Communicating 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

In the modern design environments, different modules, available in existent libraries, may obey different architectural styles and execution models. Reaching a well–behaved composition of such modules is a very important task of the system designer. In the framework of the action systems formalism, we analyze the co-existence of two models of execution, one synchronized, the other, interleaved. We devise a communication scheme, similar to the classical paradigm of polling, which allows us to model synchronized components that correctly exchange information, within the borders of a global system, with their non-synchronized partners. Derivations of such mechanisms follow specific correctness rules for 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

1 Introduction

The design of a reactive system 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 [7] separated computing systems into *interactive* and *reactive* classes. 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 [16], 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 environment of the devised reactive system does not allow a synchronized model, either because this will impose unrealistic restrictions on the model, or because the environment is highly unpredictable and therefore, the non-deterministic model covers its behavior very well. However, a simple, parallel composition of a synchronized system and its environment is not sufficient, as it may render the benefits of the synchronized model useless, due to the intrinsic non-determinism. Therefore, in such situations, communication between the environment and the synchronized system modules must follow specific rules, such that the functional goals of the design are met.

In the present study, we focus on the analysis of such rules, for which 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 [10, 13], 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, asynchronized events, are modeled here as *watched variables*.

Related work. When discussing communication between interacting systems, we usually think of resource sharing, critical regions, blocking procedures. There is a rich literature on these topics, especially coming from the Java community. However, there are several distinctions between our

approach and others described in the literature.

Early Java systems suffered from unexpected interactions between data, threads, and code. Hawblitzel and von Eicken [10], 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.

Modularity and exception handling are subjects of the study presented by Marlow at al. [13]. 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.

Rudys and Wallach [14] 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.

Seceleanu and Garlan [17] 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. The focus is only on the processing elements, and not on the user-system interaction, which is considered somehow regulated.

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 composed with the filter, by employing the 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. Section 4.4 discusses the implications of our approach, compares and contrasts it with alternative approaches. Finally, section 5 concludes the study and points to future work.

2 Action Systems

Back and Kurki-Suonio [3] 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:

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

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

The rest of the system (the *environment*) communicates with the action system via *shared* variables. In the following, we assume the following notations: the set of state variables accessed by some action A 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 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 $wA \subseteq lw\mathcal{A}$.

A statement S_i is defined by the following grammar:

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

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.

Statements in the action systems language are defined using *weakest precondition semantics*, consistent with Dijkstra's original semantics for the language of guarded commands [9]. 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 [6]. In this paper, we assume that all statements are *conjunctive monotonic predicate transformers*, that is, $\forall p, q \ \bullet \mathrm{wp}(S, (p \wedge q)) = \mathrm{wp}(S, p) \wedge \mathrm{wp}(S, q)$ (conjunctivity) and $\forall p, q \ \bullet (p \Rightarrow q) \Rightarrow$

$\mathrm{wp}(S,p) \Rightarrow \mathrm{wp}(S,q)$ (monotonicity). The guard of S is defined as $g \triangleq \neg\mathrm{wp}(S, false)$. The guard of an action system given by (1) is $gg_A \triangleq \bigvee_1^n g_k$.

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 [18] 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 $I(v\mathcal{A}) - I$ in short – is an *invariant* of the action system $\mathcal{A}$, given by (1) if: it is established by $Init$, i.e. $true \Rightarrow \mathrm{wp}(Init, I)$, and it is preserved by each action A_i, i.e. $g_i \wedge I \Rightarrow \mathrm{wp}(S_i, I), i = 1, \ldots, n$.

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 [2]. Additionally, let $I(c, z)$ 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

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

Trace Refinement of Action Systems. The semantics of a reactive action system is given in terms of behaviors [5]. 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 [4]:

Lemma 1 *Given the action systems*

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

let $I(c, z_C)$ 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:* $I(c_0, z_{C0}) \equiv true$
2. *Main action:* $A \leq_I C$
3. *Auxiliary action:* $\mathsf{skip} \leq_I X$
4. *Continuation condition:* $I \wedge gA \Rightarrow gC \vee gX$
5. *Internal convergence:* $I \Rightarrow \mathrm{wp}(\ \mathsf{do}\ X\ \mathsf{od}\ , true)$

2.1 Execution of Action Systems

Starting with the original paper by Back and Kurki-Suonio [3], 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 [16], 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 AS 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.

