This article was originally published in a journal published by Elsevier, and the attached copy is provided by Elsevier for the author's benefit and for the benefit of the author's institution, for non-commercial research and educational use including without limitation use in instruction at your institution, sending it to specific colleagues that you know, and providing a copy to your institution's administrator.

All other uses, reproduction and distribution, including without limitation commercial reprints, selling or licensing copies or access, or posting on open internet sites, your personal or institution's website or repository, are prohibited. For exceptions, permission may be sought for such use through Elsevier's permissions site at:

http://www.elsevier.com/locate/permissionusematerial
Developing adaptive systems with synchronized architectures

Tiberiu Seceleanu a,*, David Garlan b

a Department of Information Technology, University of Turku, Finland
b School of Computer Science, Carnegie Mellon University, USA

Available online 24 May 2006

Abstract

In this paper we show how to specify and refine self-adapting systems, by employing the state-based formalism called action systems. Assuming a fixed set of possible configurations, we use a recently-introduced synchronization operator to achieve separation of concerns between adaptation policy, fault tolerance and steady-state system execution. Using action systems allows us to apply standard refinement techniques, aimed for correct implementations of higher-level models. We illustrate this idea by applying it to the problem of coping with dynamically-varying user requirements and possible fault situations.

Keywords: Synchronized communication; Adaptive systems; Formal methods; Refinement

1. Introduction

A cover-term for certain theories and methods used in present-day system development is known as cybernetics, which can be traced back to Wiener (1948). Cybernetics was thought of as a concept that places together the individual notions of information, feedback, and regulation, and extends their application from specific engineering domains to systems in general, including living organisms, abstract intelligent processes, languages, etc.

It is, however, recent, that the rich interaction of goals, predictions, actions, feedback, and response in systems of all kinds, usually crowded within the (virtual) limits of a single package, force the modern day designer to think of intricate solutions for the system development process. The latest advancements in computing technology pushed the complexity of computing environments towards design-acceptable limits. Issues like component-level faults, variability in resources and processes, changes of user needs, etc, must be promptly satisfied at run time, following complex design procedures. Moreover, systems are expected to reduce their administrative overheads, allowing for smooth operation with minimal human supervision.

As an answer to such needs, a new branch of research has arisen. Extending the classical areas of fault-tolerance and high-dependability systems, the emerging field of adaptive, or autonomic systems deals with properties such as self-organization (Wermelinger, 1998), self-healing (Shaw, 2002), self-optimization (Ganak and Corbi, 2003), self-management (Cheng et al., 2004), among others. Systems that implement such characteristics must posses capabilities for discovering, diagnosing, and repairing faults, as well as adapting themselves in order to improve their overall operating behavior, in response to varying environments and user requirements.

In this paper, we propose an approach to the formal modeling of adaptive systems, using action systems (Back and Kurki-Suonio, 1983). Traditionally, the latter is a formalism intended for the rigorous development of parallel programs,
within the refinement calculus (Back and von Wright, 1998). A key feature of our approach is the explicit separation of the control aspects of the system, and the modularization of component behavior. To achieve this, we adopt a recently introduced virtual execution model for action systems, namely the synchronized environment (Cerschi Seceleanu and Seceleanu, 2004). Under the assumption that the system components satisfy certain reasonable constraints, our model permits the system-level integrator to freely use and reuse those components with different adaptation policies. In the following sections, we show how the synchronized environment can be exploited towards addressing problems of adaptability in serving computerized systems. The main construction paradigm that we use is refinement.

Related work. While the cybernetic approach to computing seems to have been embraced by IBM’s “autonomic computing” concept (Ganak and Corbi, 2003; Kephart and Chess, 2003), implementing the idea is not an easy task.

Cai et al. (2004) collect several issues that the software designer is confronted with, such as the nature of software behavior, the formalization of software development process or dependability of software systems. The word “software” in the previous sentence may be replaced, without fear of overdoing, with “hardware” or, in a more general view, even with “system”. Along the lines identified by Cai et al. (2004), our systems are composed of a control-selection part, plus (modularized) functional part. Adaptive or not, feedback control is observed at the end of execution cycles in synchronized environments.

Recent research has investigated several ways of reasoning about evolving systems. For instance, a chemical model is developed by Inverardi and Wolf (1995) for the analysis and elaboration of software architectures in the CHAM formal framework. The inspiration comes from the chemical reactions; computations are interpreted as sequences of reactions between data elements, viewed as molecules. CHAM, may be used to uncover architectural mismatches in component behaviors, leading further to architecture reconfiguration (Wermelinger, 1998). In our approach, this reconfiguration is implicitly obtained, as we will show later in the paper.

Ideally, one would like to have compositional ways of reasoning about such systems and refining their formal models into implementable ones. Such compositions should allow one to separate concerns of dynamic adaptation policy and control, from other “steady-state” behavior. Allen et al. (1998) use a distinct virtual link to connect system components—connectors—rather than using direct connections. This is useful when emulating dynamic architectures within static frameworks, as the goal is to preserve the overall system behavior in the presence of faults. While sharing with Allen et al. (1998) some of the background modeling methods (CSP vs. action systems), and some of the goals, we localize the reconfiguration issues at the level of modules, rather than at the interfaces.

