Synchronous Formalism and Behavioral Substitutability in Component Frameworks

Sabine Moisan & Annie Ressouche [1]

INRIA Sophia Antipolis, 2004 route des Lucioles
06902 Sophia Antipolis, France

Jean-Paul Rigault [2]

I3S Laboratory, Univ. of Nice Sophia Antipolis and CNRS (UMR 6070)
06902 Sophia Antipolis, France

Abstract

When using a component framework, developers need to respect the behavior implemented by the components. Static information about the component interface is not sufficient. Dynamic information such as the description of valid sequences of operations is required. In this paper we propose a mathematical model obeying the Synchronous hypothesis and a formal language to describe the knowledge about behavior. We focus on extension of components, owing to the notion of behavioral substitutability. A formal semantics for the language is defined and a compositionality result allows us to get modular model-checking facilities. From the language and the model, we can draw practical design rules that are sufficient to preserve behavioral substitutability. Associated tools may ensure correct (re)use of components, as well as automatic simulation and verification, code generation, and runtime checks.

Key words: component frameworks, protocol of use, behavioral substitutability, synchronous models, model-checking verification

1 Introduction

The motivation for this work is the correct usage of an object-oriented component framework. A component framework is dedicated to a family of problems (compiler construction, graphic user interface, knowledge-based systems, etc.). Basically, it is a well-defined architecture composed of top level classes and their relationships. To use a component framework, a developer (or framework user) selects, adapts, and assembles components to build a customized application. This

[1] Email: Sabine.Moisan@inria.fr, Annie.Ressouche@inria.fr
[2] Email: jpr@essi.fr

customization process essentially involves deriving new classes from the framework ones or composing them. The process should preserve the invariants that the framework level guarantees. This implies to understand the nature of the contract between the framework user and the components. The mere specification of a static interface (list of operation signatures) is not sufficient since it misses the information regarding the component *dynamic behavior* (state-based). In this paper, we focus on framework adaptation through subtyping (more exactly, class derivation used as subtyping). Thus derived classes must respect the behavioral protocol that their base classes implement and guarantee. In particular, we want to ensure that an invariant property at the framework base level also holds at the developer's level. Since the substitutability principle [15] offers a natural semantics for subtyping, we propose here to enlarge its scope to state-related properties. Thus the notion of *behavioral substitutability* is central to our approach to framework correct usage. To this end we elaborated a formal model of behavioral substitutability, where safety properties are preserved during subtyping. Our model relies on the Synchrony Hypothesis, a now classical means to represent behavior. Moreover, this hypothesis allows to apply model-checking techniques and leads to models more tractable by automatic tools. Our aim is to propose a verification algorithm as well as practical design rules to ensure sound framework adaptation.

There is quite a variety of object-oriented frameworks, differing in particular by the model of interaction between components: distributed, asynchronous, synchronous, etc. We do not claim to address the whole generality. Moreover, we restrict to a narrow case, although a very useful one: no distribution, no implementation level concurrency, communication through (synchronous) method calls. We shall see that the Synchrony Hypothesis suits well this restriction.

The next section defines our notion of components and their protocol of use. Section 3 presents the mathematical model and formal language to describe the behavioral part of the protocol. Section 4 depicts the model-checking verification we support in this work. Section 5 illustrates our approach on examples and shows how safety properties can be proved. Finally, section 6 discusses some issues about this approach and relates it to similar work.

2 Framework Protocol of Use

To describe how to use a component, we need to specify its *protocol of use* that describes the relationship between the state of the component and the calls to the component operations. This section is devoted to the description of this protocol.

In the object-oriented community a component framework is usually composed of several hierarchies of classes. The root class of each hierarchy corresponds to an important concept in the target domain. In this context, a component can be viewed as the realization of a class hierarchy: this complies to one of Szyperski's definitions for components [24].

Framework users both adapt the components and write some glue code. To achieve a given purpose, they will (non-exclusively) use these components directly

(like a library), specialize their classes by inheritance, compose classes together, or instantiate new classes from predefined top level ones. Among all these possibilities, class derivation is frequent. It is also the one that may raise the trickiest problems, that is why we concentrate on it in this paper. When deriving a class, users may either introduce new attributes and/or operations or redefine inherited operations. These specializations should be "semantically acceptable", i.e., they should comply with the design hypotheses of the framework.

Our work on formalizing component protocols is motivated by our experience with a framework for knowledge-based system inference engines, named BLOCKS [20]. BLOCKS's objective is to help developers create new inference engines and reuse or modify existing ones. We shall draw the paper examples from BLOCKS itself, just for illustration.

At BLOCKS top level there are presently three main components that may be extended, leading to new components. For the extension to be possible, the platform user needs information about component properties. For it to be safe, he or she should commit to some protocol. For it to be even safer, we offer proof and validity checking tools. The three high level components are associated with initial sub-trees of the three main classes: `Frame`, `Rule`, and `State`.

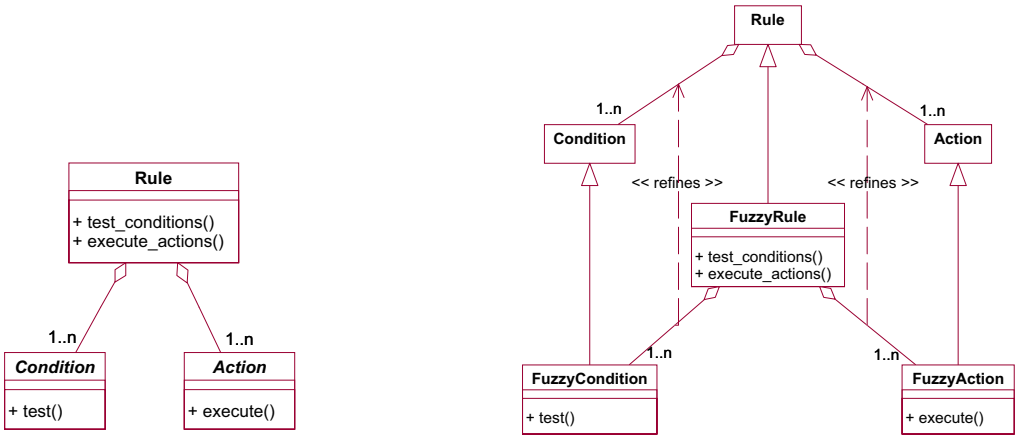
Let us take an example: the `Rule` class in BLOCKS (figure 1(a)) is composed of conditions and actions. Its implementation does not take into account fuzzy values in conditions and actions. The developer may want to define the `FuzzyRule` class as a derivative of `Rule`. He/she can access the static information (signatures of methods and associations among classes) of the UML class diagram of figure 1(a) and setup the inheritance graph shown on figure 1(b). Indeed, this information is not (totally) present at run-time, but this is easy to compensate by setting up an introspection facility.

The developer afterwards needs to redefine methods `test_conditions` and `execute_actions`. But where to get the necessary information? This raises the problem of defining which behavior is acceptable for a safe redefinition of a method. The needed information is certainly not in the UML class diagram(s).

Static information is not sufficient to ensure a correct use of a framework: specifying a protocol of use is required. This protocol is defined by two sets of constraints.

First, a *static* set enforces the internal consistency of class structures. UML-like class diagrams provide a part of this information: input interfaces of classes (list of operation signatures), specializations, associations, indication of operation redefinitions, and even constraints on the operations that a component expects from other components (a sort of *required interface*, something that will likely find its way into UML 2.0). We do not focus on this part of the protocol since its static nature makes it easy to generate the necessary information at compile-time.

A second set of constraints describes *dynamic* requirements: (1) legal sequences of operation calls, (2) specification of internal behavior of operations and of sequences of messages these operations send to other components, and (3) behavioral specification of valid redefinition of operations in derived classes. These dynamic



(a) Simplified diagram of class `Rule`

(b) `FuzzyRule` extension

Fig. 1. `Rule` and `FuzzyRule` UML diagrams

aspects are more complex to express than static ones and there is no tool (as natural as compiler-like tools for the static case) to handle and check them. While item (1) and partially item (2) are addressed by classical UML state-transition models, the whole treatment of the last two items is more challenging.

3 Behavior Description and Refinement

To cope with dynamic aspects, our approach is threefold. First, we define a mathematical model providing a consistent description of *behavioral entities*, which may be whole components, sub-components, single operations, or any assembly of these. Hence, the whole system is a hierarchical composition of communicating behavioral entities. Such a model complements the UML approach and allows to specify class and operation behavior with respect to class derivation. Second, we propose a *hierarchical* behavioral description language (BDL) to specify the dynamic aspect of components, both at the class and operation levels. Third, we define a *semantic* mapping to bridge the gap between the specification language and its meaning in the mathematical model.

