Behavior Protocols Capturing Errors and Updates*

Jiri Adamek[1], Frantisek Plasil[1,2]

[1]Charles University, Faculty of Mathematics and Physics
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{adamek, plasil}@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz

[2]Academy of Sciences of the Czech Republic
Institute of Computer Science
plasil@cs.cas.cz, http://www.cs.cas.cz

Abstract

We discuss the problem of defining a composition operator in behavior protocols in a way which would reflect false communication of the software components being composed. Here the issue is that the classical way in the ADLs supporting behavior description, such as Wright and TRACTA, is to employ a CSP-like parallel composition which inherently yields only "successful traces", ignoring non-accepted attempts for communication. We show that, resulting from component composition, several types of behavior errors can occur: bad activity, no activity, and divergence. The key idea behind bad activity is that the asymmetry of roles during event exchange typical for real programs should be honored: the caller is considered to be the initiator of the call (callee has only a passive role). In most formal systems, this is not the case. We propose a new composition operator, "consent", reflecting these types of errors by producing erroneous traces. In addition, by using the consent operator, it can be statically determined, whether the atomicity of a dynamic update of a component is implicitly guarantied thanks to the behavior of its current environment.

1 Introduction

1.1 Component models - basic ideas

Components have become an important part of software technologies. The existing component models range from simple, low-level granularity components in industrial standards COM/DCOM [9] and EJB [18], to higher-level granularity component models where Polylith [6], Darwin/Tracta [10], Wright [2] belong to the "classics", while CCM [12] and Fractal [3] are among the recent ones. Except for CCM, the higher-level models support component nesting.

To illustrate the basic idea common to all of these higher-level models, we will use the SOFA component model (SOFtware Aplliances) elaborated in our research team [13, 17, 7]; intending to keep its description here as simple as possible, we refer the reader to [17] for details on the SOFA architecture description language CDL, connection names, etc. Typically, a component is similar to an object but features more interfaces to access the services it provides *(provides interfaces)* and, moreover, it features *requires interfaces* as abstractions to capture references to other components' interfaces. In principle, a provides interface is a list of methods which can be called by clients of the component having reference to the interface, while a requires interface is a list of the methods supposed to be called by the component on the target of the reference represented by this interface. The knowledge of such a reference is reflected as *interface tie* and graphically represented by an arrow heading to the target of the reference (Fig. 1).

In Fig. 1a, the component *Alpha* features the requires interface 1 tied to the provides interface 2 *(binding)* of *Beta*. The label *a* of the arrow expressing the tie indicates that *a* is the method to be called

*The work was partially supported by the Grant Agency of the Academy of Sciences of the Czech Republic (project number A2030902) and by the Grant Agency of the Czech Republic (project number 201/03/0911).

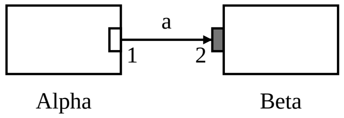
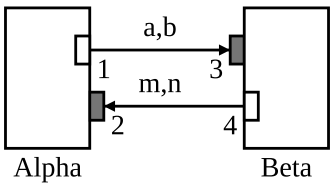
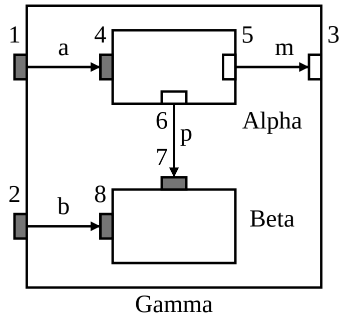
a)		$Prot_{Alpha} = !a \ = !a\uparrow; ?a\downarrow$ $Prot_{Beta} \ = ?a = ?a\uparrow; !a\downarrow$
b)		I) $Prot_{Alpha} = !a\{?m + ?n\}$ $Prot_{Beta} \ = ?a\{!m\} + ?b\{!n\}$ II) $Prot'_{Alpha} = !a\{?m\} + !b\{?n\}$ $Prot'_{Beta} \ = ?a\{!m + !n\} + ?b\{!n\}$
c)		$Prot_{Alpha} \ = ?a;\ !p;\ !m$ $Prot_{Beta} \ = ?b;\ ?p$ I) $Prot_{Gamma} = ?b;\ ?a;\ !m$ II) $Prot'_{Gamma} = ?a;\ ?b;\ !m$

Figure 1: Examples of component systems and behavior protocols

on 2 by *Alpha* via 1. In a similar vein, in Fig. 1b, there are two bindings of *Alpha* and *Beta*. Here, through 1, the component *Alpha* calls a or b on 3, and through 4, *Beta* calls m resp. n on 2 of *Alpha*. In a case of nested components (Fig. 1c), a tie can be as well from a provides interface to a provides interface of a subcomponent *(delegation)*, and from a requires interface to a requires interface of the parent component *(subsuming)*. An example of binding is the tie 6→7, while 1→4 resp. 2→8 illustrate delegation and 5→3 subsuming.[1] All interfaces "on the boundary" of a component form the *frame* of the component. In Fig. 1c, the frame of *Gamma* is formed by the interfaces 1, 2, 3, while the frame of *Alpha* is formed by 4, 5, 6, and the frame of *Beta* by 7 and 8. The internals of a component seen on the first level of nesting form the *architecture* of the component. The architecture of *Gamma* is determined by the frames of *Alpha* and *Beta*, and by all ties among *Alpha*, *Beta*, and *Gamma*. As emphasized in [14, 15], the coexistence of the frame and architecture view of a component is very important for refinement-based component design.

