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 [8]. Therefore, the *while* loop is written as while g do S od $=$ do $g \rightarrow S$ od . Here, we do not consider nested loops.

Statements in the action systems language are defined using *weakest precondition semantics*, which is consistent with Dijkstra's original semantics for the language of guarded commands [20].

For statement S and postcondition Q, the formula $\mathrm{wp}(S, Q)$, 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. In this paper, we assume that all statements are *conjunctive monotonic predicate transformers*, that is, $\forall p, q \bullet \mathrm{wp}(S, (p \wedge q)) = \mathrm{wp}(S, p) \wedge \mathrm{wp}(S, q)$.

As an example, the wp of the nondeterministic choice of n conjunctive predicate transformers is given as:

$$\mathrm{wp}(g_1 \rightarrow S_1 \,\|\, \ldots \,\|\, g_n \rightarrow S_n, Q) \ \triangleq\ g_1 \Rightarrow \mathrm{wp}(S_1, Q) \wedge \ldots \wedge g_n \Rightarrow \mathrm{wp}(S_n, Q)$$

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

Data Refinement of Actions. An action A is *refined* [3] by the action C, written $A \leq C$, if, whenever A establishes a certain postcondition, so does C:

$$A \leq C \Leftrightarrow (\forall Q.wp(A, Q) \Rightarrow wp(C, Q))$$

Next, let $R(a, c, z)$ be an abstraction relation between the abstract local variables a, the concrete local variables c and the global variables z. Then, action A is data refined by action C using the abstraction relation R, denoted $A \leq_R C$, if

$$\forall Q.R \wedge wp(A, Q) \Rightarrow wp(C, \exists a.R \wedge Q)$$

Invariants. A predicate I is an *invariant* of an AS given by (1) if: it is established by $Init$, i.e. $true \Rightarrow \mathrm{wp}(Init, I)$, and it is preserved by each action A_i, i.e. $g_i \wedge I \Rightarrow \mathrm{wp}(S_i, I), i = 1, \ldots, n$. Naturally, the only variables allowed to appear in the invariant of a given action system are the variables in $v\mathcal{A}$.

Trace Refinement of Action Systems. The semantics of an AS is given in terms of behaviors [7]. A *behavior* of an AS is a sequence of states, $b = \langle (x_0, y_0), (x_1, y_1) \ldots \rangle$, where each state has two components. The first component is the *local state* and the second is the *global state*. Behaviors can be finite or infinite. A finite behavior is called *terminating* if it ends in a proper state, or *aborting* if it ends with $\perp$ (a symbol indicating that the execution ended improperly). A *trace* of a behavior is obtained by removing all finite stuttering (no change of the visible states), and the local state component in each state of a given system.

In a general, less formal manner, we say that an action system $\mathcal{C}$ refines $\mathcal{A}$, written as $\mathcal{A} \sqsubseteq \mathcal{C}$, if every trace of $\mathcal{C}$ contains a trace of $\mathcal{A}$.

In practice, we use the following lemma to prove *trace refinement* of action systems.

Lemma 1 *Given the action systems*

$$
\begin{aligned}
\mathcal{A}(z_{\mathcal{A}}) &\triangleq \textsf{begin var } a \bullet a, z_A := a_0, z_{A0}\,;\ \textsf{do } A \textsf{ od}\\
\mathcal{C}(z_{\mathcal{C}}) &\triangleq \textsf{begin var } c \bullet c, z_C := c_0, z_{C0}\,;\ \textsf{do } C \,[\!]\, X \textsf{ od} \ \textsf{ end},
\end{aligned}
$$

let $R(a, c, z_A, z_C)$ be an abstraction relation and $I(c, z_C)$ an invariant of the system $\mathcal{C}$. The abstract system $\mathcal{A}$ is (trace) refined by the concrete system $\mathcal{C}$, denoted $\mathcal{A} \sqsubseteq_R \mathcal{C}$, if:

1. *Initialization:* $R(a_0, c_0, z_{A0}, z_{C0}) \wedge I(c_0, z_{C0}) \equiv true$
2. *Main action:* $A \leq_R C$
3. *Auxiliary action:* $skip \leq_{R \wedge I} X$
4. *Continuation condition:* $R \wedge I \wedge gA \Rightarrow gC \vee gX$
5. *Internal convergence:* $R \wedge I \Rightarrow \mathrm{wp}(\textsf{ do } X \textsf{ od }, true)$

4 The Interleaving Model

Starting with the original paper by Back and Kurki-Suonio [4], the sequential execution model was established as a *de facto* reasoning environment for AS designs.

Parallel executions are modeled by interleaving actions that have no read / write conflicts.

Thus, the execution of an AS assumes that the system is observed by a virtual external entity - the **execution controller** (**controller** in short) - which, at any moment knows what actions, in which AS, are enabled. Non-deterministically, it selects one of them for execution.

Execution of Action Systems. The initialization places the systems in a stable, starting state. The controller then selects any of the enabled actions for execution, after which the system moves to a new state. We call this operation an *execution round*. Notice that an execution round is equivalent to the execution of an action. This action may update either local or global variables, from the point of view of the containing AS. After this, the controller evaluates the new state, observes the enabled actions and starts another execution round.

4.1 Parallel Composition of Action Systems

A more recent definition of the parallel composition of action systems can be found in [10, 11]. According to this, two action systems can be composed in parallel if they have the same global variables but disjoint local variables.

Let our protagonists be $\mathcal{A}$ and $\mathcal{B}$, two action systems of the form

$$\begin{aligned} \mathcal{A}(z) &\triangleq \textsf{begin var } x \bullet Init_A\,;\ \textsf{do } g_A \rightarrow S_A \textsf{ od end} \\ \mathcal{B}(z) &\triangleq \textsf{begin var } y \bullet Init_B\,;\ \textsf{do } g_B \rightarrow S_B \textsf{ od end} \end{aligned}$$

Then, the parallel composition of $\mathcal{A}$ and $\mathcal{B}$ is the system $\mathcal{P} = \mathcal{A} \,||\, \mathcal{B}$:

$$\mathcal{P}\,(z) \triangleq \textsf{begin var } x, y \bullet Init_A\,; Init_B\,;\ \textsf{do } g_A \rightarrow S_A \,\|\, g_B \rightarrow S_B \textsf{ od end}$$

The composed action system essentially combines the variables, the initialization statements and the actions of the two subsystems. The initialization of the common variables z must be consistent, that is, they are assigned same initial values by both initialization statements, $Init_A$ and $Init_B$. Given the above formalization, $\mathcal{A} \,||\, \mathcal{B}$ is executed by first initializing the local and global variables and then interleaving the execution of enabled actions of $\mathcal{A}$ and $\mathcal{B}$. Termination occurs when both action systems can terminate, meaning that no action is enabled in any of the systems.

The non-identical global variables lists $z_{\mathcal{A}}$ and $z_{\mathcal{B}}$ of two action systems $\mathcal{A}$ and $\mathcal{B}$ can be extended to $z_{\mathcal{A}} \cup z_{\mathcal{B}}$, each of the systems performing $skip$ operations with respect to the variables not originally in their corresponding list. In the following sections, we will slightly diverge, in letter, from the above definition of the parallel composition, in the sense that we will write parallel composition of action systems that do not have the same global variables, while in spirit, the definition of the parallel composition is always preserved. Thus, we will write $\mathcal{A} \| \mathcal{B}$ even though $z_{\mathcal{A}} \neq z_{\mathcal{B}}$.

4.2 Example

Let us illustrate the interleaved execution model through a simple, still relevant example. We consider the modeling of a digital *filter*. Notions regarding this kind of applications can be found, for instance, in [24].

Briefly, a filter is a device 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 name of 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 $y(n) = \sum_{k=0}^{N-1} h(k) \cdot x(n-k)$, 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. In the end, a filter may have either software or hardware implementation.

From the above short description of the filter we can identify two sub modules of such a device: the storage FIFO–like buffer and the actual implementation of the filter. In the following, we model the signal source by system $\mathcal{S}$, the buffer by system $\mathcal{B}$ while system $\mathcal{F}$ performs the filtering. This is illustrated in Fig. 1 a).

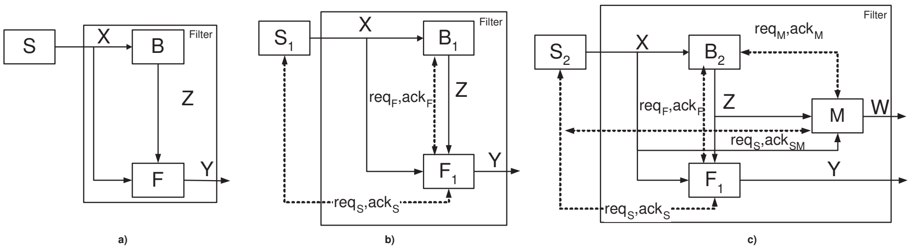

Figure 1: Simple filter representation.

The complete AS description is given as $\mathcal{P} \mathrel{\widehat{=}} \mathcal{S} \| \mathcal{B} \| \mathcal{F}$ and it is shown in Fig. 2.

Observe first that an interleaved execution of the above systems would not ensure that every signal emitted by $\mathcal{S}$ is correspondingly received by $\mathcal{B}$ and $\mathcal{F}$. Also, different values can be assigned to Y, depending on the order of selection for execution of $\mathcal{B}$ and $\mathcal{F}$, for the same sample provided by $\mathcal{S}$. Naturally, only one of these results should be a correct one. Both problems may be solved by specifying the order in which the sub-modules of $\mathcal{P}$ are required to be executed.

$$\begin{array}{ll}
& \mathcal{S}(X:T) \\
\triangleq & \mathsf{begin} \quad \bullet\, X := x_0; \\
& \quad \mathsf{do}\ X := X'.(X' \in T)\ \mathsf{od} \\
& \mathsf{end}
\end{array}$$

$$\begin{array}{ll}
& \mathcal{B}(X, Z[0..N-2]:T) \\
\triangleq & \mathsf{begin} \quad \bullet\, X, Z[0..N-2] := x_0, z_0; \\
& \quad \mathsf{do}\ Z[0], \ldots, Z[N-2] := X, Z[0], \ldots, Z[N-3]\ \mathsf{od} \\
& \mathsf{end}
\end{array}$$

$$\begin{array}{ll}
& \mathcal{F}(X, Z[0..N-2], Y:T) \\
\triangleq & \mathsf{begin\ var}\ h[0..N-1]:T \quad \bullet\, X, Z[0..N-2], h[0..N-1], Y := x_0, z_0, h_0, y_0; \\
& \quad \mathsf{do}\ Y := \sum_{k=1}^{N-1} h(k) \cdot Z(k-1) + h(0) \cdot X\ \mathsf{od} \\
& \mathsf{end}
\end{array}$$

$$\begin{array}{ll}
& \mathcal{P}(Y:T) \\
\triangleq & \mathsf{begin\ var}\ X, Z[0..N-2], h[0..N-1]:T \quad \bullet\, X, Z[0..N-2], h[0..N-1], Y := x_0, z_0, h_0, y_0; \\
& \quad \mathsf{do}\ X := X'.(X' \in T) \\
& \quad\quad [\!]\ Z[0], \ldots, Z[N-2] := X, Z[0], \ldots, Z[N-3] \\
& \quad\quad [\!]\ Y := \sum_{k=1}^{N-1} h(k) \cdot Z(k-1) + h(0) \cdot X \\
& \quad \mathsf{od} \\
& \mathsf{end}
\end{array}$$

Figure 2: Action systems filter representation.

Hence, systems should know about the status of the partners. This is realized by introducing (boolean) communication variables req_S, req_F and ack_S, ack_F, and devise a communication protocol such that the order is implemented. For instance, the systems may be remodeled as illustrated in Fig. 3.

$$\begin{array}{ll}
& \mathcal{S}_1(req_S, ack_S : Bool\,; X:T) \\
\triangleq & \mathsf{begin} \quad \bullet\, req_S, ack_S := false\,; X := x_0; \\
& \quad \mathsf{do}\ \neg req_S \wedge \neg ack_S \to X := X'.(X' \in T_X); \\
& \quad\quad\quad req_S := true \\
& \quad\quad [\!]\ req_S \wedge ack_S \to req_S := false \\
& \quad \mathsf{od} \\
& \quad \mathsf{end}
\end{array}$$

$$\begin{array}{ll}
& \mathcal{B}_1(req_F, ack_F : Bool\,; X, Z[0..N-2]:T) \\
\triangleq & \mathsf{begin} \quad \bullet\, req_F, ack_F : false\,; X := x_0\,; Z[0..N-2] := z_0; \\
& \quad \mathsf{do}\ req_F \wedge \neg ack_F \to \\
& \quad\quad\quad Z[0], \ldots, Z[N-2] := X, Z[0], \ldots, Z[N-3]; \\
& \quad\quad\quad ack_F := true \\
& \quad \mathsf{od} \\
& \quad \mathsf{end}
\end{array}$$

$$\begin{array}{ll}
& \mathcal{F}_1(req_S, req_F, ack_S, ack_F : Bool\,; X, Z[0..N-2], Y:T) \\
\triangleq & \mathsf{begin\ var}\ h[0..N-1]:T \quad \bullet\, req_S, req_F, ack_S, ack_F := false; \\
& \quad\quad X, Z[0..N-2], h[0..N-1], Y := x_0, z_0, h_0, y_0; \\
& \quad \mathsf{do}\ req_S \wedge \neg req_F \to Y := \sum_{k=1}^{N-1} h(k) \cdot Z(k-1) + h(0) \cdot X\,; req_F := true \\
& \quad\quad [\!]\ req_F \wedge ack_F \to req_F := false\,; ack_S := true \\
& \quad\quad [\!]\ \neg req_S \wedge ack_S \to ack_S := false \\
& \quad \mathsf{od} \\
& \quad \mathsf{end}
\end{array}$$

Figure 3: Communicating models.