Similar to the targets of Poladian et al. (2004), our intention is to reduce user distraction with the problem of configuration and reconfiguration of computing environments. The system-level designer, in our view, may employ system modules without knowing their detailed internal architecture, even if this may change during system operation. Despite the fact that our approach is not algorithmical, we address similar issues to the ones presented by Poladian et al. (2004), regarding user–server interaction. Hence, we combine here “capability space” with “user preferences”, that is, finding solutions to user’s functional and qualitative specifications, and also with “resource space”, that is, finding computing resources to answer (in some way) to the user’s requests. However, due to the high level representation, we avoid going into detailed specification of the systems that we examine.

Overview of the paper. The rest of the paper is organized as follows. First, the action systems formalism is briefly introduced in Section 2, together with refinement procedures and with a short presentation of the synchronized execution model. In Section 3, we analyze the modeling and construction of adaptive systems, based on a synchronized perspective of their operations. Both resource discovery and fault response policies are addressed. We continue in Section 4 with a hierarchical view on system design, within the parameters illustrated in earlier sections. We conclude the paper in Section 5, with some vision on future prospects.

2. Action systems

Action systems, originally developed by Back and Kurki-Suonio (1983, 1988), provide a formal framework for specifying and refining concurrent programs. The language has evolved since then, a generous related literature being available. An action system is defined as a collection of actions (guarded commands), executed one at a time. Multiple action systems communicate via global variables.

A generic action system is described using the following syntax:

\[
\text{SYS}(z : T_z, \ldots) \equiv \begin{array}{l}
\text{begin var } x : T_x, \ldots; \text{ Init: do } A_1 \ldots [A_n \text{ od end}} \end{array}
\] (1)

Above, the action system SYS declares a set of local variables, among which x (of a specific type \(T_x\)), followed by an initialization statement Init and a set of actions \(A_1, \ldots, A_n\). Variables z (of type \(T_z\)) are part of the global variables of SYS.

The initialization statement assigns starting values to global and local variables. After that, the enabled actions within the do–od loop are repeatedly chosen and executed. In this paper, we limit ourselves to actions \(A_i\) of the form \(g_i \rightarrow S_i\). Such an action is enabled, when the boolean condition \(g_i\) evaluates to true. In that case, the action body \(S_i\) is executed. When
multiple actions are enabled, one of them is non-deterministically chosen for execution. SYS is executed as long as one of its actions is enabled. The exit condition is $\neg(\forall X\exists S)\phi_1$.

In the following, we assume the notations: the set of state variables accessed by some action $A$, $vA$, is composed of the read variables, denoted $rA$, and the write variables, denoted $wA$. We build the same sets at the system level, considering the local/global partition of the variables: for a given action system SYS, we have the global read/write variables, $grSYS/gwSYS$ and the local read/write variables, $lrSYS/lwSYS$. We say that an action $A$ of SYS is global, if $gwSYS \cap wA \neq \emptyset$ or local, if $wA \subseteq lwSYS$.

A statement $S_1$ is defined by the following grammar:

$$S_1 ::= \text{skip (stuttering, empty statement)}$$

$$| x := e \quad (\text{multiple assignment})$$

$$| S_m; \ldots; S_n \quad (\text{sequential composition})$$

$$| g_m \Rightarrow S_m \ldots \Rightarrow g_n \Rightarrow S_n \quad (\text{non-deterministic choice})$$

$$| x := x' \cdot Q \quad (\text{non-deterministic assignment})$$

$$| \text{while } g \text{ do } S \text{ od } \quad (\text{loop})$$

Above, $S, S_m, \ldots, S_n$ are statements, $g_m, \ldots, g_n$ are predicates (functions of the form $\Sigma \rightarrow \text{Bool}$, where $\Sigma$ is the polymorphic state), $x$ a variable or a list of variables, and $e$ an expression or a list of expressions or values. If the cardinality $|e| = 1$, while $|x| > 1$, this means that the same expression or value is assigned to all the variables in $x$. The same applies if $x$ is a vector, with the result that all of its elements are set to the same value. In a non-deterministic assignment, $Q$ is the predicate that models the conditions that need to hold of $x$, $x'$. Actions can be much more general, but this simple syntax suffices for the purpose of this paper. A while loop can be reduced to iteration: while $g$ do $S$ od $= \text{do } g \rightarrow S \text{ od } (\text{Back and von Wright, 1998})$.

Weakest pre-condition semantics. Statements in the action systems language can be explained by the semantics of the weakest pre-condition predicate transformer (a function mapping predicates to predicates ($\Sigma \rightarrow \text{Bool}) \rightarrow (\Sigma \rightarrow \text{Bool})$). This semantics is consistent with Dijkstra’s original semantics for the language of guarded commands (Dijkstra, 1976). For statement $S$ and postcondition $q$, the formula $\wp(S, q)$, called the weakest pre-condition 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 $q$ (Back and von Wright, 1998). In this paper, we assume that all statements are conjunctive predicate transformers, that is, $\forall p, q \cdot \wp(S(p \land q)) = \wp(S, p) \land \wp(S, q)$. Also, we assume their monotonicity: $\forall p, q \cdot (p \Rightarrow q) \Rightarrow (\wp(S, p) \Rightarrow \wp(S, q))$.