1.2 Describing behavior of components

Frequently, the models formally describing behavior of (not only software) components observe a set of primitive actions/communication events and denote them by labels, *tokens*. Abstracting from the nature of the actions/events, a model is typically based on a transition system T where the behavior of a system of components is captured via the states and transitions of T. Frequently, T allows for reflecting the behavior (at least partially) as a set composed of all the possible/desirable sequences, *traces*, of tokens. For describing behavior of software components, the names of messages, events, resp. method calls involved in the communication among the components typically form a part of the tokens. To the transition systems designed to model software component behavior belong:

CSP [16], employed, e.g., in the Wright architecture description language [2], uses system of recursive equations and inference rules to generate a transition system; given such equations and rules, the states of the corresponding transition system are all the expressions (processes), which can be inferred from the equations by applying the rules.

[1] Many component models do not distinguish among binding, delegation and subsuming; they simply talk about ties (typically called bindings). The approach is specific to SOFA to support evaluation of behavior compliance.

TRACTA/ FSP [5], part of the Darwin ADL [10], uses a system of recursive equations as well, but the set of all allowed operators is restricted so that regularity of traces is guarantied; thus, the equations define a finite automaton accepting the desired traces.

UML [19] defines three packages for describing behavior via diagrams: (1) State machines (and their special case, activity graphs), are based on a classical transition system. (2) Collaborations reflected in the collaboration and sequence diagram concepts (equivalent in principle) are designed to capture a single, "characteristic" trace. On the contrary, (3) use cases are proposed to specify desired scenarios (sets of traces) on the boundary of a (sub)system under design. Here UML does not provide any specific means for specifying a set of scenarios (in addition to collaborations), instead, it concentrates on relationships among use cases (is subordinate, extends/includes, generalization).

Web services flow language (WSFL, [8]) is intended for behavior description of components in a specific settings where "component" is a web service provider. The related transition system represents the desired scenarios of a modeled business process as a graph capturing the desirable ordering resp. potential overlapping of activities (e.g. calls of web services).

In [14] we model the behavior of a component as traces capturing the events (a *request* and a *response* which can compose a method call) on component interfaces. For an event name $a \in EN$, request and response are denoted as $a\!\uparrow$, $a\!\downarrow$ *(events)*. Let the component *Alpha* from Fig. 1a) call the method a of *Beta*. Seen from *Alpha*, the call is written as $!a\!\uparrow; ?a\!\downarrow$ (sequence of *event tokens*), i.e *Alpha* issues (!) a request first and then accepts (?) a response. Seen from *Beta*, the method call can be written as $?a\!\uparrow; !a\!\downarrow$. If a request and response occur inside of a composed component, the corresponding sequence of event tokens takes the form $\tau a\!\uparrow; \tau a\!\downarrow$ *(internal events)*. By convention, the set of all event tokens processed (emitted and/or absorbed) by a component A forms its alphabet $S_A \subseteq ACT$. The *behavior* L_A of a component A is the set of all possible traces produced by A, forming a language $L_A \subseteq (S_A)^* \subseteq Act^*$ (L_A reflects all the possible computations of A). L_A can be approximated, *bounded*, by $L(Prot)$, the regular language generated by a *behavior protocol* $Prot$. Being a regular expression-like, a behavior protocol employs in addition to concatenation (;), alternative (+), and finite sequencing (*) also composition (Π_X, Sect. 1.3), interleaving/shuffle of traces (|), and the abbreviations: $?m = (?m\!\uparrow; !m\!\downarrow)$, $!m = (!m\!\uparrow; ?m\!\downarrow)$, $\tau m = (\tau m\!\uparrow; \tau m\!\downarrow)$, $?m\{P\} = (?m\!\uparrow; P; !m\!\downarrow)$, $!m\{P\} = (!m\!\uparrow; P; ?m\!\downarrow)$, $\tau m\{P\} = (\tau m\!\uparrow; P; \tau m\!\downarrow)$; here, P is a protocol.

To address behavior compliance throughout a hierarchy of component nesting, a frame protocol describes the external communication of a component A at the level of its frame, while an architecture protocol is associated with the architecture of A, capturing also the communication among direct subcomponents of A. Fig. 1b), part I specifies the frame protocols of *Alpha* and *Beta*. The architecture protocol of the architecture composed of *Alpha*, *Beta* is $\tau a\{\tau m\}$ (generated automatically from these frame protocols). As explained in [14], key benefits of such behavior protocols include (1) the ability to capture the behavior resulting from component composition (their bindings), and (2) the ability of testing whether a specific architecture fits into a given frame by verifying whether the corresponding architecture protocol "complies" with the frame protocol.

