Fundamenta Informaticae XX (2008) 1–27
IOS Press

Modeling Communication with Synchronized Environments

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

Axel Jantsch
Royal Institute of Technology, Stockholm, Sweden
axel@imit.kth.se

Abstract. A deterministic behavior of systems composed of several modules is a desirable design goal. Assembling a complex system from components requires also a high degree of re-usability. The compatibility of the selected components may become a problem even at abstract design levels, due to possible different degrees of model determinacy, possible different execution models, etc. In this cases, an overall deterministic system behavior is difficult to achieve. The development of communication mechanisms between such components will have then to accommodate the differences, so that both correct processing and information exchange (data and control, appropriate choices and relative timing or sequencing) are achieved. For instance, human-machine interaction offers a good example of cooperation between deterministic models (machines) communicating with highly non-deterministic counterparts (the human models, if not restricted). We analyze here such communication mechanisms by "confronting" synchronized and un-synchronized models of execution, in the framework of action systems, a state based formalism. We "force" the two models to coexist within the same context and explore the possibilities of building trustworthy communication channels between them. We base our approach on a combined polling - interrupt scheme, which allows us to mitigate communication issues that may otherwise compromise the correct input-output system behavior. More robust system models are obtained by applying specific correctness rules of refinement. We illustrate our methods on an audio system example, implementable as either a software or a hardware device.

Keywords: System modeling, synchronized / interleaved communication, action systems

*Address for correspondence: University of Turku, Finland

1. Introduction

In our daily activities we are surrounded by electronic or electro-mechanical systems that interact, among them or with humans. That is, systems that operate at different, various speeds need, from time to time, to exchange information, or to execute commands issued in possibly non-deterministic ways. Depending on the actual state the reactive system finds itself when such a command arrives, it may either continue its operation until it reaches a state where the command is acknowledged and serviced, or it must react immediately, as any delay may compromise the overall functionality. The human factor can bring certain kinds of disturbances on the desired output of the controlled systems. This happens as humans may behave in a nondeterministic style, in contradiction with the expected deterministic behavior of the systems they control. At design time, "human models" may be restricted (for instance, considered as reacting in synchrony with the device models), easing thus some of the design issues; still, this is not always a good solution, as the model then may not capture an accurate behavior.

In this study, we concentrate on the modeling of communication strategies between synchronized and un-synchronized modules, communication characterized by a high degree of non-determinism. We illustrate the design and execution problems caused by such communication channels, and offer solutions that mitigate the influence of different execution models on the correctness of data and control transfers. The example we follow describes the interaction between a human user (non-deterministic behavior) and an audio device (deterministic behavior), composed of synchronized sub-systems. The goal is to provide mechanisms that will allow a correct functional description and execution model for the overall user - device system.

While not limited to human-machine interaction, the design of a reactive system generally deals with issues like communication, composability, concurrency and preemption. 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 a solution towards partially reducing the task of the designer of complex systems. On the other hand, the employment of *formal methods* in system design tries to solve the aspects related to *correctness*.

A reasonable system design methodology requires the top-level designer, that is, the system integrator, to compose the system from parallel concurrent components called *modules*. The task of the system integrator is to identify and appropriately connect the components in order to obtain the required functionality. These components may comport certain characteristics that will require from the system-level designer to develop appropriate communication schemes in order to facilitate a correct behavior of the global system.

Berry [10] separated computing systems into *interactive* and *reactive* classes. He also argued that languages based on asynchronous models are not well suited for describing reactive systems [9]. However, in complex applications, one may find components of both classes. One of the main concerns is to accommodate a proper communication between the non-deterministic behavior of the interactive and the deterministic behavior of the reactive modules.

Such modules are modeled here in the formal framework of *action systems*. As illustrated by Cerschi Seceleanu and Seceleanu [25], the synchronized approach to system design improves the control and determinism of the reactive system, as well as its modularity characteristics. Still, it may often be the case that the communication partners (or the environment) of an internally synchronized system are not suitable to be described as synchronized modules, themselves. This, either because it would impose unrealistic restrictions, or because the modules behavior is highly unpredictable and therefore, the

non-deterministic solution covers it very well. In such situations, a simple parallel composition of the synchronized system and its environment is not sufficient, as it may render the benefits of the synchronized model useless: commands may be "lost" just because the system cannot react to them in the current execution cycle, or may be interpreted partially, leading to wrong updates. Hence, operational behaviors must be devised such that they accommodate communication models connecting synchronized modules and their system partners, or the environment.

In the present study, we focus on the characterization of synchronized - unsynchronized module cooperation. For problematic situations, that would affect the validity of results output by the synchronized systems, we offer a solution inspired by the *polling* mechanisms employed in both hardware and software systems. More clearly, we address communication issues between action systems that operate in parallel by employing a *safe points* approach [16, 21], which establishes moments during the program (system) execution, when any pending (asynchronous) exceptions can be analyzed and securely served. The "lines" through which our synchronized modules observe incoming, un-synchronized events, are modeled here as *watched variables*.

Related work. Synchronous - asynchronous communication is most often studied in the domain of hardware design. This is even more stressed lately, in the context of on-chip multiclock domains. But, in general, interaction between deterministic systems that implement certain synchronization mechanisms with systems that act in a less deterministic manner have been the focus of multiple research studies over the years.

Keller [20] established certain conditions required from participants (modules) in asynchronous systems. However, it is important to observe that the synchronized or un-synchronized execution environments that we deal with here do not necessarily map on synchronous or asynchronous hardware implementations, respectively. Our focus is not on *circuit level* asynchronous design, or on modeling speed independent implementations. Conditions [20] such as "stability" of inputs (when a signal provider does not change the output until all the readers have observed it) or "finite-blocking" (if a signal is present at the input of one module, it will eventually be read) may not always apply, and still, such systems have to be modeled and, eventually, implemented.

Berry et al. [11] studied the possibility of introducing asynchronous communication practices in the synchronous language Esterel. The model is Hoare's CSP [17], with its non-deterministic behavior, much like the action systems addressed here.

From a software perspective, concurrent programming lately turned to the solutions offered by the Java language, in the form of threads. We may see threads as independent processing modules, which, at times, must interact by exchanging data. In such situations their respective activities must be synchronized, so that the exchange operation completes in a correct (data-wise) manner. The synchronized composition that we employ here resembles the *barrier synchronization* of Java threads [15], but data updates and detailed operation requirements and mechanisms bring an extensive set of differences.

Early Java systems suffered from unexpected interactions between data, threads, and code. Hawblitzel and von Eicken [16], address the problem of isolation and communication between threads, in a Java extended framework, *Luna*. Safe points are identified and implemented such that transactions on multiple-accessed resources are not colliding. In our approach, the "safe point" is always identified as the last execution round within a cycle, in the synchronized execution model.

Modularity and exception handling are subjects of the study presented by Marlow at al. [21]. In *Haskell*, a functional language, it is not possible to use polling mechanisms for modeling non-synchronized communication. Therefore, *blocking* and *unblocking* procedures are devised. In our perspective here, we model a human command addressing an (internally) synchronized system. Blocking its activity would impose too harsh restrictions, in our opinion, on the model of the system user.

Rudys and Wallach [23] employ both blocking methods and safe points when introducing the *soft termination* concept, a general language mechanism, illustrated in Java programs. The mechanism is intended to be an interface between man and machine, allowing the former to terminate infinite loops in codelets. Therefore, it requires the presence of an administrator or a system resource monitor. We may identify a similar solution in our approach, however the purpose and the realization (the update stage) are different.

Seceleanu and Garlan [26] address the subject of adaptive system behavior in the same context of synchronized composition, by promoting changes similar, but not identical, to the ones we perform in this study. However, the cited work is complementary to the present one, the focus being only on the processing elements, and not on the user-system interaction, which is considered somehow regulated.

In general, synchronous languages, such as Esterel [10], ForSyDe [24], or hardware description languages, such as VHDL [1], do not expose problems as the ones we deal with here. In Esterel, the usage of await and emit statements build up a system of check points that help specifying a deterministic behavior; with the solution offered in [11], even synchronous - asynchronous communication can be handled. Still the presence of specific synchronization mechanisms (such as “ticks”) forces the deterministic (synchronous) components to react at “predefined” moments. In addition, the employment of ”oracles” or ”observers” in other synchronous (related) approaches (Lurette [22], Lustre [14]) are means to test or debug and are not (always) considered as part of the “normal” system execution flow. ForSyDe benefits of a synchronous model of computation that eliminates from the start any non-deterministic specification; VHDL has an execution model which, through the possibility of process re-activation, within the same clock cycle, also eliminates the possible un-desired effects of the non-deterministic selection for execution. However, all the above frameworks impose restrictions that may be seen as unsuitable in certain situations. For instance, when modeling human behavior, it may not be “natural” to consider it developing in synchrony with internal system signals, or even acting on clock edges.