The guard of the action $A_i = g_i \rightarrow S_i$ is defined as $g_i \triangleq \neg\wp(A_i, false)$. Here, we will consider only strict action bodies, that is, statements for which $\neg\wp(S_i, false) \equiv true$. This means that the guard of the action $A_i$ is, simply, $g_i$.

Notation. In the rest of the paper, for readability purposes, we will use, interchangeably, the notations $g_1 \rightarrow (g_2 \rightarrow S)$, and $g_1 \land g_2 \rightarrow S$, where $g_1$ and $g_2$ are two boolean expressions. Considering $S$ strict, and applying the wp: $\forall q \cdot \wp(g \rightarrow S, q) = \neg g \lor \wp(S, q)$, the equivalence of the notations, follows.

Properties of actions. The interaction between two actions can also be interpreted in terms of weakest precondition computations. Of interest to us in the current context are the enabling and disabling relations between actions. Thus, considering two actions, $A$ and $B$, we say that

$$A \text{ enables } B \equiv \neg gB \Rightarrow \wp(A, gB)$$

$$A \text{ disables } B \equiv gB \Rightarrow \wp(A, \neg gB)$$

Prioritizing composition. One way of expressing preemption in action systems is to use a macro operator based on the semantics of the choice operator. The prioritizing composition of two actions $A$ and $B$ is defined (Sekerinski and Sere, 1996) as

$$A / B \equiv A \parallel (\neg gA \Rightarrow B)$$

Refinement of actions. In action systems, a refinement specifies the transformation of a higher level description, into a more detailed one, while preserving the correctness properties of the initial model. Refinements follow strict mathematical rules, as established by the refinement calculus (Back, 1990). An action $A$ is refined by the action $C$, denoted $A \preceq C$, if, whenever $A$ establishes a certain postcondition, so does $C$

$$\forall q \cdot \wp(A, q) \Rightarrow \wp(C, q)$$

A simple example of such a refinement is the strengthening of the guard of an action: $g_1 \rightarrow S \leq g_1 \land g_2 \rightarrow S$. Also, the introduction of a local variable is a refinement.

Trace refinement of action systems. The semantics of a reactive action system is given in terms of behaviors (Back and von Wright, 1994). 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 the local state component in each state of a given system and all finite stuttering (no change of the visible states). During system development, through refinement, system behavior may change. If the new traces are approximated by the initial ones, we run into a trace refinement.

We say that an action system \( SYS_C \) (trace-) refines \( SYS_A \) if every trace of \( SYS_C \) contains a trace of \( SYS_A \). At the system level, one may also consider, besides the behavioral changes of existent actions, the introduction of new actions, that detail the internal behavior of the system; these actions are called auxiliary actions. The theoretical basis for trace refinement is expressed by the trace refinement lemma (Back and Sere, 1994), given in the following, in a simplified version, tailored to our immediate purposes.

**Lemma 1.** Given the action systems
\[
SYS_A(z) \triangleq \text{begin var } a \bullet a, z := a_0, z_0; \text{ do } A \text{ od}
\]
\[
SYS_C(z) \triangleq \text{begin var } c \bullet c, z := c_0, z_0; \text{ do } C \parallel X \text{ od end.}
\]
the concrete system \( SYS_C \) (trace-) refines the abstract system \( SYS_A \), denoted \( SYS_A \subseteq SYS_C \), if:

1. **Main action:** \( A \subseteq C \).
2. **Auxiliary action:** \( \text{skip} \subseteq X \)—meaning that \( wX \) are local variables of \( SYS_C \).
3. **Continuation condition:** \( gA \Rightarrow gC \lor gX \)—meaning that whenever an action is enabled in \( SYS_A \), we also have an enabled one in \( SYS_C \).
4. **Internal convergence:** \( \text{true} \equiv \wp(\text{do } X \text{ od, true}) \)—meaning that the execution of \( X \), taken separately, eventually terminates.

### 2.1. Execution of action systems

Traditionally, action systems are viewed as components acting in parallel, with respect to a non-deterministic interleaving execution model (Back and Kurki-Suonio, 1988). Thus, we assume that a parallel composition of action systems is observed by a virtual external entity—the execution controller, which, at any moment, knows what actions are enabled, in which action system. The initialization places the components in a stable, starting state. The controller non-deterministically selects any of the enabled actions for execution. We call this operation an execution round (equivalent to the execution of an action). Following this, the controller evaluates the new state, observes the enabled actions and starts another execution round.

**Synchronized environments.** An alternative virtual execution model has recently been added into the framework of action systems (Cerschi Seceleanu and Seceleanu, 2004), namely the synchronized environment. Here, the execution of the system components is synchronized with respect to updates on the global variables of the respective components. This models the unitary reaction of the compound system to a given input situation. In brief, the observable execution model is changed as follows. The controller non-deterministically selects one of the components for execution. After performing all possible execution rounds with respect to the input state, the controller marks the corresponding action system as executed. Next, it selects another unexecuted component and performs the same operation. 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 visible state of the synchronized composition, thus completing an execution cycle. Moreover, we impose that different components have different write variables.

A synchronized environment has some useful properties. The first is the increased behavior control: 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 improved modularity: the system integrator need not be aware of the details of the employed modules. Moreover, individual refinements of the latter lead to refinements of the global system, too.

A synchronized environment is obtained when several action systems evolve according to the informal execution scenario introduced above. These systems must be proper (in the sense of suitable for our intentions).