As already mentioned, our primary intent is to formalize the behavior side of class derivation, in the sense of *subtyping* [3]. In the object-oriented approach, subtyping usually obeys the classical Substitutability Principle [14]. This principle has a static interpretation leading to, *e.g.*, the well-known covariant and contravariant issues for parameters and return types. It may also be given a dynamic interpretation, leading to behavioral subtyping, or *behavioral substitutability* [12]. This is precisely the kind of interpretation we need to enforce the dynamic aspect of

[3] Note that, in this paper, derivation, inheritance, specialization all refer to the *subtyping* interpretation. In particular, we do not consider other uses or interpretations of inheritance.

framework protocols, since it provides a behaviorwise correct derivation.

Note that we focus on proving specifications, not implementations: although we use concurrency for our specification, modeling a system as disjoint parts with (more or less) independent execution, it is only a *logical* notion, not an implementation one. Moreover, this work does not address concurrent or distributed execution models, as we already mentioned in the introduction.

3.1 Description of Behavior using the Synchronous Paradigm

To deal with behavioral substitutability, we need behavior representation formalism, we propose to rely on the family of *synchronous* models [3,11], which are dedicated to specify *reactive* systems [13], such as found, for instance, in Real Time. Such systems are event-driven and discrete time: they interact with their environment, reacting to input events by sending output events.

Furthermore, they obey the *synchrony hypothesis*, i.e., the corresponding reaction is *atomic*; during a reaction, the input events are frozen, all events are considered as *simultaneous*, events are broadcast and available to any part of the system that listens to them. A reaction is also called an *instant*. The succession of instants defines a logical time. Because of synchrony, such a system is able to react to the presence of an input event as well as to its absence at a given instant.

These synchronous models provide general representations of state behavior; their semantics relies on classical finite automata theory. They are not restricted to real time or hardware systems and they can apply to modeling software component behavior as well. Moreover, they sport interesting properties to ease the verification task and to make it more efficient.

Indeed, verification of synchronous models exhibits a lower computational complexity than for asynchronous ones. Moreover, to describe complex behavior as found in component protocols of use, a hierarchical modular description is natural. Provided that certain "compositionality properties" hold, automatic proofs become modular and thus more efficient.

3.2 Mathematical Model of Behavior

Input/output labeled transition systems [17] are usual mathematical models for synchronous languages. In our approach, each reaction corresponds to a transition and obeys the synchrony hypothesis. These systems are a special kind of finite deterministic state machines (automata) and we shall denote them LFSM for short in the rest of the paper.

In our model, a LFSM is associated with a *behavioral entity*: we use LFSMs to represent the state behavior of classes as well as of operations. Each transition has a *label* representing an elementary execution step of the entity, consisting of a *trigger* (input condition) and an *action* to be executed when the transition is fired. In our case an action corresponds to emission of events, such as calling an operation of some component, whereas a trigger corresponds to starting an operation (in response to a call).

A LFSM is a tuple $M = (S, s_0, T, A)$ where S is a finite set of states, $s_0 \in S$ is the initial state, A is the *alphabet* of events from which the set of labels L is built, and T is the transition relation $T \subseteq S \times L \times S$. We introduce the set I of input events $I \subseteq A$ and the set $O \subseteq A$ of output events (or actions).

Labels
L, the set of *labels*, has elements of the form i/o, where $o \subseteq O$ is the action or output events set and i is the trigger: i has the form (i^+, i^-) where i^+, the positive (input event) set of a label, consists of the events tested for their presence in the trigger, and i^-, the negative (input event) set, consists of the events tested for their absence.
Moreover, we require that labels be *well-formed* which means that the following conditions must hold:

$$\begin{cases} i^+ \cap i^- = \emptyset & \text{(trigger consistency)} \\ i^+ \cup i^- = I & \text{(trigger completeness)} \\ i^- \cap o = \emptyset & \text{(synchrony hypothesis)} \end{cases}$$

The first condition means that an event cannot be tested for both absence and presence at the same instant. The second condition expresses that a trigger i contains either a test of presence or of absence for each event in I (this corresponds to the notion of a "completely specified" automaton). The synchrony hypothesis implies that an event tested for its absence in the trigger cannot be emitted as output in the same instant. It is clear that $i \cap o$ (that is in fact $i^+ \cap o$) can be non empty: indeed in the synchronous paradigm it is possible to test the presence of an event in the same instant it is emitted; it is even the primary way of modeling communication.

Transitions
Each transition has three parts: a source state s, a label l, and a target state s' (denoted $s \xrightarrow{l} s'$). There cannot be two transitions leaving the same state and bearing the same trigger [4]. This rule, together with the label well-formedness conditions, ensures that LFSMs are *deterministic*. Determinism constitutes one of the fundamental requirements of the synchronous approach and is mandatory for all models and proofs that follow. Moreover, LFSMs are *reactive*, i.e., there must be a transition from any state for any input. Formally, given a LFSM M and an input configuration i, for each state s in M, there is a transition $s \xrightarrow{i/o} s'$ in the transition relation of M. Reactivity implies that LFSMs have at least one input event (the input event set of any LFSM cannot be empty).

[4] Formally, if there are two transitions from the same state s such that $s \xrightarrow{i_1/o_1} s_1$ and $s \xrightarrow{i_2/o_2} s_2$, then $I_1^+ \neq i_2^+$.

Behavioral Substitutability

The substitutability principle should apply to the dynamic semantics of a behavioral entity–such as either a whole class, or one of its (redefined) operations [12]. If M and M' are LFSMs denoting respectively some behavior in a base class and its redefinition in a derivative, we seek for a relation $M' \preceq M$ stating that "M' extends M in a correct way". To comply with subtyping, this relation must be a preorder.

Following the substitutability principle, we say that M' is a correct extension of M, iff the alphabet of M' (denoted as $A_{M'}$) is a superset of the alphabet of M (denoted as A_M) and every sequence of inputs that is valid [5] for M is also valid for M' and produces the same outputs, once restricted to the alphabet of M. Thus, the behavior of M' restricted to the alphabet of M is identical to the one of M. Formally,

$$M' \preceq M \Leftrightarrow A_M \subseteq A_{M'} \wedge M \; \mathcal{R}_{sim} \; (M'\backslash A_M)$$

where $M'\backslash A_M$ is the *restriction* of M' to the alphabet of M and $\mathcal{R}_{sim}$ is a simulation relation.

First, we define the restriction $l\backslash A$ of a label l over an alphabet (A) as follows: let $l = i/o$,

$$l\backslash A = \begin{cases} (i \cap A/(o \cap A) & \text{if } i^+ \subseteq A \\ undef & \text{otherwise} \end{cases}$$

Intuitively, this corresponds to consider as undefined all the transitions bearing a positive trigger not in A, and to strip the events not in A from the outputs. The restriction of M to the alphabet A (generally with $A \subseteq A_M$) is obtained by restricting all the labels of M to A, then discarding the resulting undefined transitions.

Formally, let $M = (S, s_0, T, A_M)$ be a LFSM, $M\backslash A = (S, s_0, T\backslash A, A_M \cap A)$ where $T\backslash A$ is defined as follows:

$$s \xrightarrow{l'} s' \in T\backslash A \Leftrightarrow \exists\, s \xrightarrow{l} s' \in T \wedge l' = l\backslash A \neq undef$$

Second, the simulation relation we consider is the well-known Milner's one [19], adapted to LFSMs: let M_1 and M_2 be two LFSMs with the same alphabet: $M_1 = (S_{M_1}, s_0^{M_1}, T_{M_1}, A)$ and $M_2 = (S_{M_2}, s_0^{M_2}, T_{M_2}, A)$. A relation $\mathcal{R}_{sim} \subseteq S_{M_1} \times S_{M_2}$ is called a *simulation* iff

$$(s_0^{M_1}, s_0^{M_2}) \in \mathcal{R}_{sim} \text{ and}$$

$$\forall (s_1, s_2) \in \mathcal{R}_{sim} : s_1 \xrightarrow{l} s_1' \in T_{M_1} \Rightarrow \exists s_2 \xrightarrow{l} s_2' \in T_{M_2} \wedge (s_1', s_2') \in \mathcal{R}_{sim}$$

Simulation is the word used by R. Milner and unfortunately it may be given several interpretations. We prefer to use "substitutability" to stress the fact that both LFSMs operate under the same input scenarios, a kind of "plug-in" substitutability.

---

[5] A *path* in a LFSM M is a (possibly infinite) sequence of transitions $\pi = s_0 \xrightarrow{i_0/o_0} s_1 \xrightarrow{i_1/o_1} s_2 \ldots$ such that $\forall i (s_i, i_i/o_i, s_{i+1}) \in T$. The sequence $i_0/o_0, i_1/o_1 \ldots$ is called the *trace* associated with the path. When such a path exists, the corresponding trigger sequence $i_0, i_1, \ldots$ is said to be a *valid* sequence of M.