Paper outline. Section 2 introduces those parts of the action system formalism that are relevant for our study. In particular we review and compare parallel, prioritized and synchronized compositions. We also introduce *trace refinement* of action systems since we will demonstrate that the introduction of *watched variables* in a given action system is a trace refinement step and thus automatically leads to a correctly refined action system that does not require additional verification. Section 3 presents the case study and motivating example. An audio filter is modeled as a synchronized cooperation of sub-modules. A user model is then attached to the filter, in a parallel composition. Section 4 discusses how user and filter modules interact with each other exposing potentially unwanted system behavior. To rectify the observed problems we introduce watched variables in section 4.2. Sections 4.4 and 4.5 discuss the results of our approach, while in section 5 we compare and contrast it with alternative approaches. Finally, section 6 concludes the study and points to future work.

2. Action systems

Back and Kurki-Suonio [4] introduced the action systems formalism, providing a framework for specifying and refining concurrent programs. An *action system* is in general a collection of *actions* (guarded commands), executed one at a time. An action system is built according to the following syntax:

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

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* and x are *local* to the action system (1). 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* when the boolean condition g_i (called *guard*) evaluates to true. Executing the action changes the program state in the way described by statement S_i, to which we refer to as the *body* of the action. The change is instantaneous. 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 actions inside $\mathcal{A}$ are iterated as long as the disjunction of the guards holds. The rest of the system (the *environment*) communicates with the action system via *shared* variables, that is, variables updated by the system and read by the environment, or the other way around. Next, we assume the following notations: the set of state variables accessed by some action A, vA, is composed of the *read* variable set of action A, denoted rA, and the *write* variable set of action A, denoted wA. We build the same sets at the system level, considering the local / global partition of the variables: for a given action system $\mathcal{A}$, we have the accessed variables, $v\mathcal{A}$, the global read / write variables, $gr\mathcal{A}/gw\mathcal{A}$ and the local read / write variables, $lr\mathcal{A}/lw\mathcal{A}$. We say that an action A of $\mathcal{A}$ is *global*, if $gw\mathcal{A} \cap wA \neq \emptyset$ or *local*, if $wA \subseteq lw\mathcal{A}$.

An action A_i is defined by the following grammar:

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

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.

Statements and actions in the action systems language are defined using *weakest precondition semantics*, consistent with Dijkstra's original semantics for the language of guarded commands [13]. 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 (the weakest predicate) from which the execution of statement S is guaranteed to terminate in a state satisfying Q [7]. In this paper, we assume that all statements are *conjunctive* and therefore also *monotonic predicate transformers*, that is

$$\begin{aligned} conjunctivity: \quad & \forall p, q \bullet \mathrm{wp}(S, (p \wedge q)) = \mathrm{wp}(S, p) \wedge \mathrm{wp}(S, q) \\ monotonicity: \quad & \forall p, q \bullet (p \Rightarrow q) \Rightarrow \mathrm{wp}(S, p) \Rightarrow \mathrm{wp}(S, q) \end{aligned}$$

In addition, we also require that statements *terminate*, that is, for any given S, $\mathrm{wp}(S, true) \equiv true$. Also, we say that an action $A = g_A \rightarrow S_A$ *terminates* if $\mathrm{wp}(gA \rightarrow S_A, true) \equiv true$.

Guards. The guard of a statement S is defined as $gS \triangleq \neg\mathrm{wp}(S, false)$. The guard may not always be explicitly mentioned, in which case we should not assume it is $true$. Consider, for instance, the case of the nondeterministic assignment $S \triangleq x := x'.Q$. While there is no explicit description, however, the guard of S is is given by the above formula, together with the respective semantics (wp) of the non-deterministic assignment: $gS = \neg(\forall x' \bullet \neg Q)$. Thus, the guard of an action $gA \rightarrow S_A$ is given as $gA \wedge gS_A$. Here, we are interested in *strict* statements, that is statements for which $gS_A \equiv true$. Consequently, the guard of the action A is solely defined by g_A.

At the system level, the guard of an action system is given then by (1) is $gg_A \triangleq \bigvee_1^n g_k$, where g_k is the guard of the respective action A_k.

Parallel composition of action systems. Consider two action systems given as follows (we omit the type specification for variables, as it is not relevant).

$$\begin{aligned} \mathcal{A}(z_A) &\triangleq \ \mathsf{begin\ var}\ x_A \bullet Init_A\ ;\ \mathsf{do}\ g_A \rightarrow S_A\ \mathsf{od\ end} \\ \mathcal{B}(z_B) &\triangleq \ \mathsf{begin\ var}\ x_B \bullet Init_B\ ;\ \mathsf{do}\ g_B \rightarrow S_B\ \mathsf{od\ end} \end{aligned}$$

Then, the parallel composition [3] of $\mathcal{A}$ and $\mathcal{B}$ is the system $\mathcal{P} = \mathcal{A} \,||\, \mathcal{B}$:

$$\mathcal{P}(z_P) \triangleq \mathsf{begin\ var}\ x_P \bullet Init_A\ ; Init_B\ ;\ \mathsf{do}\ g_A \rightarrow S_A [\!]\ g_B \rightarrow S_B\ \mathsf{od\ end}$$

The composed action system essentially combines the variables, the initialization statements and the actions of the two subsystems. The initialization of the common variables $z = z_A \cap z_B$ must be consistent, that is, they are assigned the same initial values by both initialization statements, $Init_A$ and $Init_B$ (such that the order in $Init_A$; $Init_B$ is not important). Some of the previously global variables of $\mathcal{A}$ and $\mathcal{B}$, the ones that were used to model the communication between the two systems (so, part of $z_A \cap z_B$), will become local variables of $\mathcal{P}$, except if they were shared with other systems, too, in a larger context (in which case they remain global variables of $\mathcal{P}$, too). We add the former to the reunion of the individual local variables of $\mathcal{A}$ and $\mathcal{B}$, thus obtaining the set of local variables of $\mathcal{P}$, x_P. The global variables z_P are defined as $z_P \triangleq z_A \cup z_B - x_P$.

Prioritized composition. One way to express preemption, in action systems, comes in the form of a *macro* operator, based on the semantics of the choice operator. The prioritized composition of two actions A and B was defined by Sekerinski and Sere [27] as:

$$A \,//\, B \mathrel{\widehat{=}} A \ [\!]\ (\neg gA \rightarrow B)$$

At system level, the composition $\mathcal{A} \,//\, \mathcal{B}$ allows an action in $\mathcal{B}$ to be executed, only if there is no enabled action in $\mathcal{A}$.

Invariants. A predicate $I(vA)$ – I in short – is an *invariant* of the action $A \mathrel{\widehat{=}} g \rightarrow S$, if it holds prior to and after the execution of A. We then say that I is *preserved* by A, that is, $g \wedge I \Rightarrow \mathrm{wp}(S, I)$.

At the system level, a predicate $I(v\mathcal{A})$ is an *invariant* of the action system $\mathcal{A}$ if it is established by $Init$, that is, $true \Rightarrow \mathrm{wp}(Init, I)$, and also if it is preserved by each action of $\mathcal{A}$.

Refinement of actions. An action A is *refined* by the action C, written $A \leq C$, if, whenever A establishes a certain postcondition, so does C [3]. Additionally, let I be an invariant over the action C. Then, action A is refined by action C using the invariant I, denoted $A \leq_I C$, if

$$\forall Q. I \wedge \mathrm{wp}(A, Q) \Rightarrow \mathrm{wp}(C, I \wedge Q)$$

If not only an *algorithmic* refinement is intended, but one also introduces changes in the data structures, we have then a *data refinement* [7]. In such cases, an additional predicate should relate the new structures (action C) to the ones they replace (action A).

Trace refinement of action systems. The semantics of a reactive action system is given in terms of behaviors [6]. A *behavior* of an action system is a sequence of states, $b = \langle (x_0, y_0), (x_1, y_1) \ldots \rangle$, where each state has two components: the *local* and the *global* state. 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. Informally, 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 $\mathcal{A}$. The theoretical basis for the trace refinement is expressed by the following *trace* refinement lemma [5].