A synchronized environment presents some useful characteristics. The first one is an increased capability of reaction: the input stimuli are received by all the synchronized reactive components and 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 1 *Consider the action system $\mathcal{A}$:*

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

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 $\mathcal{A}$.*
- $\mathrm{wp}(\mathsf{do}\ g_L \rightarrow L\ \mathsf{od}, \neg g_L \wedge g_S) \equiv true$ *– 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 higher level action system, obtained as follows.

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

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

for which we also have that $\forall j,k = 1\ldots n, j \neq k.((gw\mathcal{A}_j \cap gw\mathcal{A}_k = \emptyset) \wedge (\bigcap_k x_k = \emptyset))$. 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:*

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

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

The system $\mathcal{P}$ introduced above represents the "flattened" model of the synchronized composition of $\mathcal{A}_1, \ldots, \mathcal{A}_n$. The guard gg_P is the disjunction of all the module guards: $gg_P \stackrel{\triangle}{=} gg_{A_1} \vee \ldots \vee gg_{A_n}$. We also 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 $false$. The action *Update* represents the integrator's choice of deciding how the actual updates of the global variables are performed. In an initial set-up, *Update* is specified as an atomic sequence:

$$Update \stackrel{\triangle}{=} wS_1 := wS_1c\,; \ldots ; wS_n := wS_nc$$

However, in order to either accommodate concurrent updates on the same variables, or in order to allow different communication situations, the content of this action can be changed by the top-level designer.

In the following, we analyze the interaction between the two presented models of execution. From the synchronized point of view, this will imply a change of the *Update* action.

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. We build on the *digital filter* example presented in [16].

3.1 The Filter

Briefly, a digital filter [11] is a device that takes as input a sequence of samples, performs certain operations on it and delivers as output a corresponding sequence of samples. The incoming sequence is described as $x(n)$, where x is the name of the input signal and n identifies the sample position; a similar notation applies to the output signal y, for which we have the samples $y(n)$. The relation between the input and output is given by $y(n) = \sum_{k=0}^{N-1} h(k) \times x(n-k)$, where the vector $h[0..N-1]$ contains the filter *coeffi cients*. 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.

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 implementation of the filter. In the following, we model the signal source by system $\mathcal{S}$, the buffer by system $\mathcal{B}$ and system $\mathcal{F}$ performs the filtering, as illustrated in Fig. 1 a).

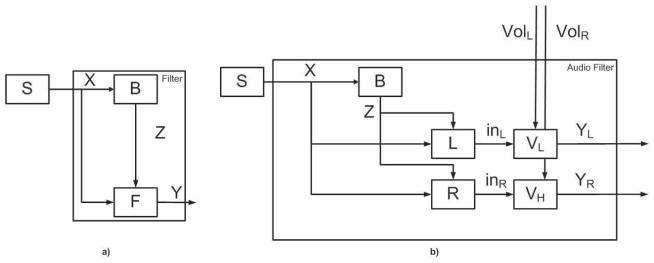

Figure 1. Simple filter representation.

For reasons given in [16], such as modularity, simplification of communication protocols, etc., a "good" (in terms of reaction and modularity) modeling of the filter device and environment is achieved when the participating modules are organized in a synchronized cooperation. The action systems descriptions of these modules are given as:

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

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

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

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

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. 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 1 b). The modules R and L are instantiations of the $\mathcal{F}$ system, whereas the modules V_R and V_L perform the desired amplification, and they are instances of the action system V, given as

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

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 at the same moment. Therefore, the audio device is modeled by the action system $\mathcal{A}$, a synchronized cooperation of its modules: $\mathcal{A} \mathrel{\widehat{=}} \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{R} \sharp \mathcal{L} \sharp V_R \sharp V_L$.