LFSMs are deterministic and it turns out that simulation coincides with trace containment in such a case. But simulation is local, since the relation between two states is based only on their successors. As a result, it can be checked in polynomial time, which is not in general the case for trace containment; hence, it is widely used as an efficient computable condition for trace-containment. Indeed, the simulation relation can be computed using a symbolic fixed point procedure, allowing to tackle large-sized state spaces.

Obviously, relation $\preceq$ is also a preorder over LFSMs; we call it the *substitution preorder*. We say that M' is substitutable for M iff $M' \preceq M$. Thus, a valid sequence of M is also a valid sequence of M' and the output traces are identical, once restricted to A_M. As a consequence, if $M' \preceq M$, M' can be substituted for M, for all purposes of M.

With such a model, the behavior description matches the class hierarchy. Hence, class and operation refinements are compatible and consistent with the static description: checking dynamic behavior may benefit from the static hierarchical organization.

3.3 Behavior Description Language (BDL)

We need a language that makes it possible to describe complex behavioral entities in a structured way. Similar to *Argos* [17], our language offers a graphical notation close to UML StateCharts with some restrictions, but with a different semantics, based on the Synchronous Paradigm. The language is easily compiled into LFSMs. Programs written in this language operationally describe behavioral entities; we call them *behavioral programs*. The mathematical model allows to express the semantics of this language, permitting an easy translation into LFSMs.

The primitive elements, from which programs are constructed, are called *flat automata*, since they cannot be decomposed (they contain no application of operators). They are the direct representation of LFSMs, with the following simplified notation: only positive (i.e., present) events appear in triggers; all other events are considered as absent. Moreover, labels are "concrete" and could involve valued datas over basic data types (boolean, integer,..). The concrete syntax of label events is detailed in [23]. The language is generated by the following grammar (where F is a flat automaton, s a state name, and Y a set of events):

$$P ::= F \mid F[P/s] \mid P \| P \mid P_{|_Y}$$

Parallel composition ($P \| Q$) is a symmetric operator which behaves as the synchronous product of its operands where labels are unioned. *Hierarchical composition* ($F[P/s]$) corresponds to the possibility for a state in an automaton to be refined by a behavioral (sub) program. This operation is able to express preemption, exceptions, and normal termination of sub-programs. *Scoping* ($P_{|_Y}$) where P is a program and Y a set of *local* events, makes it possible to restrict the scope of some events. Indeed, when refining a state by combining hierarchical and parallel composition, it may be useful to send events from one branch of the parallel com-

position to the other(s), without these events being globally visible. This operation can be seen as encapsulation: local events that fired a transition must be emitted in their scope; they cannot come from the surrounding environment.

The language offers syntactic means to build programs that reflect the behavior of components. Nevertheless, the soundness of the approach requires a clear definition of the relationship between behavioral programs and their mathematical representation as LFSMs (section 3.2). Let $\mathcal{B}$ denote the set of behavioral programs and $\mathcal{M}$ the set of LFSMs. We aim at defining a semantic function to give a meaning to each behavioral program. We want this function to be stable with respect to the previously defined operators (parallel composition, hierarchical composition, and scoping). However, the scoping operator semantics is computed from the semantics of its body, where some events are hidden. Doing that, we are not sure that the resulting model is still deterministic and reactive.

Programs whose semantics is neither deterministic nor reactive, or which have a sub-program whose semantics is not either, are considered as incorrect. Following [17] we introduce a specific value $\perp$ to express the semantics of incorrect programs. We denote $\mathcal{M}^{\perp} = \mathcal{M} \cup \{\perp\}$. In order to express the semantics of all behavioral programs (included incorrect ones), we introduce a specific state named ERROR and we denote $\mathcal{M}_{\text{error}}$ the set of LFSMs whose state space may contain ERROR. We have thus to define two semantic functions, first a complete one $\mathcal{S}_C : \mathcal{B} \longrightarrow \mathcal{M}_{\text{error}}$ and second the actual one $\mathcal{S} : \mathcal{B} \longrightarrow \mathcal{M}^{\perp}$ that will discard incorrect behavioral programs:

$$\mathcal{S}(\mathcal{P}) = \begin{cases} \mathcal{S}_C(P) & \text{iff ERROR not reachable in } \mathcal{S}_C(P) \\ \perp & \text{otherwise} \end{cases}$$

Complete Semantic Function Definition

Basic flat automata are deterministic and reactive by construction and their interpretation through $\mathcal{S}_C$ does not lead to an error (i.e. state ERROR is not reachable). The parallel composition operator is the synchronous product of its operands, with a suitable definition, it preserves determinism and reactivity. As said in [17], the only operator that can raise an error is the scoping operator.

$\mathcal{S}_C$ is structurally defined over the syntax of the language. Let P and Q be two behavioral programs, with $\mathcal{S}_C(P) = (S_P, s_0^P, T_P, A_P)$ and $\mathcal{S}_C(Q) = (S_Q, s_0^Q, T_Q, A_Q)$:

• For a *flat automaton* F, $\mathcal{S}_C(F)$ is a LFSM with the same set of states, the same initial state, and the same alphabet as F. Its transition relation T_F is the one of F where each trigger has been completed with the test of absence of all input events that the corresponding trigger of F does not contain. This satisfies the trigger completeness condition for LFSMs. Moreover, a data abstraction is performed on concrete valued data involved in labels to get LFSM abstract labels (see [23] for a formal definition of abstraction).

• For *parallel composition*, $\mathcal{S}_C(P\|Q)$ is $(S_P \boxtimes S_Q, (s_0^P, s_0^Q), T_{P\|Q}, A_P \cup A_Q)$ where $S_P \boxtimes S_Q$ is $S_P \times s_Q$ when ERROR $\notin S_P$ and ERROR $\notin S_Q$. Otherwise,

$S_P \boxtimes S_Q$ is $(S_P - \{\text{ERROR}\}) \times (S_Q - \{\text{ERROR}\}) \cup \{\text{ERROR}\}$.

The transition relation $T_{P\|Q}$ is defined by the following rules Par_1, Par_2 and Par_3, with respect to the equivalence: $(s, \text{ERROR}) = (\text{ERROR}, s') = \text{ERROR}$.

$$\frac{s_1 \xrightarrow{i_P/o_P} s'_1 \in T_P, s_2 \xrightarrow{i_Q/o_Q} s'_2 \in T_Q, i_P/o_P \oplus i_Q/o_Q \neq undef}{(s_1, s_2) \xrightarrow{i_P/o_P \oplus i_Q/o_Q} (s'_1, s'_2) \in T_{P\|Q}} \quad (Par1)$$

$$\frac{s_1 \xrightarrow{i_P/o_P} s'_1 \in T_P, s_2 \in S_Q}{(s_1, s_2) \xrightarrow{i_P/o_P} (s'_1, s_2) \in T_{P\|Q}} \quad (Par2) \qquad \frac{s_1 \in S_P, s_2 \xrightarrow{i_Q/o_Q} s'_2 \in T_Q,}{(s_1, s_2) \xrightarrow{i_Q/o_Q} (s_1, s'_2) \in T_{P\|Q}} \quad (Par3)$$

Rule Par_1 characterizes the synchronous hypothesis which allows the simultaneity of triggers. Here, the label of the resulting transition is the $\oplus$ of the respective label of each operands: the $\oplus$ operation is basically the union of trigger and output sets respectively, with the additional property that determinism is preserved and labels are well-formed. Concerning preservation of determinism, let I_P and I_Q be the respective input event sets of $\mathcal{S}_C(P)$ and $\mathcal{S}_C(Q)$. Assume $i_P/o_P \oplus i_Q/o_Q = i/o$ then $i \cap I_P = i_P$ and $i \cap I_Q = i_Q$. Otherwise, the $\oplus$ operation results in $undef$.

