Enhancing Behavior Protocols with Atomic Actions* Technical report

Jan Kofron

November 28, 2005

Department of Software Engineering
Charles University in Prague
Czech Republic
kofron @ nenya.ms.mff.cuni.cz

Institute of Computer Science
Academy of Sciences of the Czech Republic
Czech Republic
kofron @ cs.cas.cz

Abstract

Verification of software behavior and assuring thus its reliability is among current trends of applications' construction. As an arbitrary application tends to yield a large state space, using software components and a suitable abstraction mechanism enables for model checking the application piecewise, thus in reasonable time. Behavior protocols [1] are one of the component behavior specification platform. During the work on the specification of Airport Internet Access Application implemented in Fractal [3] a need of specifying both component initialization and synchronization of service requests and responses arised. In this article, we introduce a behavior protocols extension called atomic actions that enables for an easy way of specification of these commonly used programming techniques.

1 Introduction

In recent years, the use of software components as building blocks of (distributed) applications has become widely spread technique of software construction. The need of assuring the compatibility of components from various vendors is thus a legitimate requirement. Real-life experiences show that restricting this compatibility relation to the type-compatibility of bounded components' interfaces is simply not sufficient in most cases. Therefore, additional kind of semantic (behavior) specification is needed in order to test the components' behavior compatibility. By applying model checking techniques to compare components' behavior one can obtain a formal-level proof of component behavior compatibility. As model checking techniques applicable on the kind of testing stated above are usually based on exhaustive state space traversing and the size of a state space of a software piece (a software component) is usually too large to be traversed using today's system, a suitable method of software behavior abstraction is to be used.

1.1 Behavior Protocols

Behavior protocols are a method of software component behavior specification [1]. A behavior protocol describes an associated component behavior as a set of sequences of events appearing on component interfaces. The component interfaces are divided into two groups: provided (server) interfaces — *provides* and required (client) interfaces — *requires*. On provides, method calls are accepted and on requires, method calls are emitted.

Having the components' behavior described by behavior protocols, automated checking for components' behavior compatibility can be performed yielding the information about components' composition correctness.

*This work was partially supported by the Czech Academy of Sciences project 1ET400300504 and France Telecom under the external research contract number 46127110.

1.2 Goals and Structure of the Paper

As stated in Section 1, the state space of a software component usually yields a state space being to large for automated (yet lesser for manual) processing. Employing the behavior protocols in non-trivial case studies has showed up that they can be successfully used for software component behavior specification. However, in some cases, the component behavior modeling using behavior protocols resulted in a very complex, unreadable behavior protocol usually yielding unnecessarily large state space due to absence of variable abstraction, and synchronization and exception mechanisms. During the work on behavior specification it has turned out that modeling of the synchronization has the most burning problem. Therefore, we have enriched behavior protocols with atomic actions – special kind of events when a component may perform more than one single event at a time (using one state space transition). Using atomic action has substantially simplified the parts of behavior protocols dealing with synchronization which is very common mechanism in software components.

The paper is structured as follows: In Section 1 a short introduction and problem statement are presented. Section 2 contains an overview of behavior protocols as a platform for component behavior specification. Section 3 describes atomic actions, defines their syntax and semantics, while Section 4 discusses approaches to achieve synchronization and atomicity in other specification languages. Section 5 concludes the paper and proposes directions of possible future research.

2 Behavior Protocols and Component Behavior

2.1 Behavior Protocols

A behavior protocol is an expression describing the behavior of a software component as a set of sequences (traces) of events. An event may be a method call emitting (`!interface.method`↑), method call accepting (`?interface.method`↑), emitting of a return from a method call (`!interface.method`↓) and accepting such a return (`?interface.method`↓). These events can be combined together using various operators (regular and special ones) forming thus regular-like expressions describing the set of possible/allowed traces a component may perform.

As an example consider a component representing a file. It provides one interface that contains five methods to manipulate the file: `open`, `read`, `write`, `close`, and `status`. The supported behavior (i) starts with calling `open`, then an arbitrary interleaving of `read` and `write` follows and finally `close` has to be called; and (ii) allows `status` to be called at anytime (in parallel with (i)). The corresponding behavior protocol takes the form (for simplicity we use shortcut '`methodName`' for '`methodName`↑`; methodName`↓'):

```
(open; (read + write)*; close) | status*
```

2.2 Specifying Behavior of a Software Component

Software components are basic building blocks of today's application. Dividing the application logic into several smaller pieces with both semantics and interface defined more clearly enables for automatized reasoning about software component properties. A way to check for interacting components compatibility is to use of behavior protocols for their behavior specification.

Behavior of each software component [2, 3] is described by its *frame protocol* and expresses the black-box-view behavior of the component. That means that only the events on the component provided and required interfaces are taken into account. Furthermore, in the case of composite component, i.e. a component created by composing several other components, we obtain a grey-box-view behavior where the events of the first-level subcomponents are visible. The behavior at this level is described by the *architecture protocol* of a composite component, which is created by a *consent* [4] composition of the first-level subcomponents' frame protocols. The consent composition is parametrized by a set S of events corresponding to the methods of interfaces bounded between the two components being composed (if there are more than two components they are composed

in a stepwise manner). The result of this operator application is basically a parallel composition of the particular behavior protocols but the resulting traces synchronize on the complementary (in the sense of accepting vs. emitting an event) events from S.

