Modular Design of Reactive Systems

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

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

Abstract

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

1. Introduction

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

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

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

2. Action Systems

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

$$\mathcal{A}(z : T_z) \overset{\wedge}{=} \mathsf{begin\ var}\ x : T_x \bullet Init;\ \mathsf{do}\ A_1 [\!] .. [\!] A_n\ \mathsf{od\ end} \qquad (1)$$

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

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

A statement S_i is defined by the following grammar:

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

Above, $S_m, \ldots, S_n$ are statements, $g_m, \ldots, g_n$ and Q are predicates (boolean conditions), x a variable or a list of variables, and 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 [5]. Therefore, the *while* loop is written as while g do S od $=$ do $g \rightarrow S$ od . In this paper, we do not consider nested loops.

Statements in AS are defined by the *weakest precondition semantics*, consistent with Dijkstra's original semantics for the language of guarded commands [10]. 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. Here, we assume that all statements are *conjunctive predicate transformers* (functions from predicates to predicates), that is, $\forall p, q \;\bullet \mathrm{wp}(S, (p \wedge q)) = \mathrm{wp}(S, p) \wedge \mathrm{wp}(S, q)$.

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

When at least one action is enabled in a given AS $\mathcal{A}$, we say that $\mathcal{A}$ is enabled. We obtain information on the enabledness of a system $\mathcal{A}$, given by (1), by evaluating the predicate $gg_A,\ gg_A \mathrel{\widehat{=}} \bigvee_{k=1}^{n} g_k$.

3. The Traditional Model of Parallel Execution

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

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

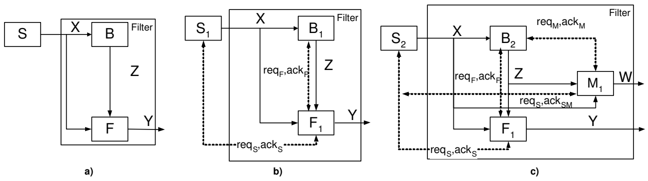

Figure 1. Simple filter representation.

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

$\mathcal{S}(X : T)$
$\triangleq$ begin $\bullet\ X := x_0;$
do $X := X'.(X' \in T)$ od
end

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

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

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

Figure 2. Filter model.

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

Consider that, in the above example, X is an audio signal and $\mathcal{F}_1$ models a low-pass filter. The output of $\mathcal{F}_1$ goes to the woofer speaker of one's audio system. We would also like to have a high-pass filter, the output of which goes to the corresponding speakers of the same audio system. We want to reuse the previously designed modules and then add one that detects the high frequencies of the input signal. The new module is modeled by the system $\mathcal{M}_1$ - Fig. 4. In or-

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

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

Figure 3. Communicating models.

der to accommodate the introduction of $\mathcal{M}_1$, $\mathcal{B}_1$ has to wait for the two filters to read its data, before updating Z. Hence, we have to change the representation of $\mathcal{B}_1$ (Fig. 4).

$$
\begin{array}{l}
\quad \mathcal{M}_1(req_S, req_M, ack_{SM}, ack_M : Bool\,;\, X, Z[0..N-2], W : T) \\
\triangleq \text{begin var } h[0..N-1] : T \quad \bullet\; req_S, req_M, ack_{SM}, ack_M := false; \\
\qquad X, Z[0..N-2], h[0..N-1], W := x_0, z_0, h_0, w_0; \\
\quad \text{do } req_S \wedge \neg(req_M \vee ack_M) \rightarrow W := \sum_{k=1}^{N-1} h[k] \times Z[k-1] + h[0] \times X; \\
\qquad req_M := true \\
\quad [\!]\; req_M \wedge ack_M \rightarrow req_M := false\,;\, ack_{SM} := true \\
\quad [\!]\; \neg req_S \wedge ack_{SM} \rightarrow ack_{SM} := false \\
\quad \text{od} \;\; \text{end}
\end{array}
$$

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

Figure 4. The systems $\mathcal{M}_1$ and $\mathcal{B}_2$.