• For *hierarchical composition*, $\mathcal{S}_C(P[Q/s])$ is $\mathcal{S}(P)$ where state s in P is refined by $\mathcal{S}(Q)$. The set of states of $\mathcal{S}_C(P[Q/s])$ is of the form $S_P \backslash \{s\} \cup \{s.s'_i | s'_i \in S_Q\}$. If $s = s_0^P$, the initial state of $\mathcal{S}_C(P[Q/s])$ is $s_0^P.s_0^Q$, otherwise it is s_0^P. To ensure error propagation, we consider that $s.\text{ERROR} = \text{ERROR}$. The set of events is $A_P \cup A_Q$ and the transition relation $T_{P[Q/s]}$ is defined by rules Ref_1, Ref_2,Ref_3, and Ref_4:

$$\frac{s \xrightarrow{i_P/o_P} s_2 \in T_P, s'_i \xrightarrow{i'_Q/o'_Q} s'_j \in T_Q}{s.s'_i \xrightarrow{i_P/o_P} s_2 \in T_{P[Q/s]}} \quad (Ref1)$$

$$\frac{s \xrightarrow{i_P/o_P} s_2 \in T_P, s'_i \xrightarrow{i'_Q/o'_Q} s'_j \in T_Q}{s.s'_i \xrightarrow{i_P/o_P \oplus i_Q/o_Q} s_2 \in T_{P[Q/s]}} \quad (Ref2)$$

$$\frac{s'_i \xrightarrow{i'_Q/o'_Q} s'_j \in T_Q, i'^+ \not\subset I_P}{s.s'_i \xrightarrow{i'_Q/o'_Q} s.s'_j \in T_{P[Q/s]}} \quad (Ref3) \qquad \frac{u \xrightarrow{i_P/o_P} u_1 \in T_P, u \neq s}{u \xrightarrow{i_P/o_P} u'_1 \in T_{P[Q/s]}} \quad (Ref4)$$

Both Ref_1 and Ref_2 are applied when a preemption transition can be fired. The preemption of the enclosing state s is done whatever the transitions of Q are. Rule Ref_1 expresses that the internal transition is not fireable (${i'_Q}^+$ does not hold) and only external actions are emitted. On the other hand, Ref_2 applies when the internal transition is fireable (${i'_Q}^+$ holds) and both internal and external actions are simultaneously performed. Rule Ref_3 applies when no preemption transition is

fireable, hence we keep the internal transition. Rule Ref_4 applies when the source state is not the refined state. Two cases may occur: if the target state of the transition in P is the refined state ($u_1 = s$), the target state of the resulting transition is the state corresponding to the initial state of Q in the resulting LFSM ($u'_1 = s.s_0^Q$). Otherwise, it is the target state of the initial transition in P ($u'_1 = u_1$).

• For the *scoping operator*, $\mathcal{S}_C(P_{|_Y})$ is basically $\mathcal{S}_C(P)$ where some transitions are discarded following a scoping principle and where occurrences of local events are hidden in the labels of the remaining transitions.
We define $\mathcal{S}_C(P_{|_Y}) = (S_P \cup \{\text{ERROR}\}, s_0^P, T_{P_{|_Y}}, A_P - Y)$ where $T_{P|_Y}$ is built following the $Sco1$ and $Sco2$ rules. In these rules, $(i_P/o_P)_{|_Y}$ means $(i_P - Y)/(o_P - Y)$.

$$\frac{\begin{array}{c} s \xrightarrow{i_P/o_P} s' \in T_P, \\ \nexists s \xrightarrow{i'_P/o'_P} \text{ERROR} \in T_P, I_P \not\subseteq Y \\ i_P^+ \cap Y \subseteq o_P, \nexists s \xrightarrow{i'_P/o'_P} s'_1 \in T_P \text{ with } i_P^+ \cap Y = {i'_P}^+ \cap Y \end{array}}{s \xrightarrow{(i_P/o_P)_{|_Y}} s' \in T_{P_{|_Y}}} \quad (Sco1)$$

$$\frac{\begin{array}{c} s \xrightarrow{i_P/o_P} s' \in T_P, \\ \left[\begin{array}{c} i_P^+ \cap Y \subseteq o_P \text{ and } \exists s \xrightarrow{i'_P/o'_P} s'_1 \in T_P \text{ and } i_P^+ \cap Y = {i'_P}^+ \cap Y \\ \text{or} \quad \exists s \xrightarrow{i'_P/o'_P} \text{ERROR} \in T_P \quad \text{or} \quad I_P \subseteq Y \end{array}\right] \end{array}}{s \xrightarrow{(i_P/o_P)_{|_Y}} \text{ERROR} \in T_{P_{|_Y}}} \quad (Sco2)$$

Rule $Sco1$ expresses the transition building rule when no errors occur. Let a transition $s \xrightarrow{i_P/o_P} s' \in T_P$, to build a valid transition in $P_{|_Y}$, we require both that there is no transition sourced in s raising an error and there is no other transition sourced in s bearing a label whose positive trigger part is equal to i_P^+ for events not in Y. Indeed, this last property preserves determinism.

Rule $Sco2$ expresses how an error is propagated, when a transition in T_P raises itself an error or when determinism or reactivity are not preserved. Determinism can fail when discarded local events leads to two transitions outgoing from the same state and bearing the same trigger. Reactivity is not preserved when the input event set of $\mathcal{S}_C(P)$ is included in the set Y of local events.

As shown in [17], for each operator, error detection and propagation is correctly done.

The following theorem expresses that relation $\preceq$ is a congruence with respect to

the language operators. The proof is out of the scope of the paper [6] and is obtained by explicit construction of the preorder relation.

Theorem 3.1 *Let P, Q_1 and Q_2 be correct behavioral programs such that $\mathcal{S}(Q_1) \preceq \mathcal{S}(Q_2)$ and both P, Q_1 and P, Q_2 have disjoint outputs; the following holds:*
$\mathcal{S}(P[Q_1/s]) \preceq \mathcal{S}(P[Q_2/s])$
$\mathcal{S}(P \| Q_1) \preceq \mathcal{S}(P \| Q_2)$
$\mathcal{S}(Q_{1_{|Y}}) \preceq \mathcal{S}(Q_{2_{|Y}})$

4 Compositional and Modular Verification

To perform model checking of behavioral programs we need a modular and incremental way to verify such programs using their natural structure: properties of a whole program can be deduced from properties of its sub-programs. Scaling up to large applications relies on this property. This makes it possible to deal with highly complex global behaviors, provided that they result from composing elementary behaviors that can be verified, modified, and understood incrementally. In particular it makes it possible to perform modular verification using some form of temporal logics.

Temporal logics are formal languages to express properties of discrete logical time systems. In these logics, a formula may specify that a particular event will *eventually* occur or will *never* happen. The formal system we consider ($\forall CTL^*$) [6] is based on first-order logics, augmented with *temporal operators* that make it possible to express properties holding for a given state, for the next state (operator **X**), eventually for a future state (**F**), for all future states (**G**), or that a property remains true until some condition becomes true (**U**). One can also express that a property holds for all the paths starting in a given state ($\forall$). For efficiency reasons, $\forall CTL^*$ does not introduce the existential path quantifier.

Following Clarke et al. [6], $\forall CTL^*$ is interpreted over *Kripke structures* in order to get a sound definition of formulae satisfaction. Such structures belong to the family of finite state machines. They possess a preorder relation ($\preceq_{\mathcal{K}}$) that preserves $\forall CTL^*$ formulae: let K_1 and K_2 be two Kripke structures such that $K_1 \preceq_{\mathcal{K}} K_2$, then $K_2 \models \phi \Rightarrow K_1 \models \phi, \phi \in \forall CTL^*$.

Relying on these results, we associate a Kripke structure ($\mathcal{K}(M)$) with each LFSM (M) and we extend the notion of satisfaction of temporal logic formulae to behavioral programs: let P a behavioral program, $P \models \phi$ means that $\mathcal{K}(\mathcal{S}(P)) \models \phi$.

4.1 Compositional Verification

As already mentioned, our major requirement for model-checking is the ability to be compositional. Indeed, the framework we want to check is a large set of rather

[6] The interested reader can find the details of this proof and of all the forthcoming ones in our technical report [23]

small entities, as it is often the case. Hence, we need a compositional verification process and thus we prove the following theorem:

Theorem 4.1 *Let P and Q two behavioral programs and ψ a $\forall CTL^*$ formula:*

if $P \models \psi$ then $P[Q/s] \models \psi$.

if $P \models \psi$ then $(P \parallel Q) \models \psi$.

Sketch of the proof

To prove theorem 4.1, we rely on Clark et al. results concerning preorders and $\forall CTL^*$ preservation through "simulation" in Kripke structures context. Unfortunately, the substitution preorder we defined is not the preorder defined by Clarke et al.: let M and M' be two LFSMs, $M' \preceq M$ does not imply $\mathcal{K}(M') \preceq_{\mathcal{K}} \mathcal{K}(M)$. Thus, we search for a preorder relation $\preceq_E$ over LFSMs so that the substitution preorder and the Kripke structures preorder are compatible.

In practice, we define this preorder as follows. Let M and M' be two LFSMs, $M' \preceq_E M \Leftrightarrow A_M \subseteq A_{M'} \wedge M \; \mathcal{E}_{Sim} \; (M' \backslash A_M)$ where $\mathcal{E}_{Sim}$ is a *simulation equivalence*. $\mathcal{E}_{Sim}$ is defined as follows: for two LFSMs M_1 and M_2, $M_1 \mathcal{E}_{Sim} M_2$ if and only if there is a pair (R^0_{Sim}, R^1_{Sim}) of simulation relations such that $M_1 R^0_{Sim} M_2$ and $M_2 R^1_{Sim} M_1$.