The development of a component application is based on hierarchical composition of primitive components (i.e. components that do not contain other components) creating thus composite components. Errors on a particular level, i.e. among first-level subcomponents of a particular component, are denoted as composition errors, while errors between adjacent composition levels, i.e. between parent and child components, are captured by the behavior compliance.

We distinguish three kinds of composition errors: *bad activity*, *no activity*, and *infinite activity*. Bad activity denotes a situation, when a component tries to emit a call on one of its required interfaces and the component bound to this required interface by its provided interface is not able (according to its frame protocol) to accept such a call. No activity denotes a deadlock and infinite activity a livelock (i.e. some components are working, but no progress to a final state can be made).

As to the behavior compliance, each component (and also a composite one) is supposed to have a frame protocol associated that is in the case of a composite component compared with the architecture protocol. The compliance is defined as the absence of errors in the composition of the architecture protocol and *inverted frame protocol* that is created by inverting (swapping provided and required interfaces) the component frame protocol. As this reduces the problem of detecting compliance errors to a problem of detecting composition errors, only composition is taken into account from now on.

3 Atomic Actions

3.1 Motivation

During the work on behavior specification of components being part of an application test bed for wireless internet access, it has turned out that a kind of atomic component initialization is needed. To clarify the problem core consider the following situation: There are three components `A`, `B` and `C`. The component `A` is a *main* component that is responsible for starting up and coordinating components `B` and `C`. Suppose components `B` and `C` need an initialization. Then, the corresponding protocols of `A`, `B` and `C` would look like following:

$Prot_A$: `!initB; !initC;` $doSomeWork_A$
$Prot_B$: `?initB;` $doSomeWork_B$
$Prot_C$: `?initC;` $doSomeWork_C$

Consent composition of the three protocols above may (in case that $doSomeWork_A$ includes communication between `A` and `B` and `A` and `C`) cause a bad activity error, since e.g. component `B` is able to enter the $doSomeWork_B$ section before the component `C` has finished its initialization. It is easy to see that even if not using the syntactical abbreviation the same problem arises. However, there is still a way to solve this problem using behavior protocols. Although it is very difficult (even maybe impossible for a developer) to construct appropriate behavior protocols from scratch, constructing a finite automaton modeling the composition of the three components and rewriting it back to behavior protocols is a feasible option. The resulting protocol is both long yet unreadable and its future maintenance is certes impossible.

3.2 Overview

Atomic actions are a proposed extension of behavior protocols. In some cases when description of component behavior results in a very long yet unreadable behavior protocol, using atomic actions provides a convenient, much faster yet more natural way of behavior description. To address the problem of the example stated above, a behavior protocol without atomic actions modeling the

behavior of the component `A` that is correct with respect to behavior protocols of the other components (i.e. not yielding composition errors) would look like following:

`(!initB; (!initC | ?`*startOfSomeWork*$_B$`); ?`*startOfSomeWork*$_C$`; `*restOfSomeWork*$_A$`)`
`+`
`(!initB; !initC; ?`*startOfSomeWork*$_C$`; ?`*startOfSomeWork*$_B$`; `*restOfSomeWork*$_A$`)`

Even if we drop readability and protocol construction issues (being much worse when more components are in game and/or the intercommunication is more complex), the protocol still does not correctly reflect the intended behavior, i.e. the initialization of the whole system precedes its *Work*. The remedy is use of atomic actions (protocol parts enclosed between `'['` and `']'`); the protocol employing atomic actions follows:

`[!initB`$\uparrow$`, !initC`$\uparrow$`]; [?initB`$\downarrow$`, ?initC`$\downarrow$`];` *doSomeWork*$_A$

Obvousily, the using of atomic actions solves both readability and construction difficulties and inconsistence of a component implementation and its behavior protocol.

3.3 Syntax

An atomic action may occur in a behavior protocol at positions where a single event and an abbreviation may. Atomic action starts with `'['` and ends with `']'`. There is a coma-separated list of events (the use of abbreviations is not allowed as their use doesn't make sense here) between `'['` and `']'` which is treated as a set, not a sequence; $[e_1, e_2]$ and $[e_2, e_1]$ are thus the same atomic actions. As an example consider the behavior protocol with atomic actions in Section 3.2.