**Definition 1.** Consider the action system \( SYS \):
\[
SYS(z : T_z) \triangleq \text{begin var } x : T_x \bullet \text{Init; do } g_S \rightarrow S \parallel g_L \rightarrow L \text{ od end}
\]
We say that \( SYS \) is a proper action system if:

- \( gwSYS \subseteq wS \)—meaning that \( S \) is the global action of \( SYS \),
- \( wL \subseteq lwSYS \)—meaning that \( L \) is the local action of \( SYS \),
- \( \wp(\text{do } g_L \rightarrow L \text{ od, } \neg g_L \land g_S) \equiv \text{true} \)—meaning that the execution of \( L \), taken separately, terminates, establishing the precondition for executing \( S \).
The synchronized composition is then defined as follows.

**Definition 2.** Consider $n$ proper action systems:

$$SYS_k(z_k) \triangleq \text{begin } x_k \bullet \text{Init}_k; \text{ do } g_k^1 \rightarrow S_k ; g_k^2 \rightarrow L_k \text{ od end, } \quad k = 1, \ldots, n$$

for which we also have that $\forall i, j = 1, \ldots, n, i \neq j : ((\text{gw}SYS_i \cap \text{gw}SYS_j = \emptyset) \land (x_i \cap x_j = \emptyset))$. The synchronized parallel composition of the above systems is a new action system $P = SYS_1 z \ldots z SYS_n$, given by:

$$P(z) \triangleq \text{begin } x : T_x, sel[1..n] : \text{Bool}, run : \text{Nat} \bullet \text{Init}; \text{ do }$$

$$\begin{align*}
\quad & (\text{run} = 0 \land \neg \text{sel}[1] \rightarrow \text{sel}[1] := \text{true} ; \text{run} := 1; \\
\quad & \quad \ldots \\
\quad & (\text{run} = 1 \land \neg \text{sel}[n] \rightarrow \text{sel}[n] := \text{true} ; \text{run} := n; \\
\quad & \quad \ldots \\
\quad & (\text{run} = 1 \land g_1^1 \rightarrow L_1 \\
\quad & \quad \quad \text{Component 1 } \\
\quad & \quad \quad \text{run} = 1 \land \neg g_1^1 \land g_1^2 \rightarrow wS_1c := wS_1 ; S'_1 ; \text{run} := 0; \\
\quad & \quad \quad \text{run} = 1 \land \neg g_1^2 \rightarrow \text{run} := 0)
\end{align*}$$

$$\ldots$$

$$\begin{align*}
\quad & (\text{run} = n \land g_n^1 \rightarrow L_n \\
\quad & \quad \text{Component n} \\
\quad & \quad \text{run} = n \land \neg g_n^1 \land g_n^2 \rightarrow wS_nc := wS_n ; S'_n ; \text{run} := 0; \\
\quad & \quad \text{run} = n \land \neg g_n^2 \rightarrow \text{run} := 0)
\end{align*}$$

$$\text{sel} \land \text{run} = 0 \rightarrow \text{Update}$$

end

The operator ‘$z$’ (‘sharp’) is called the synchronization operator. The above system $P$ is the “flattened” representation of the synchronized composition $SYS_1 z \ldots z SYS_n$.

The synchronized composition is, essentially, a cooperation between three kinds of actions: Selection, Execution, Update. While the roles of Selection and Execution are clearly stated, the action Update represents the integrator’s choice of deciding how the actual updates of the global variables (and possible other local ones) are performed.

In an initial set-up, Update is specified as an atomic assignment sequence, of the kind $\text{global variable} := \text{local copy variable}$, as the components are required to have disjoint global variable sets. 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 end, within Update, the variable $sel$ is reset ($sel := \text{false}$).

Next, we analyze how such a synchronized composition of action systems helps us to model systems that continuously adapt their execution characteristics, depending on specific changes in the environment. In the following, we assume that the requirements to be satisfied belong to a fixed set.

### 3. Adaptable synchronized model

Suppose that we have a communication system, $MS$, built (for illustration reasons) from three information delivery devices ($Delivery\_Module_1, \ldots, Delivery\_Module_3$), which are capable of producing output at three different rates, $C_1, \ldots, C_3$, correspondingly. One can imagine a web-based multimedia delivery service composed of three servers handling users with different bandwidth capabilities, related to the three supply rates. We identify each of these modules with an action system. In order to allow a proper communication between the components of $MS$, they are organized in a synchronized manner:

$$MS \triangleq Delivery\_Module_1 z \ldots z Delivery\_Module_3$$

The system $MS$ interacts with a group of users, by means of communication and data channels. The group contains a fixed, but sufficiently large number of “users”. The users may non-deterministically request services from the system, supplying the desired or constrained requirements of the connection bandwidth to the multimedia system controller. The latter analyzes the requirements and directs them further to the appropriate server or interface.

The systems modeling the users and the system $MS$ compose the top level action system, $Comm$. Although it would be easy to consider $Comm$ as a synchronized composition, too, this would impose unnatural constraints on the user model. Therefore, $Comm$ is a parallel composition, described as
If a higher bandwidth is specified, but only a narrower one is available, the controller may still direct the request to the available server, in order to satisfy the request. When no incoming requests may be serviced, the system becomes overloaded, and these requests will not be served, until termination of some of the connections.