Our examples presented here are very simple so that the reader could get the impression that focusing on interface protocols might do. However, real life examples (such as [17]) indicate that frame protocols are very important, as they provide more general information on the "interplay" of all the calls on the "component boundary".

1.3 Classical way to adress bindings and related problems

The behavior of an architecture is determined by the behavior of its subcomponents as specified by their frame protocols. A classical way to express combined behavior of the subcomponents is via parallel composition (e.g. Wright, Tracta). For this purpose, the composition operator Π_X for languages $L_1, L_2 \subseteq Act^*$ and $X \subseteq EN$ was introduced in behavior protocols [14]:

$L_1\ \Pi_X\ L_2$ is the set of traces, each formed by an arbitrary interleaving (shuffling) of a pair of traces α and β, ($\alpha \in L_1,\ \beta \in L_2$), such that, for every event $e \in X$, if e is prefixed by ? in α and by ! in β (or vice versa), any appearance of $?e; !e$ resp. $!e; ?e$ as a product of the interleaving is merged into τe in the resulting trace t (becomes an internal event). However, if some $?e$ or $!e$, ($e \in X$), remains non-merged this way in a produced trace t, t is excluded from the result. Example: Consider the $Prot_{Alpha}$ and $Prot_{Beta}$

from Fig. 1b), i.e.

$$Prot_{Alpha} = !a\{?m+?n\},\ Prot_{Beta} = ?a\{!m\}+?b\{!n\}$$

The combined behavior of *Alpha* and *Beta* as the result of their binding is described as:

$$Prot_{Alpha}\ \Pi_X\ Prot_{Beta} = \tau a\{\tau m\},\ X = \{a\uparrow,\ a\downarrow,\ b\uparrow,\ b\downarrow,\ m\uparrow,\ m\downarrow,\ n\uparrow,\ n\downarrow\}.$$

Note that X is formed by all the events from the interface bindings between *Alpha* and *Beta*. Considering these protocols, *Alpha* and *Beta* behave correctly in the sense that every request or response emitted (such as $!a\uparrow$ or $!a\downarrow$) is accepted by the other component ($?a\uparrow$ or $?a\downarrow$). However, consider $Prot'_{Alpha}$ and $Prot'_{Beta}$ from Fig. 1b), i.e.: $Prot'_{Alpha} = !a\{?m\}+!b\{?n\}$ and $Prot'_{Beta} = ?a\{!m+!n\}+?b\{!n\}$. After *Alpha* emits $!a\uparrow$ (accepted by *Beta*), *Beta* can emit $!m\uparrow$ or $!n\downarrow$. According to $Prot'_{Alpha}$, $!m\uparrow$ would be accepted, while $!n\uparrow$ would not. The bottom line is that *Beta* can attempt to call a method which invocation is not permitted on *Alpha* in that particular situation. This is an example of *bad activity*. However, such (faulty) traces are omitted in the language constructed via Π_X: $Prot'_{Alpha}\ \Pi_X\ Prot'_{Beta} = \tau a\{\tau m\} + \tau b\{\tau n\}$. This problem originates in the CCS parallel composition operator (having been an inspiration for Π_X), where it is not defined who the originator of complementary events $?a\uparrow$ and $!a\uparrow$ is. However, when modeling a procedure call a, the event $!a\uparrow$ is what starts the communication. Bad activity is analyzed in Sect. 2 and so are other types of faulty behavior, *no activity* and *divergency*.

1.4 Goals and structure of the paper

The paper has two key goals: (1) To analyze the problem that a composition of components can result in faulty behavior. Reflecting this aim, it introduces in Sect. 2 the concept of *erroneous trace* and a new operator, *consent*, for component behavior composition. (2) To show (Sect. 3) that the consent operator can be advantageously used for a static verification whether an update of a component A can take place in a particular situation (specified in A's behavior protocol), considering the behavior of A's actual environment. A short evaluation and related work is provided in Sect. 4 while conclusion and future work forms Sect. 5.

2 Capturing errors resulting from incorrect composition

Let A, B be components with the behavior L_A, L_B on alphabets S_A, S_B. Let C be a component composed of A and B. Let X be the set of all events from the connections between A, B. In this section, we show all the types of faulty computations of C resulting from the composition of A and B (on X).[2]

For a component P (with an alphabet S_P) contained in a composed component Q and a trace t produced by Q, *projection*[3] of t to P (denoted $Trace_P(t)$) is the trace processed by P while Q processes t. Thus, if C has processed a sequence of event tokens t, the subcomponent A resp. B have processed the sequence $Trace_A(t)$ resp. $Trace_B(t)$. We say that a component P *can stop* after it has processed a trace t if $t \in L_P$. Thus, A can stop after C has processed t if $Trace_A(t) \in L_A$; in a similar vein, B can stop after C has processed t if $Trace_B(t) \in L_B$. Note that C can stop after processing t iff both A and B can stop.

