Towards Formalizing Behavorial Substitutability in Component Frameworks

Sabine Moisan, Annie Ressouche, Jean-Paul Rigault

Towards Formalizing Behavioral Substitutability in Component Frameworks

Sabine Moisan & Annie Ressouche
INRIA Sophia Antipolis
2004, route des Lucioles
06902 Sophia Antipolis, France
{Sabine.Moisan,Annie.Ressouche}@inria.fr

Jean-Paul Rigault
I3S Laboratory, Univ. of Nice Sophia Antipolis
and CNRS (UMR 6070)
06902 Sophia Antipolis, France
jpr@essi.fr

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 and a formal language to describe the knowledge about behavior. We rely on a hierarchical model of deterministic finite state-machines. The execution model of these state-machines follows the Synchronous Paradigm. 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 behavorial substitutability. Associated tools may ensure correct (re)use of components, as well as automatic simulation and verification, code generation, and run-time checks.

1. Introduction

Reusability—not only of code, but also of analysis and design models—is mandatory to improve product time to market, software quality, maintenance, and to decrease development cost. The notion of frameworks was introduced as a possible answer to these needs.

A 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 generic classes and their relationships. As reusable entities, classes rapidly appeared as too fine grained. Hence, the notion of component frameworks emerged. According to Szyperski [24] a component is "a unit of [software] composition with contractually specified interfaces and explicit context dependencies...". In the object-oriented approach a component usually corresponds to a collection of interrelated classes providing a logically consistent set of services.

To use a component framework, a developer (or framework user) selects, adapts, and assembles components to build a customized application. Thus *reusing* existing components is the developer's major task. But building on reusability is not straightforward. It implies to understand the nature of the contract between the framework user and the component. The mere specification of a static interface (list of operation signatures) is not sufficient since it misses the information regarding the component *behavior*.

Adding pre- and post-conditions to operations (like in Meyer's *design by contract*) is an interesting improvement. However, contracts express only behavior local to an operation, they do not concern the global valid sequences of operations. The description of such valid sequences is the essential part of what we call the *protocol of use* of a framework. We claim that the explicit description of this protocol is an integral part of the framework, hence the importance of providing models and tools to formalize it, reason about it, and manipulate it.

Our work on formalizing component protocols relies on our experience with a framework for knowledge-based system inference engines, named BLOCKS [19]. BLOCKS's objective is to help designers create new engines and reuse or modify existing ones. It is a set of C++ classes, coming with a behavioral description of their valid sequences of operations, in the form of state-transition diagrams. Such descriptions allowed us to prove invariant properties of the framework, using model-checking techniques. As with other frameworks, the developer adapts BLOCKS classes essentially through subtyping (more exactly, class derivation used as subtyping). 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 class level. Thus the notion of *behavioral substitutability* is central to such a correct use of the frame-

work. To this end we elaborated a formal model of behavioral substitutability, where safety properties are preserved during subtyping. We then laid design rules on top of it. Our aim is to propose a verification algorithm as well as practical design rules to ensure sound framework adaptation.

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 illustrates our approach on examples and shows how safety properties can be proved. Finally, section 5 discusses some issues about this approach and relates it to similar work.

2. Target Framework Characteristics

2.1. Notion of Components

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].

As a matter of example, let us examine the problem of history management in an object-oriented environment. In BLOCKS 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). It is a rather general view of history management and any framework with a similar purpose is likely to provide classes similar to those shown in the UML class diagram of figure 1. 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).

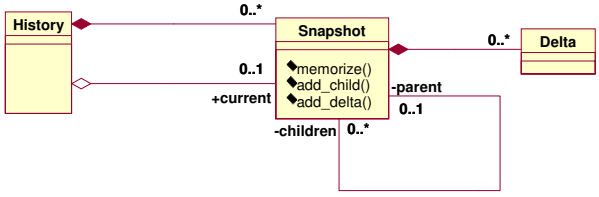


Figure 1. Simplified UML diagram of class `Snapshot`

2.2. Using a Framework

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 generic ones (template classes in C++). 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.