Discussion. In order to reduce the implicit nondeterministic behavior of the interleaving model of execution, one may introduce control channels, for ensuring that data emitted by one source is not missed by any of the intended targets, or that data is processed in a correct manner. Still, an observer of the composed system $\mathcal{P}_2 \triangleq \mathcal{B}_2 \| \mathcal{F}_1 \| \mathcal{M}_1$ (the listener, in the example) has access to both output sequences, $Y(n)$ and $W(n)$. Depending on the execution order of $\mathcal{F}_1$ and $\mathcal{M}_1$, until the listener observes the new output $(Y(n+1), W(n+1))$, it will also observe the intermediate state, either $(Y(n), W(n+1))$ or $(Y(n+1), W(n))$, which is also an incorrect aspect of the design. A solution is provided, again, by the introduction of new communication channels, between $\mathcal{F}_1$ and $\mathcal{M}_1$, on one side, and the observer, on the other. What happens if multiple, different observers become necessary in the design?

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

4. Synchronized Parallel Environments

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

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

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

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

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. *$\mathrm{wp}(\text{ do } g_L \rightarrow L \text{ od}, \neg g_L \wedge g_S) \equiv true$ – meaning that the execution of L, taken separately, terminates, leaving S enabled.*

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

$$\mathcal{A}_k(z_k) \triangleq \text{ begin var } x_k \bullet\ Init_k\,;\ \text{do } g_L^k \rightarrow L_k \;[\!]\; g_S^k \rightarrow S_k \text{ od end},$$

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

$$
\begin{array}{l}
\quad \mathcal{P}(z) \\
\triangleq \text{begin var } x : T_x\,;\, sel[1..n] : \text{Bool}\,;\, run : \text{Nat} \bullet Init; \\
\quad \text{do } gg_P \rightarrow (run = 0 \wedge \neg sel[1] \rightarrow sel[1] := true\,;\, run := 1 \\
\qquad\qquad [\!]\; \ldots \\
\qquad\qquad [\!]\; run = 0 \wedge \neg sel[n] \rightarrow sel[n] := true\,;\, run := n \\
\qquad [\!]\; run = 1 \wedge g_L^1 \rightarrow L_1 [\!]\; run = 1 \wedge \neg g_L^1 \wedge g_S^1 \rightarrow wS_1c := wS_1\,;\, S_1'\,;\, run := 0 \\
\qquad\qquad [\!]\; run = 1 \wedge \neg gg_{A_1} \rightarrow wS_1c := wS_1\,;\, run := 0 \\
\qquad [\!]\; \ldots \\
\qquad [\!]\; run = n \wedge g_L^n \rightarrow L_n [\!]\; run = n \wedge \neg g_L^n \wedge g_S^n \rightarrow wS_nc := wS_n\,;\, S_n'\,;\, run := 0 \\
\qquad\qquad [\!]\; run = n \wedge \neg gg_{A_n} \rightarrow wS_nc := wS_n\,;\, run := 0) \\
\qquad [\!]\; sel \wedge run = 0 \rightarrow Update\,;\, sel := false \\
\quad \text{od} \;\; \text{end}
\end{array}
$$

The operator '$\sharp$' ('sharp') is called the **synchronized parallel** *operator. The set z of global variables of $\mathcal{P}$ is, initially, the union of the global variable sets of all individual systems, $z = \bigcup_k z_k$, without duplicates. It may be possible* *that communication between several submodules of $\mathcal{P}$ (the composing systems $\mathcal{A}_k$) 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 appear in z.*

Further, the local variables x of the new action system $\mathcal{P}$ are the union of the local variables x_k, to which we add the hidden variables. We also add copies (wS_kc) of the original write variables of each action body S_k. They replace the original variables wS_k, therefore we have $S'_k = S_k[wS_kc/wS_k]$. Finally, the list x is completed by adding the array sel and the execution indicator, run. We have that $gg_P \widehat{=} \bigvee_1^n (g_L^k \vee g_S^k)$.