Intuitively, if M is the LSFM model of a behavioral program P and M' the LFSM of its derivative P', then the substitutability principle we want to respect implies that M simulates $M' \backslash A_M$ in Milner's sense. Conversely, to preserve $\forall CTL^*$ properties through derivation, we need to ensure that $M' \backslash A_M$ simulates M, hence the requirement for two simulation relations.

Then to prove theorem 4.1 we need the two following propositions:

Proposition 4.2 *Let M and M' be two LFSMs then*

$M' \preceq_E M \Rightarrow \mathcal{K}(M' \backslash A_M) \preceq_K \mathcal{K}(M)$.

Proposition 4.3 *Let P and Q be two behavioral programs.*

(i) $\mathcal{S}(P[Q/s]) \preceq_E \mathcal{S}(P)$

(ii) $\mathcal{S}(P \| Q) \preceq_E \mathcal{S}(P)$

Theorem 4.1 has an important consequence: it allows bottom-up verification. Properties are stable with respect to the language operators, thus a property proved for a sub-program holds for the overall program.

4.2 Modular Verification

Dually, a top-down approach similar to assume-guarantee methods [6] is possible. In such methods a property is decomposed into sub-properties according to the structural decomposition of the system: when all sub-properties are valid, their conjunction must imply the global property. Kripke structures support assume-guarantee methods. In [6], the composition of two Kripke structures (denoted $\|_K$) is defined and a sound assume-guarantee method can be expressed.

To extend the assume-guarantee methods to behavioral programs we rely both on proposition 4.2 and on the following theorem:

Theorem 4.4 *Let P and Q be two behavioral programs, then $\mathcal{K}(\mathcal{S}(P\|Q))$ is isomorphic to $\mathcal{K}(\mathcal{S}(P))\|_K\mathcal{K}(\mathcal{S}(Q))$*

To prove this theorem we first show that there is a bijective mapping between the respective set of states of $\mathcal{K}(\mathcal{S}(P\|Q))$ and $\mathcal{K}(\mathcal{S}(P))\|_K\mathcal{K}(\mathcal{S}(Q))$. Then, we prove that the respective initial states of these two Kripke structures map each other. Finally, we check that each transition in the transition relation of $\mathcal{K}(\mathcal{S}(P\|Q))$ is also a transition in the transition relation of $\mathcal{K}(\mathcal{S}(P))\|_K\mathcal{K}(\mathcal{S}(Q))$ with respect to the bijective mapping between state spaces, and conversely. □

The assume-guarantee reasoning can be applied in many different ways. For LFSMs, an assume-guarantee scheme is the following: let A be a LFSM representing an assumption, P and Q two behavioral programs:

$$\frac{\mathcal{S}(P) \preceq_E A \qquad A\|Q \models \phi}{P\|Q \models \phi}$$

For each formula ϕ, a specific Kripke structure, the *tableau* of ϕ (denoted $\Gamma[\phi]$) is defined such that for a Kripke structure K, if $K \models \phi$ then $K \preceq_K \Gamma[\phi]$ [6]. The soundness of this assume-guarantee rule is proved as follows.

First $\mathcal{S}(P) \preceq_E A \Rightarrow \mathcal{K}(\mathcal{S}(P)) \preceq_K \mathcal{K}(A)$, applying proposition 4.2. Thus, $\mathcal{K}(\mathcal{S}(P))\|_K\mathcal{K}(\mathcal{S}(Q)) \preceq_K \mathcal{K}(A)\|_K\mathcal{K}(\mathcal{S}(Q))(1)$, applying the Kripke structures compositionality result.

Second, following our definition of satisfiability: $A\|Q \models \phi \Rightarrow \mathcal{K}(\mathcal{S}(A\|Q)) \models \phi$. Thus, applying theorem 4.4 and Kripke structures tableau property, we deduce: $\mathcal{K}(A)\|_K\mathcal{K}(\mathcal{S}(Q)) \preceq_K \Gamma[\phi]$. Then, from (1) and applying the transitivity of preorder, we know that $\mathcal{K}(\mathcal{S}(P))\|_K\mathcal{K}(\mathcal{S}(Q)) \preceq_K \Gamma[\phi]$. Thus, $\mathcal{K}(\mathcal{S}(P))\|_K\mathcal{K}(\mathcal{S}(Q)) \models \phi$. From theorem 4.4, we get $\mathcal{K}(\mathcal{S}(P\|Q)) \models \phi$ and finally, $P\|Q \models \phi$. □

The major drawback of assume-guarantee method is to find the decomposition of the global property we want to prove. As a consequence, we choose this rule scheme which is simpler than the one in [6],because a learning algorithm to compute the assertion set exists [9].

5 Practical Issues

5.1 Application to BLOCKS *Components*

To illustrate our purpose, let us consider the `Rule` class in our BLOCKS framework and its `FuzzyRule` extension. Their UML representations are shown in figure 1. Figure 2 presents the behavioral program for the whole `Rule` class. This program specifies the valid sequences of operations that can be applied to `Rule`

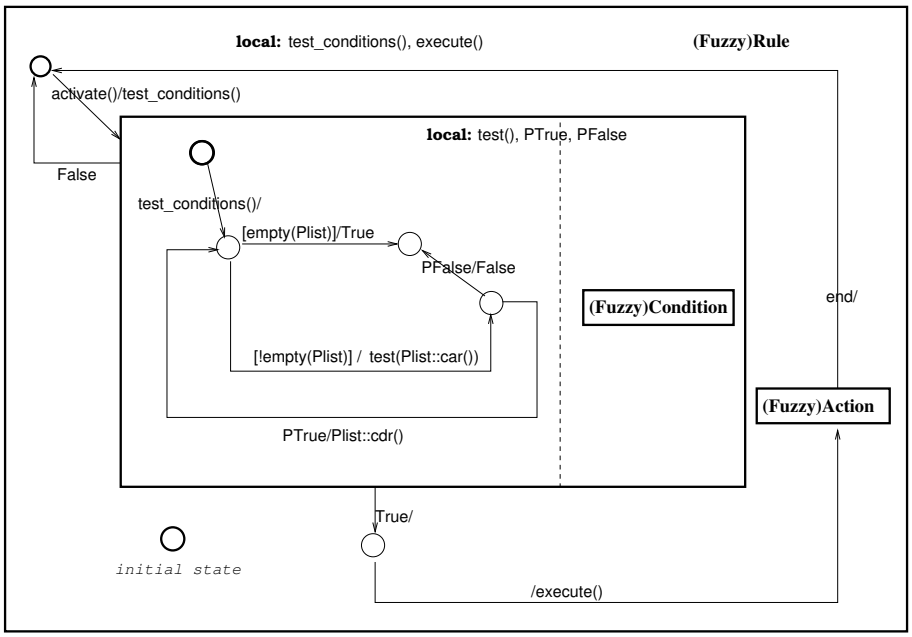


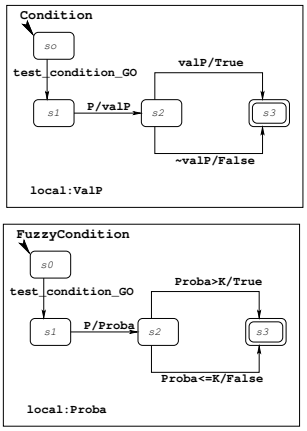
Fig. 2. Rule and FuzzyRule behavior description. *Refinement is expressed by rectangular boxes and encapsulation is specified by keyword* **local**. *The dash line represents the parallel composition. FuzzyRule behavior diagram is Rule diagram where FuzzyCondition diagram is substituted for Condition one (and FuzzyAction for Action one). Valued data in triggers are displayed between square brackets ([]) and will be abstracted in LFSM models.*

instances. The behavioral program associated with class `Condition` is a parallel composition operand and will be launched simultaneously to emitting event `test(Plist::car())`. The behavior of class `Action` is described in a refined state. `FuzzyRule` is obtained by replacing sub-program `Condition` (resp. `Action`) by `FuzzyCondition` (resp.`FuzzyAction`).