The result of such an approach is illustrated by Fig. 1 b), where the communication variables are shown as dotted lines. Notice that the after emitting one signal sample, the system $\mathcal{S}_1$ is deactivated, after which $\mathcal{F}_1$ performs the filtering, and then informs $\mathcal{B}_1$, by setting $req_F := true$, that it can perform its operation. When this is accomplished, $\mathcal{B}_1$ signals to $\mathcal{F}_1$ ($ack_F := true$). In the end, $\mathcal{F}$ signals the end of the operation to $\mathcal{S}_1$ ($ack_S := true$). Another sample can now be presented by $\mathcal{S}_1$, and so on.

Consider further that, in the above example, X is an audio signal and $\mathcal{F}_1$ models a low-pass filter. The output of $\mathcal{F}_1$ would go to the woofer speaker of ones audio system. We would also like to have a high-pass filter, the output of which would go to the corresponding speakers of the same audio system. In order to make a fast design, we want to reuse the previously designed low-pass filter and then add a module that is able to detect the high frequencies of the incoming signal. The latter is modeled by the new system $\mathcal{M}$. Structurally, $\mathcal{F}_1$ and $\mathcal{M}$ are the same, the difference residing in the values of the coefficients, $h[0..N-1]$ and $h_M[0..N-1]$, correspondingly. Moreover, $\mathcal{M}$ still uses the same input signals, X, Z, and req_S (to which it answers with ack_{SM}). The new situation is illustrated in Fig. 1 c).

In order to accommodate the introduction of $\mathcal{M}$, the systems $\mathcal{B}$ has to wait for the two filters to read its data, once a new sample has been issued by $\mathcal{S}$. We need then to change $\mathcal{B}$ in order to accommodate the new design situation. The description of this new system is given below (notice that also $\mathcal{S}_1$ has to be changed into $\mathcal{S}_2$ in order to implement the communication with $\mathcal{M}$):

$$
\begin{array}{ll}
 & \mathcal{B}_2(req_F, ack_F, req_M, ack_M : Bool\,;\, X, Z[0..N-2] : T) \\
\triangleq & \mathsf{begin} \quad \bullet\; req_S, ack_S, req_F, ack_F :\!= false\,;\, X := x_0\,;\, Z[0..N-2] := z_0; \\
 & \quad \mathsf{do}\; req_F \wedge req_M \wedge \neg ack_F \wedge \neg ack_M \rightarrow \\
 & \qquad\quad Z[0], \ldots, Z[N-2] := X, Z[0], \ldots, Z[N-3]; \\
 & \qquad\qquad ack_F, ack_M := true \\
 & \qquad [\!]\; \neg req_F \wedge \neg req_M \wedge ack_F \wedge ack_M \rightarrow ack_F, ack_M := false \\
 & \quad \mathsf{od} \\
 & \quad \mathsf{end}
\end{array}
$$

Discussion. The interleaved model of execution brings the benefit of a very simple concept, which may indeed be useful at higher levels of abstraction, in the initial phases of a design flow. In order reduce the highly nondeterministic behavior of the model, inappropriate in certain situations, as shown in our example, one could introduce control channels, so that we ensure that data emitted by one source is not missed by any of the intended targets, or that data is processed in a correct manner.

However, one should be aware of the fact that there is another aspect of the problem, not solved by the exemplified introduction of the communication channels. An observer of the composed system $\mathcal{P}_2 \widehat{=} \mathcal{S}_2 \| \mathcal{B}_2 \| \mathcal{F}_1 \| \mathcal{M}$ (the listener, in the example) has access to both output sequences, $Y(n)$ and $W(n)$. Still, depending on the execution order of $\mathcal{F}$ and $\mathcal{M}$, 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, again, can not be a correct aspect of the design. A solution is provided, again, by the introduction of new communication channels, between $\mathcal{F}$ and $\mathcal{M}$, on one side, and the observer, on the other. What happens if multiple, different observers become necessary in the design?

Hence, any extension / reduction of the design elements require an internal change of the involved subsystems. This destroys clearly any hope for a modular design flow and the reuse of components in future projects. Also, in certain situations, we may have to ensure that $\mathcal{B} \sqsubseteq \mathcal{B}_1 \sqsubseteq \mathcal{B}_2$, which may not always be an easy task. On one hand, we may assign meanings like "data valid", "operation finished", etc., to the signals of the communication channels, therefore the suitable characteristics of interleaved approaches to asynchronous designs. On the other hand, unfortunately, these signals are global variables of the system. In hardware, generally, this translates into "more wires", while in software, this violates the principle of information hiding [13].

In the following section, we propose a solution to this kind of design situations.

5 Synchronized Parallel Environments

The problem of designing a complex system modularly amounts to separately model each of the component systems such that the process requires as little information as possible about the other components, or about the system as a whole. Moreover, as the systems grow in complexity, design reuse is one of the most important requirements in order to implement a fast and correct design process. As we have seen in the previous section, the interleaved model does not encourage a modular approach to the design flow. Still, for asynchronous environments, it works well and fully captures the behavioral characteristics [34].

In our view, designing for reactivity means that the system reacts to a given stimulus by presenting a composition of the individual responses of its subsystems. Moreover, the reaction must be repetitive, when the stimulus is such. As discussed in section 4.2, an interleaved environment does not implicitly offer this kind of behavior.

In the following, we try to adapt the techniques of AS to accommodate our perception and to allow a modular design perspective. We start by introducing some necessary concepts and we afterwards develop them to cover our goals.

Synchronized environment. Detailing the view described above, 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. Thus, the subsystems must synchronize when global variables of the compound system have to be updated. Intuitively, this is achieved by extending the execution round, as described in Section 4, to an *execution cycle*. A cycle is then defined by the activities carried out by the system between two global states. Thus, 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. Notice that between rounds, the global state of the system does not change. This should imply, as we show later, that we can reason about each subsystem separately, and, at the end, we observe all the contributions to the global state.

The first, favorable implication of such an approach is that the composed system reacts "at once" when presented with a stimulus, by updating the status of the global variables. Still, the *complete* reaction may not appear in a single execution cycle. Depending on the situation, this may not always be a good thing, but an interleaved approach will also require more than a single execution round in order to complete a system reaction. The difference compared to the previous interpretation of concurrency is obvious: while interleaving means that *only the first executed global action* reacts to the stimulus of the composed system, in our perspective, *all the global variables* are updated simultaneously, as a response to the same stimulus.

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 (at least) a global variable, the system is marked as "executed" and no other action can be selected from that system. However, the other participants do not yet see the changes. Another action is then selected, from an "unexecuted" AS. The process continues until all the components are marked "executed". This also signals the end of a cycle, when all the global variables are updated.

Proper Action Systems. Translating the above scenario into our framework requires certain characteristics for the action systems in the design. These requirements are introduced by the following definition, where we describe an action system representation that takes into account a clear separation between the local and global actions.

Definition 1 *Consider the action system* $\mathcal{A}$

$$\mathcal{A}(z : T_z) \;\triangleq\; \mathsf{begin\ var}\ x : T_x \bullet\ Init\,;\ \mathsf{do}\ g_S \rightarrow S \,\|\, g_L \rightarrow L\ \mathsf{od\ \ end} \quad (2)$$

We say that $\mathcal{A}$ *is a* **proper** *action system if:*

1. $gw\mathcal{A} \subseteq wS$ *– meaning that* S *is a global action of* $\mathcal{A}$
2. $wL \subseteq lw\mathcal{A}$ *– meaning that* L *is a local action of* $\mathcal{A}$
3. $\mathrm{wp}(\ \mathsf{do}\ g_L \rightarrow L\ \mathsf{od}, \neg g_L \wedge g_S) \equiv true$ *– meaning that the execution of* L*, taken separately, can not continue forever, and when it stops,* S *is enabled.*

Notice that the specification $\mathcal{A}$ encodes more visibly the mechanism that triggers the global state changes, together with the statement that performs it. Also observe that the set of write variables of the action S may contain local variables, too, hence the above first characteristic of a proper action system. We devise, in the following, a new parallel operator, which semantics ensures that a composition of proper action systems will remain a proper action system and it would also satisfy our goals.

Synchronized parallel composition. Let us consider a parallel composition of several systems $\mathcal{A}_k$. While each of the systems will first perform updates on their local variables, and only after that on the corresponding global variables, there is no way in which we can ensure the simultaneity of the latter, even more, these updates will still appear in a nondeterministic manner. Hence, we not only have to impose requirements on the participating systems, but we must also redefine their parallel composition.

Definition 2 *Let us consider n proper action systems:*

$$\mathcal{A}_k(z_k) \triangleq \mathsf{begin\ var}\ x_k \bullet Init_k\ ;\ \mathsf{do}\ g_S^k \rightarrow S_k \parallel g_L^k \rightarrow L_k\ \mathsf{od\ end},\ k = 1 \dots n$$

for which we also have that $\forall j, k = 1 \dots n, j \neq k.((gw\mathcal{A}_j \cap gw\mathcal{A}_k = \emptyset) \wedge (\bigcap_k x_k = \emptyset))$. The **synchronized parallel composition** *of the above systems is a new action system $\mathcal{P} = \mathcal{A}_1 \sharp \dots \sharp \mathcal{A}_n$, given by:*

$$
\begin{array}{l}
\mathcal{P}(z) \triangleq \mathsf{begin\ var}\ x\ ;\ sel[1..n] : \mathsf{Bool}\ ;\ run : \mathsf{Nat} \bullet Init; \\
\quad \mathsf{do} \\
\quad\quad run = 0 \wedge (\exists j \in [1..n] \bullet \neg sel[j]) \rightarrow sel[j] := true\ ;\ run := j \\
\quad \parallel run = 1 \rightarrow (g_L^1 \rightarrow L_1 \parallel \wedge \neg g_L^1 \wedge g_S^1 \rightarrow S_1'\ ;\ run := 0 \\
\quad\quad \parallel \neg g_L^1 \wedge \neg g_S^1 \rightarrow run := 0) \\
\quad \parallel \dots \\
\quad \parallel run = n \rightarrow (g_L^n \rightarrow L_n \parallel \wedge \neg g_L^n \wedge g_S^n \rightarrow S_n'\ ;\ run := 0 \\
\quad\quad \parallel \neg g_L^n \wedge \neg g_S^n \rightarrow run := 0) \\
\quad \parallel \quad sel \wedge run = 0 \rightarrow Update\ ;\ sel := false \\
\quad \mathsf{od} \\
\quad \mathsf{end}
\end{array}
$$

The operator '$\sharp$' ('sharp') is called the **synchronized parallel** *operator.*

The set of global variables of $\mathcal{P}$, z, is, initially, the union of the global variables set of each individual system: $z = \bigcup_k z_k$, where the reunion does not contain duplicates. It may be possible that communication between several submodules of $\mathcal{P}$ (the composing systems $\mathcal{A}_k$) 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 be mentioned 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 (*$gw\mathcal{A}_k c$*) of the original global write variables of each system (*$gw\mathcal{A}_k$*). Finally,* x *is completed by the addition of the array* sel *and the execution indicator,* run.

The Init *statement is a sequential composition of the individual* Init_k *statements to which we add the initialization of variables* $gw\mathcal{A}_k c$*,* sel *and* run*:*

$$\begin{array}{lcl} Init & \triangleq & Init_1 \,;\ldots;\, Init_n \,;\, gw\mathcal{A}_1 c, \ldots, gw\mathcal{A}_n c := gw\mathcal{A}_1, \ldots, gw\mathcal{A}_n; \\ & & sel := false \,;\, run := 0 \end{array}$$

The action Update *is given as:*

$$\begin{array}{rcl} \mathit{Update} & \triangleq & \mathit{Update}_1 \,;\ldots;\, \mathit{Update}_n \\ \mathit{Update}_k & = & gw\mathcal{A}_k := gw\mathcal{A}c_k \end{array}$$

□

The above definition of the '$\sharp$' operator says that, whenever there is a change in the input, such a composition of action systems reacts based on the state of all its subcomponents, and the result is composed of the reaction of each of the subsystems. Moving certain variables to the local level within the system $\mathcal{P}$ is motivated by the containment of local communication. In the definition, the statement $sel := false$ has the meaning $sel[1], \ldots, sel[n] := false$, and the predicate sel is $sel[1] \wedge \ldots \wedge sel[n]$.

Notice further that the above definition virtually enforces our view on the new execution environment. The utilization of the run variable actually splits the system $\mathcal{P}$ into *sections* coming each from an original action system $\mathcal{A}_k$. Actions belonging to a specific section are the only ones executed for the corresponding value of run. As soon as the section operations are finished, run is set to 0. The variable sel, on the other hand, "marks" the finished sections as "executed". The way in which sel and run are updated preserves both the nondeterministic selection of sections (and thus, the order in which the original proper action systems are selected for execution) and the internal nondeterminism that may be present inside of each section.

We may further be able to find useful properties for the system $\mathcal{P}$, expressed by the following theorem.