The Init *statement is a sequential composition of the individual* Init_k *statements to which we add the initialization of variables wS_kc, sel and run, and the action $Update$ is a sequence of simple assignments:*

$$
\begin{aligned}
Init &\triangleq Init_1\,;..;Init_n\,; wS_1c,..,wS_nc := wS_1,\ldots,wS_n;\\
&\quad run := 0\,; sel := false\\
Update &\triangleq wS_1 := wS_1c\,;..;wS_n := wS_nc\,; sel := false
\end{aligned}
$$

The definition of the '$\sharp$' operator says that, whenever there is a change in the input, such a composition of AS reacts based on the state of all its subcomponents, and the result is a collection of individual reactions. The system composition reacts only if at least one subcomponent is enabled ($\exists k \in [1..n] \bullet g_L^k \vee g_S^k \equiv true$). The variable *run* identifies the system that is selected for execution. The variable *sel* stores the information on which are the executing, or already executed systems. Whenever all its elements ($sel \equiv sel[1] \wedge .. \wedge sel[n]$) become $true$ and $run = 0$, we have reached the end of an execution cycle.

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

(a)	$\mathcal{A}_1 \sharp \mathcal{A}_2$ *is a proper action system*	*(properness of $\sharp$)*
(b)	$\mathcal{A}_1 \sharp \mathcal{A}_2 = \mathcal{A}_2 \sharp \mathcal{A}_1$	*(commutativity of $\sharp$)*

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

5. Design Process

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

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

Definition 3 *A predicate I is a* **proper invariant** *of a proper AS $\mathcal{A}$, if:* $\forall z \notin wS \cdot g_S \Rightarrow (I[w'S/wS, z] \equiv I[w'S/wS])$.

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

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

$$\forall Q \bullet R \wedge I \wedge \text{wp}(A, Q) \Rightarrow \text{wp}(C, \exists a \bullet R \wedge I \wedge Q),$$

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

Lemma 1 *Given the proper action systems*

$$
\begin{aligned}
\mathcal{A}(z_\mathcal{A}) &\triangleq \textsf{begin var } a \bullet a, z_A := a_0, z_{A0}\,;\\
&\qquad \textsf{do } g_L^A \rightarrow L_A \,[\!]\, g_S^A \rightarrow S_A \textsf{ od} \;\; \textsf{end}\\
\mathcal{C}(z_\mathcal{C}) &\triangleq \textsf{begin var } c \bullet c, z_C := c_0, z_{C0}\,;\\
&\qquad \textsf{do } g_L^C \rightarrow L'_A \,[\!]\, g_S^C \rightarrow S'_A \,[\!]\, g_X \rightarrow X \textsf{ od} \;\; \textsf{end},
\end{aligned}
$$

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

1. *Initialization:* $R(a_0, c_0, z_{A0}, z_{C0}) \wedge I(c_0, z_{C0}) \equiv true$
2. *Main actions:* $(g_L^A \rightarrow L_A \leq_{R,I} g_L^C \rightarrow L'_A) \wedge (g_S^A \rightarrow S_A \leq_{R,I} g_S^C \rightarrow S'_A)$
3. *Auxiliary action:* $skip \leq_{R,I} g_X \rightarrow X$
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. *Properness:* $R \wedge I \Rightarrow \text{wp}(\textsf{ do } g_X \rightarrow X \,[\!]\, g_L^C \rightarrow L'_A \textsf{ od}, \neg(g_X \vee g_L^C) \wedge g_S^C)$

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

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

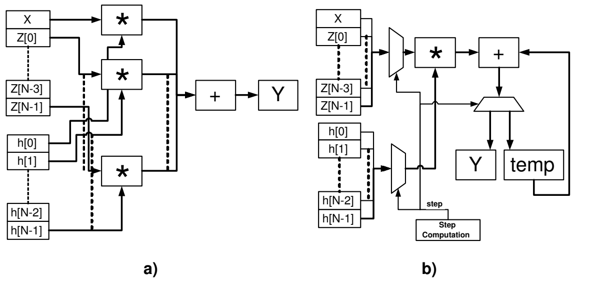

Figure 5. Implementation of the filter.