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

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

$$
\begin{array}{rcl}
Q_L & \equiv & V_L' = V_L \vee (V_L' = V_L + \Delta_V \wedge\ 0 < V_L' \leq Vol_{Max}) \\
 & & \quad \vee (V_L' = V_L - \Delta_V \wedge\ 0 \leq V_L' < Vol_{Max}) \\
Q_R & \equiv & V_R' = V_r \vee (V_R' = V_R + \Delta_V \wedge\ 0 < V_R' \leq Vol_{Max}) \\
 & & \quad \vee (V_R' = V_R - \Delta_V \wedge\ 0 \leq V_R' < Vol_{Max})
\end{array}
$$

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.

4.1 Modeling the audio system

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

$\mathcal{A}(V_R, V_L : integer\ 0..Vol_{Max}\ ; Y_R, Y_L : T_D)$
begin
  var $X, X_c, Z[0..N-2], Z_c[0..N-2], Y_{Rc}, Y_{Lc},$
    $in_R, in_L, in_{Rc}, in_{Lc} : T_D\ ; sel[1..6] : \mathsf{Bool};$
    $run : integer\ 0..6\ \bullet$
    $V_R, V_L := Vol_{Max}/2;$
    $in_R, in_L, in_{Rc}, in_{Lc} := in_0;$
    $X, X_c := X_0\ ; Z, Z_c := Z_0;$
    $Y_R, Y_L, Y_{Rc}, Y_{Lc} := Y_0\ ; run := 0\ ; sel := false;$
  do
    $Selection$
  $[\!]\ run = 1 \rightarrow Y_{Rc} := V_R \times in_R\ ; run := 0$
  $[\!]\ run = 2 \rightarrow Y_{Lc} := V_L \times in_L\ ; run := 0$
  $[\!]\ run = 3 \rightarrow X_c := X'.(X' \in T)\ ; run := 0$
  $[\!]\ run = 4 \rightarrow$
    $Z_c[0], \ldots, Z_c[N-2] := X, \ldots, Z[N-3];$
    $run := 0$
  $[\!]\ run = 5 \rightarrow in_{Rc} := \sum_{k=1}^{N-1} h_R(k) \times Z(k-1) +$
    $h(0) \times X\ ; run := 0$
  $[\!]\ run = 6 \rightarrow in_{Lc} := \sum_{k=1}^{N-1} h_L(k) \times Z(k-1) +$
    $h(0) \times X\ ; run := 0$
  $[\!]\ sel \wedge run = 0 \rightarrow Update\ ; sel := false$
  od
end,
$Selection = run = 0 \wedge \neg sel[1] \rightarrow$
    $run := 1\ ; sel[1] := true$
  $[\!]\ \ldots$
  $[\!]\ run = 0 \wedge \neg sel[6] \rightarrow$
    $run := 6\ ; sel[6] := true$
$Update = X := X_c\ ; Z := Z_c\ ; \ldots$

System execution. 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 $run = 2 \rightarrow Y_{Lc} := V_L \times in_L\ ; run := 0$, 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: $V_L := V'_L.Q_L$. However, this option cannot be followed by a reaction of $\mathcal{A}$, in this execution cycle, as the intermediate update on Y_{Lc} 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.

One possible illustration of the above described scenario is described in statecharts-like representation of Figure 2, 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 V_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.

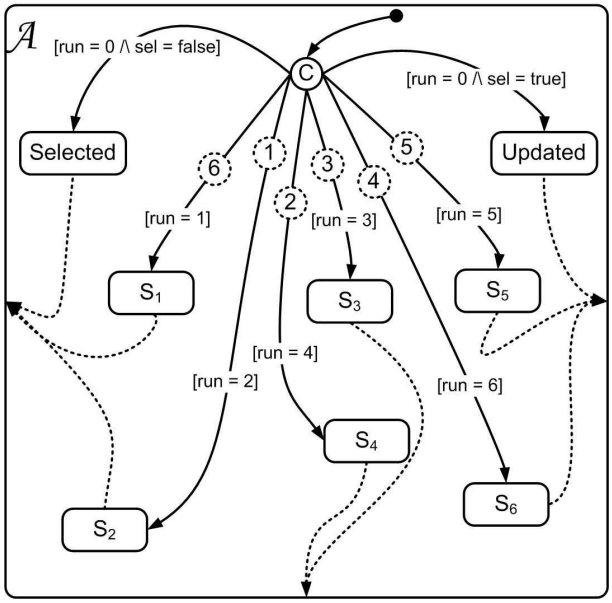

Figure 2. Execution of the system $\mathcal{A}$.