Lemma 2.1. Given the action systems

$$\begin{aligned} \mathcal{A}(z) : T_Z) &\triangleq \mathsf{begin\ \ var}\ \ a : T_a \ \ \bullet\ \ a, z_A := a_0, z_0\ ;\ \mathsf{do}\ \ A\ \mathsf{od}\ \ \mathsf{end} \\ \mathcal{C}(z : T_Z) &\triangleq \mathsf{begin\ \ var}\ \ c : T_c \ \ \bullet\ \ c, z_C := c_0, z_0\ ;\ \mathsf{do}\ \ C \parallel X\ \mathsf{od}\ \ \mathsf{end}, \end{aligned}$$

let $I(v\mathcal{C})$ be an invariant of the system $\mathcal{C}$. The concrete system $\mathcal{C}$ (trace) refines the abstract system $\mathcal{A}$, denoted $\mathcal{A} \sqsubseteq_I \mathcal{C}$, if:

1. Initialization: $I(c_0, z_0) \equiv true$. This means that I is established by the initialization statement.
2. Main action: $A \leq_I C$. Refinement of actions.
3. Auxiliary action: $\mathsf{skip} \leq_I X$. The possibly new added actions behave like skip, that is, they do not update global variables of the system $\mathcal{C}$.
4. Continuation condition: $I \wedge gA \Rightarrow gC \vee gX$. Whenever an action is enabled in $\mathcal{A}$, one action is also enabled in $\mathcal{C}$.
5. Internal convergence: $I \Rightarrow \mathrm{wp}(\ \mathsf{do}\ X\ \mathsf{od}\ , true)$. The newly introduced actions may not execute for ever. We direct the reader to [13] for the definition of loops and the corresponding weakest precondition computation.

2.1. Execution of action systems

Starting with the original paper by Back and Kurki-Suonio [4], the sequential execution model was established as a *de facto* reasoning environment for action systems designs. Parallel executions are modeled by interleaving actions that have no read / write conflicts.

Thus, the execution of an action system 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 action system, are enabled. Non-deterministically, it selects one of them for execution. 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* (equivalent to the execution of an action). After this, the controller evaluates the new state, observes the enabled actions and starts another execution round.

Synchronized environments. An additional virtual execution model has recently been added into the framework of action systems [25], the *synchronized environment*. Here, the execution of the components of the system under design is synchronized with respect to the updates on the global variables of the respective components. This models the unitary reaction of a composition of action systems to a given input situation. In brief, the *observable* execution model is changed as follows.

The controller selects one of the components for execution, in a nondeterministic manner. After performing all the possible execution rounds, with respect to the input state, the controller *marks* the corresponding action system as *executed*. However, the global variables of the action systems component are not updated at this stage; instead, the new values are stored in local copies of the respective global variables. Next, the controller selects another *un-executed* component and performs the same operation. Due to the updates of the copies, instead of the actual global variables, between selections, the visible state of the composition does not change. When all the components have been executed, the controller runs a final round in which the appropriate values are assigned to the global variables of the synchronized composition. This also signals the end of an execution *cycle*, followed, if the system remains enabled, by the beginning of another.

A synchronized environment presents some useful characteristics. The first one is an increased capability of reaction: no special attention must be given to the order in which elements of the composition are selected for execution. The second impact on design is reflected by an improved system modularity: responsibility of upgrading the modules stands only in the hands of the module designer, and this information is transparent to the system level integrator, concerned only with the overall functionality and the interface of the employed components. A synchronized environment assumes certain properties of the composing action systems: they must be *proper* action systems.

Definition 2.1. *Consider the action system $\mathcal{A}$:*

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

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

• $gw\mathcal{A} \subseteq wS$ *– meaning that S is a global action of $\mathcal{A}$.*

• $wL \subseteq lw\mathcal{A}$*– meaning that L is a local action of $\mathcal{A}$.*

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

A synchronized environment is realized when a certain number of proper action systems evolve following the informal execution scenario introduced above. Their composition is a new action system, obtained as follows.

Definition 2.2. Let us consider n proper action systems:

$$\mathcal{A}_k(z_k : T_{z_k}) \quad \triangleq \quad \mathsf{begin\ var}\ x_k : T_k \bullet\ Init_k\ ;\ \mathsf{do}\ g_S^k \rightarrow S_k\ []\ g_L^k \rightarrow L_k\ \mathsf{od\ end},\ k = 1 \ldots n$$

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

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

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

The system $\mathcal{P}$ introduced above represents the "flattened" model of the synchronized composition of $\mathcal{A}_1, \ldots, \mathcal{A}_n$, viewed above as *components*.

The set z of global variables of $\mathcal{P}$ is, initially, the union of the global variables sets of each individual system: $z = \bigcup_k z_k$. 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 that model 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 (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*. The guard gg_P is the disjunction of all the module guards: $gg_P \triangleq gg_{A_1} \vee \ldots \vee gg_{A_n}$. The *Init* statement is the sequential composition of the individual *Init*$_k$ statements to which we add the initialization of variables $wS_kc,\ k \in 1,..,n$, *sel* and *run*:

$$Init \quad \triangleq \quad Init_1\ ;..;\ Init_n\ ;\ wS_1c,..,wS_nc := wS_1,..,wS_n\ ;\ run := 0\ ;\ sel := false$$

The action *Update* represents the integrator's choice of deciding how the actual updates of the global variables are performed. We consider $sel = sel[1] \wedge \ldots \wedge sel[n]$, while the assignment $sel := false$ sets all the vector components $sel[1], \ldots, sel[n]$ to $false$ - as in the above definition of $Init$. In an initial set-up, *Update* is specified as the atomic sequence:

$$Update \triangleq wS_1 := wS_1c\ ;\ \ldots\ ;\ wS_n := wS_nc\ ;\ sel := false$$

Two easy to asses properties of the '$\sharp$' operator are commutativity and preservation of properness - that is, the action system $\mathcal{P}$ is a proper action system, too [25]. In addition to these, we mention that, at this moment, it is not proven that $\sharp$ is associative. Even though, intuitively, one may guess so, the proof is difficult and is one of the on-going work topics. Thus, certain features of hierarchical design may be hindered, especially system decomposition. However, building up hierarchies is not difficult. The systems do not require representation at the "flattened" level, and, as long as iterated composition - decomposition activities are not required, "bottom-up" design techniques are applicable.

2.2. Synchronized composition in system design

The motivation behind employing the synchronized composition instead of the traditional parallel one in certain system development is given in details in [25]. However, we restate briefly the main issues here, as we will use the approach in the example description of section 3.

Let us analyze the model of a simple digital filter [18] as 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 $y(n) = \sum_{k=0}^{N-1} h(k) \times x(n-k)$, 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. In the end, a filter may have either a software or a hardware implementation.

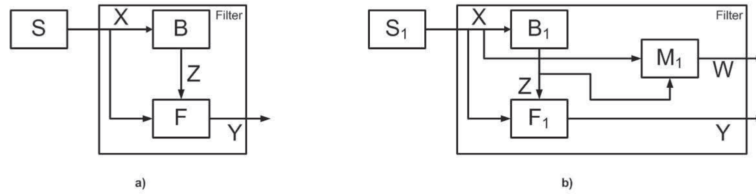

Figure 1. Filter representation: a) single channel; b) two channel.

From the above short description of the filter one can identify two sub modules of such a device: the storage FIFO–like buffer, and the actual filter. In the following, we model the buffer by system $\mathcal{B}$ and system $\mathcal{F}$ performs the filtering. In addition, we also need a signal source, modeled here by system $\mathcal{S}$. The whole system is illustrated in Fig. 1 a), with the action systems descriptions given as:

$\mathcal{S}(X : T)$
$\triangleq$ 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 $Z[0], \ldots, Z[N-2] := X, \ldots, Z[N-3]$ od
end

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

where $h[0..N-1]$ are the filter coefficients that do not change during the execution of the system.

Suppose now that we organize the composed system ($\mathcal{A}$) based on the parallel operator:

$$\mathcal{A} = \mathcal{S} \,||\, \mathcal{F} \,||\, \mathcal{B}$$

The system $\mathcal{A}$ executes in an interleaved, non-deterministic manner. The immediate problems that one may notice is that we cannot ensure the correct processing of the samples emitted by $\mathcal{S}$: a valid execution scenario of $\mathcal{A}$ can be $\mathcal{S}\,;\mathcal{S}\,;\mathcal{F}\,;\mathcal{F}\,;\mathcal{B}\,;\mathcal{S}\,;\mathcal{B}\,;\ldots$, which affects the result presented by $\mathcal{F}$, either because of losing incoming X samples, or by reading wrong operands when computing the multiplication.

This can be solved by implementing a communication channel between $\mathcal{S}$, on one hand, and $\mathcal{F}$ and $\mathcal{B}$ on the other, in order to disable $\mathcal{S}$ until the current signal sample has been processed by $\mathcal{F}$ and $\mathcal{B}$.

There is still the problem of ordering the execution of $\mathcal{F}$ and $\mathcal{B}$, otherwise consecutive X samples will be processed with the same elements stored by $\mathcal{B}$. Hence, another communication channel must be conceived between $\mathcal{B}$ and $\mathcal{F}$, in order to impose a consistent execution order. The communication channels are modeled by request - acknowledge pairs of boolean signals, $req_S, ack_S, req_F, ack_F$, the system descriptions resulting as

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

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

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

One may now consider that a good filter model has been devised. The execution of $\mathcal{S}_1||\mathcal{F}_1||\mathcal{B}_1$ always follows the same trace, thus the output is the expected one. The models $\mathcal{S}_1, \mathcal{F}_1$ and $\mathcal{B}_1$ can be stored in a design library for later utilization.