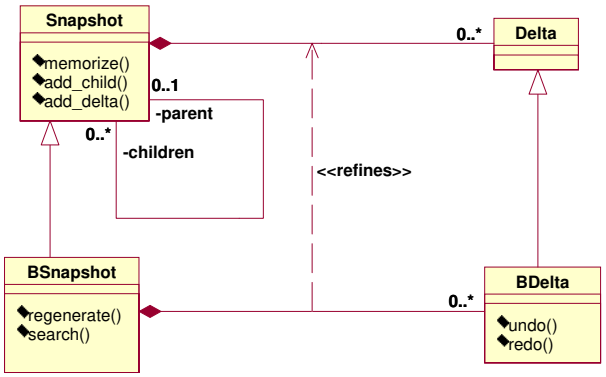


Figure 2. UML class diagram of `BSnapshot`; above, the original classes, below, the derived ones

Let us continue with our example: 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 2). `BSnapshot` defines two new 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 2).

2.3. Protocol(s) of Use

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 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 specification language to describe 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*[1]. In the object-oriented approach, subtyping usually obeys the classical Substitutability Principle [15]. 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* [13]. This is precisely the kind of interpretation we need to enforce the dynamic aspect of framework protocols, since it provides a behaviorwise correct derivation.

1 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.

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 execution models.

To deal with behavioral substitutability, we need behavior representation formalisms: we propose to rely on the family of *synchronous* models [3, 12], which are dedicated to specify *reactive* systems [14], such as found for instance in Real Time systems. 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*: 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. 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.

3.1. Mathematical Model of Behavior

Input/output labeled transition systems [17] are usual mathematical models for synchronous languages. 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 i is the trigger and $o \subseteq O$ the action or output events

set; 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, a label must 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 "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'; $s \xrightarrow{l} s'$ denotes the transition (s, l, s').
There cannot be two transitions leaving the same state and bearing the same trigger. Formally, if there are two transitions from the same state s such that $s \xrightarrow{i_1/o_1} s_1$ and $s \xrightarrow{i_2/o_2} s_2$, with $s_1 \neq s_2$, then $i_1 \neq i_2$.

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.

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 [13]. 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' ($A_{M'}$) is a superset of the alphabet of M (A_M) and every sequence of inputs that is valid [2] 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 Milner's 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, Milner's simulation relation [18] is defined as follows: 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 $(s_0^{M_1}, s_0^{M_2}) \in \mathcal{R}_{sim}$ and

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

LFSMs are deterministic and it turns out that simulation coincides with trace containment relation 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 of 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.

Milner's simulation relation is a preorder and preserves satisfaction of the formulae of a subset of temporal logic, expressive enough for most verification tasks (namely $\forall CTL^*$ [7]). Moreover, this subset has efficient model checking algorithms. Obviously, relation $\preceq$ is also a preorder over LFSMs; we call it *substitution* preorder. We say that M' is substituable 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.

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

3.2. Behavior Description Language

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 applications of operator). 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. 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 composition 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 3.1). Let $\mathcal{B}$ denote the set of behavioral programs and $\mathcal{M}$ the set of LFSMs. We define a *semantic* function $\mathcal{S} : \mathcal{B} \longrightarrow \mathcal{M}$ that is stable with respect to the previously defined operators (parallel composition, hierarchical composition, and scoping). $\mathcal{S}$ is structurally defined over the syntax of the language. Let P and Q be two behavioral programs, with $\mathcal{S}(P) = (S_P, s_0^P, T_P, A_P)$ and $\mathcal{S}(Q) = (S_Q, s_0^Q, T_Q, A_Q)$:

• For a *flat automaton* F, $\mathcal{S}(F)$ 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

• For *parallel composition*, $\mathcal{S}(P \| Q)$ is $(S_P \times S_Q, s_0^P \times s_0^Q, T_{P\|Q}, A_P \cup A_Q)$ where $T_{P\|Q}$ is defined by rules Par_1, Par_2 and Par_3 described in figure 3. Rule Par_1 characterizes the synchronous hypothesis whitch 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 the union of trigger and output sets respectively, ensuring that the result leads to a well-formed label (see [22] for a formal definition).