In *isolation*, one may prove that the system $\mathcal{F}_S$ is a refinement of $\mathcal{F}$, under the invariant I (R is the identity relation):

$$\mathcal{F} \sqsubseteq_I \mathcal{F}_S, \;\; I \widehat{=} step \in [2..N] \Rightarrow temp = \sum_{k=1}^{step-1} h[k] \times Z[k-1]$$

From a system level point of view, we should check that $\mathcal{B} \parallel \mathcal{F} \sqsubseteq_I \mathcal{B} \parallel \mathcal{F}_S$. Unfortunately, as $\mathcal{B}$ does not *respect* I, the refinement is not possible (see [7] for details). This fact has a simple explanation; since the controller may choose an enabled action from either $\mathcal{B}$ or $\mathcal{F}_S$, let us suppose that it chooses only actions from $\mathcal{F}_S$, until $step = N$, after which it selects $\mathcal{B}$ for execution. Hence, following the update of Z, the invariant I is no longer valid ($\mathcal{B}$ *interferes* with I).

$$\begin{array}{ll} & \mathcal{F}_S(X, Z[0..N-2], Y : T) \\ \triangleq & \text{begin var } h[0..N-1], temp : T\,;\, step : 0..N \quad \bullet \\ & \quad X, Z, h, Y := x_0, z_0, h_0, y_0\,;\, temp := 0\,;\, step := 0; \\ & \quad \text{do } step = 0 \rightarrow temp := 0\,;\, step := step + 1 \\ & \quad\quad [\!]\; step \in [1..N-1] \rightarrow temp := temp + h[step] \times Z[step-1]\,;\, step := step + 1 \\ & \quad\quad [\!]\; step = N \rightarrow Y := temp + X \times h[0]\,;\, step := 0 \\ & \quad \text{od end} \end{array}$$

Figure 6. The new system $\mathcal{F}_S$.

Theorem 2 *Consider the system $\mathcal{P} \triangleq \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_n$, where each of the component systems preserves the proper invariants $I_1, \ldots, I_n$, respectively. We then have that $I \triangleq I_1 \wedge \ldots \wedge I_n \wedge \bigwedge_{k \in [1..n]} (sel[k] \wedge \neg(run = k) \Rightarrow I'_k)$, $I'_k \triangleq I_k[wS_k c/wS_k]$ is a proper invariant of $\mathcal{P}$.*

The theorem states that in a synchronized environment, the global properties of the system are obtained from the individual properties of the components, as $I \Rightarrow I_1 \wedge .. \wedge I_n$. The additional terms of I help us make the connection between the copies of the write variables and the respective original variables, at the moment when the action *Update* is executed.

Corollary 1 *Consider the proper action systems $\mathcal{A}_k$ as in Definition 2, and an abstraction relation R_j. The systems $\mathcal{A}_k$ preserve the proper invariants I_k, respectively. Then*

$$\frac{\mathcal{A}_k \sqsubseteq_{R_k, I_k} \mathcal{A}'_k}{\mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_k \sharp \ldots \sharp \mathcal{A}_n \sqsubseteq_{R_k, I_k} \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}'_k \sharp \ldots \sharp \mathcal{A}_n}, \forall j \in [1 \ldots n]$$

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

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

Refinement Example. If we check $\mathcal{F} \sqsubseteq_I \mathcal{F}_S$ in the context of Lemma 1, (we consider a synchronized perspective on the system composition), we will immediately obtain that $\mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F} \sqsubseteq_I \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F}_S$ (notice that $\mathcal{F}_S$ is a proper AS). Besides this, a previous addition of module $\mathcal{M}$ would not change the refinement, and we could have $\mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F} \sharp \mathcal{M} \sqsubseteq_I \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F}_S \sharp \mathcal{M}$.