Suppose further, that we want to re-use the above modules, but with the addition of another filtering unit, hence to process data on two channels, with different filter coefficients (Fig. 1 b)). One can think of just introducing the module $\mathcal{M}_1$, an instance of the $\mathcal{F}_1$ description above, but with different filter coefficients. However, due to the interleaved model of execution, again, $\mathcal{S}_1$ must accommodate the presence of $\mathcal{M}_1$, such that no samples emitted by $\mathcal{S}$ are "lost" by $\mathcal{M}_1$. Hence, a separate communication channel must connect the two systems. This means a re-design of the initial system $\mathcal{S}$. The same applies to $\mathcal{B}$. In addition, the output delivered by $\mathcal{F}_1$ and $\mathcal{M}_1$ will be sequential; even more, the sequence will not be consistent, as the controller will freely select either one of them for execution, with no priority.

In conclusion, the traditional parallel model with the associated interleaved execution may not be supportive for a modular approach to system design. The proposed solution comes in the form of employing the '$\sharp$' operator instead of the parallel one. Observe that, if the initial system description is

$$\mathcal{A} = \mathcal{S} \sharp \mathcal{F} \sharp \mathcal{B},$$

there is no need for the additional communication channels. Still, the order in which modules are selected for execution is not relevant, as the input space is "stable" during one execution round, and, therefore, no wrong updates are performed. Furthermore, a possible re-utilization of the same modules with an extra filtering device ($\mathcal{M}$, an instance of $\mathcal{F}$), there is no need to re-design any of the components, we can just re-write the composition as $\mathcal{A}' = \mathcal{S} \sharp \mathcal{F} \sharp \mathcal{M} \sharp \mathcal{B}$. Moreover, the output from $\mathcal{F}$ and $\mathcal{M}$ will be updated simultaneously.

The modularity features offered by the synchronized composition are completed by relaxed system refinement capabilities. However, these are not relevant for the present study, and we direct the reader for details to [25].

It is important to notice that, above, the synchronized composition was considered in synchronized environments, that is, where *all* the system components are connected with the '$\sharp$' operator. This may not be a generic solution for various system properties and requirements. Therefore, in the following, we analyze the interaction between the two presented models of execution, the interleaved and the synchronized models. For the latter, this will imply a change of the *Update* action. We present the necessary modification on a running example.

3. Design example

In this section, we introduce the reader to a case study that combines the two styles of design presented in the previous section.

Out of a variety of synchronized systems that interact in a un-synchronized manner with other systems, or with the environment, as potential exemplifications of our approach, we mention here:

- sensor based traffic light controllers (lights are synchronized on all directions) and car / pedestrian traffic (non-deterministic vehicle / pedestrian arrival at cross-roads),
- elevator controller (supervising, in a synchronized manner the elevator movement, door opening and closing, etc) and human commands (unaware of the elevator positioning, therefore non-deterministic from the controller point of view),

- train junction control (similar to the traffic light controller).
- application / operating system (totally non-deterministic, given the wide range of possible requests) and hardware platform (acting not in a synchronized, but even in a synchronous way).
- ATM machines (data transfers synchronized with the available controls) and users (non-deterministic in their choices).

All the mentioned examples raise the problem of *system reaction* to input stimuli, which may come at un-predictable moments, with respect to the current system state. However, while, for instance, it would be unpleasant for one to press the elevator button and the elevator to continue its movement without stopping, because it could not react in proper time to the command, this can be resolved at a later moment, without affecting the basic functional specification of the system.

The above examples emphasize, at the same time, the need for a correct design methodology, as errors in design may lead to human victims. Still, we settle here for a simpler example, that of an *audio system* design. In brief, our audio system is composed of a signal source, a two-channel audio filter and volume controllers for each channel. A user may operate the device by modifying the volume on either or both channels, whenever desired. The problem is to model a system that reacts both to the signal source and to the commands of the user, and produces the expected output.

The reason for our selection is that, apart from the *reactivity* issue - resolved by a synchronized design as detailed in section 2.2, the chosen example raises an additional problem dimension, and this regards the constraint of simultaneous updates on the two channels, in the presence of external, non-deterministic commands. For instance, in the case of the traffic light controller, due to security reasons, red light - in one direction, is not supposed to switch *at the same time* with the green light, on the orthogonal direction. A delay in turning green in one direction is actually required. Therefore, a sequential update is observed. On the contrary, in the stereo audio system that we propose, the two-channel output must present both updates at the same time, regardless of the issued volume changes, otherwise distortions are observed. A sequential update solution would, thus, be unacceptable in this case.

In a final note on the example selection, we acknowledge the fact that one can find solutions to the problems raised by each of the mentioned examples. Frequency selection, speed of the elevator engine, regulations on vehicle speed, or human latency in observing changes are just a few mechanisms or reasons why working solutions can be accepted. However, our focus here is the accuracy of the model and the stepwise correct approach to system development, even if certain requirements may seem too strict, when observed in daily utilization.

3.1. The design elements

In the following, we shortly introduce the elements of our example.

The filter. A very common utilization of filters can be observed for instance in modern day audio applications. Tasks such as channel separation, equalizers, etc are implemented using various kinds of (digital) filters. We will use the filter models that we described in section 2.2.

The volume controllers. The interaction audio system-user is realized through the volume controllers on the two channels.

Each of them is an instance of the action system $\mathcal{V}$, given as

$$\begin{array}{l} \mathcal{V}(Vol : integer\ 0..10\ ; in, out : T_D) \\ \quad \text{begin} \quad \bullet\ Vol := 0\ ; in, out := in_0, out_0\ ; \\ \quad\quad \text{do}\ out := Vol \times in\ \text{od} \\ \quad \text{end} \end{array}$$

The above system performs a simple update on the output variable out, by multiplying the input in with the user selection captured in the value of the variable Vol.

The audio system. The setup for a two-channel audio system, where the user has the possibility of selecting the desired volume for either of the channels is introduced in Figure 2. The modules R and L are instantiations of the $\mathcal{F}$ system, whereas the modules Vol_R and Vol_L perform the desired amplification, and they are instances of the action system $\mathcal{V}$.

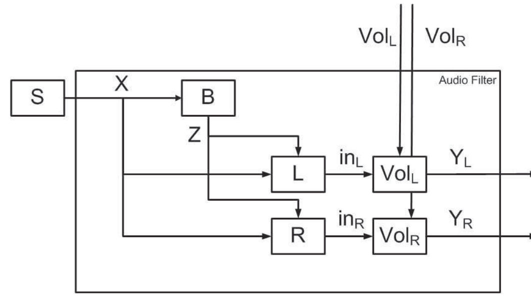

Figure 2. The audio system.

One of the requirements of the audio system of Fig. 1 b) is that the changes that affect both outputs Y_R and Y_L must be observed simultaneously. Therefore, the audio device is modeled by the action system $\mathcal{A}$, a synchronized cooperation of its modules: $\mathcal{A} \mathrel{\widehat{=}} \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{R} \sharp \mathcal{L} \sharp Vol_R \sharp Vol_L$

Observe that one can build the audio system $\mathcal{A}$ starting from the original description of section 2.2, that is, $\mathcal{A} \sharp \mathcal{B} \sharp \mathcal{F}$, by identifying $\mathcal{F}$ with either $\mathcal{R}$ or $\mathcal{L}$, adding the other filter and the respective volume controllers. If the starting point would have been a parallel system description, such an easy update would not have been possible.

In the next sections, we study the interaction of a synchronized model - the audio system $\mathcal{A}$ - with an environment considered *fair*, but unpredictable.

3.2. Modeling the user

As listeners to our stereo audio device, we expect to be able to change the audio characteristics, by raising or lowering the volume of either the "left" or "right" channel. Such behavior is captured by the system

$$\begin{array}{l} \mathcal{U}ser(Vol_L, Vol_R : integer\ 0..Vol_{Max}\ ; in, out : T_D) \\ \quad \text{begin var}\ \Delta_V : integer\ \bullet \\ \quad\quad\quad Vol_L, Vol_R := Vol_{Max}/2\ ; \Delta_V := 1\ ; \\ \quad\quad \text{do}\ Vol_L := V'_L.Q_L\ [\!]\ Vol_R := V'_R.Q_R\ \text{od} \\ \quad \text{end} \end{array}$$

where

$$Q_L \equiv V'_L = Vol_L \vee (V'_L = Vol_L + \Delta_V \wedge\ 0 < V'_L \leq Vol_{Max})$$
$$\vee (V'_L = Vol_L - \Delta_V \wedge\ 0 \leq V'_L < Vol_{Max})$$
$$Q_R \equiv V'_R = V_r \vee (V'_R = Vol_R + \Delta_V \wedge\ 0 < V'_R \leq Vol_{Max})$$
$$\vee (V'_R = Vol_R - \Delta_V \wedge\ 0 \leq V'_R < Vol_{Max})$$

Observe that the above system is always enabled, that is, $gg\mathcal{U}ser \equiv true$. In the following sections, however, we assume that actions of the system $\mathcal{U}ser$ are not selected infinitely often by the controller. In this case, we are interested in analyzing the moments when such actions are selected, and the impact of their execution on the behavior of the audio system, modeled by $\mathcal{A}$.