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

(a)	$\mathcal{A} \sharp \mathcal{B}$ *is a proper action system*	*(properness of* $\sharp$*)*
(b)	$\mathcal{A} \sharp \mathcal{B} = \mathcal{B} \sharp \mathcal{A}$	*(commutativity of* $\sharp$*)*
(c)	$\mathcal{A} \sharp (\mathcal{B} \sharp \mathcal{C}) = (\mathcal{A} \sharp \mathcal{B}) \sharp \mathcal{C},$ *with* $\mathcal{C}$ *also a proper AS*	*(associativity of* $\sharp$*)*

Proof. The proof is presented in Appendix A. □

Observe that the first statement of the above theorem shows that utilization of '$\sharp$' is scalable, therefore, one can approach hierarchical system development in a synchronized environment.

The Update Action. In a more general view, we do not want to impose the restriction that $\forall j, k = 1 \dots n, j \neq k.gw\mathcal{A}_j \cap gw\mathcal{A}_k = \emptyset$. Designs where write-write conflicts may appear are not necessarily examples of bad designs. A well known example of such situations is the bus-based design of digital systems, where multiple participants in the processing effort share a common resource, the bus lines. Of course, the situation requires a thorough analysis, and there are also multiple solutions that resolve the inevitable conflicts. Therefore, the action $Update$ may not be merely the action that writes the corresponding values into the variables of $gw\mathcal{P}$, but it can also be the action that "resolves" such conflicts. Intuitively, this means that the system $\mathcal{P}$ must also allow the designer to separately specify the action $Update$. This is subject to further studies.

5.1 Controlling Synchronized Parallel Compositions

We revisit briefly the example proposed in Section 4. Consider that instead of the parallel composition $\mathcal{P} = \mathcal{S} \parallel \mathcal{B} \parallel \mathcal{F}$, we write the description of our system as $\mathcal{P} = \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F}$. It is easy to check that the components $\mathcal{S}, \mathcal{B}, \mathcal{F}$ are proper AS. Therefore, we do not have to add communication channels to any of the subsystems, which remain as described in Fig. 2. Also, in case of a synchronized environment, the multiplicity of targets stops being an issue for the system $\mathcal{P}$. We can introduce as many $\mathcal{F}$-like systems as required, without modifying $\mathcal{B}$ in order to accommodate the presence of the new systems. 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 extra system $\mathcal{M}$ are selected for execution.

5.2 Trace Refinement in a Synchronized Parallel Environment

Crucial to a module-based design context is the possibility to separately analyze and, if necessary, improve the functionality of the subsystems. In our formal environment, performing such actions implies checking the result against a set of rules that would eventually ensure that the modification is done correctly. One of these rules is expressed by the trace refinement lemma (Section 3). Below, we reformulate the requirements of this lemma, so that it also covers proper action systems.

Lemma 2 *Given the proper action systems*

$$\begin{array}{lcl}\mathcal{A}(z_A) & \triangleq & \mathsf{begin\ var}\ a\ \bullet\ a, z_A := a_0, z_{A0}\ ;\\ & & \quad \mathsf{do}\ g_L^A \to L_A \| \neg g_L^A \wedge g_S^A \to S_A\ \mathsf{od}\\ & & \mathsf{end}\\ \mathcal{C}(z_{\mathcal{C}}) & \triangleq & \mathsf{begin\ var}\ c\ \bullet\ c, z_C := c_0, z_{C0}\ ;\\ & & \quad \mathsf{do}\ g_L^C \to L'_A \| \neg(g_L^C \vee g_X) \wedge g_S^C \to S'_A \| \, g_X \to X\ \mathsf{od}\\ & & \mathsf{end},\end{array}$$

let $R(a, c, z_A, z_C)$ be an abstraction relation and $I(c, z_C)$ an invariant of the system $\mathcal{C}$. The abstract system $\mathcal{A}$ is (trace) refined by the concrete system $\mathcal{C}$, denoted $\mathcal{A} \sqsubseteq_R \mathcal{C}$, if:

1. *Initialization:*

$$R(a_0, c_0, z_{A0}, z_{C0}) \wedge I(c_0, z_{C0}) \equiv true$$

2. *Main actions:*

$$(g_L^A \to L_A \leq_R g_L^C \to L'_A) \wedge (\neg g_L^A \wedge g_S^A \to S_A \leq_R \neg(g_L^C \vee g_X) \wedge g_S^C \to S'_A)$$

3. *Auxiliary action:*

$$skip \leq_{R \wedge I} g_X \to X$$

4. *Continuation condition:*

$$R \wedge I \wedge (g_L^A \vee g_S^A) \Rightarrow g_L^C \vee g_S^C \vee g_X$$

5. *Preserving properness:*

$$R \wedge I \Rightarrow \mathrm{wp}(\ \mathsf{do}\ g_X \to X \| \, g_L^C \to L'_A\ \mathsf{od}\ , \neg(g_X \vee g_L^C) \wedge g_S^C)$$

The first four requirements of the lemma are mere rewritings of the original one. The fifth, however, strengthens the original request by specifying that not only the auxiliary action $g_X \to X$, taken in isolation, must terminate, but also that the new group of local actions, $g_X \to X \| \, g_L^C \to L'_A$ must terminate and establish the necessary conditions for the (possibly) new global action $g_S^C \to S'_A$ to execute. Therefore, given Lemma 1, we only have to show that $\mathrm{wp}(\ \mathsf{do}\ g_X \to X \| \, g_L^C \to L'_A\ \mathsf{od}\ , \neg(g_X \vee g_L^C) \wedge g_S^C) \Rightarrow \mathrm{wp}(\ \mathsf{do}\ g_X \to X\ \mathsf{od}\ , true)$, to prove that Lemma 2 satisfies the conditions required by Lemma 1 for $\mathcal{A} \sqsubseteq_R \mathcal{C}$. The proof is described in Appendix B.

5.3 Modularity

Continuing the line established by the above lemma, we prove the following theorem.

Theorem 2 *Consider the synchronized environment $\mathcal{P} \triangleq \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_n$, where each of the component systems preserves the invariants $I_1, \ldots, I_n$, respectively. We then have that $I \triangleq I_1 \wedge \ldots \wedge I_n$ is an invariant of $\mathcal{P}$.*

Proof. See Appendix C.

The theorem states that in a synchronized environment, the designer may employ the modules without knowing their internal details of functionality. The module designer is responsible with upgrading (improving) the performance of the modules, and this is transparent for the integrator designer.

Corollary 1 *Consider the proper action systems $\mathcal{A}_k$, as in Definition 2, and some relation R. Then*

$$\frac{\mathcal{A}_j \sqsubseteq_R \mathcal{A}'_j}{\mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_j \sharp \ldots \sharp \mathcal{A}_n \sqsubseteq_R \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}'_j \sharp \ldots \sharp \mathcal{A}_n}, \forall j = 1, \ldots, n$$

Proof. The proof is a direct consequence of Lemma 2 and Theorem 2. □

The interpretation of the Corollary is that each component of a synchronized parallel composition may be refined in isolation, without knowledge about the invariants of the other components. This is a consequence of the fact that the systems exchange information at the end of an execution cycle rather than after each execution round.

5.4 Refinement Example

Let us see how the refinement procedure is applied to the design example offered in section 5.1. Consider that not only we want to separate the low-frequencies of the signal X through the filter $\mathcal{F}$, but we also want to amplify them. Therefore, we intend to change $\mathcal{F}$ into $\mathcal{F}_A$ (Fig. 4) under some invariant I, given as $I \triangleq (X = x(n)) \Rightarrow (Z[0] = x(n-1)) \wedge \ldots \wedge (Z[N-2] = x(n-N+1))$. In *isolation*, it is easy to check that the requirements of Lemma 1 are satisfied, hence we have $\mathcal{F} \sqsubseteq_I \mathcal{F}_A$. However, this is not our full intention. We should also check that, for instance, $\mathcal{B} \parallel \mathcal{F} \sqsubseteq_I \mathcal{B} \parallel \mathcal{F}_A$. Unfortunately, as $\mathcal{B}$ does not *respect* I [10], the refinement is not possible.

The solution comes, again, from employing communication channels as described in section 5.1. The invariant I has to be rewritten to take into account these channels, and thus the systems will gain some independence in this respect. Still, the same problems arise concerning the introduction of another filtering unit ($\mathcal{M}$), in which case not only the systems, but also the invariant must be reshaped. In addition, what we have to check is actually that $\mathcal{S} \parallel \mathcal{B} \parallel \mathcal{F} \sqsubseteq_I \mathcal{S} \parallel \mathcal{B} \parallel \mathcal{F}_A$, which will make things even more complicated.

$$
\begin{array}{l}
\mathcal{F}_A(X, Z[0..N-2], Y : T) \triangleq \mathsf{begin\ var}\ h'[0..N-1], a : T\, ; amp : \mathsf{Bool} \ \bullet \\
\quad X, Z[0..N-2], h'[0..N-1], Y, a, amp := x_0, z_0, h'_0, y_0, a_0, false; \\
\quad \mathsf{do}\ amp \rightarrow Y := \sum_{k=1}^{N-1} h'(k) \cdot Z_c(k-1) + h(0) \cdot X_c\, ; amp := false \\
\quad\quad [\!] \ \neg amp \rightarrow X_c := a \cdot X\, ; amp := true \\
\quad \mathsf{od} \\
\quad \mathsf{end}
\end{array}
$$

Figure 4: The new system $\mathcal{F}_A$.

However, if we check $\mathcal{F} \sqsubseteq_I \mathcal{F}_A$ in the context of Lemma 2, meaning that we intend a synchronized composition of the subsystems, we will at once have that $\mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F} \sqsubseteq_I \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F}_A$. Besides this, a previous addition of the module $\mathcal{M}$ would not change the refinement, and we could have $\mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F} \sharp \mathcal{M} \sqsubseteq_I \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F}_A \sharp \mathcal{M}$.

6 Synchronized Parallel Continuous Action Systems

Designing concurrent hybrid (systems that combine discrete and continuous behavior) / real-time systems (systems whose correct functioning depends also on various timing constraints) is one of the most challenging areas of software development. This is mostly because the process of combining synchronization and interleaving, which is essential to system specification, is in general harder for systems with required timing constraints. This justifies our extension of the synchronized model of parallel action systems to *continuous action systems*. In the following, the reader is presented with a succinct description of the generalization of action systems formalism to hybrid systems, and then with the synchronized hybrid approach.

6.1 Continuous Action Systems

Back et al. [5] defined a *continuous action system* (in short, CAS) as a finite set of actions that act upon a finite set of variables that can range over discrete or continuous valued time functions. The variables form the state of the system. A CAS has the form

$$
\begin{array}{rcl}
\mathcal{C}(z) & \triangleq & \mathsf{begin}\ \ \mathsf{var}\ x : \mathsf{Real}_+ \rightarrow T \ \bullet Init; \\
 & & \quad \mathsf{do}\ g_1 \rightarrow S_1 [\!] \ \ldots [\!] \ g_m \rightarrow S_m \ \ \mathsf{od}\ \ \mathsf{end}
\end{array}
$$

Here, x are the *local variables* of the system, $Init$ is the initialization statement, while $A_i = g_i \rightarrow S_i$, $i = 1, \ldots, m$ are the *actions* of the system. The global variables z are defined in the environment of the CAS. Real_+ stands for the non-negative reals, and is used as the time domain.

The execution of a CAS uses an implicit variable now, which shows the present time. The guards of the actions may refer the value of now, as may also expressions in the action bodies (but they can not change now). The initialization $Init$ assigns initial time functions to the variables x. The system will then start evolving according to these functions, with time (measured by now) moving forward continuously. Afterwards, 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 to the function e.

We explain the meaning of $\mathcal{C}$ by translating it into an ordinary action system. Its semantics is given by the following (discrete) action system $\bar{\mathcal{C}}$:

$$
\begin{array}{lll}
\bar{\mathcal{C}}(z) & \stackrel{\wedge}{=} & \mathsf{begin}\ \mathsf{var}\ now : \mathsf{Real}_+, x : \mathsf{Real}_+ \rightarrow T \ \bullet now := 0\,; Init\,; N; \\
& & \quad \mathsf{do}\ g_1.now \rightarrow S_1\,; N \,[\!]\ \ldots [\!]\ g_m.now \rightarrow S_m\,; N\ \ \mathsf{od} \\
& & \mathsf{end}
\end{array}
$$

Here, the attribute now is declared, initialized and updated explicitly. It models the moments of time that are of interest for the system: the starting time and the succeeding moments when some action is enabled. The value of now is updated by the statement N, defined as $N \stackrel{\wedge}{=} now := \mathsf{next}.gg.now$. Here, $gg = g_1 \vee ... \vee g_m$, and next is defined by

$$
\mathsf{next}.gg.t \stackrel{\wedge}{=} \begin{cases} min\{t' \geq t \mid gg.t'\}, & \text{if exists } t' \geq t \text{ such that } gg.t' \\ t, & \text{otherwise} \end{cases}
$$

Thus, 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 the variable now will denote the moment of time when the last discrete action was executed. In this case the system terminates with the last assigned values for the variables.

Parallel CAS - Interleaving Model. The parallel composition of two or more CAS is defined by using essentially the same method as for composing ordinary action systems [5]. Thus, it combines the variables and the actions of the component systems, and in consequence their continuous evolution, and the interleaved execution of enabled actions applies here as well. The difference from the discrete case is that the local / global variables of the composed system are discrete/continuous timed functions. Also, one needs to combine the component CAS before translating them into discrete action systems, to ensure that the composed system uses a unique now variable, which is checked by actions from all subsystems.