The faulty computations caused by a "bad" component composition can be split into two categories: 1) At some point a computation cannot continue - *no continuation* error which includes two specific error types: *bad activity* and *no activity*; 2) A computation is infinite (*divergency* error) - recall that our model does not allow infinite traces.

To capture computations with errors, we introduce *error tokens* $\varepsilon n\uparrow$, $\varepsilon n\downarrow$, $\varepsilon\oslash$ and $\varepsilon\propto$, ($n \in EN$), and *erroneous traces* of the form $w<e>$, where w is a trace formed of non-error event tokens ($w \in ((ACT\backslash ErrorTokens)^*)$[4] and e at the end of the trace stands for the error token reflecting the type of

[2] In this section, $Prefix(L) = \{u : (\exists v)(uv \in L)\}$, $Tokens_\tau(E) = \{\tau e : e \in E\}$, $Tokens_!(S) = \{!e : e \in E\}$, $Tokens_?(E) = \{?e : e \in E\}$, S^∞ is the set of all infinite sequences of elements of S.

[3] More formally: Let Y be the set of all events from connections between P and other components inside Q. $Trace_P(t)$ is obtained from t by deleting all of its tokens which are not elements of the set $Tokens_\tau(Y) \cup S_P$ and renaming those of t's tokens which are of the form τe, $e \in Y$ to the only element of the set $\{?e, !e\} \cap S_P$ (i.e.: $?e$ if $?e \in S_P$, $!e$ if $!e \in S_P$).

[4] $ErrorTokens = \{\varepsilon\oslash,\ \varepsilon\propto\}\ \cup\ \{\varepsilon n\uparrow\ :\ n \in EN\}\ \cup\ \{\varepsilon n\downarrow\ :\ n \in EN\}$

the error occurred. In a case of a no-continuation error, the error occurs at the end of the trace (just where e is located). In a case of divergency, an erroneous trace is a finite prefix followed by $\varepsilon\propto$ to represent an infinite continuation.

In a case of *bad activity*, A tries to emit $!n\uparrow$ or $!n\downarrow$ ($n \in EN$), but B at the other side of the respective binding is not ready to accept (to issue $?n\uparrow$ resp. $?n\downarrow$), i.e. no suitable trace is defined in B's behavior. A bad activity is expressed by an error token of the form $\varepsilon n\uparrow$ or $\varepsilon n\downarrow$. For example, the composition of P_{Alpha} and P_{Beta} from Sect. 1.3 — $!a\{?m\}+!b\{?n\}$ and $?a\{!m+!n\}+?b\{!n\}$ on $X = \{a\uparrow, a\downarrow, b\uparrow, b\downarrow, m\uparrow, m\downarrow, n\uparrow, n\downarrow\}$, results in $\tau a\{\tau m\} + \tau a\uparrow; \varepsilon n\uparrow + \tau b\{\tau n\}$. In general, the *bad activity set* $BA(L_A, L_B, X)$ of the erroneous traces such that A tries to emit an event which cannot be accepted by B, is defined as follows:

$$\begin{aligned} BA(L_A, L_B, X) = \; & \{\, w{<}\varepsilon e{>} : (\exists u)(\exists v)(u{<}!e{>} \in Prefix(L_A) \wedge \\ & v \in Prefix(L_B) \wedge v{<}?e{>} \notin Prefix(L_B) \wedge \\ & w \in (u\ \Pi_X\ v) \wedge e \in X) \,\}. \end{aligned}$$

In a case of *no activity*, none of A and B is able to emit an event, and at least one of A and B cannot stop. For example, if the behavior protocols on Fig. 1b) were defined as $P_{Alpha} = (?m; ?n)$, $P_{Beta} = (!m; ?a)$, their composition would yield the (only) trace: $<\tau m\uparrow; \tau m\downarrow; \varepsilon\oslash>$. Formally, the *no activity set* $NA(LA, LB, X)$ is defined as follows:

$$\begin{aligned} NA(L_A, L_B, X) = \; & \{\, w{<}\varepsilon\oslash{>} : (\exists u)(\exists v)(u \in Prefix(L_A) \wedge \\ & v \in Prefix(L_B) \wedge (u \notin L_A \vee v \notin L_B) \wedge \\ & w \in (u\ \Pi_X\ v) \wedge (\forall t \in (S_A \cup S_B)\backslash Tokens_?(X))(\\ & u{<}t{>} \notin Prefix(L_A) \wedge v{<}t{>} \notin Prefix(L_B))) \,\}. \end{aligned}$$