6. Continuous Action Systems

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

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

Here, Real_+ stands for the non-negative reals, and models the time domain.

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

$$\begin{array}{l} \bar{\mathcal{C}}(z) \triangleq \text{begin var } x : \mathsf{Real}_+ \rightarrow T \;\bullet\; now := 0; Init; N; \\ \quad\quad \text{do } g_1.now \rightarrow S_1; N [\!] \ldots [\!]\, g_m.now \rightarrow S_m; N \text{ od end} \\ N \triangleq now := \mathsf{next}.gg_C.now, \\ \mathsf{next}.gg_C.t \triangleq \begin{cases} min\{t' \geq t \mid gg_C.t'\}, & \text{if } \exists\, t' \geq t \bullet\; gg_C.t', \\ t, & \text{otherwise} \end{cases} \end{array}$$

In $\bar{\mathcal{C}}$, the variable now is declared, initialized and updated explicitly. It models the starting time and the succeeding moments when some action is enabled. The value of a variable v or of an expression e at a given moment of time t is identified by $v.t$ or $e.t$, respectively. Their values at the current moment are consequently given by $v.now$ and $e.now$. The value of now is updated by the statement N. The function next gives a moment of time when at least one action is enabled. If no action will ever be enabled, then the second branch of the definition will be followed, and now will denote the moment of time when the last discrete action was executed, the system terminating with the last assigned values for the variables.

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

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

$$\begin{array}{l}\bar{\mathcal{A}}(z) \triangleq \mathsf{begin\ var}\ now : \mathsf{Real}_+, x : \mathsf{Real}_+ \to T \bullet now := 0; Init; N; \\ \qquad\quad \mathsf{do}\ g_L.now \to L \,\|\, g_S.now \to S\ ; N\ \ \mathsf{od}\ \ \mathsf{end}\end{array}$$

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

$$\begin{array}{lll}\mathcal{A}_k(z_k) & \triangleq & \mathsf{begin\ var}\ x_k : \mathsf{Real}_+ \to T \bullet Init_k; \\ & & \qquad \mathsf{do}\ g_L^k \to L_k \,\|\, g_S^k \to S_k\ \ \mathsf{od}\ \ \mathsf{end}\end{array}$$

Their **synchronized parallel composition** is a new CAS, $\mathcal{P} = \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_n$. Its semantics is given following the lines of Definition 2, as illustrated in Fig. 7. The variables of $\bar{\mathcal{P}}$ are functions from Real_+ to some type T, and the variables $sel[1..n]$ and run are also written as functions from time to types Bool and Nat, respectively. The action guards of the component systems are evaluated at time point now. The time is not advanced before all CAS components have updated their global variables, indicated by $sel \wedge run = 0 \equiv true$. Also observe that, if no component system is supposed to react to a specific input situation, the composition is disabled ($gg_P \equiv false$). The theoretical results obtained for discrete synchronized AS apply to synchronized CAS, also, due to the discrete AS representation of the latter.

Example - Hybrid system analysis. Let us consider an abstract model of a simple heating-cooling hybrid control system, which keeps the temperature inside a place where some thermic processes happen, between a minimum and a maximum value. The system is equipped with a controller that either increases the temperature (modeled by θ) until it reaches the maximum allowed value (θ_M), or decreases θ, until it reaches a minimum value (θ_m). These processes develop at speeds v_h, for heating, and v_c, for cooling. There also exists a counter (variable $counter$ in the model) that records the number of times when $\theta = \theta_M$. When $counter = 9$, the system sets the boolean function $beep$ to true, and then it stops. Even if the system is simple enough to be designed as a monolith, we would rather design it modularly, to create the premises for further extensions, which may require the addition of other modules. We use the subsystems $\mathcal{S}_1$ (the heating-cooling system) and $\mathcal{S}_2$ (the counter) (Fig. 8).