In the following, we keep the details of our example servers hidden, on purpose, as the problems that we deal with are situated at the immediate higher level of abstraction.

System modeling—the users. The simple Userₖ action system, given below, models an arbitrary user. The whole environment of the multimedia system is composed of \( n \in \text{Nat} \) such systems, acting in parallel (the system Users).

\[
\text{User}_k(\text{req}_{IN}[k], \text{ack}_{OUT}[k] : \text{Bool}, c_{\text{req}}[k] : \{C_1, C_2, C_3\}, \text{data}_k : T_D)
\]

\[
\begin{align*}
\text{begin} & \quad \text{req}_{IN}[k], \text{ack}_{OUT}[k] := \text{false}; c_{\text{req}}[k], \text{data}_k := c_0, \text{data}_0; \\
& \text{do} \\
& \quad \neg \text{req}_{IN}[k] \land \neg \text{ack}_{OUT}[k] \\
& \quad \rightarrow (\text{req}_{IN}[k] := \text{true}; c_{\text{req}}[k] := c \cdot (c \in \{C_1, C_2, C_3\})] \text{ skip}) \\
& \quad \left[ \text{req}_{IN}[k] \land \text{ack}_{OUT}[k] \\
& \quad \rightarrow \text{processing data; (req}_{IN}[k] := \text{false} \right] \text{ skip}) \\
& \text{od} \\
\text{end}
\]

The communication channel between the user and the multimedia system is composed of two variables: req—request and ack—acknowledge, which model a basic handshake mechanism. The variable \( c_{\text{req}} \) models the quality requirements as illustrated by the set of available values, \( \{C_1, C_2, C_3\} \); the variable \( \text{data} \) (of some abstract type \( T_D \)) represents the content of the communication. Any other protocol may be employed, to accommodate the communication between users and the multimedia system, but this simple one suffices for reaching our goals. Further, we are not interested in how the user proceeds in processing the received data, therefore this aspect is hidden in the procedure \( \text{processing data} \). The sequence of states for the variables that compose the communication channel is given in Fig. 1.

Whenever the user is not involved in any communication with the multimedia system, that is, when \( \neg \text{req}_{IN}[k] \land \neg \text{ack}_{OUT}[k] \) holds, it has the freedom of issuing a new request, accompanied by a new value of the respective \( c_{\text{req}} \), or by idling (represented by the \( \text{skip} \) statement). The same applies if the user decides to maintain the connection with the multimedia system, after processing the received data. In this latter case, the previous quality requirements are preserved.

System modeling—the multimedia system. Consider that we are in possession of processing modules that deliver multimedia content, and that they correspond to the selected communication protocol. Their generic representation is

\[
\text{Delivery Module}(\text{req}, \text{ack} : \text{Bool}, \text{data} : T_D)
\]

\[
\begin{align*}
\text{begin} & \quad \text{req}, \text{ack} := \text{false}; \text{data} := \text{data}_0; \\
& \text{do} \\
& \quad \text{req} \rightarrow \text{Service}; \text{ack} := \text{true} \\
& \quad \left[ \neg \text{req} \land \neg \text{ack} \rightarrow \text{ack} := \text{false} \\
& \text{od} \\
\text{end}
\]

Fig. 1. Communication protocol.
For simplicity, the actual service provided by Delivery Module is not detailed. Its behavior is contained within the generic action Service, which updates the variable data.

We continue by considering the synchronized composition of three similar data delivery modules, each characterized by a specific delivery rate. The multimedia system results, then, as MS $\triangleq D_1 \triangleright D_2 \triangleright D_3$, where $D_1, D_2, D_3$ are instances of the Delivery Module template described above. The “flattened” version of MS is illustrated in Fig. 2. The Update action is given as

$$Update \triangleq \text{sel} \land \text{run} = 0$$

$$\rightarrow \text{data} \leftarrow \text{data}_c; \ \text{ack}_{OUT} \leftarrow false; \ \text{ack}_{OUT}[ID[1]] \leftarrow \text{ack}[1]; \ \text{ack}_{OUT}[ID[2]] \leftarrow \text{ack}[2]; \ \text{ack}_{OUT}[ID[3]] = \text{ack}[3]; \ \text{sel} \leftarrow false,$$

where $\text{sel} \equiv \text{sel}[1] \land \text{sel}[2] \land \text{sel}[3]$.

In this version of the MS system, we deferred the specification of the update on the vector ID, which identifies the user request serviced by the devices. The variable data, is replaced by the local copy, data_c. In consequence, we have also replaced the initial Service_X actions ($\forall \in \{1, 2, 3\}$) with Service_X:

$$\text{Service}_X = \text{Service}_X[data \leftarrow \text{data}_c]$$

Furthermore, the variables req, ack, of each of the Delivery Module components, are mapped to the elements of the local vector variables req[1..3] and ack[1..3], respectively. This is the reason why the components of the global variable ack_{OUT}[1..n], updated by the modules, do not need copies; these are considered to be the elements of the local vector ack. Due to this mapping, the condition gg_D, defined as the disjunction of the guards of the delivery modules, is:

$$gg_D \triangleq \text{req}[1] \lor \text{ack}[1] \lor \text{req}[2] \lor \text{ack}[2] \lor \text{req}[3] \lor \text{ack}[3]$$

Hence, as req[1], ..., ack[3] are local variables of MS, and are initialized to false, the system will never start executing, even though the external request lines (req_IN) may evaluate to true. In addition, we have to ensure that, from one execution cycle to the next, the user that has obtained access to the requested resource does not lose the connection.