In a case of *divergence*, A and B can emit events, but after each event, at least one of the components cannot stop. Such computation can be formally captured by an infinite meta-trace of the form wt, $w \in T^*$, $t \in T^\infty$, $T = Tokens_\tau(X) \cup (S_A\backslash\ (Tokens_?(X) \cup Tokens_!(X))) \cup (S_B\backslash\ (Tokens_?(X) \cup Tokens_!(X)))$. Here, w is a (finite) correct prefix, i.e. w is also a prefix of a non-erroneous trace. Until the whole w is processed, it would be always possible to chose another path of computation such that C could stop. The second part of the meta-trace, an incorrect (infinite) postfix t, expresses the part of computation, where stopping is already impossible. Because our model does not allow infinite traces, t is represented in the resulting behavior by infinite number of all (finite) sequences of the form $w\beta{<}\varepsilon\propto{>}$, where β is a finite prefix of t such that one of A, B can stop after C has processed $w\beta$, but the other cannot stop. For example, let the frame protocols in Fig. 1b) be: $P_{Alpha} = (!a; (?m; !a)^*)$, $P_{Beta} = (?a; !m)^*$. Their composition results into the infinite meta-trace $<\tau a\uparrow;\ \tau a\downarrow;\ \tau m\uparrow;\ \tau m\downarrow;\ \tau a\uparrow;\ \tau a\downarrow;\ \tau m\uparrow;\ \tau m\downarrow;\ ...>$. It is represented by the set of erroneous traces of the form $\{<\tau a\uparrow;\ \tau a\downarrow;\ \varepsilon\propto>,\ <\tau a\uparrow;\ \tau a\downarrow;\ \tau m\uparrow;\ \tau m\downarrow;\ \varepsilon\propto>,\ <\tau a\uparrow;\ \tau a\downarrow;\ \tau m\uparrow;\ \tau m\downarrow;\ \tau a\uparrow;\ \tau a\downarrow;\ \varepsilon\propto>,\ ...\ \}$. Formally, the *divergence set* $DIV(L_A, L_B, X)$, containing the erroneous traces of an infinite activities such that A can stop (and B cannot), is defined as follows:

$$\begin{aligned} DIV(L_A, L_B, X) = \; & \{\, w{<}\varepsilon\propto{>} : (\exists u)(\exists v)(u \in L_A \wedge v \in Prefix(L_B) \wedge \\ & v \notin L_B \wedge w \in (u\ \Pi_X\ v) \wedge \\ & w \notin Prefix(NC(L_A, L_B, X) \cup (L_A\ \Pi_X\ L_B))) \,\}, \\ NC(L_A, L_B, X) = \; & NA(L_A, L_B, X) \cup BA(L_A, L_B, X) \cup BA(L_B, L_A, X).^5 \end{aligned}$$

Now, we can define *consent* operator ∇_X for languages L_A, L_B, where X consists of all the events from connections between components A, B: $L_A \nabla_X L_B$ is the set containing all the traces from $L_A\ \Pi_X\ L_B$ and all the erroneous traces induced by the composition of A and B. Formally:

$$\begin{aligned} & L_A \nabla_X L_B = (L_A\ \Pi_X\ L_B) \cup ER(L_A, L_B, X), \\ & ER(L_A, L_B, X) = NC(L_A, L_B, X) \cup DIV(L_A, L_B, X) \cup DIV(L_B, L_A, X).^6 \end{aligned}$$

[5] $NC(L_A, L_B, X)$ stands for the *no continuation set*.

[6] $ER(L_A, L_B, X)$ is the set of all erroneous traces. Note that $DIV(L_B, L_A, X)$ captures the divergences when B can stop and A cannot.

If (at least) one of L_A and L_B describes the behavior of a composed component, it can contain an erroneous trace; this trace will trigger the existence of other erroneous traces in $L_A \nabla_X L_B$. It can be proved that the operator ∇_X preserves regularity. Although defined on languages, it can be easily extended to protocols, so that: $L(Prot_A \nabla_X Prot_B) = L(Prot_A) \nabla_X L(Prot_B)$.

To demonstrate the difference between ∇_X and Π_X, we present the following simple examples, illustrating the three types of errors described above (we assume $X = \{a\uparrow, a\downarrow, m\uparrow, m\downarrow\}$):

$$
\begin{array}{ll}
(1a) & ?a\ \Pi_X\ (!m + !a)\ =\ \tau a \\
(1b) & ?a\ \nabla_X\ (!m + !a)\ =\ \varepsilon m\uparrow\ +\ \tau a \\
(2a) & ?a\ \Pi_X\ ((\tau i; ?m) + !a)\ =\ \tau a \\
(2b) & ?a\ \nabla_X\ ((\tau i; ?m) + !a)\ =\ (\tau i; \varepsilon\oslash)\ +\ \tau a \\
(3a) & (?a; !m)^*\ \Pi_X\ (!a; (?m; !a)^*\ +\ (!a; ?m)^*)\ =\ (\tau a; \tau m)^* \\
(3b) & (?a; !m)^*\ \nabla_X\ (!a; (?m; !a)^*\ +\ (!a; ?m)^*)\ = \\
 & =\ ((\tau a; \tau m)^*; \varepsilon\propto)\ +\ (\tau a; (\tau m; \tau a)^*; \varepsilon\propto)\ +\ (\tau a; \tau m)^*
\end{array}
$$

The following lemma shows that the only no continuation errors are bad activity and no activity.