• For *hierarchical composition*, $\mathcal{S}(P[Q/s])$ is $\mathcal{S}(P)$ where state s in P is refined by $\mathcal{S}(Q)$. The set of states of $\mathcal{S}(P[Q/s])$ 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 $\mathcal{S}(P[Q/s])$ is $s_0^P.s_0^Q$, otherwise it is s_0^P. 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 described in figure 3.

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 *scoping operator*, $\mathcal{S}(P_{|_Y})$ is basically $\mathcal{S}(P)$ 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 $\mathcal{S}(P_{|_Y}) = (S_P, s_0^P, T_{P_{|_Y}}, A_P - Y)$ where $T_{P|_Y}$ is built following the Sco rule described in figure 3. In this rule, $(i_P/o_P)_{|_Y}$ means $(i_P - Y)/(o_P - Y)$.

The following theorem expresses that relation $\preceq$ is a congruence with respect to the language operators. The proof [22] is out of the scope of the paper and is obtained by explicit construction of the preorder relation.

Theorem 1 *Let P, Q_1 and Q_2 be behavioral programs*

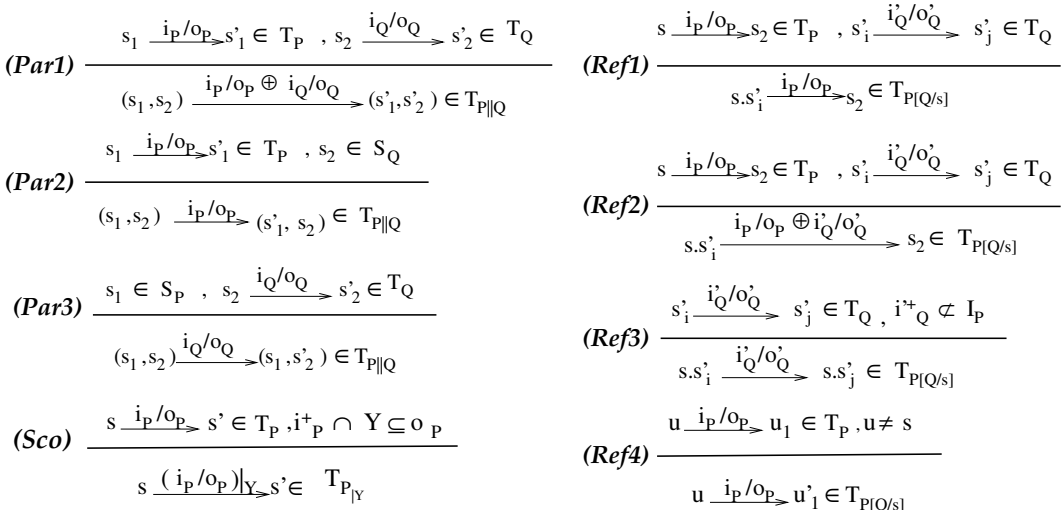


Figure 3. Semantic Rules for Behavior Description Language Operators.

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:

$\mathcal{S}(P[Q_1/s]) \preceq \mathcal{S}(P[Q_2/s])$
$\mathcal{S}(P \| Q_1) \preceq \mathcal{S}(P \| Q_2)$
$\mathcal{S}(Q_{1|_Y}) \preceq \mathcal{S}(Q_{2|_Y})$

3.3. Modular Verification

To perform model checking of behavorial 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, since 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 logic we consider ($\forall CTL^*$) [7] is based on first-order logic, 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. [7], $\forall CTL^*$ 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 ($\mathcal{K}(M)$) whith each LFSM (M) and we extend the notion of satisfaction of temporal logic formulae to behavioral programs: let P a behavorial program, $P \models \phi$ means that $\mathcal{K}(\mathcal{S}(P)) \models \phi$.
The main result of our approach is the following theorem:

Theorem 2 *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 First, we introduce a stronger preorder than the substitution preorder: let M and M' be two LFSM, $M' \preceq_E M \Leftrightarrow A_M \subseteq A_{M'} \wedge M \; \mathcal{E}_{Sim} \; (M' \backslash A_M)$ where $\mathcal{E}_{Sim}$ is a *simulation equivalence* [3].
Second, we prove two propositions (proof detailed in [22]):