The solution comes as an additional control action, operating on the internal request lines. The modified version of the system is:

$$MS(req_IN[1..n], ack_{OUT}[1..n] : \text{Bool}, \ c\_req[1..n] : \{C_1, C_2, C_3\}, \ \text{data}[1..n] : T_D)$$

$$\triangleq \begin{align*}
\text{begin} \ \text{var} \ \text{req}[1..3], \ \text{ack}[1..3], \ \text{sel}[1..3] : \text{Bool}, \ \text{ID}[1..3], \ \text{run} : \text{Nat}, \ \text{data}_c : T_D; \\
\text{do} \ \text{Control} [/ (gg_D \rightarrow \text{Selection} \ [\text{Execution} \ [\text{Update}\ ] \text{od}] \text{end}
\end{align*}$$

Fig. 2. The system MS.
In $MS_1$, the actions $\text{Execution}$ and $\text{Update}$ are as described earlier (Fig. 2), whereas the action $\text{Control}$ is given by

$$\text{Control} \equiv \text{Control}^1 \parallel \text{Control}^2 \parallel \text{Control}^3,$$

$$\text{Control}^p \equiv \neg \text{sel}[k] \land \neg \text{ack}[k] \land \left( \forall j \left( \text{req}_{\text{IN}}[j] \land (c \cdot \text{req}[j] = C_p) \land j \notin \{ID[(k+1) \mod 3], \, ID[(k+2) \mod 3]\} \right) \right)$$

$$\rightarrow ID[k] := m \cdot (\text{req}_{\text{IN}}[m] \land c \cdot \text{req}[m] = C_p \land \forall n \neq k \cdot m \neq ID[n])$$

$$\parallel \neg \text{sel}[k] \land \text{ack}[k] \land \text{req}[k] \land \neg \text{req}_{\text{IN}}[ID[k]] \rightarrow \text{req}[k] := false$$

In the description of the actions $\text{Control}^k$, the index $p$ stands for the selection of the quality level ($C_1$, $C_2$, $C_3$) whereas $k$ identifies the delivery module ($k \in \{1, 2, 3\}$).

Observe that the $\text{Control}$ action ensures the continuation of service for an acquired connection, by testing the value of the local variables $\text{ack}[k]$. Also notice that the “fall” of the request line from the user is followed by the reset of the corresponding element of the vector $\text{req}$. The updates of the connection identification variable $ID$ are expressed by the action $\text{Control}$, too.

**Lemma 2.** $MS_1$ is a refinement of $MS$: $MS \sqsubseteq MS_1$.

**Proof.** There are actually two refinement steps concentrated in the transformation of $MS$ into $MS_1$, which we outline below.

1. The first step considers adding the action $\text{Control}$ to the original action composition of the $MS$ do–od loop, resulting in a new choice composition. The corresponding intermediate system, $MS'$, has the same global and local variables as $MS$, the same initialization, the difference residing in the description of the do–od loop. The excerpted description $MS'$ is:

$$MS'(\cdots) \triangleq \text{begin} \cdots \cdot \cdots; \text{do}$$

$$\text{Control} \parallel \text{ggD} \rightarrow (\text{Selection} \parallel \text{Execution} \parallel \text{Update})$$

$$\text{od}$$

$$\text{end}$$

Notice that $w\text{Control} \in l \text{w}MS'$, that is, $\text{Control}$ is a local auxiliary action of $MS'$, refining $\text{skip}$, so satisfying the second requirement of Lemma 1. Next, the actions $\text{Control}^k$ are self-disabling actions, and independent of each other (they can not enable or disable each other). Thus, the execution of $\text{Control}$ stops after at most three iterations of the system loop. It follows then that the internal convergence requirement of the refinement lemma is also satisfied. There is no change regarding the original actions of $MS$, meaning that also the other specifications of Lemma 1 are trivially respected, hence $MS \sqsubseteq MS'$.

2. As proved by Sekerinski and Sere (1996), giving priority to one of the actions of a choice construct is always a refinement step. No other changes are performed, thus we may state that $MS' \sqsubseteq MS_1$.

Based on the transitivity of the refinement relation, we obtain that $MS \sqsubseteq MS_1$. \hfill \Box

### 3.1. System development—quality preferences

Notice that the $\text{Control}$ action introduced in the previous section only allows service to requests matching the available servers in their $C_p$ group, respectively. This is a strong requirement and presumes that the incoming requests will always cover each of the servers. Otherwise, some of these components will idle. In this section, we look for a complete coverage of the servers, together with some simple priority schemes.

Suppose that we decide to treat preferentially the requests asking for higher quality communication—group $C_1$. If there are not enough requests of this kind, then requests from the group $C_2$ should also be served before the ones requiring only low quality connections—group $C_3$.