Lemma. Let a component C be composed of A and B having the behavior L_A and L_B, X be the set of all events from communication between A and B. Let $L_A \nabla_X L_B$ contain no erroneous traces ending with $\varepsilon n\uparrow, \varepsilon n\downarrow$ (for a method name n) nor $\varepsilon\oslash$. Then, in every step of any computation, C can always stop or process an event.

Proof sketch: Let C have already processed a trace t, $t_A = Trace_A(t)$, $t_B = Trace_B(t)$. If $t_A \notin L_A$ or $t_B \notin L_B$ (i.e. C cannot stop), there exists an event token k such that $k \neq ?e$ for any $e \in X$, and either $t_A{<}k{>} \in Prefix(L_A)$ or $t_B{<}k{>} \in Prefix(L_B)$, otherwise a no activity error would occur. If $k = !e$ and $e \in X$, the call is accepted, otherwise bad activity error would occur. Thus, the computation can continue (by τe if $k = !e$, $e \in X$, or by k otherwise).

3 Dynamic updates

In SOFA, a component C with frame F and architecture A_i (denoted as C/A_i) can be updated at run-time by replacing its implementation (i.e. A_i) by another architecture A_{i+1} and, potentially, converting the current state of C/A_i to the initial state of C/A_{i+1}. It is assumed that C/A_{i+1} has the same frame (F) as C/A_i. The update is started by the *component manager* CM_C, which is responsible for managing lifecycle of C; the lifecycle can be viewed as the sequence C/A_1, C/A_2... . Assuming there is no communication activity between C and its environment (how this is achieved is discussed below), CM_C during an update deletes the old architecture (A_i) and instantiates and calls the *component builder* of A_{i+1}; consequently, the actual new component C/A_{i+1} is created, obtaining its initial state as a transformation of the last state of C/A_i. The problem of transforming the state of C/A_i to C/A_{i+1} (for which a conversion code in A_{i+1} is responsible) is out of scope of this paper. The mechanism of SOFA component update is described in detail in [13, 20]. During the life-cycle of a component, a *dynamic checker* associated with CM_C is monitoring the behavior of the component (using F's frame protocol), allowing to predict what the next event on the component may be.

A component update has to be *atomic* - i.e. there should be no communication between C and its environment while C is being updated.[7] In this section, we first show how updates are reflected in behavior protocols (extending the approach introduced in [20]). Further we apply the consent operator to detecting violations of an update's atomicity.

There are several ways to ensure atomicity during an update of a component C: 1) The whole system is stopped. 2) All the interfaces of C are locked, so that during the update any method call to C is deferred, as in [13]. 3) By analyzing behavior protocols, it is statically tested whether, during an update, another component could call a method of C. The first two techniques worsen performance of the system (as stopping all components is too pessimistic, and locking means employing a wrapper). This is why

[7] Being a task of CM_C, the problem of coping with the internal communication in C during its update, is outside of the scope of this paper.

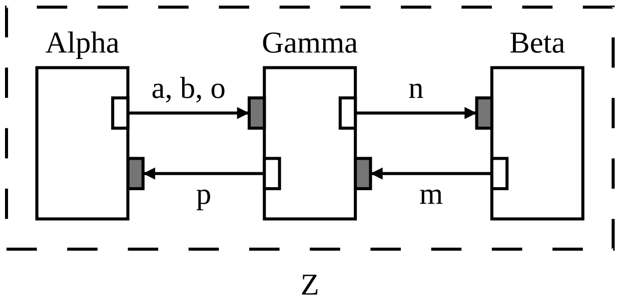


Figure 2: Updating the *Alpha* component

we focus on 3) and propose the following method of statical testing of update atomicity via behavior protocols.

A natural way to incorporate the update events in behavior specification of C is to include the "update" method call on CM_C into the frame protocol of C. By convention, we denote the corresponding event tokens (*update tokens*) as $?\pi\uparrow$ (beginning of update), $!\pi\downarrow$ (end of update) or $?\pi_n\uparrow$, $!\pi_n\downarrow$ (n being an integer distinguishing different updates in one protocol). In every trace t generated by the frame protocol of C, if t contains the update tokens $?\pi\uparrow$, $!\pi\downarrow$, they have to be present in this order and there can be no other token between them. Using the method described below, we can statically check weather the atomicity of a particular update is ensured (the negative cases will be reflected as erroneous traces). Thus, what remains to be done at run time is to make sure that the moment of a π call complies with the frame protocol. Here, e.g. if update calls are orthogonal to business calls, the component manager CM_C can cooperate with the dynamic checker and postpone the actual update as if it came in the right moment specified in the frame protocol.