6.2 Synchronized Execution of Parallel CAS

The idea behind the synchronized parallel composition of CAS resembles very much the corresponding discrete case introduced in section 5. The differences arise from the interaction of the discrete controller with the continuous processes. Thus, the semantics of a proper CAS $\mathcal{A}(z)$ is given by the discrete translation:

$$\begin{array}{lcl}\bar{\mathcal{A}}(z) & \triangleq & \mathsf{begin\ \ var}\ now : \mathsf{Real}_+, x : \mathsf{Real}_+ \rightarrow T \ \bullet now := 0 \,;Init \,; N; \\ & & \quad \mathsf{do}\ g_L.now \rightarrow L [\!] \neg g_L.now \wedge g_S.now \rightarrow S \,; N \ \ \mathsf{od} \\ & & \mathsf{end}\end{array}$$

Further, let us consider n proper continuous AS as being ($k = 1 \ldots n$):

$$\begin{array}{lcl}\mathcal{A}_k(z_k) & \triangleq & \mathsf{begin\ \ var}\ \ x_k : \mathsf{Real}_+ \rightarrow T \ \bullet \ Init_k \,; \\ & & \qquad \mathsf{do}\ g_L^k \rightarrow L_k \ [\!] \ \neg g_L^k \wedge g_S^k \rightarrow S_k \ \ \mathsf{od\ \ end}\end{array}$$

Their **synchronized parallel composition** is a new CAS, $\mathcal{P} = \mathcal{A}_1 \sharp \ \ldots \ \sharp \mathcal{A}_n$. The semantics of $\mathcal{P}$ is given by the discrete AS $\bar{\mathcal{P}}$, illustrated in Fig. 5. The definitions of the local variables x, the global variables z, the initialization statement $Init$ and the $Update$ statement are similar to those of Definition 2. However, there are some specific differences. In Fig. 5, the local variables of $\bar{\mathcal{P}}$ are functions from the non-negative reals to some type T, and the variables $sel[1..n], run$ are also written as functions from time to type Bool, Nat, respectively. In addition, the variable now, which captures the global transition time for $\bar{\mathcal{P}}$, is explicitly declared, initialized, and advanced. Also, the guards of both local and global actions of the component systems are evaluated at time point now. The principle of hiding specific "not-to-disclose" communication variables of systems $\mathcal{A}_k$ applies here as well. In this way, we are able to prune internal "interleaving" variables from the list of global variables of $\bar{\mathcal{P}}$.

The controller nondeterministically selects for execution actions of any of the systems $\mathcal{A}_k$, which are enabled at now. The selected system is memorized by the variable run, and is marked as executed in the array sel. Further, if a local action has been executed, it will eventually be followed by a global action of the selected system. This is ensured by the fact that the component systems $\mathcal{A}_k, k = 1 \ldots n$, are proper. Notice that the time is not advanced before all CAS components have updated their global variables, indicated by $sel \wedge run = 0 \equiv true$. This means

$$
\begin{array}{rl}
\bar{\mathcal{P}}(z) \triangleq & \mathsf{begin}\ \ \mathsf{var}\ x : \mathsf{Real}_+ \to T\,;\, sel[1..n] : \mathsf{Real}_+ \to \mathsf{Bool}\,;\, run : \mathsf{Real}_+ \to \mathsf{Nat}\,;\, now : \mathsf{Real}_+ \ \bullet \\
& now := 0\,;\, Init\,;\, N; \\
& \ \mathsf{do} \\
& \quad run.now = 0 \wedge (\exists j \in 1..n \ \ \bullet \neg sel[j].now) \to sel[j] :- (\lambda t \cdot true)\,;\, run :- (\lambda t \cdot j) \\
& [\!]\ run.now = 1 \to (g_L^1.now \to L_1 [\!]\ (\neg g_L^1.now \wedge g_S^1.now \to S_1'\,;\, run :- (\lambda t \cdot 0)) \\
& \quad [\!]\ (\neg g_L^1.now \wedge \neg g_S^1.now \to run :- (\lambda t \cdot 0)) \\
& [\!]\ \ldots \\
& [\!]\ run.now = n \to (g_L^n.now \to L_n [\!]\ (\neg g_L^n.now \wedge g_S^n.now \to S_1'\,;\, run :- (\lambda t \cdot 0)) \\
& \quad [\!]\ (\neg g_L^n.now \wedge \neg g_S^n.now \to run :- (\lambda t \cdot 0)) \\
& [\!] \quad sel.now \wedge run.now = 0 \to Update\,;\, sel :- (\lambda t \cdot false); \\
& \qquad\qquad\qquad\qquad\qquad\qquad now := min\{t' \geq now \mid gg.t'\} \\
& \ \mathsf{od} \\
& \mathsf{end}
\end{array}
$$

Figure 5: The system $\bar{\mathcal{P}}$.

that, at this abstract level, we assume that the internal computation performed by each CAS component takes no time.

The way in which we enforce the update of the global state of the composed system $\bar{\mathcal{P}}$ guarantees that it indeed reacts to all sets of inputs, concurrently. In section 7.2 we illustrate a typical synchronized composition of CAS.

7 Concrete Design Situations

In this section, we analyze in more detail two examples, one with discrete characteristics and a second one within a continuous, real-time environment. The examples also present more common applications.

7.1 Discrete system analysis

We illustrate the benefits brought by the synchronized parallel composition to discrete system design in action systems, with the help of a more concrete example, inspired from [10].

Thus, consider that we want to model a keyboard - monitor system (illustrated in Fig. 6) where the *numbers* (integers) typed on the keyboard, modeled by the action system $\mathcal{A}$, are first transferred to a buffer, modeled by the vector $x[0 \ldots n-1] : \mathsf{Int}$. From here, the monitor, system $\mathcal{D}$, reads them into the variable $disp$ (modeling the display). The numbers are supposed to be different from the ones already in the buffer, and different than 0. Both $\mathcal{A}$ and $\mathcal{D}$ access x by means of local pointer variables, pos. The models are presented as follows:

$$
\begin{array}{ll}
& \mathcal{A}(x[0\ldots n-1] : \mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos, a : \mathsf{Int} \quad \bullet\ pos, x := 0; \\
& \quad \mathsf{do} \\
& \qquad a = 0 \rightarrow a := a'.(\exists\, z \notin x.a' = z \wedge z \neq 0) \\
& \quad [\!] \ x := x'.(x'[pos] = a \wedge \forall j := 0\ldots n, j \neq pos.x'[j] = x[j]); \\
& \qquad\qquad pos := (pos + 1) \mod n \; ; a := 0 \\
& \quad \mathsf{od} \\
& \mathsf{end}
\end{array}
\qquad (3)
$$

$$
\begin{array}{ll}
& \mathcal{D}(x[0\ldots n-1], disp : \mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos : \mathsf{Int} \quad \bullet\ x, pos, disp := 0; \\
& \quad \mathsf{do}\ disp := d'.(d' = x[pos]) \; ; pos := (pos + 1) \mod n\ \mathsf{od} \\
& \mathsf{end}
\end{array}
$$

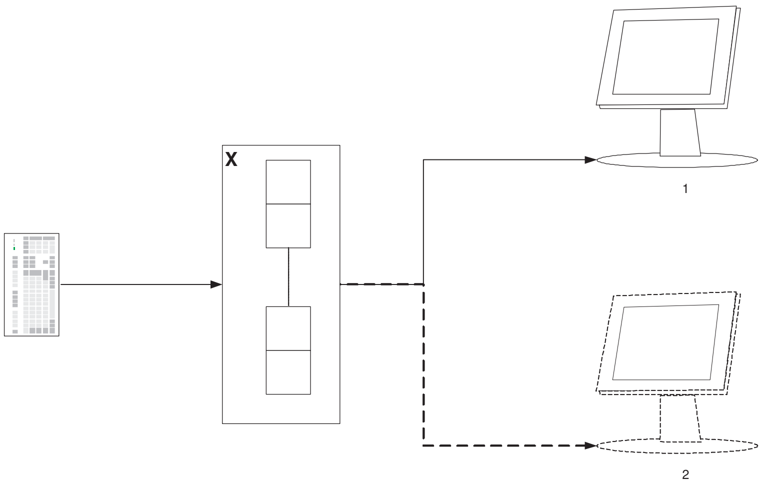

Figure 6: The keyboard - monitor system.

Interleaved execution analysis. We can describe the keyboard - monitor system as the parallel composition of $\mathcal{A}$ and $\mathcal{D}$: $\mathcal{A}\|\mathcal{D}$. Whatever is input by the keyboard, it will also appear on the display, at some moment. First thing to notice is that the way that systems are described does not ensure that the order writing - displaying is preserved. For a more explicit illustration of this effect, consider the following refinement of $\mathcal{D}$ which specifies that only numbers larger than the previous are displayed:

$$
\begin{array}{ll}
 & \mathcal{D}'(x[0\ldots n-1], disp : \mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos : \mathsf{Int} \quad \bullet\ x, pos, disp := 0; \\
 & \quad \mathsf{do}\ disp := d'.((x[pos] > disp \Rightarrow d' = x[pos]) \wedge \\
 & \qquad\qquad\qquad (x[pos] \leq disp \Rightarrow d' = disp)); \\
 & \qquad\qquad\quad pos = (pos + 1) \mod n \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

Observe now that if $\mathcal{D}'$ is not selected for execution at the appropriate moment (after each execution of $\mathcal{A}$), we may even reach the state when the monitor only displays the initial value of $disp$, 0, as if nothing has been typed on the keyboard.

This can be solved, as shown in section 4 by indicating in the models when the monitor is supposed to read the data stored in the buffer. It may either happen after each update performed by the keyboard, or just after the buffer has been filled. In both scenarios, we have to describe a communication channel between the two elements. We may re-describe the systems as:

$$
\begin{array}{ll}
 & \mathcal{A}(x[0\ldots n-1] : \mathsf{Int}, req_D, ack_D : \mathsf{Bool}) \\
\triangleq & \mathsf{begin\ var}\ pos, a : \mathsf{Int} \quad \bullet\ pos, x, a := 0; \\
 & \quad \mathsf{do}\ \neg req_D \wedge \neg ack_D \rightarrow a := a'.(\exists z \notin x. a' = z \wedge z \neq 0) \\
 & \qquad\qquad [\!]\ x[pos] := x'.(x'[pos] = a \\
 & \qquad\qquad\qquad \wedge \forall j := 0 \ldots n, j \neq pos. x'[j] = x[j]); \\
 & \qquad\qquad\quad pos = (pos + 1) \mod n\, ; req_D := true \\
 & \quad [\!]\ req_D \wedge ack_D \rightarrow req_D := false \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathcal{D}(x[0\ldots n-1], disp : \mathsf{Int}, req_D, ack_D : \mathsf{Bool}) \\
\triangleq & \mathsf{begin\ var}\ pos : \mathsf{Int} \quad \bullet\ x, pos, disp := 0; \\
 & \quad \mathsf{do}\ req_D \rightarrow disp := d'.(d' = x[pos])\, ; pos = (pos + 1) \mod n; \\
 & \qquad\quad ack_D := true \\
 & \quad [\!]\ \neg req_D \rightarrow ack_D := false \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

The introduction of another monitor ($\mathcal{D}_1$, as indicated by the dotted lines in Fig. 6) into the system will obviously create difficulties with respect to the communication channel between $\mathcal{A}$ and $\mathcal{D}$. Not only induces possible sources of error related to the updating sequence of the communication variables req_D (and the necessary req_{D_1}) and ack_D (and the necessary ack_{D_1}), but also destroys the modularity of design: if we were to fetch a keyboard module from a given library, we have to replace the one that communicates with a single monitor with one that can deal with the presence of two monitors, and so on.

Synchronized environment analysis. Let us discover now how the above situation can be analyzed in a synchronized environment, built by employing the synchronized parallel operator '$\sharp$' instead of '$\|$'. Notice that systems described by (3) are proper action systems.

The detailed description of $\mathcal{A} \sharp \mathcal{D}$ is given by the system $\mathcal{P}$ in Fig. 7.

$$
\begin{array}{ll}
 & \mathcal{P}(disp : \mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos_{\mathcal{A}}, pos_{\mathcal{D}}, a, x[0 \ldots n-1], disp_c, x_c[0 \ldots n-1] : \mathsf{Int}; \\
 & \quad sel[1..2] : \mathsf{Bool}, run : \mathsf{Nat} \ \bullet \\
 & \quad pos_{\mathcal{A}}, pos_{\mathcal{D}}, disp, x, run := 0 \,;\, x_c := x \,;\, disp_c := disp \,;\, sel := false; \\
 & \quad \mathsf{do} \\
 & \qquad run = 0 \wedge (\exists j \in [1..2] \ \bullet \neg sel[j]) \rightarrow sel[j] := true \,;\, run := j \\
 & \qquad [\!]\ run = 1 \rightarrow ((a = 0 \rightarrow a := a'.(\exists\, z \notin x.a' = z \wedge z \neq 0) \\
 & \qquad\qquad [\!]\ (a \neq 0 \rightarrow \\
 & \qquad\qquad\qquad x_c := x'.(x'[pos_{\mathcal{A}}] = a \wedge \\
 & \qquad\qquad\qquad\qquad \forall j = 1 \ldots n, j \neq pos_{\mathcal{A}}.x'[j] = x[j]); \\
 & \qquad\qquad\qquad pos_{\mathcal{A}} := (pos_{\mathcal{A}} + 1) \mod n \,;\, a := 0; \\
 & \qquad\qquad\qquad run := 0)) \\
 & \qquad [\!]\ run = 2 \rightarrow disp_c := d'.(d' = x[pos_{\mathcal{D}}]); \\
 & \qquad\qquad\qquad pos_{\mathcal{D}} := (pos_{\mathcal{D}} + 1) \mod n; \\
 & \qquad\qquad\qquad run := 0 \\
 & \qquad [\!]\ sel \wedge run = 0 \rightarrow x := x_c \,;\, disp := disp_c \,;\, sel := false \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

Figure 7: System $\mathcal{P}$.