4. System interaction

When analyzing the interaction between the systems $\mathcal{U}ser$ and $\mathcal{A}$, one should notice that the user actions are independent of the activity of the audio system. Therefore, even though this would considerably simplify the modeling of the communication between these two entities, one should not consider a synchronized composition of the two action systems. Instead, the whole system is modeled as a parallel composition of the two models: $\mathcal{U}ser \parallel \mathcal{A}$.

In the following, we study how the above composition satisfies the requirement that once the volume level has been changed by the user, the audio system appropriately reacts to this new situation. Informally, this means that if the current execution session of the audio system is not finished, one must consider the new values of the volume lines, for the current input sample. Therefore, part of the processing may be required to be re-executed. Hence, the actions of the user act as interrupt generators for the digital device, and the latter has to respond to such events. As the focus of our study is located "within" the synchronized composition, we will describe and analyze the synchronized compositions in their respective flattened versions.

4.1. Modeling the audio system

In order to study the actual realization of the above scenario, we analyze in more detail the system $\mathcal{A}$:

$\mathcal{A}(Vol_R, Vol_L : integer\ 0..Vol_{Max}\ ; Y_R, Y_L : T_D)$
begin
  var $X, X_c, Z[0..N-2], Z_c[0..N-2], h_R[0..N-1], h_L[0..N-1] : T_D;$
    $Y_{R_c}, Y_{L_c}, in_R, in_L, in_{R_c}, in_{L_c} : T_D\ ; sel[1..6]$: Bool ; $run : integer\ 0..6$ $\bullet$
    $Vol_R, Vol_L := 0\ ; in_R, in_L, in_{R_c}, in_{L_c} := 0\ ; h_R[0..N-1], h_L[0..N-1] := H_R, H_L;$
    $X, X_c, Z, Z_c, Y_R, Y_L, Y_{R_c}, Y_{L_c} := 0\ ; run := 0\ ; sel := false;$
  do
    $Selection$
    $[\!]\ run = 1 \rightarrow Y_{R_c} := Vol_R \times in_R\ ; run := 0$
    $[\!]\ run = 2 \rightarrow Y_{L_c} := Vol_L \times in_L\ ; run := 0$
    $[\!]\ run = 3 \rightarrow X_c := X'.(X' \in T)\ ; run := 0$

$\| \ run = 4 \rightarrow Z_c[0], \ldots, Z_c[N-2] := X, \ldots, Z[N-3] \ ; run := 0$
$\| \ run = 5 \rightarrow in_{R_c} := \sum_{k=1}^{N-1} h_R(k) \times Z(k-1) + h(0) \times X \ ; run := 0$
$\| \ run = 6 \rightarrow in_{L_c} := \sum_{k=1}^{N-1} h_L(k) \times Z(k-1) + h(0) \times X \ ; run := 0$
$\| \ sel \wedge run = 0 \rightarrow Update$
od
end,
$Selection = run = 0 \wedge \neg sel[1] \rightarrow run := 1 \ ; sel[1] := true$
$\| \ \ldots$
$\| \ run = 0 \wedge \neg sel[6] \rightarrow run := 6 \ ; sel[6] := true$
$Update = X := X_c \ ; Z := Z_c \ ; \ldots ; sel := false$

Effects of the un-synchronized interaction between $\mathcal{U}ser$ and $\mathcal{A}$. Let us analyze a possible scenario regarding the execution of the composition $\mathcal{U}ser \parallel \mathcal{A}$. Suppose that the $Selection$ action performs $run := 2$, thus enabling the action $run = 2 \rightarrow Y_{L_c} := Vol_L \times in_L \ ; run := 0$, in $\mathcal{A}$. After the latter is executed, the *controller* may choose to select now one action of the system $\mathcal{U}ser$, and, for instance, lowers the volume on the left channel: $Vol_L := V'_L.Q_L$. However, this option cannot be followed by a reaction of $\mathcal{A}$, in this execution cycle, as the intermediate update on Y_{L_c} has already been executed. Hence, the immediate next time when the user observes a change in the output of the system $\mathcal{A}$, he / she will not observe the modification of the volume on the left channel, as selected.

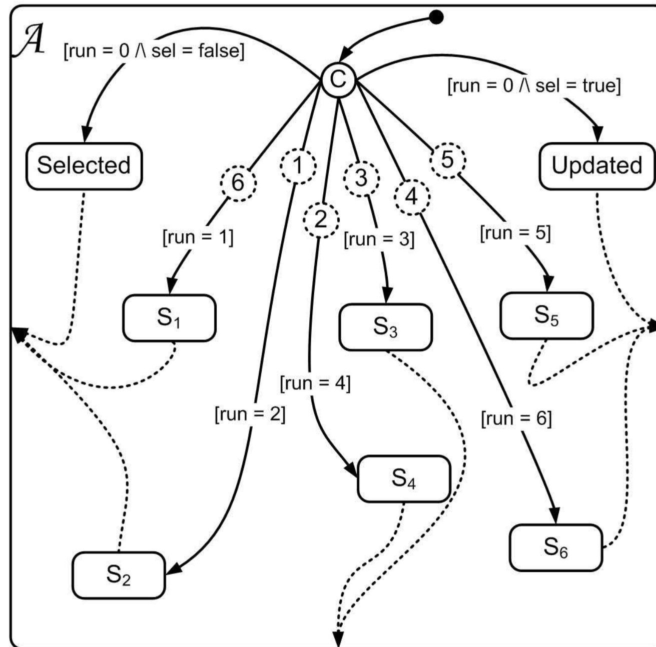

Figure 3. Execution of the system $\mathcal{A}$.

One possible illustration of the above scenario is described in statecharts-like representation of Figure 3, where the execution controller is identified as the choice operator. The actions of the audio system

$\mathcal{A}$ are to be identified by the corresponding value of the transition guard. After their execution, the system ends in one of the states $S_1, \ldots, S_6$, and then it returns (the dotted lines) for another **controller** decision. One possible order for executing these actions, based on the non-deterministic results provided by the $Selection$, is illustrated by the circled numbers 1. . .6. Notice that the modification of the left volume is the first one to be executed. The user intervention that changes Vol_L appears at any later moment, but before the termination of the current execution cycle. After all the transitions $[run = 1], \ldots, [run = 6]$ have been taken, the system goes into the $Updated$ state, and a new execution cycle may begin. In addition to the above scenario, suppose that the user does not only modify the left channel value, during the same execution cycle, but he also changes the volume on the right channel. Before the controller selects the action $Y_{R_c} := Vol_R \times in_R; run := 0$, the module $\mathcal{User}$ is chosen, and action $Vol_R := V'_R.Q_R$ is executed. Due to the fact that this change occurs prior to the update in system $\mathcal{A}$, its effect on the final result will be visible. Hence, the user performs two modifications, but the effect of a single one can be observed.

This problem is caused by the fact that there are two execution models, the interleaved and the synchronized, which is a grouping of interleaved executions. Necessarily, we have to devise a manner in which these two models can coexist.

4.2. Watched variables

In order to solve the problem described above, we first introduce the concept of *watched variable*. From the point of view of a synchronized composition, such a variable is a global connection with the environment. Its value may change during any execution round, and the latest value must be taken into consideration when the final updates are presented as the output of the composition.

The mechanism that we propose for modeling this kind of behavior affects two of the generic actions present in the flattened representation of a synchronized composition. Observe that the first action to be executed in an execution cycle is the selection action, identified as $Selection$. This is the place where we specify the watched variables and assign their initial values. In the audio system modeled in our example, a new selection action can be specified as follows.

$$\begin{array}{ll} & Selection_1 \\ = & \;\; run = 0 \wedge \neg sel[1] \rightarrow sel[1] := true \,;\, run := 1 \\ & [\!] \;\ldots \\ & [\!] \; run = 0 \wedge \neg sel[6] \rightarrow sel[6] := true \,;\, run := 6 \\ & [\!] \; proc \rightarrow Vol_L^{start} := Vol_L \,;\, Vol_R^{start} := Vol_R \,;\, proc := false \end{array}$$

The new selection action stores the values of the watched variables Vol_L and Vol_R into the local variables Vol_L^{start} and Vol_R^{start}.