Formally, we introduce the set of *update event names* $UN = \{\pi, \pi_1, \pi_2, ...\}$, $UN \subseteq EN$. Further, let the component C with frame F be a part of an architecture Z. Let C be the only component in Z being able to be updated ($?\pi\uparrow; !\pi\downarrow$ in C's frame protocol $Prot_C$). To describe how C behaves within Z, we construct the architecture protocol $Prot_Z$ of Z by using the consent operator applied on the behavior $L(Prot_C)$ and the behavior of other components in Z (specified by their frame protocols). In the resulting language $L(Prot_Z)$, there can be two types of erroneous traces: 1) Erroneous traces of the form $w{<}?\pi\uparrow; \varepsilon n\uparrow{>}$ where $w \in ACT^*$, $\pi \in UN$ and $n \in EN \setminus UN$. Such errors are caused by an attempt to deliver an event $n\uparrow$ to C during the update π — recall that no event tokens between $?\pi\uparrow$, $!\pi\downarrow$ are allowed, specially no token of the form $?e$ expressing acceptance of an event. Here, atomicity of π is not ensured. 2) Erroneous traces of the form $w{<}k{>}$, where k is any error token and for every $\pi \in UN$, either both the tokens $?\pi\uparrow$, $!\pi\downarrow$ occur in w, or none of them. Here, the erroneous behavior has nothing common with the update-process itself, however C does not behave the way the other frames in Z expect.

The components in Fig. 2 model a server (*Gamma*) and two clients (*Alpha*, *Beta*) forming an architecture Z. Assume *Alpha* with the architecture A_1 has been updated by A_2. We present two examples of Z's behavior after the update: (Ex. 1): Let the components have the frame protocols

$$\begin{aligned} Prot_{Alpha} &= ((!a; ?p)^*; ?\pi_1\uparrow; !\pi_1\downarrow)^*, \\ Prot_{Beta} &= (!m; ?n)^*, \\ Prot_{Gamma} &= (?a; !p)^* \mid (?m; !n)^*, \end{aligned}$$

$\pi_1 \in UN$. Then the architecture protocol of Z is

$$\begin{aligned} Prot_Z &= (Prot_{Alpha} \nabla_X Prot_{Gamma}) \nabla_Y Prot_{Beta} = \\ &= ((\tau a; \tau p)^*; \tau\pi_1\uparrow; \tau\pi_1\downarrow)^* \mid (\tau m; \tau n)^*, \end{aligned}$$

where $X = \{a\uparrow, a\downarrow, b\uparrow, b\downarrow, o\uparrow, o\downarrow, p\uparrow, p\downarrow\}$ and $Y = \{m\uparrow, m\downarrow, n\uparrow, n\downarrow\}$. From $Prot_Z$ we see that: 1) Atomicity of π_1 is ensured. 2) During the update π_1, *Beta* and *Gamma* can communicate, not violating the atomicity of π_1. (Ex. 2): Let *Beta* have the frame protocol *NULL* (i.e., it does nothing) and let

$Prot_{Alpha} = ((!a; ?\pi_1\uparrow; !\pi_1\downarrow) + (!b; ?\pi_2\uparrow; !\pi_2\downarrow)); (!o \,|?p)$, $Prot_{Gamma} = (?a; ?o; !p) + (?b; (?o \,|!p))$, where $\pi_1, \pi_2 \in UN$. In this case

$$\begin{aligned} Prot_Z &= Prot_{Alpha} \nabla_X Prot_{Gamma} = \\ &= (\tau a; \tau\pi_1\uparrow; \tau\pi_1\downarrow; \tau o; \tau p) + (\tau b; ((\tau\pi_2\uparrow; \tau\pi_2\downarrow; (\tau o \mid \tau p)) + (\tau\pi_2\uparrow; \varepsilon p\uparrow))), \end{aligned}$$

X is the same as in ex. 1. We see that: 1) The update π_1 is always atomic (there is no erroneous trace of the form $w{<}\tau\pi_1\uparrow; \varepsilon e{>}$ for an event e and $w \in ACT^*$). 2) The atomicity of π_2 is not ensured. This fact is indicated by erroneous traces in the resulting behavior (error token occurs always after $\tau\pi_2\uparrow$). 3) All erroneous traces are caused by violating the atomicity of an update (thus no erroneous trace contains $\tau\pi_2\downarrow$). Note, however, when the atomicity of π_2 was ensured by another mechanism (interface locking, etc.), *Alpha* could be updated successfully.

Our update mechanism allows to do unanticipated changes in the structure of a component system: at the design stage, the architecture A_{i+1} updating a C/A_i is unknown. The only part of the C's design which reflects the possibility of an update is the C's frame protocol where the π tokens are to be stated — however, this is not a crucial problem: 1) Even though the update moments are given by the behavior protocol, an update can be asked at any time, as CM_C cooperating with dynamic checker may postpone such a request until an update is possible; 2) Because behavior protocols are clearly separated from the code, they can be easily modified (not affecting the actual implementation).