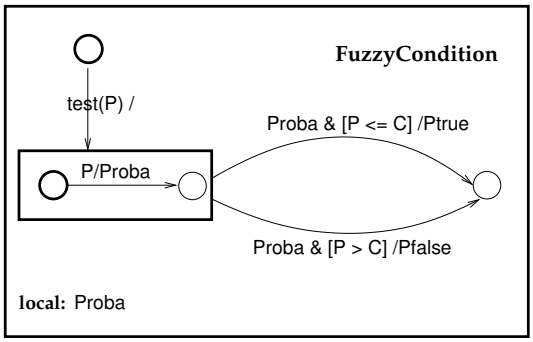
Figure 3(b) presents the expected behavioral program for class `FuzzyCondition` which derives from `Condition`. In particular, a state of `Condition` has been refined to take into account fuzzy values in condition testing.

The behavioral program of `FuzzyRule` has been obtained from the one of `Rule` by redefining the states corresponding to `Condition` and `Action`. To prove that the extension is safe, we must ensure that $\mathcal{S}(\texttt{FuzzyRule}) \preceq \mathcal{S}(\texttt{Rule})$. Particularly, $\mathcal{S}(\texttt{FuzzyCondition}) = \mathcal{S}(\texttt{Condition})$, according to both semantics definition of hierarchical composition and data abstraction to get abstract labels from concrete ones. Particularly, the abstraction on valued boolean expressions belonging to triggers evaluates [P <= C] and [P] to $true$ and [P > C] and [!P] to $false$ (see [23] for a detailed definition of the data abstraction definition). Hence $\mathcal{S}(\texttt{Condition})\ \mathcal{R}_{Sim}$ $\mathcal{S}(\texttt{FuzzyCondition})\backslash A_{\texttt{Condition}}$ and $\mathcal{S}(\texttt{FuzzyCondition}) \preceq \mathcal{S}(\texttt{Condition})$. Similarly, we could prove that $\mathcal{S}(\texttt{FuzzyAction}) \preceq \mathcal{S}(\texttt{Action})$. Thus, applying theorem 3.1, we can deduce that `FuzzyRule` is substitutable for `Rule`. Therefore, the extension of `FuzzyRule` has no influence when `FuzzyRule` is used as `Rule`. As a result, every trace of `Rule` is also a trace for `FuzzyRule`.

But, proving substitutability implies running a possibly heavy algorithm. It re-



(a) Behavioral program of class `Condition`.



(b) Behavioral program of class `FuzzyCondition`. It is similar to `Condition` with a refined state, a local event Proba.

Fig. 3. Behavioral programs of classes `Condition` and `FuzzyCondtion`.

quires to compute the LFSM models of behavioral programs and to perform both restriction and simulation operations. As a consequence, we introduce a set of suitable *design rules* that we apply at *language level* to ensure behavioral substitutability of behavioral programs.

5.2 Design Rules

We have drawn a set of *practical design rules* to avoid systematically applying the *behavioral substitutability algorithm* described in section 3.2. These rules are applicable at BDL level. When a behavioral program P (called the base program in the following) is extended by another behavioral program P' (called the derived program in the following), respecting these rules ensures that we obtain a new deterministic automaton for which behavioral substitutability holds ($P' \preceq P$). These rules correspond to *sufficient* conditions that save us the trouble of a formal proof for each derived program.

To express these practical rules, we shall use the following notations: as before, for a program P, I_P and O_P denote respectively the input and output event sets of P, that is the input and output alphabets of P. For a state s, $\mathcal{I}_s$ (resp. $\mathcal{O}_s$) denotes the preemption trigger set (resp. the preemption action set) of s, that is the set of the triggers (resp. of the actions) of all outgoing transitions of s [7]. We also define the set of triggers and of actions of a program P, $\mathcal{I}_P$ and $\mathcal{O}_P$, as well as the input and output alphabets, I_s and O_s, of a state s: $\mathcal{I}_P = \bigcup\{\mathcal{I}_s | s \in S_P\}$, $\mathcal{O}_P = \bigcup\{\mathcal{O}_s | s \in S_P\}$, $I_s = \bigcup\{t | t \in \mathcal{I}_s\}$ and $O_s = \bigcup\{t | t \in \mathcal{O}_s\}$, where S_P is the set of states of P.

For the time being we have identified eight practical rules, that are listed below.

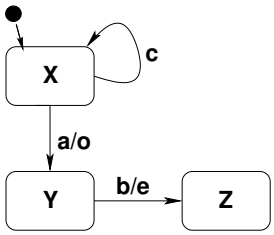
[7] Note that, since triggers and actions are themselves sets of events, $\mathcal{I}_s$ and $\mathcal{O}_s$ are sets of subsets of events, whereas I_s, O_s, I_P, and O_P are simply sets of events.

Some of these rules are illustrated with the base program presented on figure 4(a).

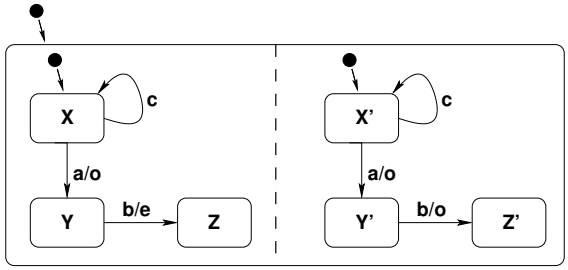
(i) *No modification of the base program structure*: no state, nor transition from the base program should be deleted; in the same way, target and source state of an existing transition should not be changed;

(ii) *Addition of independent transitions*: if a new transition t'/a' is added from an existing state s in the base program, then its trigger should not belong to the preemption trigger set of the state ($t' \notin \mathcal{I}_s$); there are no other constraints on the states and transitions reachable through the new t'/a' transition;

(iii) *Parallel composition with a program with different actions*: it is allowed to compose a new behavioral program Q in parallel with the base program P if the output set of Q is disjoint from the one of P ($O_P \cap O_Q = \emptyset$); figure 4(b) shows a violation of this rule;



(a) Original (base) program



(b) The base program and the program in parallel share the output event o. The original program has the trace $\{a/o, b/e\}$ whereas the new one has $\{a/o, b/oe\}$

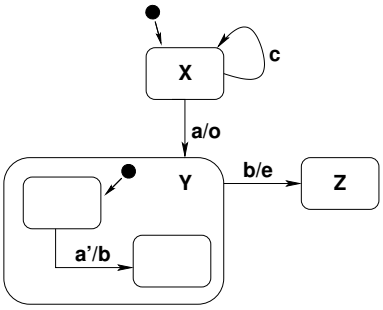
Fig. 4. Counter-example for rule 3

(iv) *Parallel composition with a program with a different initial trigger*: it is allowed to compose a new behavioral program Q in parallel with the base program P if the preemption trigger sets of the initial states of both P and Q are disjoint ($\mathcal{I}_{s_0^P} \cap \mathcal{I}_{s_0^Q} = \emptyset$);

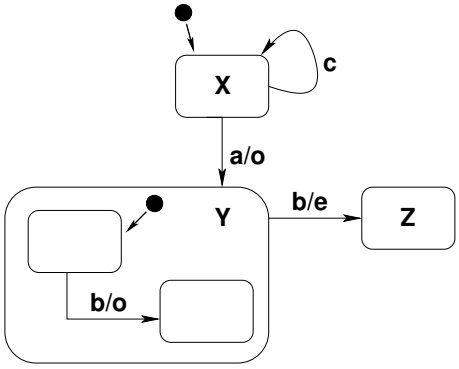
(v) *Parallel composition with a substitutable program*: it is allowed to compose a new behavioral program Q in parallel with the base program P if Q is substitutable for P (that is $Q \preceq P$); this is a consequence of the compositionality property theorems (and of the fact that $P \parallel P$ is P);

(vi) *Hierarchical composition without auto-preemption*: it is allowed to refine a state s of the base program P with a program Q ($P[Q/s]$) if no action of Q is also a trigger of s ($\mathcal{O}_Q \cap \mathcal{I}_s = \emptyset$); this means that the new nested program cannot terminate the state it refines; figure 5(a) shows a violation of this rule;

(vii) *Hierarchical composition with a program with different triggers or actions*: it is allowed to refine a state s of the base program P with a program Q ($P[Q/s]$) if



(a) Counter-example for rule 6: in the base program, state Y was waiting for external event b, whereas, in the refinement, b is emitted from the inside



(b) Counter-example for rule 7(b): when state Y is preempted by event b, the action is e in the original program, whereas it is oe in the new program, and with o belonging also to the base alphabet

Fig. 5. Counter-example for rules 6 and7(b)

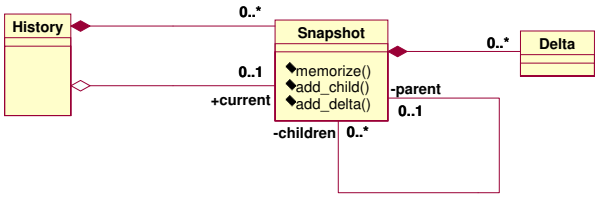
(a) either the set of triggers of Q is disjoint from the preemption trigger set of s ($\mathcal{I}_Q \cap \mathcal{I}_s = \emptyset$),