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 $Y_{Rc} := V_R \times in_R\ ; run := 0$, the module $\mathcal{U}ser$ is chosen, and action $V_R := V'_R.Q_R$ 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 the cooperation between 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.

$$\begin{aligned} & Selection_1 \\ = \quad & run = 0 \wedge \neg sel[1] \rightarrow sel[1] := true \,;\, run := 1 \\ & [\!] \ \ldots \\ & [\!] \ run = 0 \wedge \neg sel[6] \rightarrow sel[6] := true \,;\, run := 6 \\ & [\!] \ proc \rightarrow V_L^{start} := V_L \,;\, V_R^{start} := V_R \,;\, proc := false \end{aligned}$$

The new selection action stores the values of the watched variables V_L and V_R into the local variables V_L^{start} and V_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 [18]. Thus, we obtain the action $Selection_2$:

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

It is easy to check that the above described refinement steps lead to a trace refinement of the original system, as specified by Lemma 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.
Consequently, we have proved the trace refinement $\mathcal{A} \sqsubseteq \mathcal{A}_1$. The system $\mathcal{A}_1$ is obtained from the original system $\mathcal{A}$, by introducing the new local variable $proc$, and replacing the action $Selection$ with $Selection_2$.

$\mathcal{A}_1(V_R, V_L : integer\ 0..Vol_{Max} \,;\, Y_R, Y_L : T_D)$
begin
  var $X, X_c, Z[0..N-2], Z_c[0..N-2], Y_{Rc}, Y_{Lc},$
    $in_R, in_L, in_{Rc}, in_{Lc} : T_D \,;\, proc, sel[1..6] :$ Bool;
    $run : integer\ 0..6$ •
    $V_R, V_L := Vol_{Max}/2;$
    $in_R, in_L, in_{Rc}, in_{Lc} := in_0;$
    $X, X_c := X_0 \,;\, Z, Z_c := Z_0;$
    $Y_R, Y_L, Y_{Rc}, Y_{Lc} := Y_0;$
    $run := 0 \,;\, proc, sel := false;$
  do
      $Selection_2$
    $[\!] \ run = 1 \rightarrow Y_{Rc} := V_R \times in_R \,;\, run := 0$
    $[\!] \ \ldots$
    $[\!] \ sel \wedge run = 0 \rightarrow Update \,;\, sel := false$
  od
end

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:

$$\begin{aligned} Update_1 \quad = \quad & sel \wedge run = 0 \rightarrow \\ & \quad (V_R^{start} \neq V_R \rightarrow sel[1] := false \\ & [\!] \quad V_L^{start} \neq V_L \rightarrow sel[2] := false) \\ & // \, Update \,;\, proc := true \,;\, sel := false \end{aligned}$$

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 V_L or V_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 $Y_{Rc} := V_R \times in_R \,;\, run := 0$ or (and) $Y_{Lc} := V_L \times in_L \,;\, run := 0$, 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 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

$\mathcal{A}_2(V_R, V_L : integer\ 0..Vol_{Max}\ ; Y_R, Y_L : T_D)$
begin
var $X, X_c, Z[0..N-2], Z_c[0..N-2], Y_{Rc}, Y_{Lc},$
$in_R, in_L, in_{Rc}, in_{Lc} : T_D\ ; proc, sel[1..6] :$ Bool;
$run : integer\ 0..6\ ; V_R^{start}, V_L^{start} : integer\ 0..Vol_{Max}$ •
$in_R, in_L, in_{Rc}, in_{Lc} := in_0;$
$V_R, V_L, V_R^{start}, V_L^{start} := Vol_{Max}/2;$
$X, X_c := X_0\ ; Z, Z_c := Z_0\ ; Y_R, Y_L, Y_{Rc}, Y_{Lc} := Y_0;$
$run := 0\ ; proc, sel := false;$
do
$Selection_2$
$[\!]\ run = 1 \rightarrow Y_{Rc} := V_R \times in_R\ ; run := 0$
$[\!]\ run = 2 \rightarrow Y_{Lc} := V_L \times in_L\ ; run := 0$
$[\!]\ run = 3 \rightarrow X_c := X'.(X' \in T)\ ; run := 0$
$[\!]\ run = 4 \rightarrow Z_c[0], \ldots, Z_c[N-2] := X, \ldots, Z[N-3];$
$run := 0$
$[\!]\ run = 5 \rightarrow in_{Rc} := \sum_{k=1}^{N-1} h_R(k) \times Z(k-1) +$
$h(0) \times X\ ; run := 0$
$[\!]\ run = 6 \rightarrow in_{Lc} := \sum_{k=1}^{N-1} h_L(k) \times Z(k-1) +$
$h(0) \times X\ ; run := 0$
$[\!]\ sel \wedge run = 0 \rightarrow$
$((V_R^{start} \neq V_R \rightarrow sel[1] := false$
$[\!]\ V_L^{start} \neq V_L \rightarrow sel[2] := false)$
$// Update\ ; proc := true\ ; sel := false)$
od
end