3.4 Semantics

3.4.1 Informal Description

An atomic action is as an ordinary event executed in a single step. Let `PA` and `PB` be two behavior protocols and `ma` be a method on a provided interface `IA1` of the component `A` and `mb` be a method on a provided interface `IB1` of the component `B` corresponding to protocols `PA` and `PB`, respectively. Let there be two required interfaces `IA2` and `IB2` of components `A` and `B`, respectively. Let the interface `IA2` is bound to the interface `IB1` and the interface `IB2` is bound to the interface `IA1`. Let `c` be a method of another required interface `IA3` of component `A` bound to a different component. Then the composition of protocols containing atomic actions using the consent operator will result in a behavior protocol with atomic action again:

`...[?IA1.ma`$\uparrow$`,!IA3.c`$\uparrow$`]...`∇`...!IB2.ma`$\uparrow$`... -> ...[`τ`IB2-IA1.ma`$\uparrow$`,!IA3.c`$\uparrow$`]...`

Hence, the event inside of an atomic action is transformed by the consent operator to an internal event just like events outside atomic actions are. We say the event has been bound.

To preserve the associativity of the consent operator, consent composition of behavior protocols needs to be redefined from [4]. Consider two behavior protocols not containing atomic actions not bound each to the other:

Prot$_A$: `!I1.m1` (an abbreviation for `!I1.m1`$\uparrow$`; ?I1.m1`$\downarrow$)
Prot$_B$: `?I2.m2` (an abbreviation for `?I2.m2`$\uparrow$`; !I2.m2`$\downarrow$)

The consent composition of these protocols yields a behavior protocol (generating no error traces) containing atomic actions:

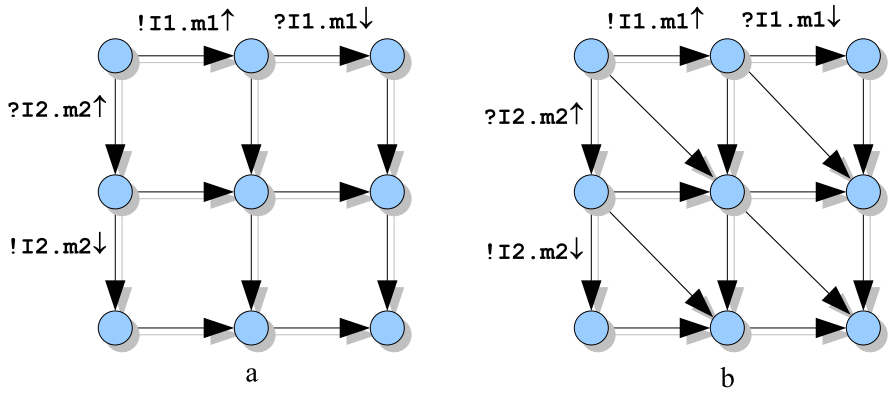

Figure 1: State space generated by consent composition using (a) former semantics and (b) extended semantics (with atomic actions)

$Prot_{A \nabla B}$:
```
(!I1.m1↑; ?I1.m1↓; ?I2.m2↑; !I2.m2↓) +
(!I1.m1↑; [?I1.m1↓, ?I2.m2↑]; !I2.m2↓) +
(!I1.m1↑; ?I2.m2↑; ?I1.m1↓; !I2.m2↓) +
(!I1.m1↑; ?I2.m2↑; [?I1.m1↓, !I2.m2↓]) +
(!I1.m1↑; ?I2.m2↑; !I2.m2↓; ?I1.m1↓) +
([!I1.m1↑, ?I2.m2↑]; ?I1.m1↓; !I2.m2↓) +
([!I1.m1↑, ?I2.m2↑]; [?I1.m1↓, !I2.m2↓]) +
([!I1.m1↑, ?I2.m2↑]; !I2.m2↓; ?I1.m1↓) +
(?I2.m2↑; !I1.m1↑; ?I1.m1↓; !I2.m2↓) +
(?I2.m2↑; !I1.m1↑; [?I1.m1↓, !I2.m2↓]) +
(?I2.m2↑; !I1.m1↑; !I2.m2↓; ?I1.m1↓) +
(?I2.m2↑; [!I1.m1↑, !I2.m2↓]; ?I1.m1↓) +
(?I2.m2↑; !I2.m2↓; !I1.m1↑; ?I1.m1↓)
```

The state space of a corresponding finite automata is shown in Figure 1b. For comparison with the state space generated by an automaton corresponding to the previous version consider the Figure 1a.

3.4.2 Formal Definition

A formal definition of consent operator extended with atomic action is done in the context of the finite automata theory. As the special protocol operators, such as and-parallel, do not break the regularity of the language generate by behavior protocols [4], in this sense, the definition is correct.

Definition 3.1 (The consent operator extended by atomic actions):
Let us denote the set of traces specified by a protocol P by L(P) and the set of traces accepted by a finite automaton A by L(A). Let P, Q be group protocols specifying behavior of component groups G_1, G_2. As justified in [4], L(P), L(Q) are regular languages, and, therefore, there exist nondeterministic finite automata A, B s.t. L(P) = L(A), L(Q) = L(B) and the number of the states of both A, B is minimal. Let S be a set of events on bindings between the groups G_1, G_2, and U be the set of events on unbound interfaces in G_1 and G_2, $S \cap U = \{\}$ ($\{\}$ denotes the empty set). Furhtermore, let us denote $\overline{e}$ the complement event to e, i.e. $\overline{!e}$ is ?e and vice versa.