We obtain such behavior modification by employing techniques similar to the ones that we have adopted in the previous section. Thus, we only concentrate on the $\text{Control}$ action. Instead of the original solution, which tried to satisfy equally three requests characterized by different communication features, we build a more elaborate system, where $\text{Control}$ is replaced by $\text{Control}^1$: 
The behavior of the system $MS_2$ is characterized by the preference given to higher quality requests:

1. The first group of control actions, having the highest priority, tries to find resources for two $C_1$ requests (on servers 1 and 2). Subsequently, one $C_2$ request may be also served, by server 3.
2. If there are not two $C_1$ requests, additional $C_2$ ones may be served (actions in the second control group), by either servers 1 or 2; if there is another $C_1$ request, it may be served by server 3, this time.
3. Eventually $C_1$ requests may be answered, only by the available servers.

### 3.2. System development—faulty servers

Suppose, next, that the designer of delivery modules improves their behavior by providing an additional flag that signals the status of the respective component. A possible new representation of the module may be as follows:

```plaintext
Delivery_Module(!req, ack : Bool, data : T_D)
≡ begin
  var status : \{OK, Fault\} •
  status := OK; req, ack := false; data := data_0;
  do
    status = OK
    \[ (\neg req \wedge \neg ack \rightarrow ack := false);
    status := s \cdot (s \in \{OK, Fault\})
  od
end
```

How does this affect the behavior of the multimedia system? The answer stands in the resulting expression of the $gg_D$ guard. The new $gg_D$ is:


This further means that the synchronized composition, by definition, reacts to such a situation, without requiring the system level designer to specify any reaction to this kind of situation. He or she may even not be aware of the change performed by the module designer.

The behavior is affected in the following way. Consider, for instance, that the third server becomes unusable ($status[3] = Fault$). This will not influence the decisions taken by the control actions. Hence, suppose that the faulty server receives permission to execute its tasks ($req[3]$ is set to $true$). In consequence, the execution eventually proceeds with the assignments $sel[3] := true; run := 3$. Now, the execution controller will have to select the execution action corresponding to the third server. However, due to the fact that $gg_3 \triangleq status[3] \wedge (req[3] \vee ack[3])$ evaluates to $false$, the controller will only be able to select the action $run = 3 \land \neg gg_3 \rightarrow run := 0$. Hence, no output variables are updated.

Nevertheless, the system will observe an increase in the servicing time (measured in execution cycles) reported to the same number of incoming requests. This can be regarded as a “graceful degradation” process, as the performance gradually lowers, but no faulty data streams are delivered.

In contrast, whenever the faulty module is repaired, again, this is transparent both to the user and to the system developer. The observable behavior changes now towards shorter times of data delivery, as defined above.

### 3.3. Discussion

Cheng et al. (2004) have identified three technical challenges that need to be addressed in order to support the design of correctly coordinated multiple self-management systems. These are: consistent system access, consistent model and non-conflicting decision. The first two challenges are, in our approach, solved by employing of the synchronized environment: by
eliminating the interleaved execution model at the system level, the consistency with respect to input states of a synchro-
nized composition depends only on the state itself. The non-conflicting aspect, as a final action in taking a certain decision,
may also be applied to the synchronized execution model, at the level of action Update. However, a more complex descrip-
tion must be considered in this case, as the definition of the synchronized composition does not allow, yet, for such a
situation.

Another related aspect, and subject of on-going work, is the interaction between the users (in our case) and the multi-
media system. In principle, non-deterministic descriptions model quite well the unpredictable behavior of the user. One
may not want to constrain the users, in the sense of establishing, a priori, when they may decide to request a connection
to the server. Nevertheless, the non-deterministic execution may interfere with the separation in rounds and cycles, which
characterizes the synchronized environments.

The same situation arises if one considers the possible interleaved execution of control and user actions. Here, the con-
sistency of the execution is given, on one hand, by the careful selection of the guards of control actions: the execution con-
troller may not select again any control action before the termination of the current execution cycle. On the other hand, by
the prioritized composition between Control and the original synchronized system actions, it is ensured that the require-
ments of the latter (proper assignments to the req variables) are satisfied.

Observe, next, that in the multimedia system example the prioritized composition allows us to construct, in some sense,
a resource discovery mechanism. The model gives higher priority to requests characterized by high transmission rates. The
users connections characterized by a slow speed are the last ones to be granted access to the server resources, and this hap-
pens only in situations where there are not three requests with higher transmission speeds. However, this is not a problem
of the model; instead, it belongs to the servicing policies of the respective system. Distinct resource selection policies, or
“search” paths, may be applied for each of the control branches of the construct, ours being just one of them.

While the above mentioned policy decisions are the privilege of the system integrator, the internal behavior of the com-
ponents (in the example, the servers) is not relevant at this level. Explicitly, if the “server” designer decides to offer an
improved version of its product, and this is taken into consideration by the top-level designer, it will not affect the model
or its properties. This is one of the most important benefits brought by the synchronized composition.

4. Hierarchical development

As we have implicitly suggested in previous chapters, in line with the current design practices, complex systems are built
from less complicated available modules. As the synchronized composition of proper action systems preserves properness
(Cerschi Seceleanu and Seceleanu, 2004) and therefore, also modularity, it is easy to think of a hierarchy of synchronized
components, where possible selection or control policies are applied at different levels.

We analyze, in this section, such a lower architectural-level description of the action system Delivery_Module as a syn-
hronized composition of three sub-systems. Suppose, thus, that the multimedia content consists of video, audio and tex-
tual data components.