The new local variable, $proc$ (*processed*), is intended to identify the starting of a new execution cycle, whenever this variable becomes $false$. Notice that in the above description of $Selection_1$, this is not implemented. The choice composition may allow the update on $proc$ to be executed after several of the other actions of $Selection_1$ have been processed. This problem can be solved by assigning to the last action of $Selection_1$ a higher priority than the other components of $Selection_1$, as a refinement step

[27]. Thus, we obtain the action $Selection_2$:

$$
\begin{array}{ll}
 & Selection_2 \\
= & \quad proc \rightarrow Vol_L^{start} := Vol_L \,; Vol_R^{start} := Vol_R \,; proc := false \\
 & // \,(run = 0 \wedge \neg sel[1] \rightarrow sel[1] := true \,; run := 1 \\
 & \quad [\!] \ \ldots \\
 & \quad [\!] \ run = 0 \wedge \neg sel[6] \rightarrow sel[6] := true \,; run := 6)
\end{array}
$$

By introducing the new local variable $proc$, and replacing the action $Selection$ with $Selection_2$, we obtain a new system, $\mathcal{A}_1$, as follows.

$$
\begin{array}{l}
\mathcal{A}_1(Vol_R, Vol_L : integer\ 0..Vol_{Max} \,; Y_R, Y_L : T_D) \\
\textsf{begin} \\
\quad \textsf{var}\ X, X_c, \ldots \,; proc, sel[1..6] : \textsf{Bool} \,; run : integer\ 0..6 \ \bullet \\
\quad\quad Vol_R, Vol_L := 0 \,; \ldots \,; run := 0 \,; proc := true \,; sel := false; \\
\quad \textsf{do} \\
\quad\quad\quad Selection_2 \\
\quad\quad [\!] \ run = 1 \rightarrow Y_{R_c} := Vol_R \times in_R \,; run := 0 \\
\quad\quad [\!] \ \ldots \\
\quad\quad [\!] \ sel \wedge run = 0 \rightarrow Update \\
\quad \textsf{od} \\
\textsf{end}
\end{array}
$$

It is easy to check that the above described refinement steps lead to the trace refinement $\mathcal{A} \sqsubseteq \mathcal{A}_1$, as specified by Lemma 2.1:

(1) we do not consider any specific invariant, hence the initialization state is not affected;

(2) there are no changes of the existent actions;

(3) the new action refines (behaves like) $skip$ (no updates of the global variables);

(4) enabledness of the system is not affected;

(5) the added action disables itself, thus, it terminates.

4.3. Catching and processing events

Storing the initial values of watched variables at the beginning of an execution cycle is just the first step towards observing and processing events that may appear during the execution of a synchronized environment. The next step is the notification of the event and its consequent processing.

The execution of the *Update* action comes at the end of an execution cycle. This is the moment when we should check if, in parallel with the previous execution rounds, anything worth of system's

attention occurred at the interface with the environment. In our example, we are interested in observing any possible change in the values of the volume variables. Hence, we remodel the *Update* action as follows:

$$Update_1 \;=\; (Vol_R^{start} \neq Vol_R \rightarrow sel[1] := false \,\|\, Vol_L^{start} \neq Vol_L \rightarrow sel[2] := false) \;//\; (Update\,;proc := true)$$

Before acting on the global variables of the synchronized system $\mathcal{A}_1$, as specified by the initial action *Update*, the new version, $Update_1$ starts by checking if there are any changes in the values of the watched variables Vol_L or Vol_R. We have assigned a higher priority to this activity, by using the prioritized composition. The new specification triggers a re-execution of the actions $Y_{R_c} := Vol_R \times in_R\,;run := 0$ or (and) $Y_{L_c} := Vol_L \times in_L\,;run := 0$, as necessary.

Similar to the case of action $Selection$, at the system level, the specification of *Update*$_1$ leads to a trace refinement, in the sense of Lemma 2.1.

Providing a new form for the initial *Selection* and *Update* actions is consistent with our view on system design, expressing that the system-level integrator is responsible for the set-up of the necessary modules and the communication inside and outside the synchronized system.

The flattened new version of the audio system is given by

$$
\begin{array}{l}
\mathcal{A}_2(Vol_R, Vol_L : integer\ 0..Vol_{Max}\,; Y_R, Y_L : T_D)\\
\textsf{begin}\\
\quad \textsf{var}\ X, X_c, \ldots \quad \bullet\\
\qquad in_R, in_L, in_{R_c}, in_{L_c} := in_0\,; \ldots;\\
\qquad run := 0\,; proc := true\,; sel := false;\\
\quad \textsf{do}\\
\qquad\quad Selection_2\\
\qquad \|\ run = 1 \rightarrow Y_{R_c} := Vol_R \times in_R\,; run := 0\\
\qquad \|\ \ldots\\
\qquad \|\ sel \wedge run = 0 \rightarrow Update_1\\
\quad \textsf{od}\\
\textsf{end}
\end{array}
$$

We recall now the execution scenario described at the beginning of section 3.2. Running now the composition $User \parallel \mathcal{A}_2$ (Figure 4), observe the additional execution of the left-channel update $run = 2 \rightarrow Y_{L_c} := Vol_L \times in_L\,; run := 0$ (at number 7).

4.4. Model analysis

We may now claim that we have a reliable overall system description, given as $\mathcal{U}ser \parallel \mathcal{A}_2$, as far as system reactiveness to user commands is concerned.

We wanted to ensure that system $\mathcal{A}_2$ outputs two values, Y_L, Y_R that are consistent both with the user's selection of volume and with the signal offered by the source $\mathcal{S}$. This can be checked by assessing

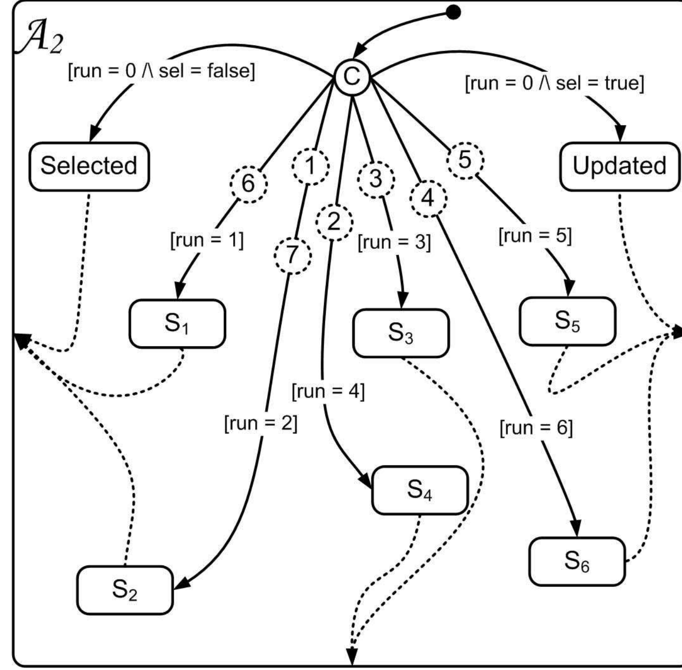

Figure 4. Execution of the system $\mathcal{A}_2$.

the invariance of the predicate

$$\begin{aligned} I \ \widehat{=} \quad & sel \wedge run = 0 \wedge (Vol_L^{start} = Vol_L) \wedge (Vol_R^{start} = Vol_R) \\ & \Rightarrow (Y_L = Vol_L \times in_L) \wedge (Y_R = Vol_R \times in_R) \end{aligned} \tag{2}$$

over the actions of $\mathcal{A}_2$.

The relation I above shows that the update performed by $\mathcal{A}_2$ is consistent with the values of the Vol_R and Vol_L controls. The term $sel \wedge run = 0$ suggest that the important location where the invariance must be checked is after executing $Update_2$, as prior to this we have $sel \wedge run = 0 \equiv false$. It also supports the validity of I when considering the action $Selection_2$, where $sel \wedge run = 0 \equiv false$. The term $(Vol_L^{start} = Vol_L) \wedge (Vol_R^{start} = Vol_R)$ similarly ensures that there has been no changes from the user, otherwise it evaluates to $false$ and I holds trivially, but we also know that no actual updates are performed.

Observe that the existence of several points in the execution of $\mathcal{A}_2$ where I holds trivially is not a weakness of the model. These points are considered either after the execution of internal actions of the composing modules, and, therefore, they should not be concerned with any global properties of the system, or they are considered after set-up procedures (action $Selection_2$) where also the actual output of the system is not affected. The situation where I holds trivially after the execution of $Update_1$ is when $(Vol_L^{start} = Vol_L) \wedge (Vol_R^{start} = Vol_R) \equiv false$. However, in this case again, there are no actual updates on the output variables, as the system must re-execute certain actions.

In the context of the original system $\mathcal{A}$, a relation similar to I:

$$I_1 \ \widehat{=} \quad sel \wedge run = 0 \Rightarrow (Y_L = Vol_L \times in_L) \wedge (Y_R = Vol_R \times in_R) \tag{3}$$

would not hold as an invariant, due to possible changes of volume values during the system execution. Hence, the changes performed in the previous section help offer the possibility of a correct model with respect to requirements, as also supported by the invariant I.