We define the semantics of the extended consent operator by construction of a (nondeterministic) finite automaton D s.t. L(D) = L(P $\nabla_{S,U}$ Q). The construction is done in four steps: In the first step a (nondeterministic) automaton C is constructed, which accepts all the traces reflecting correct communication between G_1 and G_2 (i.e. the traces containing no composition errors), and

also the traces with bad activity, no activity, unbound requires and bad atomic activity (caused by an atomic action) errors, while the third and the fourth steps defines the behavior for cases with atomic actions. Unbound requires errors are denoted by a token of the form $\epsilon\not{?}e$, where e is the event occurring on an unbound requires interface, while to denote a conditional bad atomic activity and a conditional unbound requires error the tokens $\epsilon?e$ and $\epsilon\not{?}?e$, respectively, are used, where e is an event inside of an atomic action occurring on an bound requires interface.

In the last step, D is constructed in such a way that it accepts all the traces which are accepted by C and in addition it accepts also the traces expressing divergence errors, denoted by the event token $\epsilon\infty$. Basically, if a divergence error occurs in $P \nabla_{S,U} Q$, the groups G_1, G_2 might never stop their communication.

Let $A = (Act, Q_A, q_A, F_A, N_A)$ where Act is the set of all event tokens forming the alphabet, Q_A is the set of states, q_A in Q_A is the initial state, $F_A \subseteq Q_A$ is the set of accepting states and $N_A \subseteq Q_A \times Act \times Q_A$ is the transition relation. In a similar way, $B = (Act, Q_B, q_B, F_B, N_B)$.

Let error tokens be all the tokens of the form ϵe, $\epsilon\oslash$, $\epsilon\infty$, $\epsilon\not{?}e$, $\epsilon?e$, or $\epsilon\not{?}?e$ where e is an event. We define $C = (Act, Q_C, q_C, F_C, N_C)$, where $Q_C = Q_A \times Q_B \cup error$, $q_C = (q_A, q_B)$, $F_C = F_A \times F_B \cup error$. A transition is an element of N_C if and only if it is deduced by one of the following rules:

$((q_1, q_2), ?e, (q_1', q_2)) \in N_C$ *if* $(q_1, ?e, q_1') \in N_A$ *and* $e \notin (S \cup U)$

$((q_1, q_2), ?e, (q_1, q_2')) \in N_C$ *if* $(q_2, ?e, q_2') \in N_B$ *and* $e \notin (S \cup U)$

$((q_1, q_2), !e, (q_1', q_2)) \in N_C$ *if* $(q_1, !e, q_1') \in N_A$ *and* $e \notin (S \cup U)$

$((q_1, q_2), !e, (q_1, q_2')) \in N_C$ *if* $(q_2, !e, q_2') \in N_B$ *and* $e \notin (S \cup U)$

$((q_1, q_2), t, (q_1', q_2)) \in N_C$ *if* $(q_1, t, q_1') \in N_A$, *where* t *is either* τe *or an error token*

$((q_1, q_2), t, (q_1, q_2')) \in N_C$ *if* $(q_2, t, q_2') \in N_B$, *where* t *is either* τe *or an error token*

$((q_1, q_2), \tau e, (q_1', q_2')) \in N_C$ *if* $(q_1, !e, q_1') \in N_A$ *and* $(q_2, ?e, q_2') \in N_B$ *and* $e \in S$

$((q_1, q_2), \tau e, (q_1', q_2')) \in N_C$ *if* $(q_1, ?e, q_1') \in N_A$ *and* $(q_2, !e, q_2') \in N_B$ *and* $e \in S$

$((q_1, q_2), \epsilon e, error) \in N_C$ *if* $(q_1, !e, q_1') \in N_A$ *and there is no* $q_2' \in Q_B$ *s.t.* $(q_2, ?e, q_2') \in N_B$ *or* $(q_2, [?e, \ldots], q_2') \in N_B$ *and* $e \in S$

$((q_1, q_2), \epsilon e, error) \in N_C$ *if* $(q_2, !e, q_2') \in N_B$ *and (there is no* $q_1' \in Q_A$ *s.t.* $(q_1, ?e, q_1') \in N_A$ *or* $(q_1, [?e, \ldots], q_1') \in N_A$ *and* $e \in S$

$((q_1, q_2), \epsilon\oslash, error) \in N_C$ *if (there is no* $(q_1, t, q_1') \notin N_A$ *s.t.* t *is either* $!e$, $?e$, *or an error token) and (there is no* $(q_2, t, q_2') \notin N_B$ *s.t.* t *is either* $!e$, $?e$, *or an error token) and* $(q_1 \notin F_A$ *or* $q_2 \notin F_B)$

$((q_1, q_2), \epsilon\not{?}e, error) \in N_C$ *if* $(q_1, !e, q_1') \in N_A$ *and* $e \in U$

$((q_1, q_2), \epsilon\not{?}e, error) \in N_C$ *if* $(q_2, !e, q_2') \in N_B$ *and* $e \in U$

Furthermore, additional rules defining concerning atomic actions are included within the rule set. As an atomic action contains events from S of various consent operator instances, the topmost consent operator (i.e. the last instance of consent operator applied in the component composition hierarchy) has to be treated in a special way.