We recall now the execution scenario described at the beginning of section 3.2. Running now the composition $User \parallel \mathcal{A}_2$ (Figure 3), observe the additional execution of the left-channel update $run = 2 \rightarrow Y_{Lc} := V_L \times in_L\ ; run := 0$ (at number 7).

4.4 Model Analysis and Comparisons

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.

This can be checked by assessing the invariance of the predicate

$$I \ \widehat{=}\ proc \wedge (V_L^{start} = V_L) \wedge (V_R^{start} = V_R) \Rightarrow (Y_L = V_L \times in_L) \wedge (Y_R = V_R \times in_R) \quad (2)$$

over the actions of $\mathcal{A}_2$. This is a simple task, as the invariance must only be checked in case of the action $Update_1$, following the updates on the global variables. The predicate I holds trivially for $Selection_2$ and the other actions do not write variables accessed by I.

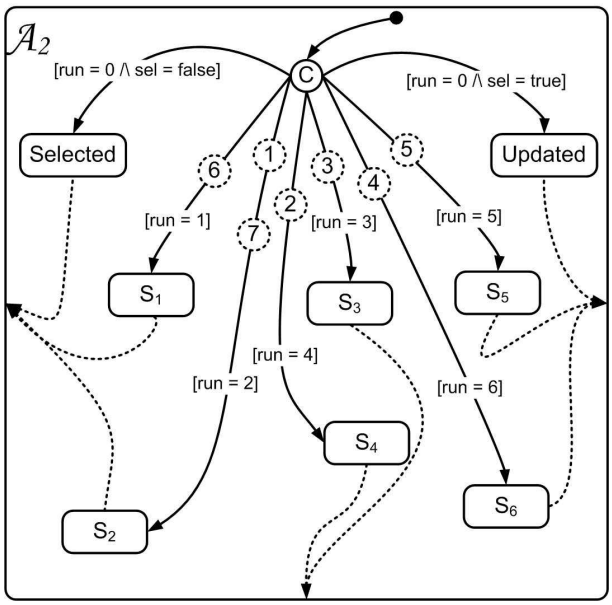

Figure 3. Execution of the system $\mathcal{A}_2$.

Moreover, the benefit of the 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 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 performed 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.

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 [13, 14]. 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 [16].

In [10], 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 [8], regarding the *watched statements* (variables in our case). 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.

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 reactivated, 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.

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 that paradigm.

In Esterel [7], 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, in our approach. 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.

In ForSyDe [15] and in the synchronous model of computation described in [12] 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.

Our study demonstrates a technique to obtain synchronized reactive behavior similar to perfectly synchronous models such as Esterel and ForSyDe in the general framework of action systems. Thus one is allowed to mix non-deterministic, non-synchronized modules with synchronized, reactive modules.

Observe that, at the same time, the improved modularity property offered by the utilization of the synchronized composition is preserved after the changes performed on the selection and update actions. Hence, a more performant, or a more accurate filter may replace the current system description, without the higher level, that is, the synchronized composition, to be affected. These improvements stay at the level of the module designer, while the system integrator may only select the appropriate solution.

5 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 improve 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] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

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

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T. Seceleanu, D. Garlan. Developing Adaptive Systems with Synchronized Architectures. To appear. Journal of Software and Systems, 2006.

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