Proposition 1 *Let M_1 and M_2 be two LFSMs, then $M_1 \preceq_E M_2 \Rightarrow \mathcal{K}(M_1 \backslash A_{M_2}) \preceq_K \mathcal{K}(M_2)$.*

Proposition 2 *Let P and Q be two behavioral programs.*

1. $\mathcal{S}(P[Q/s]) \preceq_E \mathcal{S}(P)$
2. $\mathcal{S}(P \| Q) \preceq_E \mathcal{S}(P)$

3 For two LFSMs M_1 and M_2, $M_1 \preceq_E M_2$ 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 $M_2 R^1_{Sim} M_1$.

Theorem 2 has an important consequence: it allows a 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.

Dually, a top-down approach similar to assume-guarantee method [7] is possible. In this method a property is decomposed into sub-properties according to the structural decomposition of the system: when all sub-properties are valid, their conjonction must imply the global property. Kripke structures support an assume-guarantee method. In [7], the composition of two Kripke structures (denoted $\|_K$) is defined and the assume-guarantee method corresponds to the following proof scheme [4]. Let K and K' be two Kripke structures, and A a Kripke structure representing a set of assumptions:

$$\frac{\begin{array}{c} K \preceq_K A \\ A\|_K K' \models \phi \\ \mathcal{T}(\phi)\|_K K \models \psi \end{array}}{K\|_K K'\psi}$$

To extend the assume-guarantee method to behavioral programs we first show that the $\preceq_E$ preorder implies the $\preceq_K$ preorder and, second, that parallel composition of behavioral programs is isomorphic to the composition of their associated Kripke structures. The latter property (proved in [22]) is fundamental to ensure that an assume-guarantee mechanism can be applied in our language.

4. Practical Issues

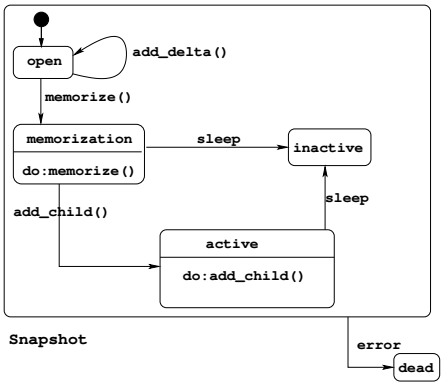
4.1. Design Rules

>From our model, we obtain practical design rules that can be applied at the behavioral language level. When a behavioral program P (called the base program) is extended by another behavioral program P', 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.

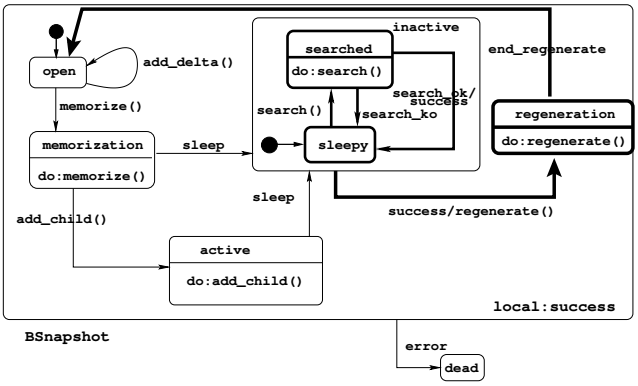
At this time we have identified eight such practical rules. Their formal descriptions can be found in [22]. We briefly list them here: (1) modification of the base program structure is not allowed (no deletion nor modification of transitions or states); (2) it is possible to add trigger-disjoint transitions for a given state; (3) parallel composition is possible with a program with disjoint actions(3a) or different initial trigger(3b) or with a substitutable program(3c); (4) hierarchical composition is possible with a program without auto-preemption(4a) or with disjoint triggers and actions(4b); (5) program top-level events cannot be made local.

4.2. Application to Components

To illustrate our purpose, let us consider the previously mentioned history mechanism (section 2.1). Figure 4(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.



(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.

Figure 4. Behavioral programs of classes `Snapshot` and `BSnapshot`.