The following rules are valid for all consent operator instances:

$((q_1,q_2),[e_1,e_2],(q_1',q_2')) \in N_C$ *if*
$(q_1,e_1,q_1') \in N_A$ *and* $(q_2,e_2,q_2') \in N_B$ *and* $e_1,e_2 \notin (S \cup U)$

$((q_1,q_2),[e_1,e_2,\ldots,e_n],(q_1',q_2')) \in N_C$ *if*
$(q_1,e_1,q_1') \in N_A$ *and* $(q_2,[e_2,\ldots,e_n],q_2') \in N_B$ *and* $\{e_1,\ldots,e_n\} \in Act\backslash(S \cup U)$

$((q_1,q_2),[e_1,e_2,\ldots,e_n],(q_1',q_2')) \in N_C$ *if*
$(q_2,e_2,q_2') \in N_B$ *and* $(q_1,[e_1,\ldots,e_n],q_1') \in N_A$ *and* $\{e_1,\ldots,e_n\} \in Act\backslash(S \cup U)$

$((q_1,q_2),[e_1,e_2,\ldots,e_i,e_{i+1},\ldots,e_n],(q_1',q_2')) \in N_C$ *if*
$(q_1,[e_1,\ldots,e_i],q_1') \in N_A$ *and* $(q_2,[e_{i+1},\ldots,e_n],q_2') \in N_B$ *and* $\{e_1,\ldots,e_n\} \in Act\backslash(S \cup U)$

$((q_1,q_2),[e_1,e_2,\ldots,\tau e_i,\ldots,\tau e_j,\ldots,e_n],(q_1',q_2')) \in N_C$ *if*
$\{e_1,e_2,\ldots,e_{i-1},e_{j+1},\ldots,e_n\} \subseteq Act\backslash(S \cup U)$ *and*
$\{e_i,\ldots,e_j\} \subseteq S$ *and*
$(q_1,[e_1,\ldots,e_j],q_1') \in N_A$ *and*
$(q_2,[\overline{e_i},\ldots,\overline{e_j},e_{j+1},\ldots,e_n],q_2') \in N_B$

$((q_1,q_2),[\epsilon!e_1,\ldots,\epsilon!e_i,\ldots,\epsilon!e_j,\tau e_{j+1},\ldots,\tau e_k,!e_{k+1},\ldots,!e_l,\ldots,!e_n],error) \in N_C$ *if*
$(q_1,[\epsilon?e_1,\ldots,\epsilon?e_i,e_{j+1},\ldots,!e_k,!e_{k+1},\ldots,!e_l],q_1') \in N_A$ *and*
$(q_2,[\epsilon?e_{i+1},\ldots,\epsilon?e_j,\overline{e_{j+1}},\ldots,\overline{e_k},!e_{l+1},\ldots,!e_n],q_2') \in N_B$ *and*
$\{e_{j+1},\ldots,e_k\} \subseteq S, \{e_1,\ldots,e_j,!e_{k+1},\ldots,!e_n\} \subseteq Act\backslash(S \cup U)$,
$i,j \geq 0, i+j \geq 1, i \leq j \leq k \leq l \leq n$

$((q_1,q_2),[\cancel{\epsilon}e_1,\ldots,\cancel{\epsilon}e_i,\ldots,\cancel{\epsilon}e_j,\tau e_{j+1},\ldots,\tau e_k,!e_{k+1},\ldots,!e_l,\ldots,!e_n],error) \in N_C$ *if*
$(q_1,[\epsilon?e_1,\ldots,\epsilon?e_i,e_{j+1},\ldots,!e_k,!e_{k+1},\ldots,!e_l],q_1') \in N_A$ *and*
$(q_2,[\epsilon?e_{i+1},\ldots,\epsilon?e_j,\overline{e_{j+1}},\ldots,\overline{e_k},!e_{l+1},\ldots,!e_n],q_2') \in N_B$ *and*
$\{e_{j+1},\ldots,e_k\} \subseteq U, \{e_1,\ldots,e_j,!e_{k+1},\ldots,!e_n\} \subseteq Act\backslash(S \cup U)$,
$i,j \geq 0, i+j \geq 1, i \leq j \leq k \leq l \leq n$

The following rules are valid for all the consent operator instances except for the top-most one:

$((q_1,q_2),[?e_1,\ldots,?e_i,\ldots,?e_j,\epsilon?e_{j+1},\ldots,\epsilon?e_k,\ldots,\epsilon?e_l,\tau e_{l+1},\ldots,\tau e_m,e_{m+1},\ldots,e_n,\ldots,e_o]$,
$(q_1',q_2')) \in N_C$ *if*
$(q_1,[?e_1,\ldots,?e_i,!e_{j+1},\ldots,!e_k,e_{l+1},\ldots,e_m,e_{m+1},\ldots,e_n],q_1') \in N_A$ *and*
$(q_2,[?e_{i+1},\ldots,?e_j,!e_{k+1},\ldots,!e_l,\overline{e_{l+1}},\ldots,\overline{e_m},e_{n+1},\ldots,e_o],q_2') \in N_B$ *and*
$\{e_1,\ldots,e_j\} \subseteq Act\backslash(S \cup U), \{e_{l+1},\ldots,e_m\} \subseteq S, \{e_{m+1},\ldots,e_o\} \subseteq Act\backslash(S \cup U)$ *and*
$i \geq 0, j \geq 0, i+j \geq 1$ *and there is no atomic action* $\in N_B$ *containing all of* $?e_{j+1}\ldots?e_m$

$((q_1,q_2),[?e_1,\ldots,?e_i,\ldots,?e_j,\epsilon?e_{j+1},\ldots,\epsilon?e_k,\ldots,\epsilon?e_l,\tau e_{l+1},\ldots,\tau e_m,e_{m+1},\ldots,e_n,\ldots,e_o]$,
$(q_1',q_2')) \in N_C$ *if*
$(q_1,[?e_1,\ldots,?e_i,!e_{j+1},\ldots,!e_k,e_{l+1},\ldots,e_m,e_{m+1},\ldots,e_n],q_1') \in N_B$ *and*
$(q_2,[?e_{i+1},\ldots,?e_j,!e_{k+1},\ldots,!e_l,\overline{e_{l+1}},\ldots,\overline{e_m},e_{n+1},\ldots,e_o],q_2') \in N_A$ *and*
$\{e_1,\ldots,e_j\} \subseteq Act\backslash(S \cup U), \{e_{l+1},\ldots,e_m\} \subseteq S, \{e_{m+1},\ldots,e_o\} \subseteq Act\backslash(S \cup U)$ *and*
$i \geq 0, j \geq 0, i+j \geq 1$ *and there is no atomic action* $\in N_A$ *containing all of* $?e_{j+1}\ldots?e_m$

$((q_1,q_2),[?e_1,\ldots,?e_i,\ldots,?e_j,\cancel{\epsilon}?e_{j+1},\ldots,\cancel{\epsilon}?e_k,\ldots,\cancel{\epsilon}?e_l,\tau e_{l+1},\ldots,\tau e_m,e_{m+1},\ldots,$
$e_n,\ldots,e_o],(q_1',q_2')) \in N_C$ *if*
$(q_1,[?e_1,\ldots,?e_i,!e_{j+1},\ldots,!e_k,e_{l+1},\ldots,e_m,e_{m+1},\ldots,e_n],q_1') \in N_A$ *and*
$(q_2,[?e_{i+1},\ldots,?e_j,!e_{k+1},\ldots,!e_l,\overline{e_{l+1}},\ldots,\overline{e_m},e_{n+1},\ldots,e_o],q_2') \in N_B$ *and*
$\{e_1,\ldots,e_j\} \subseteq Act\backslash(S \cup U), \{e_{l+1},\ldots,e_m\} \subseteq U, \{e_{m+1},\ldots,e_o\} \subseteq Act\backslash(S \cup U)$ *and*
$\{e_{l+1},\ldots,e_m\} \subseteq S$ *and* $i \geq 0, j \geq 0, i+j \geq 1$

$((q_1,q_2),[?e_1,\ldots,?e_i,\ldots,?e_j,\cancel{\epsilon}?e_{j+1},\ldots,\cancel{\epsilon}?e_k,\ldots,\cancel{\epsilon}?e_l,\tau e_{l+1},\ldots,\tau e_m,e_{m+1},\ldots,$

$e_n, \ldots, e_o], (q_1', q_2')) \in N_C$ *if*
$(q_1, [?e_1, \ldots, ?e_i, !e_{j+1}, \ldots, !e_k, e_{l+1}, \ldots, e_m, e_{m+1}, \ldots, e_n], q_1') \in N_B$ *and*
$(q_2, [?e_{i+1}, \ldots, ?e_j, !e_{k+1}, \ldots, !e_l, \overline{e_{l+1}}, \ldots, \overline{e_m}, e_{n+1}, \ldots, e_o], q_2') \in N_A$ *and*
$\{e_1, \ldots, e_j\} \subseteq Act \backslash (S \cup U), \{e_{l+1}, \ldots, e_m\} \subseteq U, \{e_{m+1}, \ldots, e_o\} \subseteq Act \backslash (S \cup U)$ *and*
$\{e_{l+1}, \ldots, e_m\} \subseteq S$ *and* $i \geq 0, j \geq 0, i + j \geq 1$

The following rules are valid for the top-most consent operator instance:

$((q_1, q_2), [?e_1, \ldots, ?e_i, \ldots, ?e_j, \epsilon e_{j+1}, \ldots, \epsilon e_k, \ldots, \epsilon e_l, \tau e_{l+1}, \ldots, \tau e_m, e_{m+1}, \ldots, e_n, \ldots, e_o],$
$error) \in N_C$ *if*
$(q_1, [?e_1, \ldots, ?e_i, !e_{j+1}, \ldots, !e_k, e_{l+1}, \ldots, e_m, e_{m+1}, \ldots, e_n], q_1') \in N_A$ *and*
$(q_2, [?e_{i+1}, \ldots, ?e_j, !e_{k+1}, \ldots, !e_l, \overline{e_{l+1}}, \ldots, \overline{e_m}, e_{n+1}, \ldots, e_o], q_2') \in N_B$ *and*
$\{e_1, \ldots, e_j\} \subseteq Act \backslash (S \cup U), \{e_{l+1}, \ldots, e_m\} \subseteq S, \{e_{m+1}, \ldots, e_o\} \subseteq Act \backslash (S \cup U)$ *and*
$i \geq 0, j \geq 0, i + j \geq 1$ *and there is no atomic action* $\in N_B$ *containing all of* $?e_{j+1} \ldots ?e_m$

$((q_1, q_2), [?e_1, \ldots, ?e_i, \ldots, ?e_j, \epsilon e_{j+1}, \ldots, \epsilon e_k, \ldots, \epsilon e_l, \tau e_{l+1}, \ldots, \tau e_m, e_{m+1}, \ldots, e_n, \ldots, e_o],$
$error) \in N_C$ *if*
$(q_1, [?e_1, \ldots, ?e_i, !e_{j+1}, \ldots, !e_k, e_{l+1}, \ldots, e_m, e_{m+1}, \ldots, e_n], q_1') \in N_B$ *and*
$(q_2, [?e_{i+1}, \ldots, ?e_j, !e_{k+1}, \ldots, !e_l, \overline{e_{l+1}}, \ldots, \overline{e_m}, e_{n+1}, \ldots, e_o], q_2') \in N_A$ *and*
$\{e_1, \ldots, e_j\} \subseteq Act \backslash (S \cup U), \{e_{l+1}, \ldots, e_m\} \subseteq S, \{e_{m+1}, \ldots, e_o\} \subseteq Act \backslash (S \cup U)$ *and*
$i \geq 0, j \geq 0, i + j \geq 1$ *and there is no atomic action* $\in N_A$ *containing all of* $?e_{j+1} \ldots ?e_m$

$((q_1, q_2), [?e_1, \ldots, ?e_i, \ldots, ?e_j, \not{\epsilon} e_{j+1}, \ldots, \not{\epsilon} e_k, \ldots, \not{\epsilon} e_l, \tau e_{l+1}, \ldots, \tau e_m, e_{m+1}, \ldots, e_n, \ldots, e_o],$
$error) \in N_C$ *if*
$(q_1, [?e_1, \ldots, ?e_i, !e_{j+1}, \ldots, !e_k, e_{l+1}, \ldots, e_m, e_{m+1}, \ldots, e_n], q_1') \in N_A$ *and*
$(q_2, [?e_{i+1}, \ldots, ?e_j, !e_{k+1}, \ldots, !e_l, \overline{e_{l+1}}, \ldots, \overline{e_m}, e_{n+1}, \ldots, e_o], q_2') \in N_B$ *and*
$\{e_1, \ldots, e_j\} \subseteq Act \backslash (S \cup U), \{e_{j+1}, \ldots, e_l\}$
$\subseteq U, \{e_{l+1}, \ldots, e_m\} \subseteq S, \{e_{m+1}, \ldots, e_o\} \subseteq Act \backslash (S \cup U)$ *and* $i \geq 0, j \geq 0, i + j \geq 1$

$((q_1, q_2), [?e_1, \ldots, ?e_i, \ldots, ?e_j, \not{\epsilon} e_{j+1}, \ldots, \not{\epsilon} e_k, \ldots, \not{\epsilon} e_l, \tau e_{l+1}, \ldots, \tau e_m, e_{m+1}, \ldots, e_n, \ldots, e_o],$
$error) \in N_C$ *if*
$(q_1, [?e_1, \ldots, ?e_i, !e_{j+1}, \ldots, !e_k, e_{l+1}, \ldots, e_m, e_{m+1}, \ldots, e_n], q_1') \in N_B$ *and*
$(q_2, [?e_{i+1}, \ldots, ?e_j, !e_{k+1}, \ldots, !e_l, \overline{e_{l+1}}, \ldots, \overline{e_m}, e_{n+1}, \ldots, e_o], q_2') \in N_A$ *and*
$\{e_1, \ldots, e_j\} \subseteq Act \backslash (S \cup U), \{e_{j+1}, \ldots, e_l\} \subseteq U, \{e_{l+1}, \ldots, e_m\} \subseteq S,$
$\{e_{m+1}, \ldots, e_o\} \subseteq Act \backslash (S \cup U)$ *and* $i \geq 0, j \geq 0, i + j \geq 1$

We say that a state $q = (q_1, q_2)$ *of the automaton* C *is a divergence state, if the following two conditions are satisfied:*