4.5. Disabling conditions

Let us further develop the analysis on the system interaction and focus on additional characteristics of the $\mathcal{U}ser||\mathcal{A}$ composition.

Notice that all the modules of the system $\mathcal{A}$, that is, the signal source $\mathcal{S}$, the right channel filter $\mathcal{R}$, etc, are *always enabled* systems ($gg_S = gg_R = \ldots \equiv true$). Consider next that we detail the corresponding models by introducing on / off mechanisms that are capable of enabling / disabling the systems, respectively. The user will also be given the possibility to operate these mechanisms.

In the following descriptions we use the boolean variables: S_{on}, to model the "system on" signal, and the variables R_{on}, L_{on}, that enable or disable the right or left filters, respectively. In the volume controller we identify the boolean $filter_{on}$ with either R_{on} or L_{on}. The respective system models become

$\mathcal{S}^1(X : T \,;\, S_{on} : \mathsf{Bool})$
begin $\bullet\ X := x_0 \,;\, S_{on} := false;$
  do $S_{on} \rightarrow X := X'.(X' \in T)$ od
end

$\mathcal{R}^1(X, Z[0..N-2], in_R : T \,;\, R_{on}, S_{on} : \mathsf{Bool})$
begin var $h_R[0..N-1] : T \quad \bullet R_{on}, S_{on} := false \,;\ldots;$
  do $R_{on} \wedge S_{on} \rightarrow in_R := \sum_{k=1}^{N-1} h_R(k) \times Z(k-1) + h(0) \times X$ od
end

$\mathcal{L}^1(X, Z[0..N-2], in_L : T \,;\, L_{on}, S_{on} : \mathsf{Bool})$
begin
...
end

$\mathcal{V}^1(Vol : integer\ 0..10 \,;\, in, out : T_D \,;\, filter_{on}, S_{on} : \mathsf{Bool})$
  begin $\bullet\, filter_{on}, S_{on} := false \,;\ldots;$
    do $filter_{on} \wedge S_{on} \rightarrow out := Vol \times in$ od
  end

$\mathcal{U}ser^1(Vol_L, Vol_R : integer\ 0..Vol_{Max} \,;\, in, out : T_D \,;\, R_{on}, L_{on}, S_{on} : \mathsf{Bool})$
  begin var $\Delta_V : integer \quad \bullet R_{on}, L_{on}, S_{on} := false \,;\ldots;$
    do
      $Vol_L := V'_L.Q_L \parallel Vol_R := V'_R.Q_R$

$$\begin{array}{l} \quad [\!]\ S_{on} := false\ [\!]\ S_{on} := true \\ \quad [\!]\ R_{on} := false\ [\!]\ R_{on} := true \\ \quad [\!]\ L_{on} := false\ [\!]\ L_{on} := true \end{array}$$

od

end

Observe that, given the lack of correlation between the activities of the user and those of the system $\mathcal{A}$, it is possible that the user sets, for instance, $S_{on} := false$ during an execution cycle of the synchronized system. Identifying that $gg_A = S_{on}$, this will force the audio system $\mathcal{A}_3 = \mathcal{S}^1 \sharp \mathcal{R}^1 \sharp \ldots$ to not execute the *Update* action, as, possibly, $sel \equiv false$. This, however, will not mean a termination of the composition activities, as the $User$ system is still enabled.

The consequence of the above scenario is that, whenever the user will re-start the audio system ($S_{on} := true$), the synchronized composition will start from the state where it was disabled at the previous execution. The $Selection$ action will pick for execution the modules k for which the respective $sel[k]$ was left $false$ before the system was disabled.

The designer has the option of alleviating the effects of such behavior, or not. Depending on the system application domain, in general, preserving the sate when shut-down has occurred may be a desirable feature.

However, if such a state saving is not desired, one may appeal again to the watched variable mechanism. Thus, if one treats S_{on} as a watched variable, we may refine the action $Update_1$ of $\mathcal{A}_2$ into $Update_2$, where, in addition to the volume processing check, we will also deal with the "shut-off" procedure, by clearing the present selection state:

$$\begin{array}{rcl} Update_2 & = & \neg S_{on} \wedge (sel[1] \vee \ldots \vee sel[6]) \rightarrow sel := false \\ & & //\ (Vol_R^{start} \neq Vol_R \rightarrow sel[1] := false\ [\!]\ Vol_L^{start} \neq Vol_L \rightarrow sel[2] := false) \\ & & //\ (Update\ ; proc := true) \end{array}$$

Further, notice that the new added action, $\neg S_{on} \wedge (sel[1] \vee \ldots \vee sel[6]) \rightarrow sel := false$, does not update global variables and it self-disables. Therefore it behaves like skip (requirement 3 of Lemma 2.1) and its execution cannot continue forever (requirement 5 of Lemma 2.1). Thus, as the fourth point of the lemma is also easy to asses, we then have a system level refinement $\mathcal{A}_2 \sqsubseteq \mathcal{A}_3$, where $\mathcal{A}_3 = \mathcal{A}_2[Update_2/Update_1]$.

The invariant that one has to check now must also consider the S_{on} signal, as we are only interested to assess the property whenever the system is on:

$$\begin{array}{rl} I_2\ \widehat{=} & S_{on} \wedge sel \wedge run = 0 \wedge (Vol_L^{start} = Vol_L) \wedge (Vol_R^{start} = Vol_R) \\ & \Rightarrow (Y_L = Vol_L \times in_L) \wedge (Y_R = Vol_R \times in_R) \end{array}$$

5. Discussion

We have characterized, in the previous sections, the interaction between two models of execution: the interleaved and the synchronized models. Our example provided insights of the communication between a system with a completely non-deterministic behavior and a synchronized system with an external deterministic behavior.

Firstly, we assumed that the unsynchronized systems must behave in a fair manner with respect to the provided updates. This is not an assumption uniquely considered within the focus of the present study. On the contrary, it is an usual supposition when designing "regular" action systems.

Secondly, with the use of the watched variables we showed that commands issued at random stages of the synchronized execution model can be successfully interpreted and satisfied. Here, the benefit of our approach is reflected by the correct-by-construction result, as we have $\mathcal{A} \sqsubseteq \mathcal{A}_1 \sqsubseteq \mathcal{A}_2$. This is easily derived by checking the requirements of Lemma 2.1, and is consistent with the separation of concerns from a module-designer / system-integrator perspective. The selection of watched variables and the reaction to new values updated while the synchronized composition is executing fall in the responsibility of the system–level integrator, therefore they do not characterize the imported modules. Furthermore, as the refinement steps are not performed under a specific invariant, also the global system $\mathcal{U}ser \parallel \mathcal{A}_2$ is a refinement of the initial representation, $\mathcal{U}ser \parallel \mathcal{A}$ (we only add local variables and the corresponding actions that update them). Notice, though, that I (relation (2)) is only an invariant of the system $\mathcal{A}_2$. It is not an invariant of the composition $\mathcal{U}ser \parallel \mathcal{A}_2$, and the mentioned refinement is not done under I.

A thorough analysis of parallel system execution in the presence of invariants is given by Back and von Wright [8]. Each component must "respect" the invariants of other composition components. If such an environment can be constructed, even individual refinement can hold at the composition level. In our case, if it was desired, or possible to restrict the user actions, then the predicates I_2 (relation (3)) could actually be an invariant of the $\mathcal{U}ser \parallel \mathcal{A}$ composition. However, in our presented case, and to our best thinking, also in similar example situations, such restrictions are not desired or not even possible.

We could have employed, with apparently same results, a prioritized composition between the participating systems: $\mathcal{U}ser$ // $\mathcal{A}$. The disadvantage of this model is twofold. First, the $\mathcal{U}ser$ is an always enabled system, as both of its actions are. Therefore, no action of $\mathcal{A}$ could *ever* get executed. The solution would be to model a self-disabling mechanism for the $\mathcal{U}ser$, such that, when it is disabled, $\mathcal{A}$ has the chance of updating, in its turn, the composition variables. This approach would resemble those that adopt blocking procedures, as presented, for instance, in [21, 23]. However, we do not want to apply such restrictions to a naturally independent system, as the one modeled by $\mathcal{U}ser$. This takes us further to the second reason that prevents us from using a prioritized composition: reusability. Even though this is not within the scope of the present study, we may mention that such a solution will only fit the current design and would not allow the same system to be reused in a slightly different situation [25] - for instance if considering extending the audio system to process signals on four channels.

In [16], the authors resorted to a similar safe point check, to establish if (certain) permissions, to run code in loops, are revoked. Still, even though the concept is very similar, these check-ups are performed at the *beginning*, rather than at the end of one loop execution. In our case, we do not know "when" the $\mathcal{U}ser$ system will modify the volume. Hence, we have to give to the audio system a chance to execute. This motivates our "last line" evaluation of the watched variables.