$$\begin{array}{ll}
 & \bar{\mathcal{P}}(z) \\
\triangleq & \mathsf{begin\ \ var}\ x : \mathsf{Real}_+ \to T\ ; sel[1..n] : \mathsf{Real}_+ \to \mathsf{Bool}; \\
 & run : \mathsf{Real}_+ \to \mathsf{Nat}\ ; now : \mathsf{Real}_+\ \bullet now := 0\ ; Init\ ; N; \\
 & \ \mathsf{do} \\
 & \quad gg_P.now \to (run.now = 0 \wedge \neg sel[1].now \to sel[1] :-\ (\lambda t \cdot true)\ ; run :-\ (\lambda t \cdot 1) \\
 & \qquad\qquad \| \ \ldots \\
 & \qquad\qquad \| \ run.now = 0 \wedge \neg sel[n].now \to sel[n] :-\ (\lambda t \cdot true)\ ; run :-\ (\lambda t \cdot n) \\
 & \qquad \| \ run.now = 1 \wedge g_L^1.now \to L_1 \\
 & \qquad\qquad \| \ run.now = 1 \wedge \neg g_L^1.now \wedge g_S^1.now \to wS_1c :-\ wS_1\ ; S_1'\ ; run :-\ (\lambda t \cdot 0) \\
 & \qquad\qquad \| \ run.now = 1 \wedge \neg gg_{A_1} \to wS_1c :-\ wS_1\ ; run :-\ (\lambda t \cdot 0) \\
 & \qquad \| \ \ldots \\
 & \qquad \| \ run.now = n \wedge g_L^n.now \to L_n \\
 & \qquad\qquad \| \ run.now = n \wedge \neg g_L^n.now \wedge g_S^n.now \to wS_nc :-\ wS_n\ ; S_n'\ ; run :-\ (\lambda t \cdot 0) \\
 & \qquad\qquad \| \ run.now = n \wedge \neg gg_{A_n} \to wS_nc :-\ wS_n\ ; run :-\ (\lambda t \cdot 0)) \\
 & \quad \| \quad sel.now \wedge run.now = 0 \to Update; \\
 & \qquad\qquad sel :-\ (\lambda t \cdot false)\ ; now := min\{t' \geq now \mid gg.t'\} \\
 & \ \mathsf{od}\ \ \mathsf{end}
\end{array}$$

Figure 7. The system $\bar{\mathcal{P}}$.

$$\begin{array}{ll}
 & \mathcal{S}_1(\theta : \mathsf{Real}_+ \to \mathsf{Real}_+\ ; beep : \mathsf{Real}_+ \to \{false, true\}) \\
\triangleq & \mathsf{begin}\ beep :-\ (\lambda t \cdot false)\ ; \theta :-\ (\lambda t \cdot v_h * (t - now)); \\
 & \quad \mathsf{do}\ \ \neg beep.now \wedge \theta.now = \theta_M \to \theta :-\ (\lambda t \cdot \theta_M - v_c * (t - now)) \\
 & \qquad \| \ \neg beep.now \wedge \theta.now = \theta_m \to \theta :-\ (\lambda t \cdot \theta_m + v_h * (t - now)) \\
 & \quad \mathsf{od}\ \ \mathsf{end} \\
 & \\
 & \mathcal{S}_2(\theta : \mathsf{Real}_+ \to \mathsf{Real}_+\ ; beep : \mathsf{Real}_+ \to \{false, true\}) \\
\triangleq & \quad \mathsf{begin\ var}\ counter : \mathsf{Real}_+ \to \mathsf{Nat}\ \ \bullet \\
 & counter :-\ (\lambda t \cdot 0)\ ; beep :-\ (\lambda t \cdot false)\ ; \theta :-\ (\lambda t \cdot v_h * (t - now)); \\
 & \quad \mathsf{do}\ \ \neg beep.now \wedge counter.now < 9 \wedge \theta.now = \theta_M \to \\
 & \qquad\qquad\qquad\qquad counter :-\ (\lambda t \cdot counter.now + 1) \\
 & \qquad \| \ \neg beep.now \wedge counter.now = 9 \wedge \theta = \theta_M \to beep :-\ (\lambda t \cdot true) \\
 & \quad \mathsf{od}\ \ \mathsf{end}
\end{array}$$

Figure 8. The systems $\mathcal{S}_1$ and $\mathcal{S}_2$.