(b) or the set of triggers of Q intersects the preemption trigger set of s and the output alphabet of Q is disjoint from the output alphabet of s ($O_Q \cap O_s = \emptyset$); this prevents from adding spurious output actions of the base program when preempting both the nested and the enclosing state; figure 5(b) shows a violation of this rule,

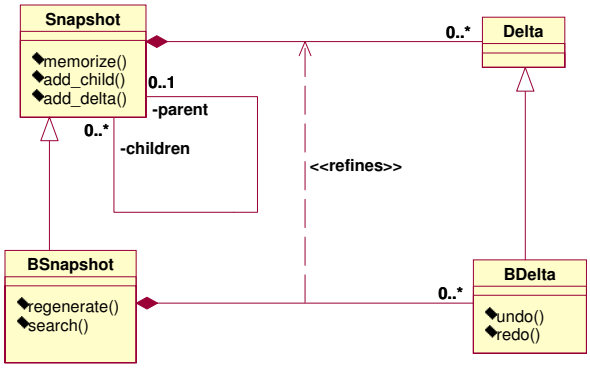
(viii) *No localization of global events*: a global input or output event (be it a trigger or an action) cannot be made local; thus, local events are acceptable if they do not belong to the base program alphabet.

As a matter of example of design rules application, let us examine the problem of the management of execution history in BLOCKS. In our framework, a *history* is composed of several successive *snapshots*, each one gathering the modifications (or *deltas*) to object attributes that have happened since the previous snapshot (that is during an execution step). As shown in figure 6(a), class `Snapshot` memorizes the modification of objects during an execution step in its attached `Delta` set; it displays several operations: memorize the deltas and other contextual information, add a new delta, and add a child snapshot (i.e., close the current step and start a new one).

The `Snapshot` class originally implements a linear history and does not take into account a possible "backtrack". However, in searching activities, a "branching" history is necessary: a common practice is to backtrack to past milestones in order to try a different action or to modify some contextual information and see what happens. To cope with such requirements, the user can introduce a `BSnapshot` class as a derivative of `Snapshot` (figure 6(b)). `BSnapshot` defines two new



(a) Simplified diagram of class `Snapshot`



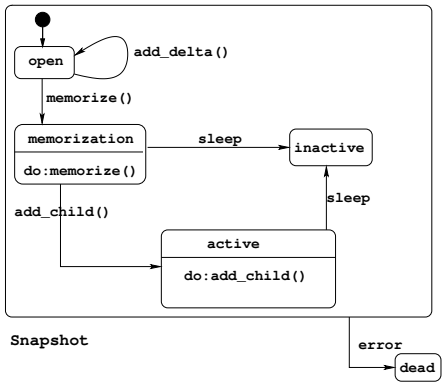
(b) `BSnapshot` extension

Fig. 6. `Snapshot` and `BSnapshot` UML diagrams

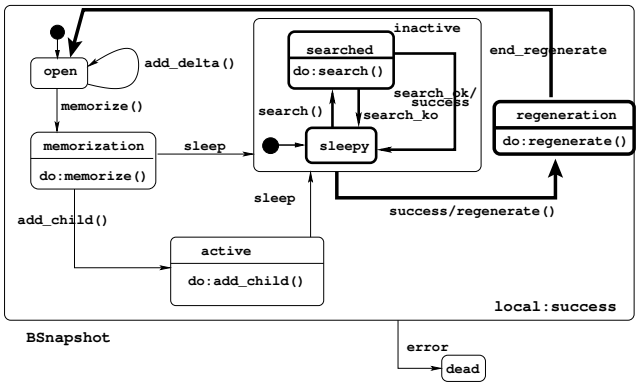
operations: `regenerate` that reestablishes the memorized values and `search` that checks whether a condition was true in a previous state. The regeneration feature implies that deltas have the ability to redo and undo their changes; hence the new class `BDelta` replaces `Delta` (figure 6(b)).

Figure 7(a) presents the behavioral program for the whole `Snapshot` class. This program specifies the valid sequences of operations that can be applied to `Snapshot` instances. Two states correspond to execution of operations (`memorize` and `add_child`); they are to be refined by behavioral programs describing these operations. Figure 7(b) presents the expected behavioral program for class `BSnapshot` which derives from `Snapshot`. In particular, `BSnapshot` necessitates a new operation, `regenerate`, called when backtracking the history (i.e., when `search` returns `success`). The new class has the extra possibilities to search inside a sleeping snapshot and to call `regenerate` when `success` occurs.

The behavioral program of `BSnapshot` has been obtained from the one of `Snapshot` by applying a combination of our design rules. Obviously no state nor transition have been deleted from `Snapshot` (rule 1). The new transition from `inactive` to `regeneration` bears a completely new trigger (rule 2). The program that refines state `inactive` has no trigger belonging to the preemption trigger set of this state (rule (4a)). Finally, the local event `success` was not part of the `Snapshot` program (rule 5). Thus, by construction, `BSnapshot` is substitutable for `Snapshot`; no other



(a) Behavioral program of class `Snapshot`.



(b) Behavioral program of class `BSnapshot`. It is similar to `Snapshot` with a refined inactive state, a local event success, and the possibility of launching regenerate from the inactive state. Restriction $\mathcal{S}(\texttt{BSnapshot})\backslash A_{\texttt{Snapshot}}$ is obtained by removing states and transitions displayed with thick lines.

Fig. 7. Behavioral programs of classes `Snapshot` and `BSnapshot`.

verification is necessary to assert that `BSnapshot` $\preceq$ `Snapshot`. Therefore, the extension of `BSnapshot` has no influence when a `BSnapshot` is used as a `Snapshot`. As a result, every trace of `Snapshot` is also a trace of `BSnapshot`.

Design rules act at language level and they offer an effective means to ensure behavioral substitutability property. Hence, they are useful within an automatic substitutability analyzer.

5.3 Stability of Properties

Continuing with the previous example, to prove that every temporal property in $\forall CTL^*$ true for `Snapshot` is also true for its extension `BSnapshot`, we need to en-

sure that $\mathcal{S}(\texttt{Snapshot}) \preceq_E \mathcal{S}(\texttt{Snapshot})$. But, obviously $\mathcal{S}(\texttt{BSnapshot})\backslash A_{\texttt{Snapshot}} = \mathcal{S}(\texttt{Snapshot})$ and so the proof is immediate.

For instance, suppose we wish to prove the following property: "It is possible to add a child to a snapshot (i.e., to call the `add_child()` operation) only after memorization has been properly done". Looking at the behavioral program (figure 7(a)), this property (referred to as P_{child}) corresponds to the following behavior. When exiting successfully from state `memorization`, if `add_child()` is received, then control enters state `active`. Then label `sleep` leads to the `inactive` state. Otherwise, operation `memorize()` emits `error` which provokes global preemption.

We decompose the P_{child} property into two $\forall CTL^*$ specifications:

$$\forall \mathbf{G}(\texttt{add\_child}()\&\forall \mathbf{G}(\neg \texttt{error})) \Rightarrow \forall \mathbf{F}\texttt{state} = \texttt{inactive}$$

$$\forall \mathbf{G}(\texttt{error} \Rightarrow \forall \mathbf{G}(\neg \texttt{state} = \texttt{inactive}))$$

Intuitively, the first formula corresponds to memorization success: if `add_child()` is received and if no `error` occurs, then state `inactive` is reached. The second formula corresponds to memorization failure: `error` occurred, and state `inactive` will never be reached.

We are developing a tool that allows us to describe BLOCKS component behavior and to automatically achieve proofs of safety properties. In this example, our tool automatically transforms the description of the behavioral program of `Rule` and the above $\forall CTL^*$ specification into inputs acceptable for *NuSMV* model checker [5]. The tool returns that the specification is true for `Rule`. Conversely, if a formula turns out to be false, the diagnosis returned by *NuSMV* is a counter-example. Our tool interprets and displays a user friendly version of this diagnosis for the user.

In frameworks a component (a set/pattern of related classes in our case) usually implements a given service, such as rule management in the example. Components can be extended to satisfy users' needs. Provided that they are small enough to be individually verified by model-checking tools (a sound assumption in most cases), the modularity property allows to verify a complex large scale framework that would not be tractable as a whole.

6 Related Work and Discussion

Modeling component behavior and protocols and ensuring correct use of component frameworks through a proof system is a recent research line. Most approaches concentrate on the composition problem [16,1,7], whereas we are focusing on the substitutability issue.