The server system can then be described as

\[ \text{Delivery\_Module}_{HI} = V\_Service\_Module \, \star \, A\_Service\_Module \, \star \, T\_Service\_Module \]

![Fig. 3. The structure of the system Delivery\_Module_{HI}.](image-url)
Each of the sub-modules of Delivery_ModuleH are instances of the template:

\[ XServiceModule(req, ack : \text{Bool}, data : T_X) \]

\[ \equiv \begin{aligned} & \text{begin } req, \text{ack} := \text{false}; \text{data} := \text{data}_0; \\
& \text{do} \\
& \quad (req \to XService; \text{ack} := \text{true}) \\
& \quad \left[ (\neg req \land \text{ack} \to \text{ack} := \text{false}) \\
& \text{od} \\
& \end{aligned} \]

end,

where the preceding X stands for either “V” (video), “A” (audio) or “T” (text).

Notice that the “policy” for launching into execution the three sub-components of Delivery_ModuleH follows the original solution, that is, modules are selected for execution in an arbitrary manner, according to Definition 2. This can be compared with the execution selection policy at the level of MS (Fig. 2).

If we rebuild the multimedia system consisting of three delivery modules, but considering Delivery_ModuleH instead of Delivery_Module, we obtain the system MS_H, illustrated in Fig. 4.

4.1. Discussion

Employing the synchronization operator in the description of Delivery_ModuleH, instead of the parallel one, is motivated by the need of eliminating the non-deterministic interleaved interaction between the users and the components of
the system. In a synchronized operation, the output of a system composed of sub-modules is observed simultaneously, at the end of the execution cycle, hence not allowing for partial reactions that are possible within an interleaved approximation of concurrency. Thus, the correlation between the distinct data-flows representing textual, video and audio information, respectively, is enforced by the model.

We can imagine, next, that each of the service modules $X_{Service\_Module}$ may be changed as suggested in Section 3.2, that is, the module designer supplies a flag $(status_V, status_A, status_T)$, respectively) that reflects the internal state of the module. This can be captured at the level of $Delivery\_Module_H$, by a global variable, $status$. Several response policies may be applied, considering an appropriate severity characterization of the fault, based on its localization.

For instance, consider that the $Update$ action contains the assignment $status := status_V \land status_A$. This means that the $Delivery\_Module_H$ will still deliver data (video and audio), even when the text module is faulty ($status_T = false$). A possible malfunction of the system is pondered only when either the video or the audio delivery modules are at fault ($status = false$). Thus, again, the system will offer a graceful degradation in service. Observe that this solution is also supported by the way in which the action $Update$ processes the changes on the variable $ack$, based on the values of the local vector $ack_L$.

5. Conclusions

Focusing on the special case of adaptation in which a fixed set of configurations is given, we have proposed an environment for both rigorous modeling and reasoning about adaptive systems, using the formal framework of action systems. Based on a synchronized perspective on system design, our study shows how quality of service requirements and response to fault situations may be satisfied and answered, along with functional services.

According to the lines identified by Cai et al. (2004), our systems are composed of a control-selection part, plus (modularized) functional part. Adaptive or not, feedback control is observed at the end of execution cycles. Refinement techniques applicable to the employed modules modeled as action systems are certified ways of obtaining correct system implementations and maintenance. As distinct from other approaches, ours does not allow the modeling of the dynamic behavior of such systems, as, for instance, described by Poladian et al. (2004), and Schmerl and Garlan (2002). Therefore, certain parameters of the example are fixed, such as the number of users and the number of resources. However, interpreting the limits as worst-case situations, real systems may still be represented. The situation is similar to the one analyzed by Allen et al. (1998): we mimic dynamic connectivity by means of a static architecture.

Starting from the original definition of synchronized composition, through refinement techniques, one is capable to obtain a system description where different selection and search algorithms may be used as required by the system integrator. These features are not supported by most approaches targeting self-adapting system design. Based on the characteristics of synchronized environments, modularity and hierarchical design flow are two other benefits of our approach.
Similar to the CHAM language (Inverardi and Wolf, 1995), where the communication style between components is not implicitly specified, action systems allow for the existence of both interleaved and synchronized execution paradigms, as illustrated. However, we have advocated the usage of synchronized architectures for system modeling, as an improvement to consistency, transparency and modularity. While CHAM requires the specification of rewriting rules in order to cope with architectural reconfigurations, in synchronized environments, the presence or absence of computing modules is implicitly observed, based on the evaluation of system guards.

6. Future work

The present study can be completed with a similar analysis on system development, but adding the axis of time. Action systems in general, and specifically the synchronized environments, allow reasoning about timed systems (Cerschi Seceleanu and Seceleanu, 2004). One of the challenging future goals is to see how issues like system adaptability and fault tolerance can be examined from a timed perspective, too.

Acknowledgements

The authors thank Cristina Cerschi Seceleanu for valuable comments on this paper. The comments from the anonymous reviewers also helped improving the quality of the manuscript.

This work was partially supported by the Army Research Office (ARO) through grant DAAD19-02-1-0389 (“Perpetually Available and Secure Information Systems”) to Carnegie Mellon University’s CyLab, by ARO Grant DAAD19-01-1-0485 (“Verification Tools for Embedded Systems”), and by the National Science Foundation (NSF) through grants CCR-0205266 (“ITR: Proactive Self-Tuning Systems for Ubiquitous Computing”) and CCR-0113810 (“ITR: Compositional Connectors”). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARO or the NSF.

References