Interleaved model. The parallel composition of the CAS $\mathcal{S}_1$ and $\mathcal{S}_2$ gives a new CAS, $\mathcal{S} = \mathcal{S}_1 || \mathcal{S}_2$, with an implicit, unique variable now. Following the interleaved execution model, at some moment, both first actions of $\mathcal{S}_1$ and $\mathcal{S}_2$ will be simultaneously enabled (when $\theta = \theta_M$). However, only one of them is selected by the controller. If the chosen action is the one of $\mathcal{S}_1$, the action with the same guard in $\mathcal{S}_2$ becomes disabled, since the temperature is decreased. Thus, the counter misses to record the respective event of $\theta = \theta_M$, therefore presenting a wrong output.

Synchronized model. We now compose $\mathcal{S}_1$ and $\mathcal{S}_2$ by using our newly defined operator '$\sharp$' (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, $\bar{\mathcal{S}}_{new}$ (Fig. 9), with explicit time, by applying the definition given in Fig. 7.

If we repeat the scenario described above, when $\theta = \theta_M$, the semantics of $\bar{\mathcal{S}}_{new}$ does not let time progress unless all the global variables are updated. Therefore, both enabled actions are executed at the same moment of time, and, in

$$
\begin{array}{ll}
 & \bar{\mathsf{S}}_{new}(\theta : \mathsf{Real}_+ \to \mathsf{Real}_+) \\
\triangleq & \mathsf{begin\ var}\ \theta_c : \mathsf{Real}_+ \to \mathsf{Real}_+\ ;\ beep, beep_c, sel[1..2] : \mathsf{Real}_+ \to \mathsf{Bool}; \\
 & \quad counter, run : \mathsf{Real}_+ \to \mathsf{Nat}\ ;\ now : \mathsf{Real}_+\ \bullet \\
 & \quad now :\!= 0\ ;\ counter, run :- (\lambda t \cdot 0)\ ;\ beep, beep_c, sel :- (\lambda t \cdot false); \\
 & \quad \theta, \theta_c :- (\lambda t \cdot v_h * (t - now))\ ;\ now :\!= min\{t' \geq now \mid gg.t'\}; \\
 & \mathsf{do}\ gg_S \to (\neg sel[1].now \wedge run.now = 0 \to sel[1] :- (\lambda t \cdot true)\ ;\ run :- (\lambda t \cdot 1) \\
 & \qquad [\!] \ \neg sel[2].now \wedge run.now = 0 \to sel[2] :- (\lambda t \cdot true)\ ;\ run :- (\lambda t \cdot 2) \\
 & \ [\!] \ run.now = 1 \to (\neg beep.now \wedge \theta.now = \theta_M \to \theta_c :- (\lambda t \cdot \theta_M - v_c * (t - now)) \\
 & \qquad [\!] \ (\neg beep.now \wedge \theta.now = \theta_m \to \theta_c :- (\lambda t \cdot \theta_m + v_h * (t - now)) \\
 & \qquad [\!] \ \neg gg_{S_1} \to \theta_c :- \theta)\ ;\ run :- (\lambda t \cdot 0)) \\
 & \ [\!] \ run.now = 2 \to (\neg beep.now \wedge counter.now < 9 \wedge \theta.now = \theta_M \to \\
 & \qquad\qquad\qquad counter :- (\lambda t \cdot counter.now + 1) \\
 & \qquad [\!] \ (\neg beep.now \wedge counter.now = 9 \wedge \theta = \theta_M \to beep_c :- (\lambda t \cdot true) \\
 & \qquad [\!] \ \neg gg_{S_2} \to beep_c :- beep)\ ;\ run :- (\lambda t \cdot 0))) \\
 & \ [\!] \ sel.now \wedge run.now = 0 \to \theta :- \theta_c\ ;\ beep :- beep_c; \\
 & \qquad sel :- (\lambda t \cdot false)\ ;\ now :\!= min\{t' \geq now \mid gg.t'\} \\
 & \mathsf{od}\ \ \mathsf{end} \\
 & gg_S = \neg beep.now,\ gg_{S_1} = \neg beep.now \wedge (\theta.now = \theta_M \vee \theta.now = \theta_m) \\
 & gg_{S_2} = \neg beep.now \wedge counter \leq 9 \wedge \theta = \theta_M
\end{array}
$$