4 Evaluation and related work

The consent operator. The consent operator provides a means for modeling behavior of composed components (i.e. constructing architecture protocols), covering three types of errors: bad activity, no activity, and divergency. The approach is similar to the failures semantics (capturing unaccepted events), deadlocks and divergence in CSP [16]. Also, the operator Π_X combines traces in a similar way as the parallel operator $\|_X$ in CSP, provided the hiding operator $\setminus X$ is also applied. The main difference is that in CSP the communication is symmetric, i.e. in $P \|_X Q$ the events in P and Q denoted by the same token are synchronized, and a failure occurs when an event from X appears just in one of the processes P, Q. However, we claim that the semantics of a method call is asymmetric - emitting events without absorbing them is an error, however absorbing without emitting is not one (except for the case of a deadlock). Thus, modeling of a method call in CSP fails to capture this asymmetry. As to CCS [11], the issue persists, since the composition operator | allowing to synchronize inputs with outputs by generating internal actions does not address the asymmetry mentioned above. In fact, it handles composition in the same way as our Π_X operator (or, better put, vice versa). As an aside, we do not define the consent operator via the structural operational semantic (SOS) rules (approach used e.g. in CSP and CCS), because of: 1) We use finite traces, while SOS rules define a label transition system not employing any accepting states and considering all traces infinite. 2) Our divergency is a global property (of our transition system), while SOS rules are focused on the local properties of states. As to other work, the authors of [4] analyze various methods of component composition; however, parallel composition of behavior is not considered.

Dynamic updates. Using the consent operator, it can be statically evaluated whether a dynamic update of a component A can be atomic "for free", i.e. it is ensured that, while the rest of the system is running, none of the other components can call a method on A during the update (performance benefit). Moreover, this technique can also selectively identify which of the several update events specified in the frame protocol of A would violate update atomicity and which would not. To our knowledge, there is no similar work addressing the issue. Naturally, a testing of component update atomicity could be realized via CSP [16] with failure semantics as well; in general, we consider behavior protocols much easier to apply in a real architecture description language than CSP [14].

5 Conclusion and future work

We presented a method of identifying errors in behavior of composed components. In addition, we have shown how the method can be used for verifying the atomicity of run-time component updates. In [1], other applications of our consent operator (not mentioned here) can be found. As a future work related to component updating, we intend to focus on: 1) Analyzing the semantics of stopping composed components. The question is how each component should contribute to the decision about the end of processing and how such a decision should be coordinated. Addressing the issue may even need to modify the consent operator. 2) Analyzing different strategies of a component manager CM_C as for coping with the internal communication in C during an update.

References

[1] J. Adamek, F. Plasil, "Behavior Protocols: Tolerating Faulty Architectures and Supporting Dynamic Updates", Technical Report 02/10, Department of Computer Science, University of New Hampshire, 2002

[2] R. J. Allen, D. Garlan, "A Formal Basis For Architectural Connection", ACM Transactions on Software Engineering and Methodology, Jul. 1997

[3] E. Bruneton, T. Coupaye, J. B. Stefani, "The Fractal Composition Framework", Proposed Final Draft of Interface Specification Version 0.9, The ObjectWeb Consortium, Jun. 2002

[4] A. Farias, M. Sudholt, "On the construction of components with explicit protocols", Technical Report No. 02/4/INFO, Departement Informatique, Ecole des Mines de Nantes, Nantes, France, 2002

[5] D. Giannakopoulou, J. Kramer, S. C. Cheung, "Analysing the Behaviour of Distributed Systems using Tracta", Journal of Automated Software Engineering, special issue on Automated Analysis of Software, vol. 6(1), Jan. 1999

[6] C. R. Hofmeister, J. Atlee and J. Purtilo: "Writing Distributed Programs in Polylith", University of Maryland, Nov. 1990

[7] T. Kalibera, P. Tuma, "Distributed Component System Based On Architecture Description: The SOFA Experience", Accepted at DOA 2002, Copyright (C) Springer-Verlag, Oct. 2002

[8] F. Leyman, "Web Services Flow Language (WSFL 1.0)", IBM Software Group, May 2001

[9] Microsoft COM Technology, http://www.microsoft.com/com

[10] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, "Specifying Distributed Software Architectures", 5th European Software Engineering Conference, Barcelona, Spain, 1995

[11] R. Milner, "Communication and Concurrency", Prentice Hall, 1989

[12] OMG Corba Component Model Specification,
http://www.omg.org/technology/documents/formal/components.htm

[13] F. Plasil, D. Balek, R. Janecek, "SOFA/DCUP Architecture for Component Trading and Dynamic Updating", Proceedings of the ICCDS '98, Annapolis, IEEE Computer Soc. Press, 1998, pp. 43 52

[14] F. Plasil, S. Visnovsky, "Behavior protocols for Software Components", IEEE Transactions on SW Engineering, 28(9), 2002

[15] F. Plasil, S. Visnovsky, M. Besta, "Bounding Behavior via Protocols", Proceedings of TOOLS USA 99, Santa Barbara, CA, Aug. 1999

[16] A. W. Roscoe, "The Theory and Practice of Concurrency", Prentice-Hall, 1998

[17] SOFA project, http://nenya.ms.mff.cuni.cz/thegroup/SOFA/sofa.html

[18] Sun Enterprise Java Beans, http://java.sun.com/products/ejb

[19] UML Resource Page, http://www.omg.org/technology/uml/index.htm

[20] S. Visnovsky, "Modeling Software Components Using Behavior Protocols", doctoral dissertation, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2002