An addition of an extra monitor to the system, transforms the composition into $\mathcal{A} \sharp \mathcal{D} \sharp \mathcal{D}_1$, with the representation given by the system $\mathcal{P}_1$ in Fig. 8. System $\mathcal{P}_1$ displays on two monitors the content of x, as typed in by $\mathcal{A}$, no elements being missed.

Suppose further that we transform $\mathcal{D} \sqsubseteq \mathcal{D}'$ and $\mathcal{D}_1 \sqsubseteq \mathcal{D}_1'$, so that $\mathcal{D}'$ displays only if the current element is greater than the previous one, and $\mathcal{D}_1'$ displays only if the current element is smaller than the previous one. The corresponding

$$
\begin{array}{ll}
 & \mathcal{P}_1(disp : \mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos_{\mathcal{A}}, pos_{\mathcal{D}}, a, x[0 \ldots n-1], disp_c, disp_{c1}, x_c[0 \ldots n-1] : \mathsf{Int}; \\
 & \quad sel[1..3] : \mathsf{Bool}, run : \mathsf{Nat} \quad \bullet \\
 & \quad pos_{\mathcal{A}}, pos_{\mathcal{D}}, disp, disp_1, x, run := 0 \,;\, x_c := x; \\
 & \quad disp_c, disp_{c1} := disp, disp_1 \,;\, sel := false; \\
 & \quad \mathsf{do} \\
 & \qquad run = 0 \wedge (\exists j \in [1..3] \quad \bullet \neg sel[j]) \rightarrow sel[j] := true \,;\, run := j \\
 & \qquad [\!] \ run = 1 \rightarrow ((a = 0 \rightarrow a := a'.(\exists\, z \notin x.a' = z \wedge z \neq 0) \\
 & \qquad\qquad [\!] \ (a \neq 0 \rightarrow \\
 & \qquad\qquad\qquad x_c := x'.(x'[pos_{\mathcal{A}}] = a \wedge \\
 & \qquad\qquad\qquad\qquad \forall j = 1 \ldots n, j \neq pos_{\mathcal{A}}.x'[j] = x[j]); \\
 & \qquad\qquad\qquad pos_{\mathcal{A}} := (pos_{\mathcal{A}} + 1) \mod n \,;\, a := 0; \\
 & \qquad\qquad\qquad run := 0)) \\
 & \qquad [\!] \ run = 2 \rightarrow disp_c := d'.(d' = x[pos_{\mathcal{D}}]); \\
 & \qquad\qquad\qquad pos_{\mathcal{D}} := (pos_{\mathcal{D}} + 1) \mod n; \\
 & \qquad\qquad\qquad run := 0 \\
 & \qquad [\!] \ run = 3 \rightarrow disp_{c1} := d'.(d' = x[pos_{\mathcal{D}_1}]); \\
 & \qquad\qquad\qquad pos_{\mathcal{D}_1} := (pos_{\mathcal{D}_1} + 1) \mod n; \\
 & \qquad\qquad\qquad run := 0 \\
 & \qquad [\!] \ sel \wedge run = 0 \rightarrow x := x_c \,;\, disp, disp_1 := disp_c, disp_{c1} \,;\, sel := false \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

Figure 8: System $\mathcal{P}_1$.

descriptions are given as:

$$
\begin{array}{ll}
 & \mathcal{D}'(x[0 \ldots n-1], disp : \mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos : \mathsf{Int} \quad \bullet\ x, pos, disp := 0; \\
 & \quad \mathsf{do} \\
 & \qquad disp := d'.((x[pos] > disp \Rightarrow d' = x[pos]); \\
 & \qquad\qquad\qquad \wedge (x[pos] \leq disp \Rightarrow d' = disp)) \\
 & \qquad\qquad pos := (pos + 1) \mod n \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

and

$$
\begin{array}{ll}
 & \mathcal{D}_1'(x[0 \ldots n-1], disp_1 : \mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos : \mathsf{Int} \quad \bullet\ x, pos, disp_1 := 0; \\
 & \quad \mathsf{do} \\
 & \qquad disp_1 := d'.((x[pos] \leq disp \Rightarrow d' = x[pos]) \\
 & \qquad\qquad\qquad \wedge (x[pos] > disp \Rightarrow d' = disp)); \\
 & \qquad\qquad pos := (pos + 1) \mod n \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

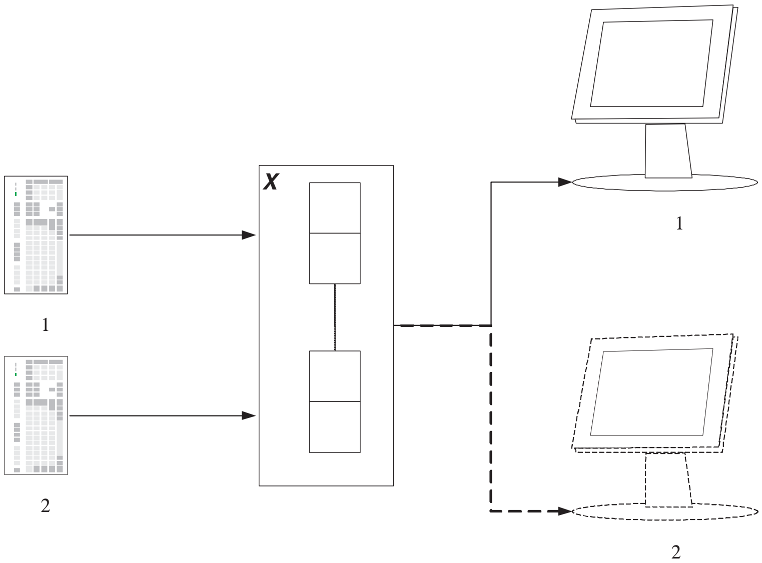

Figure 9: The keyboards - monitors system.

Observe that both the introduction of $\mathcal{D}_1$ and the above refinements do not affect the correctness of the synchronized environment operation. Even more, consider the following refinement of $\mathcal{A}$:

$$
\begin{array}{ll}
 & \mathcal{A}'(x[0\ldots n-1] : \mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos, a, less, great : \mathsf{Int} \quad \bullet\ pos, x, a, less, great := 0; \\
 & \quad \mathsf{do} \\
 & \qquad\quad a = 0 \rightarrow less := l'.(\exists\ z \notin x.z \leq x[pos-1] \wedge z \neq 0 \wedge l' = z); \\
 & \qquad\qquad great := g'.(\exists\ z \notin x.z > x[pos-1] \wedge z \neq 0 \wedge g' = z); \\
 & \qquad\qquad a := less \| \ a := great \\
 & \qquad \| \ a \neq 0 \rightarrow x := x'.(x'[pos] = a \wedge \forall j = 1\ldots n, j \neq pos.x'[j] = x[j]); \\
 & \qquad\qquad pos := (pos+1) \mod n; \\
 & \qquad\qquad a := 0 \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

The system $\mathcal{A}$’ (a proper action system, by inspection) can be further *decomposed* (operation which is not of interest in this study) into $\mathcal{A}_1 \| \mathcal{A}_2$, with the corresponding definitions:

$$
\begin{array}{ll}
 & \mathcal{A}_1(x[0\ldots n-1], a:\mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos, less : \mathsf{Int}\ \bullet pos, x, a, less := 0; \\
 & \quad \mathsf{do} \\
 & \quad\quad less = 0 \wedge a = 0 \rightarrow less := l'.(\exists\, z \notin x.z \leq x[pos-1] \wedge \\
 & \quad\quad\quad\quad z \neq 0 \wedge l' = z)\ ;\ a := less \\
 & \quad\quad [\!]\ less \neq 0 \wedge a \neq 0 \rightarrow x := x'.(x'[pos] = less \wedge \\
 & \quad\quad\quad\quad \forall j = 1\ldots n, j \neq pos.x'[j] = x[j]); \\
 & \quad\quad\quad\quad pos := (pos+1) \mod n; \\
 & \quad\quad\quad\quad less, a := 0 \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

and

$$
\begin{array}{ll}
 & \mathcal{A}_2(x[0\ldots n-1], a:\mathsf{Int}) \\
\triangleq & \mathsf{begin\ var}\ pos, great : \mathsf{Int}\ \bullet pos, x, a, great := 0; \\
 & \quad \mathsf{do} \\
 & \quad\quad great = 0 \wedge a = 0 \rightarrow great := g'.(\exists\, z \notin x.z > x[pos-1] \wedge \\
 & \quad\quad\quad\quad z \neq 0 \wedge g' = z)\ ;\ a := great \\
 & \quad\quad [\!]\ great \neq 0 \wedge a \neq 0 \rightarrow x := x'.(x'[pos] = great \wedge \\
 & \quad\quad\quad\quad \forall j = 1\ldots n, j \neq pos.x'[j] = x[j]); \\
 & \quad\quad\quad\quad pos := (pos+1) \mod n; \\
 & \quad\quad\quad\quad great, a := 0 \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

As a consequence, the keyboard - display system becomes $(\mathcal{A}_1 \| \mathcal{A}_2) \sharp \mathcal{D}' \sharp \mathcal{D}'_1$, where all the systems are proper action systems. The two system sourced from the initial $\mathcal{A}$ play the role of *two* keyboards (Fig. 9), one always writing a number smaller than the last one present in the buffer, the other one greater than the same number. Their execution is interleaved, as suggested by the parallel operator, '$\|$'. The composition above may be rewritten as $(\mathcal{A}_1 \sharp \mathcal{D}' \sharp \mathcal{D}'_1) \parallel (\mathcal{A}_2 \sharp \mathcal{D}' \sharp \mathcal{D}'_1)$.

Discussion. In the previous paragraphs we analyzed the modeling of a simple keyboard - monitor system, with the possibility of introducing additional monitors, if required. The traditional interleaved model, as expected, raised the problem of introducing communication channels between the components of the system, while in a synchronized environment this is not necessary. Separately refining the behavior of components is also possible when employing the synchronized parallel operator '$\sharp$'.

The description $(\mathcal{A}_1 \| \mathcal{A}_2) \sharp \mathcal{D}' \sharp \mathcal{D}'_1$ may be interpreted in another context, too. Suppose that $\mathcal{A}_1$ and $\mathcal{A}_2$ are two sources of signals, received together on the abstract channel x. The systems $\mathcal{D}'$ and $\mathcal{D}'_1$, in turn act as filters capable of reconstructing the initial signals from the transmittion channel. The modeling as such shows that the coexistence of both the traditional parallel operator and the synchronized one is possible and useful.

Observe that systems $\mathcal{A}_1$ and $\mathcal{A}_2$ write on the same variable x. This is allowed by the interleaving of the corresponding updates. It may seem that a *control monitor* is already in place and that it appropriately selects one of the systems to perform the required update. This is not possible in the synchronized approach, for which a monitor has to be constructed separately, and this is a subject for future studies. In addition, it may not always be correct to transform $(\mathcal{A}_1 \| \mathcal{A}_2) \sharp \mathcal{D}' \sharp \mathcal{D}'_1$ into $(\mathcal{A}_1 \sharp \mathcal{D}' \sharp \mathcal{D}'_1) \parallel (\mathcal{A}_2 \sharp \mathcal{D}' \sharp \mathcal{D}'_1)$, unless $\mathcal{A}_1$ and $\mathcal{A}_2$ are also proper action systems.

7.2 Hybrid system analysis

Having presented an example for the synchronized modular design of discrete AS, we now continue with exemplifying our view of the parallel composition of CAS. We consider an abstract model of a simple heating-cooling hybrid control system, which keeps the temperature inside a closed place between a minimum and a maximum value. The system is equipped with a controller that decreases the temperature when it has reached the maximum allowed value, at (constant) speed v_c, and starts heating the place when the temperature has decreased to the minimum value, at rate v_h. In our model, we denote the temperature with θ, and its maximum and minimum given values with θ_M, θ_m, respectively.

This system works in parallel with a counter (modeled by variable $counter$) that has to count the number of times when the temperature reaches the maximum value, and when that number is 10, it signals this by setting the boolean function $beep$ to true, after which the entire system stops.