Our refinement-based approach to communication comes quite close to the principles described in [12], regarding the *watched statements* (variables here). However, we may not call our methods *interrupts*. This is because the classical execution method for interrupts assumes storing the current system status, executing an interrupt handling procedure and then restoring the saved environment and resuming the interrupted activity. At least the last two steps are not subsumed by our development process.

The synchronized composition preserves *properness* [25]. This is an important factor towards analyzing possible deadlock situations. By requiring that the composing modules are proper, that is, that, eventually, their global state *must* change, and due to the above mentioned property, the system level

designer is relieved of considering deadlock as one of the problems. Still, this does not ensure that the operation of a certain module is not stalled, that is, the module provides an infinite chain of similar output values for various input ones. In this case, again, the problem lies at the module designer level.

An additional design issue that one has to consider, even after the introduction of a specific set of watched variables, is the relative fairness in selecting for execution the actions of $\mathcal{U}ser$ or $\mathcal{A}_2$ in the example audio system. It is easy to imagine that, in case the user selects very often to change the volume, the system $\mathcal{A}_2$ may never reach the point where it actually updates the output values of the volume. The execution may cycle from $Selection_2$ to $Update_1$ through the component's actions, without reaching the end of an execution cycle.

A digital filter, as the one that we have used in our example, may be implemented either as a software or as a hardware unit. We may compare our model, from the hardware point of view, with a VHDL [1] specification. In VHDL, we also witness a two-level execution perspective. Firstly, inside the processes, the execution follows a sequential path. Secondly, processes, viewed as parallel running entities, are selected in a non-deterministic manner. Distinctly from our approach, the "global variables" - signals in VHDL, are properly updated after the execution of the respective process. Hence, in a VHDL simulation cycle, it is often the case that processes that have already executed, become re-activated, and therefore re-executed. This is due to the fact that other processes may change the values of the read values of the already executed ones, therefore scheduling these for a re-execution. This activity settles down, eventually, as, in an implementable description, the system has to stabilize. Notice that the same symptoms characterize our synchronized environment, too. However, we may not assume any stabilization moment; therefore, the synchronized updates happen once, at the end of the execution cycle.

Furthermore, the watched variables are "local" to the synchronized composition, but "global" from the component's perspective. They are actually accessible to any module in the composition. This brings us close to the concept of shared variables also possible to be employed in VHDL. Nonetheless, as described by Ashenden and Wilsey [2], the utilization of shared variables in VHDL raises problems with respect to non-deterministic behavior, solved by a more restricted mechanism, the protected types. Our watched variables may not raise such issues, as the module designer is not aware of the future points of interest required by a specific employment of the module in a synchronized composition. Therefore, it may not take advantage of the possible existence of such variables at the upper levels.

A third topic of our study analyzed the impact of un-synchronized commands acting on the enabling conditions for the synchronized environments. The result is that the synchronized system does not change the external state, but the internal state may not correspond to the one at the beginning of the execution cycle. A combination of the solutions described in sections 4.3 and 4.5 provide a refinement-based improvement of the initial audio system, "protected" now to both volume change or shut-down commands that are issued *during* an execution cycle. Thus, we do not have to impose constraints on the unsynchronized systems that communicate with synchronized ones.

Deterministic synchronous models do not experience similar problems for two reasons. First, they have the notion of clock cycles that represent a global time. Clock cycles are totally ordered on the time axis. Two events occurring in different clock cycles, occur unambiguously at different time instances, one earlier than the other. Events occurring in the same clock cycle occur indistinguishable at the same time instance. Second, these models are fully deterministic and no particular execution order of processes will lead to an overall different system behavior. Esterel and ForSyDe are two examples following this paradigm.

In Esterel [10], a fully *synchronous* language, exceptions are modeled by *traps*. Checking for trap conditions is done in parallel with the execution of other functional blocks. These conditions correspond to the comparison of start-of–cycle and end-of–cycle values of the watched variables. Global variables are updated only at the end of an execution cycle, therefore we can allow the detection of new events on the watched variables to be detected at the end of the cycle. On the other hand, traps may be inserted "anywhere", while our "check-points" are fixed.

In ForSyDe [24] and in the synchronous model of computation described in [19] all input and output events are deterministically synchronized. All user inputs received in the same *evaluation cycle* (or *clock cycle*) are processed and the corresponding outputs are generated during that cycle. Since the clock cycles are globally ordered and synchronized between all signals in the system, two user events (controlling the left and right volume, respectively) either occur in the same cycle or one event occurs in an earlier cycle than the other. In both cases the filter process will properly react to the inputs received in a particular cycle.

Berry et al. [11] studied mixed synchronous and asynchronous design from an Esterel and CSP [17] perspective. The result was an "import" of CSP-like communication into Esterel, with the help of on additional commands, exec and rendezvous. Therefore, the communication is seen from the perspective of the synchronous environment, Esterel, and the actual non-deterministic behavior of the asynchronous design parts is not analyzed.

Our study demonstrates a technique to obtain synchronized reactive behavior similar to perfectly synchronous models such as Esterel and ForSyDe in the framework of action systems. The advantage here comes in the non-restricted model of the environment, that may behave, related to the synchronized systems in a non-deterministic manner. While here we identified such environment with a human, it is not rarely, nowadays, that multiple (same chip or distributed) devices operate with a globally non-synchronized behavior. Instead of the user, sensor information or processing requirements with immediate deadlines can be considered. Memory accesses in a distributed processor system is also another point where the analysis of read / write operations is of utmost importance.

6. Conclusions

In this study, we addressed the problem of devising a correct communication procedure between non-deterministic environments and synchronized reactive modules. The respective procedures emerge as (successive) refinements of the initial synchronized composition, thus offering correct-by-construction results. The final system model preserves the behavioral characteristics of the initial one, while allowing un-synchronized, but important events to be intercepted and processed. The selection of these events is done by the system integrator, leaving the module designer the freedom to concentrate only on the functionality of the modules, rather than on the communication schemes.

Acknowledgements.
The authors wish to thank Cristina Cerschi Seceleanu for improvement of the paper content, through helpful comments. The remarks of the anonymous reviewers are also gratefully acknowledged.

References

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

[2] Protected Shared Variables in VHDL: IEEE Standard 1076a. IEEE Design & Test, Volume 16, Issue 4, pp: 74 - 83, 1999.

[3] 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.

[4] R. J. R. Back, R. Kurki-Suonio. Distributed Cooperation with Action Systems. ACM Transactions on Programming Languages and Systems, Vol. 10, No. 4, pp. 513-554, 1988.

[5] 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.

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

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

[8] R. J. R. Back and J. von Wright. Compositional Action System Refinement. Formal Aspects of Computing (2-3): 103-117 2003.

[9] G. Berry. Real-time programming: special purpose or general purpose languages. Proceedings of the IFIP Congress, North-Holland, Amsterdam, 1989.

[10] G. Berry. The Foundations of Esterel. Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, eds., MIT Press, 1998.

[11] G. Berry, S. Ramesh, R.K. Shyamasundar. Communicating Reactive Processes. Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Charleston, South Carolina, United States, 1993, pp: 85 - 98.

[12] J. Borkowski. Interrupt and Cancellation as Synchronization Methods. R. Wyrzykowski et al. (Eds.): PPAM 2001, LNCS 2328, pp. 3-9.

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

[14] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous dataflow programming language Lustre. Proceedings of the IEEE, 79(9):1305–1320, September 1991.

[15] S.J. Hartley. Concurrent Programming Using Java. Oxford University Press, 1998.

[16] C. Hawblitzel, T. von Eicken. Luna: a Flexible Java Protection System. USENIX Association: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, 2002.

[17] C.A.Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

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

[19] A. Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Systems on Silicon. Morgan Kaufmann Publishers, June 2003.

[20] R.M. Keller. *Towards a theory of speed independent modules.* IEEE Transactions on Computers, C-23(1):2133, Jan. 1974.

[21] S. Marlow at al. Asynchronous Exceptions in Haskell. Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, 2001, pp. 274-285.

[22] P. Raymond, D. Weber, X. Nicollin, N. Halbwachs. Automatic testing of reactive systems. In 19th IEEE RealTime Systems Symposium, Madrid, Spain, December 1998.

[23] A. Rudys, D. S. Wallach. Termination in Language-Based Systems. ACM Transactions on Information and System Security Vol. 5, Issue 2, 2002, pp. 138-168.

[24] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004.

[25] C. Cerschi Seceleanu, T. Seceleanu. Synchronization Can Improve Reactive Systems Control and Modularity. Journal of Universal Computer Science (J.UCS), Vol. 10, Nr. 10, 2004, pp. 1429 - 1468.

[26] T. Seceleanu, D. Garlan. Developing Adaptive Systems with Synchronized Architectures. The Journal of Systems and Software 79 (2006) 15141526.

[27] E. Sekerinski, K. Sere. A Theory of Prioritized Composition. The Computer Journal, VOL. 39, No 8, pp. 701–712. University Press.