Figure 4(b) presents the expected behavioral program for class `BSnapshot` which derives from `Snapshot`.

4 For each $\forall CTL*$ formulae ϕ, a specific Kripke structure: the *tableau* of ϕ denoted by $\mathcal{T}(\phi)$ is defined and $M \models \phi$ if and only if $M \preceq_K \mathcal{T}(\phi)$.

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 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.

In frameworks a component (a set/pattern of related classes in our case) usually implements a given service, such as history 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 trackable as a whole by model-checking tools.

4.3. 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 ensure 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 4(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 Snapshot and the two above specifications into inputs acceptable for *NuSMV*model checker [6]. The tool returns that both specifications are true for Snapshot. Conversly, 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.

5. 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, 8], whereas we are focusing on the substitutability issue.

Many approaches use finite state machine to model the behavior of components. In [9], *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 [23], Counter-Constrained Finite State Machines (*CC-FSM*) are introduced to model component interfaces. *CC-FSMs* are finite state machines extended with constraints on counters related to enumerable ressources. This approach complements ours since it allows to take into account specific properties of resource values. However it cannot describe the whole behavior of components even at an abstract level. Moreover, it is not liable to standard verification tools.

Holmquist et al. [10] introduce the virtual finite state machine (VFSM) formalism to specify the control behavior of a software module. VFSMs are well-suited to reduce the gap between specification and code generation. Thus, VFSMs are just basic FSMs extended "with little more than boolean variables". This low-level dedication is not adapted to substitutability analysis, a specification level issue.

In the field of Software Architecture, most works for modeling behavior [2] address component compatibility and adaptation in a distributed environment. They are often based on process calculi [20, 25, 21]. In particular, works to formalize and verify Statechart-like languages (UML state

diagrams [26] and μ-Charts [11]) use CSP process algebra. These approaches differ from our's. 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. μ-Charts also differ from behavorial programs and their refinement operation is not equivalent to our substitution preorder. Moreover, both works use a model checker based on the notion of CSP refinement, not well-suited to verify temporal logic properties.

Some authors put a specific emphasis on the substitutability problem. For instance [4] proposes static subtype checking relying on Nierstrasz's notion of regular types [20]. As another example, in [5], 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 than ours in their objective, 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.

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 model checking techniques. Indeed, model checkers rely on verification algorithms based on the exploration of a state space and they can be made automatic since tools are available. They are robust and can be made transparent to framework users. The problem with model checkers is the possible explosion of the state space. Fortunately, this problem has become less limiting over the last decade owing to symbolic algorithms. Furthermore, 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 1 and 2).

6. 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. For instance, once the analysis completed, the design of a new planning engine based on the BLOCKS framework took only two months (instead of about two years for a similar former project started from scratch) and more than 90 $\%$ of the code reused existing components [19]. While performing these extensions, we realized the need to formalize 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 the language into 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. Moreover, this model 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 3.1) and a usual partitioning simulation algorithm.

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 M. 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] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and Tool Support for Debugging Object Protocols. In *Proc. of the 8th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, pages 50–59, San Diego, CA, USA, 2000. ACM Press.

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

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: an OpenSource Tool for Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *Proc. of the 14th Int. Conf. on Computer-Aided Verification*, number 2404 in LNCS, pages 359–364, Copenhagen, Danmark, July 2002. Springer-Verlag.

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

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

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

[10] A. Flora-Holmquist, E. Morton, J. O'Grady, and M. Staskauskas. The virtual finite-state machine design and implementation paradigm. Technical report, Lucent Technologies Inc., winter 1997.

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

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

[13] D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Trans. Soft. Eng.*, 28:9:889–903, 2002.

[14] 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.

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

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

[19] 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.

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

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

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

[23] R. Reussner. Counter-constraint finite state machines: A new model for resource-bounded component protocols. In B. Grosky, F. Plasil, and A. Krenek, editors, *Proc. of the 29th Conf. in Current Trends in Theory and Practice of Informatics (SOFSEM 2002), Milovy, Tschechische Republik*, volume 2540 of *LNCS*, pages 20–40, Nov. 2002.

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

[25] 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.

[26] 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.