We can design the hybrid system described above, modularly, by composing in parallel the subsystems $\mathcal{S}_1$ and $\mathcal{S}_2$, defined by the following CAS, respectively:

$$
\begin{array}{ll}
 & \mathcal{S}_1(\theta : \mathsf{Real}_+ \to \mathsf{Real}_+ \,;\, beep : \mathsf{Real}_+ \to \mathsf{Bool}) \\
\triangleq & \mathsf{begin} \\
 & \quad\quad beep :- (\lambda t \cdot false) \,;\, \theta :- (\lambda t \cdot v_h * (t - now)); \\
 & \quad \mathsf{do}\ \neg beep.now \wedge \theta.now = \theta_M \to \theta :- (\lambda t \cdot \theta_M - v_c * (t - now)) \\
 & \quad\quad [\!] \ \neg beep.now \wedge \theta.now = \theta_m \to \theta :- (\lambda t \cdot \theta_m + v_h * (t - now)) \\
 & \quad \mathsf{od} \\
 & \mathsf{end}
\end{array}
$$

$$
\begin{array}{ll}
& \mathcal{S}_2(\theta : \mathsf{Real}_+ \rightarrow \mathsf{Real}_+ \,; beep : \mathsf{Real}_+ \rightarrow \mathsf{Bool}) \\
\triangleq & \mathsf{begin\ var}\ counter : \mathsf{Real}_+ \rightarrow \mathsf{Nat}\ \bullet \\
& counter :- (\lambda t \cdot 0) \,; beep :- (\lambda t \cdot false) \,; \theta :- (\lambda t \cdot v_h * (t - now)); \\
& \quad \mathsf{do}\ \neg beep.now \wedge \theta.now = \theta_M \rightarrow counter :- (\lambda t \cdot counter + 1) \\
& \quad\quad [\!] \ \neg beep.now \wedge counter.now < 10 \rightarrow skip \\
& \quad\quad [\!] \ \neg beep.now \wedge counter.now = 10 \rightarrow beep :- (\lambda t \cdot true) \\
& \quad \mathsf{od} \\
& \mathsf{end}
\end{array}
$$

Interleaved model. When composing in parallel the CAS $\mathcal{S}_1$ and $\mathcal{S}_2$, we get a new CAS, $\mathcal{S} = \mathcal{S}_1 || \mathcal{S}_2$, having an implicit unique variable now, which denotes the present moment of time:

$$
\begin{array}{ll}
& \mathcal{S}(\theta : \mathsf{Real}_+ \rightarrow \mathsf{Real}_+ \,; beep : \mathsf{Real}_+ \rightarrow \mathsf{Bool}) \\
\triangleq & \mathsf{begin\ var}\ counter : \mathsf{Real}_+ \rightarrow \mathsf{Nat}\ \bullet \\
& counter :- (\lambda t \cdot 0) \,; beep :- (\lambda t \cdot false) \,; \theta :- (\lambda t \cdot v_h * (t - now)); \\
& \quad \mathsf{do}\ \neg beep.now \wedge \theta.now = \theta_M \rightarrow \theta :- (\lambda t \cdot \theta_M - v_c * (t - now)) \\
& \quad\quad [\!] \ \neg beep.now \wedge \theta.now = \theta_m \rightarrow \theta :- (\lambda t \cdot \theta_m + v_h * (t - now)) \\
& \quad\quad [\!] \ \neg beep.now \wedge \theta.now = \theta_M \rightarrow counter :- (\lambda t \cdot counter + 1) \\
& \quad\quad [\!] \ \neg beep.now \wedge counter.now < 10 \rightarrow skip \\
& \quad\quad [\!] \ \neg beep.now \wedge counter.now = 10 \rightarrow beep :- (\lambda t \cdot true) \\
& \quad \mathsf{od} \\
& \mathsf{end}
\end{array}
$$

The execution of $\mathcal{S}$ proceeds as follows. The local and global variables are initialized to time functions that characterize their behavior from time point $now = 0$ to the next time instance when at least one guard evaluates to $true$. Therefore, the variable $beep$ is initially set to $false$, and the temperature starts increasing from 0, as stipulated by the assigned function $(\lambda t \cdot v_h * (t - now))$. Suppose that at some point in time, the temperature reaches θ_M, that is, the actions 1 and 3 are enabled. According to the interleaved semantics of parallel CAS, the controller nondeterministically selects one action for execution, and after that, it checks for the next moment of time when at least one guard holds. If the chosen action is action 1, after its execution the variable θ will be assigned the function $(\lambda t \cdot \theta_M - v_c * (t - now))$, which decreases the temperature and in turn disables action 3. This means that the latter will not be executed at the same moment of time, thus the counter misses to record at least one time when the temperature reaches the maximum allowed value. Thus, the final result output by the counter will be altered.

Clearly, this situation can be solved by adding extra information to the component systems, regarding their communication, or by employing other operators of AS, which could make the component systems react at all stimuli received as input. However, either of the solutions implies extra coding effort, and moreover deteriorates the system modularity in case more heating-cooling systems and more counters have to be added. The highest inconvenience of the traditional parallel composition of CAS is that it does not scale up easily.

Synchronized model. Now, we compose $\mathcal{S}_1$ and $\mathcal{S}_2$ by using our newly defined operator '$\sharp$', given the fact that the components are proper CAS. As a result, we get the CAS $\mathcal{S}_{new} = \mathcal{S}_1 \sharp \mathcal{S}_2$. Then, we translate $\mathcal{S}_{new}$ into an ordinary AS, $\bar{\mathcal{S}}_{new}$, with explicit time, by applying the definition given by Fig. 5, as follows:

$$
\begin{array}{ll}
& \bar{\mathcal{S}}_{new}(\theta : \mathsf{Real}_+ \to \mathsf{Real}_+) \\
\triangleq & \mathsf{begin\ var}\ \theta_c : \mathsf{Real}_+ \to \mathsf{Real}_+ \,;\, beep, beep_c, sel[1..2] : \mathsf{Real}_+ \to \mathsf{Bool}; \\
& \quad counter, run : \mathsf{Real}_+ \to \mathsf{Nat} \,;\, now : \mathsf{Real}_+ \ \bullet \\
& \quad now := 0 \,;\, counter, run :- (\lambda t \cdot 0) \,;\, beep, beep_c, sel :- (\lambda t \cdot false); \\
& \quad \theta, \theta_c :- (\lambda t \cdot v_h * (t - now)) \,;\, now := min\{t' \geq now \mid gg.t'\}; \\
& \ \mathsf{do} \\
& \quad run.now = 0 \wedge (\exists j \in [1..2] \ \bullet \neg sel[j].now) \to sel[j] :- (\lambda t \cdot true); \\
& \qquad run :- (\lambda t \cdot j) \\
& \quad [\!]\ run.now = 1 \to \\
& \qquad (\neg beep_c.now \wedge \theta_c.now = \theta_M \to \theta_c :- (\lambda t \cdot \theta_M - v_c * (t - now)) \\
& \qquad [\!]\ \neg beep_c.now \wedge \theta_c.now = \theta_m \to \theta_c :- (\lambda t \cdot \theta_m + v_h * (t - now))); \\
& \qquad run :- (\lambda t \cdot 0) \\
& \quad [\!]\ run.now = 2 \to \\
& \qquad (\neg beep_c.now \wedge \theta_c.now = \theta_M \to counter :- (\lambda t \cdot counter + 1) \\
& \qquad [\!]\ \neg beep_c.now \wedge counter.now < 10 \to skip \\
& \qquad [\!]\ \neg beep_c.now \wedge counter.now = 10 \to beep_c :- (\lambda t \cdot true)); \\
& \qquad run :- (\lambda t \cdot 0) \\
& \quad [\!]\ sel.now \wedge run.now = 0 \to \theta :- \theta_c \,;\, beep :- beep_c; \\
& \qquad sel :- (\lambda t \cdot false) \,;\, now := min\{t' \geq now \mid gg.t'\} \\
& \ \mathsf{od} \\
& \mathsf{end}
\end{array}
$$

Notice that the variable *beep* is hidden from the interface of $\bar{\mathcal{S}}_{new}$.

Repeating the previous scenario, if the first action and the third action get enabled, the semantics of the synchronized parallel composition of CAS does not let time progress unless all global variables are updated. Therefore, the counter records all times when the temperature reaches θ_M, correctly. Thus, we can truly design the system modularly without adding communication information between subsystems, and moreover, in case we add similar modules to the composed system, the synchronized parallel composition lets us reuse already existing (library) components just by plugging them in, at the same time ensuring correct outputs to all sets of inputs.

This example emphasizes the practical application of the synchronized parallel composition of CAS.

8 Conclusions

This study was motivated by an analysis of control aspects and of modular design techniques, as supported by the current action systems formal framework. We illustrated that the interleaved model of concurrency may not suffice for modeling parallel reactive systems. Our solution comes as a synchronization mechanism, implying a new virtual execution model of AS. This is applicable to both discrete environments and hybrid system design, where complexity arises from the intricate nature of the discrete controller-continuous processes interaction, in addition to which timing constraints may be added to the system requirements. Within a synchronized environment, we eliminate intermediate results that could affect the global state. The system gives complete answers to the stimuli provided by the environment.

The result of a synchronized composition is not required when reasoning about the functionality of the composed system. The representation of $\mathcal{P}$ (considering $\mathcal{P} = \mathcal{A}_1 \sharp \ \ldots \ \sharp\mathcal{A}_n$) is only meant to offer the semantics of the '$\sharp$' operator. Therefore, the informal description of the execution rounds and cycles should be sufficient for understanding the new virtual execution environment.

Cavalcanti and Woodcock [17] and Charpentier [19] needed to build new reasoning environments in order to address issues related to correctness and composability of (reactive) systems. Both approaches have roots in the weakest precondition semantics of Dijkstra. These aspects are already included in our framework, and we have shown how to use them in order to achieve our goals. The main idea was to isolate the local updates performed by an action system from the global ones, which will appear to be updated at the same moment with the corresponding global variables of other systems. This is consistent with the views presented by Gupta et al. in [21], where they use concurrent constraint programming. Within a synchronized environment, we eliminated intermediate results that could affect the global state. The system gives a complete answer to a particular stimulus provided by the environment.

Perhaps the most important 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. It should be easy to show that one can obtain, from a synchronized environment either a traditional parallelism – whenever the synchronized subsystems actually operate in turns – or, at the extreme, a synchronous representation, where clock related aspects come into the picture. Hence, we can imagine parts of a larger system model operating in a synchronized environment, connected to parts executing in the traditional fashion, as exemplified in section 7.

The closest related work comes from researchers working in the same framework. In this respect, we can mention here the work of Back and von Wright [10], Butler [15] and Back and Xu [11]. In the first paper, the authors established conditions that enable the designer to perform individual refinements of the components in a parallel composition of action systems. Thus, achieving the intended results requires that the subsystems are carefully analyzed and their invariants fulfill certain *noninterference* conditions. As a consequence, this does not allow a module designer to independently modify its work, as he has to be informed about the structure of the other components in a higher level parallel composition. A more relaxed approach to this problem is given by Butler, in [15], where features of the state-based action systems and of the algebraic CSP are combined, in search for modularity.

One of the most recent studies analyzing composability issues is tailored in [12], to the event B systems [2]. The B-method in itself is a formalism very close to action systems. The present work extends even the set of common characteristics, as Bellegarde et al. introduced the idea of synchronous parallel composition for B event systems [12]. However, the similarities stop here. In opposition to our model, which increases the *external* determinacy while preserving the *internal* non-determinism, the event-B solution preserves also the external non-determinism. Moreover, the cited study requires the existence of a *gluing invariant* when synchronized modules are refined. This requirement comes from the fact that the synchronization is performed only with regard to selected events, collected in a synchronization specification. Therefore, the supplier of modules should deliver to the system integrator, besides the modules themselves, the synchronization specification, too. The requirement of the gluing invariant and of the synchronization specification is, hence, closely related to the noninterference invariant of [10]. The approach is also similar to the non-interleaved specifications of Lamport [25], where the designer may employ *joint actions*. The conclusion, however, supports our point of view, i.e., that it “blurs” the distinction between the components used in design.

The analysis presented in this study addresses the possibility of independently designing modules and their interactions, and the consistent correct control over their execution. For similar reasons, Butler [16] and Treharne and Schneider [35] employed CSP specifications in order to control the behavior of B state machines. The newly introduced synchronized parallel composition, while providing means for improved modularity of design, offers also, *within the same* framework, a simpler understanding of the behavior, as the utility of communication channels, in the sense that we used them here, becomes obsolete.

Another important aspect when analyzing interleaved models is related to *fair-*

ness issues. This was also implicitly present in our interleaved model examples. *Action fairness*, as used in [4], means that an enabled action will, sooner or later, be chosen for execution. More aspects on fairness in AS can be found in [11]. In a synchronized environment, fairness is not an issue of the system-level designer. It may be found again, as one descends in the system hierarchy.

The action systems formalism offers an environment at higher levels of abstraction. This means that our descriptions, down to a certain point, can be viewed as either software or hardware models. We already mentioned semaphores, mutexes, monitors and barriers as synchronization methods in software approaches. For hardware designers, synchronization bears a connection with virtual time values. Compared to the techniques of hardware design, where component based development has a long history, our approach is somehow similar with the ones employed within VHDL [1] programming environments. Hence, the updates of the local actions can be mapped on the variable updating statements, while the global updates correspond to signal updates. Moreover, our execution cycle may be viewed as a δ - simulation cycle in VHDL. However, VHDL allows only one process to be executed at a time, while all the '$\sharp$' connected action systems are executed at once - from the point of view of the updates performed on signals.

As illustrated in section 7.1, the existence of both the original parallel and the synchronized parallel operators is possible (with the condition that the considered action systems are proper) and, in certain situations, even required (systems updating same variables). This brings us the advantage of being able to reason in the same framework about interleaved and simultaneous events. Actually, notice that the synchronized parallel composition may be seen as a "lifting" of the simultaneous assignment defined in [33], from the action level, to the system level.