(i) There exists a cycle (in C*) containing the state* q*, i.e. there exists a sequence* $q_1, ..., q_n \in Q_C$ *s.t.* $q_1 = q, q_n = q$ *and* $(\forall i \in 1, ..., n-1)(\exists$ *event token or an atomic action* $t)((q_i, t, q_{i+1}) \in N_C)$*.*

(ii) From q *no accepting state is reachable, i.e. there exists no sequence* $q_1, ..., q_n \in Q_C$ *s.t.* $q_1 = q, q_n \in F_C$ *and* $(\forall i \in 1, ..., n-1)(\exists$ *event token or an atomic action* $t)((q_i, t, q_{i+1}) \in N_C)$*.*

The resulting automaton is defined as $D = (Act, Q_C, q_C, F_C, N_D)$*, where*

$N_D = N_C \cup \{(q, \epsilon\infty, error) :$ *q is a divergence state*$\}$*.*

4 Related Work

A way for specifying atomic or uninterruptable sequences of operation is certainly necessary once describing behavior of a software piece. Otherwise, the state space growth caused by generating unnecessary interleavings of multiple threads instruction is too large to maintain for automated

reasoning. As the software model checking is in its early stages of research and development, a standard way (a language or even a method) for describing software behavior hasn't been established so far. On the other hand, a method of achieving desired effect (synchronization, uniterruptable sequence) is available in most specification languages.

In Promela [5] — the Spin model checker language — there are two ways to express two kinds of uninterruptable sequences. The first option is the use of `atomic` statement, which embodies a sequence of statements intended to be executed without interruption; however, once the thread executing instruction inside an atomic statement is blocked, other thread is selected for execution. A stronger construct is the `d_step` (stands for *deterministic step*) statement, where blocking of the thread leads to a deadlock and is thus considered as an error. Both constructs, if used correctly, may significantly reduce the resulting state space achieving thus an acceptable checking time requirements. Furthermore, they allow for specification behavior that would be very difficult or impossible in some cases to describe without these constructs.

Java PathFinder [6] is a tool for checking of models written in the Java programming language [7]. It features a special Java class `Verify` to instrument the input code written in Java for reducing the state space and introduces also an easy way for employing of the non-determinism into the Java code. The `Verify` class contains the `beginAtomic` and `endAtomic` methods for denoting code intended to be performed without interruption by other threads. Although the JPF implements very beneficent partial-order-reduction algorithm, the `beginAtomic` and `endAtomic` methods are useful in cases when POR doesn't work well.

Symbolic Model Verifier [8] is a tool for symbolic model checking finite state systems against a specification in CTL. The input language of SMV is a parallel assignment language describing finite Kripke structures. It defines state variables, their initial values and changes in each steps. All the assignments of variables are performed in parallel in each step, modeling of atomicity is thus easy to achieve. On the other hand, one can still model asynchronous systems by declaring multiple processes, whose assignments are interleaved — in each step a process is chosen and one step of this process is performed, while the state variables of other process remain unchanged. SMV allows thus for specification both synchronous and asynchronous systems with the implicit ability of expressing atomicity of more operations.

In process algrebras like CSP [9] and CCS [10], uninterruptable sequences can be easily modeled while exploiting the process synchronization (join operation) and the fact that these formalisms don't require a strict mapping between code and specification allowing thus for creating a coarser-grained abstraction.

5 Conclusion and Future Work

Introducing of the atomic actions to the behavior protocols specification platform has significantly simplified the behavior specification of special behavior constucts like initialization and synchronization. In cases when a specification without atomic actions is not feasible due to enourmous protocol complexity, the use of atomic actions provides a very convenient way. On the other hand, there are situations impossible to be described by a finite behavior protocol without atomic actions. Atomic actions thus extend the expressing power of behavior protocols. They can be also used in situations where one doesn't want to check all possible interleavings of components' behavior knowing that a particular part of the behavior doesn't affect the result of checking for compliance and composition errors.

The future work can be divided into two main areas: (i) further enhancement of behavior protocols as a component behavior specification platform enabling it for use in real-life applications and (ii) on fastening (i.e. looking for both suitable state space representation and its efficient traversing) of the behavior protocols checker implementation.

References

[1] Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002

[2] SOFA — `http://sofa.objectweb.org`

[3] E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, J-B. Stefani: An Open Component Model and Its Support in Java. 7th SIGSOFT International Symposimum on Component-Based Software Engineering (CBSE7), LNCS 3054, Edinburgh, Scotland, May 2004.

[4] Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance and Evolution: Research and Practice 17(5), Sep 2005

[5] Promela language reference — `http://spinroot.com/spin/Man/promela.html`

[6] Java PathFinder — `http://javapathfinder.sourceforge.net`

[7] Java programming language — `http://java.sun.com`

[8] Symbolic Model Verifier — `http://www.cs.cmu.edu/~modelcheck/smv.html`

[9] C.A.R Hoare: Communicating Sequential Processes, Prentice Hall International, London, ISBN: 0131532715, 1985

[10] R. Milner: A Calculus of Communicating Systems, Springer-Verlag New York, Inc., ISBN:0387102353