Many approaches use finite state machine to model the behavior of components. In [8], *interface automata* are defined to model the "temporal" aspects of components. This formalism intends to check the compatibility between components viewed both statically and dynamically. However, the notion of refinement defined for interface automata intends to prove that an implementation meets its

specification; it differs from our substitution preorder, since it addresses a kind of "inverse" problem.

In the field of Software Architecture, most works on modeling behavior [2] address component compatibility and adaptation in a distributed environment. They are often based on process calculi [21,26,22]. In particular, works to formalize and verify Statechart-like languages (UML state diagrams [27] and μ-Charts [10]) use CSP process algebra. These approaches differ from our's. First, UML state diagrams are intrinsically non-deterministic and formalizing them in CSP produces automata larger than synchronous ones; hence, model-checking tends to be more complex. Second, μ-Charts also differ from behavioral programs and their refinement operation is not equivalent to our substitution preorder. Moreover, both works use a model checker based on CSP refinement, not well-suited to verify temporal logic properties. On another hand, in [25] the authors rely on synchronous formalism to express models for UML state-machines. They define a language consisting of synchronous transition systems with preorders and they provide a translation of UML state-machines into this language. Hence they get models "much easier to deal with than classical, non deterministic, asynchronous concurrency". This approach allows to compile, optimize and verify distributed software. As we address another kind of problem, this modeling does not fit our purpose.

Some authors put a specific emphasis on the substitutability problem. For instance, in [4], the authors focus on inheritance and extension of behavior, using the π-calculus as formal model. Both consider a distributed environment. They are more general in their objective than we are, although quite similar as far as behavioral description is concerned. In contrast, we restrict to the problem of substitutability in a non-distributed world, because it is what we needed for BLOCKS. Again, this restriction allows us to adopt models more familiar to software developers (UML StateCharts-like), easier to handle (deterministic systems), efficient for formal analysis (model-checking and simulation), and for which there exist effective algorithms and tools. The Synchronous Paradigm offers good properties and tools in such a context. This is why we could use it as the foundation of our model.

In the domain of modular model-checking, some authors address the problem of modular verification of Argos programs [18]. Their approach is similar to ours, but to establish their results, they relies on a cast of Argos programs into Boolean Automata. Indeed Boolean Automata and Kripke structures are very close models but we prefer to translate BDL programs into Kripke structures and to prove that our translation preserves verification results. Doing that, we benefit from all model-checking techniques available for Kripke structures.

7 Conclusion and Perspectives

The work described in this paper is derived from our experience in providing support for correct use of a framework. We first adapted framework technology to the design of knowledge-based system engines and observed a significant gain in development time. While performing these extensions, we realized the need to for-

malize and verify component protocols, especially when dealing with subtyping. The corresponding formalism, the topic of this paper, has been developed in parallel with the engines. As a consequence of this initial work, developing formal descriptions of BLOCKS components led us to a better organization of the framework, with an architecture that not only satisfies our design rules but also makes the job easier for the framework user to commit to these rules.

Our behavioral formalism relies on a mathematical model, a specification language, and a semantic mapping from the language to the model. The model supports multiple levels of abstraction, from highly symbolic (just labels) to merely operational (pieces of code); thus users can consider the specification level they need. Whereas this model is a classical synchronous model, it is original in the sense that it can cover both static and dynamic behavioral properties of components. To use our formalism, the framework user has only to describe behavioral programs, by drawing simple StateCharts-like graphs with a provided graphic interface. The user may be to a large extend oblivious of the theoretical foundations of the underlying models and their complexity.

Our aim is to accompany frameworks with several kinds of dedicated tools. Currently, we provide a graphic interface to display existing descriptions and modify them. In the future, the interface will watch the user activity and warn about possible violation of the design rules. Since these rules are only sufficient, it is possible for the user not to apply them or to apply them in such a way that they cannot be clearly identified. To cope with this situation, we will also provide a static substitutability analyzer, based on our model (section 3.2) and a partitioning simulation algorithm.

As already mentioned our notion of substitutability guarantees the stability of interesting (safety) properties during the extension process. Hence, at the user level as well as at the framework one, it may be necessary to automatically verify these properties. To this end, we have chosen formal verification and we prove that usual model checking techniques can be applied in our model. The problem with model checkers is the possible explosion of the state space. But, taking advantage of the structural decomposition of the system allows modular proofs on smaller (sub-)systems, a key for scaling up. This requires a formal model that exhibits the *compositionality property*, which is the case for our model (theorems 4.1). At the present time we have designed a complete interface with *NuSMV*. This tool makes it possible to represent synchronous finite state systems and to analyze specifications expressed in $\forall CTL^*$ temporal logic. It uses both symbolic BDD-based and SAT-based (based on propositional satisfiability) model checking techniques. These techniques solve different classes of problems and therefore can be seen as complementary. First, our description language can be translated into *NuSMV* specifications, and our tool provides also a user friendly way to express the properties the users may want to prove. Second, *NuSMV* diagnosis and return messages are displayed in a readable form: users can browse the hierarchies of behavioral derivations and follow the steps of the proofs. The next step is to implement the substitutability analysis tool.

The model has also a pragmatic outcome: it allows simulation of resulting applications and generation of code, of run-time traces, and of run-time assertions. Indeed the behavioral description is rather abstract and may be interpreted in a variety of ways. In particular, automata and associated labels can be given a code interpretation. The generated code would provide skeletal implementations of operations. This code will be correct, by construction—at least with respect to those properties which have been previously checked. Furthermore, the generated code can also be instrumented to build run-time traces and assertions into components.

Developing such tools is a heavy task. Yet, as frameworks are becoming more popular but also more complex, one cannot hope using them without some kind of active assistance, based on formal modeling of component features and automated support. Our work shows that combining formal techniques issued from different computer science domains can be of practical value to make the use of component frameworks safer and easier.

References

[1] F. Achermann and O. Nierstrasz. Applications = Components + Scripts - A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[3] G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stearling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honor of Robin Milner*. MIT Press, 2000.

[4] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, (41):105–138, 2001.

[5] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model verifier, available at http://nusmv.irst.itc.it.

[6] E. M. Clarke, O.Grumberg, and D.Peled. *Model Checking*. MIT Press, 2000.

[7] J. Costa Seco and L. Caires. A Basic Model of Typed Components. In Elisa Bertino, editor, *ECOOP 2000*, volume 1850 of *LNCS*, pages 108–128. Springer, 2000.

[8] L. de Alfaro and T. A. Henzinger. Interface automata. *Proc. of the Foundation of Soft. Eng.*, 26:109–122, 2001.

[9] D. Giannakopoulo, C. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of the 7th IEEE Int. Conf. on Auto.Soft. Eng.* IEEE Computer Society Press, 2002.

[10] D. Goldson. Formal Verification of μ-Charts. In *Proc. of the 9th Asia-Pacific Soft. Eng. Conf.*, Gold Coast, Australia, 2002. IEEE Computer Society Press.

[11] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[12] D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Transactions on Soft. Eng.*, 28(9):889–903, 2002.

[13] D. Harel and A. Pnueli. On the development of reactive systems. In *NATO, Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*. Springer Verlag, 1985.

[14] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[15] B. Liskov and J. L. Wing. A New Definition of the Subtype Relation. In *ECOOP'93*, volume 707 of *LNCS*, pages 119–141. Springer-Verlag, 1993.

[16] K. Mani Chandy and M. Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, 20(1):7–21, January 2002.

[17] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Composition. *LNCS: Concur*, 630, 1992.

[18] A. Merceron and M. Pinna. Modular Verification of Argos Programs. In *Proc. of the 6th Australasian Conf. on Parallel and Real-Time Systems (Part'99)*. Springer, 1999.

[19] R. Milner. An algebraic definition of simulation between programs. *Proc. Int. Joint Conf. Artificial Intelligence*, pages 481–489, 1971.

[20] S. Moisan, A. Ressouche, and J-P. Rigault. BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems. *Informatica, Special Issue on Component Based Software Development*, 25:501–507, 2001.

[21] O. Nierstrasz. *Object-Oriented Software Composition*, chapter Regular Types for Active Objects, pages 99–121. Prentice-Hall, 1995.

[22] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Soft. Eng.*, 28(11), Nov 2002.

[23] A. Ressouche, S. Moisan, and J.-P. Rigault. A Behavioral Model of Component Frameworks. Technical report, INRIA, December 2003. available at: http://www.inria.fr.

[24] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[25] Y. Wang, J-P. Talpin, A. Benveniste, and P. Le Guernic. A semantics of UML state-machines using synchronous pre-order transition systems. In *Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2000)*. IEEE Press, march 2000.

[26] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.

[27] M. Yong and M. Butler. Towards Formalizing UML State Diagrams. In *Proc. of the 1st Conf. on Soft. Eng. and Formal Methods*, Brisbane, Australia, 2003. IEEE Computer Society Press.