Further work is required by the specification of the *Update* action. If we intend to allow systems to write on the same variables, if so required by the design constraints / style, an appropriate solution has to be found in order to resolve the write - write conflicts. Another aspect is the realization of a simulation tool, which can "run" both approaches.

Acknowledgements. The authors would like to express their gratitude to Victor Bos and Viorel Preoteasa for their excellent comments and suggestions on this paper.

References

[1] P. Ashenden. The Designer's Guide to VHDL - Second Edition. Morgan Kaufmann Publishers, 2002.

[2] J.-R. Abrial. Extending B without Changing it (for developing distributed systems). In 1^{st}t Conference on the B method, pages 169190, Nantes, France, November 1996.

[3] R. J. R. Back. Refinement Calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings. 1989, vol. 430, LNCS, pp. 67-93. Springer-Verlag, 1990.

[4] R. J. R. Back and R. Kurki-Suonio. Distributed Cooperation with Action Systems. In ACM Transactions on Programming Languages and Systems, Vol. 10, No. 4.1988, pp. 513-554.

[5] R. J. Back, L. Petre, and I. Porres-Paltor. Continuous Action Systems as a Model for Hybrid Systems. Nordic Journal of Computing, vol. 8, 2001, pp. 2-21.

[6] R.-J. R. Back and K. Sere. From Action Systems to Modular Systems. Formal Methods Europe'94, Spain, October 1994, Lecture Notes in Computer Science. Springer–Verlag, 1994.

[7] R. J. R. Back and J. von Wright. Trace refinement of action systems. In Proceedings of CONCUR-94, Springer–Verlag, August 1994.

[8] R. J. R. Back and J. von Wright. Refinement Calculus: A Systematic Introduction. Springer–Verlag, 1998.

[9] R. J. R. Back, J. von Wright. Reasoning algebraically about loops. Acta Informatica 36, pp. 295334, Springer–Verlag, 1999.

[10] R. J. R. Back and J. von Wright. Compositional Action System Refinement. Technical Report Nr. 464, Turku Centre for Computer Science, 2002.

[11] R.J.R. Back and Q. W. Xu. Refinement of Fair Action Systems. Acta Informatica 35, 131-165, 1998.

[12] F.Bellegarde, J.Julliand, and O.Kouchnarenko. Synchronized Parallel Composition of Event Systems in B. D. Bert et al. (Eds.): ZB 2002, Springer-Verlag LNCS 2272, pp. 436457, 2002.

[13] R. Behrends and R. Stirewalt. The Universe Model: An Approach for Improving the Modularity and Reliability of Concurrent Programs. Proc. of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'2000).

[14] S. Bornot and J. Sifakis. On the composition of hybrid systems. In First International Workshop Hybrid Systems : Computation and Control (HSCC'98), Springer-Verlag, LNCS 1386, pp. 49-63, 1998.

[15] M. Butler. Stepwise Refinement of Communicating Systems. Science of Computer Programming 27(2):139-173, 1996.

[16] M. Butler. csp2B: A Practical Approach to Combining CSP and B. Formal Aspects of Computing (2000) 12: 182–198.

[17] A. Cavalcanti and J. Woodcock. A Predicate Transformer Semantics for a Concurrent Language of Refinement. Communicating Process Architectures 2002 J.Pascoe, P.Welch, R.Loader and V.Sunderam (Eds.) IOS Press, 2002, 147-165.

[18] K. M. Chandy and J. Misra. Parallel Program Design. A Foundation. Addison-Wesley, Reading, MA, 1988.

[19] M. Charpentier. An Approach to Composition Motivated by wp. In The 5^{th} International Conference on Fundamental Approaches to Software Engineering, 2002, LNCS 2306, pages 1-14.

[20] E. W. Dijkstra. A Discipline of Programming. Prentice-Hall International, 1976.

[21] V. Gupta, R. Jagadeesan, and V.A. Saraswat. Truly concurrent constraint programming. Theoretical Computer Science 278, 223-255, 2002.

[22] S.J. Hartley. Concurrent Programming Using Java. Stephen J. Hartley, 1998.

[23] C. A. R. Hoare. Communicating sequential processes. Prentice–Hall, New-York, 1984.

[24] E.C.Ifeachor and B.W.Jervis Digital Signal Processing Practical Approach. Addison Wesley Publishing Company, 1997.

[25] L. Lamport. Specifying Systems - The TLA+ Language and Tools for Hardware and Software Engineers. Addison Wesley Publishing Company, 2002.

[26] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. CWI Quarterly 2, pp. 219-246, 1989.

[27] R. Milner. Communication and Concurrency. Prentice–Hall, London, 1989.

[28] C. Morgan. Programming from Specifications. PrenticeHall International, 1998.

[29] U. Montanari and F. Rossi. Concurrency and Concurrent Constraint Programming. In A. Podelski ed., Constraint Programming: Basics and Trends, Springer-Verlag, LNCS 910, pp. 171-193, 1995.

[30] E-R. Olderog. Correctness of Concurrent Processes. In G.Goos, J. Hartmanis, Mathematical Foundations of Computer Science 1989.SpringerVerlag, LNCS 379, pp. 107-132.

[31] C. A. Petri. Communication with Automata. Tech. Rep. RADC-TR-65-377, Vol. 1, Suppl. 1, Applied Data Research, Princeton, NJ, 1966.

[32] J. Plosila. Self-Timed Circuit Design - The Action Systems Approach. Ph.D. Thesis, University of Turku, Dep. of Applied Physics, Lab. of Electronics and Information Technology, Turku, Finland, 1999.

[33] T. Seceleanu. Systematic Design of Synchronous Digital Circuits. Ph.D. Thesis, Abo Akademi, Turku, Finland, 2001.

[34] G. K. Theodoropoulos. Distributed Simulation of Asynchronous Hardware: The Program Driven Synchronization Protocol. Journal of Parallel and Distributed Computing, Vol. 62, Issue 4, April 2002, Pages 622-655.

[35] H.Treharne and S.Schneider. Using a Process Algebra to control B OPERATIONS. In K.Araki, A.Galloway, K.Taguchi (Eds.), Proceedings of the 1^{st} International Conference on Integrated Formal Methods, IFM 99. Springer 1999, pp. 437-456.

Appendix

Before starting to prove the statements of the paper, we present first some results that help us achieve our goals.
• We will make use of the Corollary 27 proved in [9]:

Corollary 2 *Assume that G and H are conjunctive predicate transformers and that $g \wedge h = false$. Then*

$$\mathsf{do}\ g \rightarrow G [\!] \ h \rightarrow H \ \mathsf{od} \ = \ \mathsf{do}\ h \rightarrow H \ \mathsf{od} \ ; \ \mathsf{do}\ g \rightarrow (G\ ; \ \mathsf{do}\ h \rightarrow H \ \mathsf{od}\ )\ \mathsf{od}$$

In the above, intuitively speaking, the condition $g \wedge h = false$ states that the statements G and H exclude each other; they cannot be enabled simultaneously [9].
• We adapt the special case of Theorem 31 proved in [9], to our context.

Theorem 3 *Assume that G and H are conjunctive predicate transformers. Assume further that:*

1. $\mathrm{wp}(H, \neg g) \equiv true$*, which means that statement H terminates and establishes (or preserves) $\neg g$.*

2. $g \wedge h \equiv false$*, which means that at most one of the statements is enabled at a time.*

Then

$$\mathsf{do}\ g \rightarrow G [\!] \ h \rightarrow H \ \mathsf{od} \ = \ \mathsf{do}\ g \rightarrow G \ \mathsf{od} \ ; \ \mathsf{do}\ h \rightarrow H \ \mathsf{od}$$

Remark. Observe that, due to the commutativity of the choice operator, the order of the sequence is arbitrary.
• We recall some of the weakest precondition rules [20], where A_1, A_2 are conjunctive statements:

1. wp rule for sequential composition:

$$\mathrm{wp}(A_1\ ; A_2, Q) \ \triangleq \ \mathrm{wp}(A_1, \mathrm{wp}(A_2, Q))$$

2. wp rule for choice:

$$\mathrm{wp}(A_1 [\!] \ A_2, Q) \ \triangleq \ \mathrm{wp}(A_1, Q) \wedge \mathrm{wp}(A_2, Q))$$

3. wp rule for assignment statement:

$$\mathrm{wp}(x := e, Q) \ \triangleq \ Q[e/x]$$

• We state here another helpful theorem, as follows:

Theorem 4 *Assume that G, H and W are conjunctive predicate transformers. Then*

$$(G \,[\!]\, H)\,;W = (G\,;W) \,[\!]\, (H\,;W)$$

Proof.

$$\begin{array}{ll}
 & \mathrm{wp}((G \,[\!]\, H)\,;W, Q) \\
= & \{ \text{ wp rule 1 } \} \\
 & \mathrm{wp}((G \,[\!]\, H), \mathrm{wp}(W, Q)) \\
= & \{ \text{ wp rule 2 } \} \\
 & \mathrm{wp}(G, \mathrm{wp}(W, Q)) \wedge \mathrm{wp}(H, \mathrm{wp}(W, Q)) \\
= & \{ \text{ wp rule 2 } \} \\
 & \mathrm{wp}((G\,;W) \,[\!]\, (H\,;W), Q) \\
\Box &
\end{array}$$

A. Proof of Theorem 1

(a) By Definition 2, the synchronized parallel composition of the proper action systems

$$\begin{array}{lcl}
\mathcal{A}_1(z_1) & \triangleq & \mathsf{begin\ \ var}\ \ x_1 \bullet\ Init_1\,;\ \mathsf{do}\ g_S^1 \rightarrow S_1 \,[\!]\, g_L^1 \rightarrow L_1\ \mathsf{od\ \ end} \\
\mathcal{A}_2(z_2) & \triangleq & \mathsf{begin\ \ var}\ \ x_2 \bullet\ Init_2\,;\ \mathsf{do}\ g_S^2 \rightarrow S_2 \,[\!]\, g_L^2 \rightarrow L_2\ \mathsf{od\ \ end}
\end{array}$$

is given by the system

$$\begin{array}{lcl}
\mathcal{P}(z) & \triangleq & \mathsf{begin\ \ var}\ x\,;sel[1..2] : \mathsf{Bool}\,; run : \mathsf{Nat} \bullet Init; \\
 & & \quad \mathsf{do} \\
 & & \qquad run = 0 \wedge (\exists j \in [1..2]\ \ \bullet \neg sel[j]) \rightarrow sel[j] := true\,; run := j \\
 & & \quad [\!]\ run = 1 \rightarrow (g_L^1 \rightarrow L_1 [\!]\ \neg g_L^1 \wedge g_S^1 \rightarrow S_1'\,; run := 0 \\
 & & \qquad [\!]\ \neg g_L^1 \wedge \neg g_S^1 \rightarrow run := 0) \\
 & & \quad [\!]\ run = 2 \rightarrow (g_L^2 \rightarrow L_2 [\!]\ \neg g_L^2 \wedge g_S^2 \rightarrow S_2'\,; run := 0 \\
 & & \qquad [\!]\ \neg g_L^2 \wedge \neg g_S^2 \rightarrow run := 0) \\
 & & \quad [\!]\quad sel \wedge run = 0 \rightarrow Update\,; sel := false \\
 & & \quad \mathsf{od} \\
 & & \mathsf{end}
\end{array}$$

We denote the actions of $\mathcal{P}$ as:

$$
\begin{array}{lcl}
Sel & \triangleq & run = 0 \wedge (\exists j \in [1..2] \;\; \bullet \neg sel[j]) \rightarrow sel[j] := true \,; run := j \\
A_1 & \triangleq & run = 1 \rightarrow (g_L^1 \rightarrow L_1 [\!] \neg g_L^1 \wedge g_S^1 \rightarrow S_1' \,; run := 0 \\
 & & \quad [\!] \neg g_L^1 \wedge \neg g_S^1 \rightarrow run := 0) \\
A_2 & \triangleq & run = 2 \rightarrow (g_L^2 \rightarrow L_2 [\!] \neg g_L^2 \wedge g_S^2 \rightarrow S_2' \,; run := 0 \\
 & & \quad [\!] \neg g_L^2 \wedge \neg g_S^2 \rightarrow run := 0) \\
U & \triangleq & sel \wedge run = 0 \rightarrow Update \,; sel := false
\end{array}
$$

In order to show that $\mathcal{P}$ is a proper AS, we have to show first that the local action terminates and that its execution enables the global action. We identify here the local and global actions of $\mathcal{P}$ as:

$$
\begin{array}{lclllcl}
L_{\mathcal{P}} & \triangleq & Sel [\!]\, A_1 [\!]\, A_2, & guardL_{\mathcal{P}} & \triangleq & \neg sel \vee run \neq 0 \\
S_{\mathcal{P}} & \triangleq & U, & guardS_{\mathcal{P}} & \triangleq & sel \wedge run = 0
\end{array}
\tag{4}
$$

Let us analyze in more detail the components of action A_1, for instance. They are given as follows:

$$
\begin{array}{lcllcl}
G & \triangleq & (run = 1 \wedge g_L^1 \rightarrow L_1), & guardG & \triangleq & (run = 1 \wedge g_L^1) \\
H_1 & \triangleq & (run = 1 \wedge \neg g_L^1 \wedge g_S^1 \rightarrow S_1' \,; run := 0), & guardH_1 & \triangleq & (run = 1 \wedge \neg g_L^1 \wedge g_S^1) \\
H_2 & \triangleq & (run = 1 \wedge \neg g_L^1 \wedge \neg g_S^1 \rightarrow run := 0), & guardH_1 & \triangleq & (run = 1 \wedge \neg g_L^1 \wedge \neg g_S^1)
\end{array}
$$

Further, observe that the above actions have the following properties:

1. $guardG \wedge guardH_1 \wedge guardH_2 = false$
2. G terminates, i.e. $\neg guardG$, since the system $\mathcal{A}_1$ is proper.
3. H_1 terminates, i.e. $\neg guardH_1$, since $\mathrm{wp}(H_1, run = 0) \equiv true$ and hence, also preserves $\neg guardG$.
4. H_2 terminates, i.e. $\neg guardH_2$, since $\mathrm{wp}(H_2, run = 0) \equiv true$ and hence, also preserves $\neg guardG \wedge \neg guardH_1$.