Figure 9. The system $\bar{\mathcal{S}}_{new}$.

consequence, the counter records all times when θ reaches θ_M, correctly. Additionally, in case we need to add similar modules to the composed system, the synchronized parallel composition lets us reuse the already existing components, at the same time ensuring correct outputs to all inputs.

7. Conclusions and Related Work

The motivation behind the product operator of Milner's SCCS [13] is the same as ours: the system response to stimuli is the composition of the individual reactions of the included subsystems. However, while we synchronize on the updates of a group of variables, the SCCS approach is based on simultaneous execution of actions, which we only get in the last execution round of a synchronized composition.

In [7], Back and von Wright established conditions that enable the designer to perform individual refinements of the components in a parallel composition of AS, by checking noninterference conditions. Still, this does not allow a module designer to independently modify his work, as he needs information about the behavior of the other components.

Bellegarde et al. introduced a similar idea of synchronized parallel composition for event-B systems [8]. In opposition to our model, which increases the *external* determinacy, while preserving the *internal* nondeterminism, the event-B solution preserves also the external nondeterminism. A *gluing invariant* is also necessary when synchronized modules are refined, as the synchronization is performed only with regard to selected events, collected in a synchronization specification. Therefore, the supplier of modules should also deliver to the system integrator, besides the modules themselves, the synchronization specification. Thus, the approach is similar to the one adopted in [7], except for the synchronization idea.

Parallel composition of hybrid system models has also been studied extensively. In the temporal logic of actions of Lamport [12], synchronization is specified as a way of applying *non-interleaving* to system design. This is reached by employing *joint actions*, a concept non-existent in our framework. The conclusion, however, supports our point of view: interleaving "blurs" the distinction among the components. Bornot and Sifakis [9] analyze compositions of timed systems expressed as communicating processes. The authors strive for *maximal progress*, ensured in our case by the synchronized semantics.

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

References

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

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

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

[4] R. J. R. Back, J. von Wright. "Trace refinement of action systems". Proceedings of CONCUR-94, Springer–Verlag, 1994.

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

[6] R.J.R. Back, K. Sere. "Action Systems with Synchronous Communication". Programming Concepts, Methods and Calculi. In E.-R. Olderog. IFIP Trans. A-56, pp. 107-126, 1994.

[7] R. J. R. Back and J. von Wright. "Compositional Action System Refinement". *Formal Aspects of Computing*, Vol. 15, No. 2 pp. 103-117, 2003.

[8] F.Bellegarde, J.Julliand, O.Kouchnarenko. "Synchronized Parallel Composition of Event Systems in B". D. Bert et al.: ZB 2002, Springer-Verlag LNCS 2272, pp. 436-457, 2002.

[9] 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.

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

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

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

[13] R. Milner. "Calculi for synchrony and asynchrony". *Theoretical Computer Science*, Vol. 25, Issue 3, pp. 267-310, 1983.

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

[15] C. Seceleanu, T. Seceleanu. "On Designing for Modularity". TUCS Technical Report 534, June 2003. http://www.tucs.fi