Thus, we can safely apply Theorem 3:

$$\mathsf{do}\ G [\!]\, H_1 [\!]\, H_2\ \mathsf{od} \ = \ \mathsf{do}\ G\ \mathsf{od}\ ;\ \mathsf{do}\ H_1\ \mathsf{od}\ ;\ \mathsf{do}\ H_2\ \mathsf{od}$$

In the above, each of the right hand side loops terminates (properties of G, H_1, H_2), thus, the loop $\mathsf{do}\ A_1\ \mathsf{od}$ terminates. Moreover, it also establishes $run = 0$.The same applies for the action A_2. We can then write the cumulative result as:

$$\mathrm{wp}(A_1, run = 0) \equiv true \wedge \mathrm{wp}(A_2, run = 0) \equiv true \tag{5}$$

At the next level, we consider the loop $\mathsf{do}\ Sel \,[\!]\, A_1 \,[\!]\, A_2\ \mathsf{od}$. We can obtain:

$$
\begin{array}{cl}
 & \mathsf{do}\ Sel \,[\!]\, A_1 \,[\!]\, A_2\ \mathsf{od} \\
= & \{\text{ Corollary 2 }\} \\
 & \mathsf{do}\ Sel\ \mathsf{od}\ ;\ \mathsf{do}\ ((A_1 \,[\!]\, A_2)\ ;\ \mathsf{do}\ Sel\ \mathsf{od}\ )\ \mathsf{od} \\
= & \{\text{ Theorem 4 }\} \\
 & \mathsf{do}\ Sel\ \mathsf{od}\ ;\ \mathsf{do}\ (A_1\ ;\ \mathsf{do}\ Sel\ \mathsf{od}\ ) \,[\!]\, (A_2\ ;\ \mathsf{do}\ Sel\ \mathsf{od}\ )\ \mathsf{od} \\
= & \{\text{ Theorem 3 }\} \\
 & \mathsf{do}\ Sel\ \mathsf{od}\ ;\ \mathsf{do}\ (A_1\ ;\ \mathsf{do}\ Sel\ \mathsf{od}\ )\ \mathsf{od}\ ;\ \mathsf{do}\ (A_2\ ;\ \mathsf{do}\ Sel\ \mathsf{od}\ )\ \mathsf{od}
\end{array}
$$

In the above, we considered that the first execution of Sel establishes the conditions for the execution of A_1, i.e., $run = 1$. Due to the remark on Theorem 3, if we consider that the first execution of Sel establishes $run = 2$ then we could have written the second and third elements of the sequence in reverse order. Observe however, that, independently of the order in the sequence, as $sel \notin wA_1 \cup wA_2$, the second time we execute Sel, this action disables itself by establishing sel:

$$
\begin{array}{rl}
 & sel[0] \wedge \neg sel[1] \wedge run = 0 \Rightarrow \mathrm{wp}(Sel, sel) \equiv true \\
\vee & \neg sel[0] \wedge sel[1] \wedge run = 0 \Rightarrow \mathrm{wp}(Sel, sel) \equiv true
\end{array}
\qquad (6)
$$

Hence, the result of the above derivations is the proof that the loop $\mathsf{do}\ Sel \,[\!]\, A_1 \,[\!]\, A_2\ \mathsf{od}$ terminates, since the equivalent sequence $\mathsf{do}\ Sel\ \mathsf{od}\ ;\ \mathsf{do}\ (A_1;\ \mathsf{do}\ Sel\ \mathsf{od}\ )\ \mathsf{od}\ ;\ \mathsf{do}\ (A_2\ ;\ \mathsf{do}\ Sel\ \mathsf{od}\ )\ \mathsf{od}$ terminates. Even more, we have that termination establishes $run = 0 \wedge sel$ (relations 5, 6). Therefore, using the notations given by (4), it follows:

$$
\mathrm{wp}(L_{\mathcal{P}}, \neg guardL_{\mathcal{P}} \wedge guardS_{\mathcal{P}}) \equiv true,
$$

which means that the local action terminates, and it enables the execution of the global action of the system $\mathcal{P}$.

It is easy to check that the other requirements of Definition 1 are also satisfied, thus, $\mathcal{P}$ is a proper AS. □
(b) Follows from the commutativity of the choice operator. □
(c) Follows from the associativity of the choice operator. □

B. Proof of Lemma 2

We consider the definition of the weakest precondition of a loop, as given by Dijkstra in [20]:

$$
\begin{array}{rcl}
\mathrm{wp}(\ \mathsf{do}\ g \rightarrow A\ \mathsf{od}\ , Q) & = & \exists k \geq 0 \bullet H_k, \\
H_0 & = & Q \wedge \neg g, \\
H_k & = & H_0 \vee wp(g \rightarrow A, H_{k-1})
\end{array}
\qquad (7)
$$

$$
\begin{array}{ll}
& \mathrm{wp}(\ \mathsf{do}\ g_x \to X \,\|\, g_L^C \to L'_A\ \mathsf{od}\ , \neg(g_X \vee g_L^C) \wedge g_S^C) \\
= & \{(7)\} \\
& \exists k \geq 0 \ \bullet H_k \\
= & \{(7)\} \\
& \exists k \geq 0 \ \bullet H_0 \vee \mathrm{wp}(g_x \to X \,\|\, g_L^C \to L'_A, H_{k-1}) \\
= & \{\text{ wp rule 2}\} \\
& \exists k \geq 0 \ \bullet H_0 \vee \mathrm{wp}(g_x \to X, H_{k-1}) \wedge \mathrm{wp}(g_L^C \to L'_A, H_{k-1}) \\
\Rightarrow & \{g_L^C \to L'_A \leq g_L^A \to L_A\} \\
& \exists k \geq 0 \ \bullet H_0 \vee \mathrm{wp}(g_x \to X, H_{k-1}) \wedge \mathrm{wp}(g_L^A \to L_A, H_{k-1}) \\
\Rightarrow & \{L_A \text{ terminates establishing the postcondition }\} \\
& \exists k \geq 0 \ \bullet H_0 \vee \mathrm{wp}(g_x \to X, H_{k-1}) \\
\Rightarrow & \{(7)\} \\
& \mathrm{wp}(\ \mathsf{do}\ g_x \to X\ \mathsf{od}\ , \neg(g_X \vee g_L^C) \wedge g_S^C) \\
\Rightarrow & \{\neg(g_X \vee g_L^C) \wedge g_S^C \Rightarrow true \text{ and monotonicity of the } \mathrm{wp}\} \\
& \mathrm{wp}(\ \mathsf{do}\ g_x \to X\ \mathsf{od}\ , true)\ \square
\end{array}
$$

C. Proof of Theorem 2

We consider the system $\mathcal{P}$, as follows:

$$
\begin{array}{lll}
\mathcal{P}(z) & \triangleq & \mathsf{begin\ \ var}\ x\,;sel[1..n] : \mathsf{Bool}\,; run : \mathsf{Nat} \bullet Init; \\
& & \mathsf{do} \\
& & \quad run = 0 \wedge (\exists j \in [1..n] \ \ \bullet \neg sel[j]) \to sel[j] := true\,; run := j \\
& & \| \ run = 1 \to (g_L^1 \to L_1 \| \ \wedge \neg g_L^1 \wedge g_S^1 \to S'_1\,; run := 0 \\
& & \quad \| \ \neg g_L^1 \wedge \neg g_S^1 \to run := 0) \\
& & \| \ \ldots \\
& & \| \ run = n \to (g_L^n \to L_n \| \ \wedge \neg g_L^n \wedge g_S^n \to S'_n\,; run := 0 \\
& & \quad \| \ \neg g_L^n \wedge \neg g_S^n \to run := 0) \\
& & \| \quad sel \wedge run = 0 \to Update\,; sel := false \\
& & \mathsf{od} \\
& & \mathsf{end}
\end{array}
$$

We denote the actions of $\mathcal{P}$ as:

$$
\begin{array}{lll}
Sel & \triangleq & run = 0 \wedge (\exists j \in [1..n] \ \ \bullet \neg sel[j]) \to sel[j] := true\,; run := j \\
A_1 & \triangleq & run = 1 \to (g_L^1 \to L_1 \| \neg g_L^1 \wedge g_S^1 \to S'_1\,; run := 0 \\
& & \quad \| \ \neg g_L^1 \wedge \neg g_S^1 \to run := 0) \\
& & \ldots \\
A_n & \triangleq & run = n \to (g_L^n \to L_n \| \neg g_L^n \wedge g_S^n \to S'_n\,; run := 0 \\
& & \quad \| \ \neg g_L^n \wedge \neg g_S^n \to run := 0) \\
U & \triangleq & sel \wedge run = 0 \to Update\,; sel := false
\end{array}
$$

We have stated that each action A_j respects an invariant I_j: $I_j \Rightarrow \mathrm{wp}(A_j, I_j),\ j = 1..n$. We have to prove that the invariant of $\mathcal{P}$ is $I = \bigwedge_{k=1}^{n} I_j$.

Since $Init = Init_1\,;\ldots;Init_n\,;gw\mathcal{A}c_1 :\!= gw\mathcal{A}_1\,;\ldots;gw\mathcal{A}c_n :\!= gw\mathcal{A}_n\,;sel :\!= false\,;run := 0$, the proof that $Init$ establishes the invariant I is straightforward. We are left to prove that I is preserved by every action of the system, that is:

$$I \Rightarrow \mathrm{wp}(Sel \parallel A_1 \parallel \ \ldots \parallel A_n \parallel U, I)$$

We have that:

$$\begin{array}{ll} & \mathrm{wp}(Sel \parallel A_1 \parallel \ \ldots \parallel A_n \parallel U, I) \\ \equiv & \{ \text{ wp rule 2}\} \\ & \mathrm{wp}(Sel, I) \wedge \mathrm{wp}(A_1, I) \ldots \mathrm{wp}(A_n, I) \wedge \mathrm{wp}(U, I) \end{array}$$

The action Sel updates only the local variables of the system $\mathcal{P}$, that is, sel and run. These do not appear in any of the individual invariants I_j. Therefore

$$\begin{array}{ll} & \mathrm{wp}(Sel, I) \wedge \mathrm{wp}(A_1, I) \ldots \mathrm{wp}(A_n, I) \wedge \mathrm{wp}(U, I) \\ \equiv & \mathrm{wp}(A_1, I) \ldots \mathrm{wp}(A_n, I) \wedge \mathrm{wp}(U, I) \end{array}$$

Hence, we are left to prove that

$$I \Rightarrow \mathrm{wp}(A_1, I) \ldots \mathrm{wp}(A_n, I) \wedge \mathrm{wp}(U, I)$$

By construction, A_j modifies only variables mentioned in I_j, therefore it preserves $I_1 \wedge \ldots I_{j-1} \wedge I_{j+1} \wedge \ldots I_n$. Also by construction, $I_j \Rightarrow \mathrm{wp}(A_j, I_j[gwAc_j/gwA_j])$, which means that in the invariant I_j the variables gwA_j are substituted by their copies, $gwAc_j$. It follows that we have to prove:

$$\begin{array}{ll} & I \Rightarrow I \wedge \mathrm{wp}(A_1, I_1[gwAc_1/gwA_1]) \wedge \ldots \wedge \mathrm{wp}(A_n, I_n[gwAc_n/gwA_n]) \wedge \mathrm{wp}(U, I) \\ \equiv & \{\text{logic}\} \\ & I \Rightarrow \mathrm{wp}(A_1, I_1[gwAc_1/gwA_1]) \wedge \ldots \wedge \mathrm{wp}(A_n, I_n[gwAc_n/gwA_n]) \wedge \mathrm{wp}(U, I) \\ \equiv & \{\text{hypothesis, } I_j \Rightarrow \mathrm{wp}(A_j, I_j),\ j = 1..n, A_j \text{ preserves } I\} \\ & I \Rightarrow \mathrm{wp}(U, I) \\ \equiv & \{\text{definition of } U\} \\ & I \Rightarrow \mathrm{wp}(gwA_1, \ldots, gwA_n :\!= gwAc_1, \ldots, gwAc_n\,;sel :\!= false, I) \\ \equiv & \{ \text{ successive applications of wp rule 1 and 3 } \} \\ & I \Rightarrow I[gwAc_1, \ldots, gwAc_n/gwA_1, \ldots, gwA_n] \\ & true \ \square \end{array}$$

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi

University of Turku
- **Department of Information Technology**
- **Department of Mathematics**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**