

BEHAVIOR PROTOCOLS: TOLERATING FAULTY
ARCHITECTURES AND SUPPORTING DYNAMIC
UPDATES

by

Jiri Adamek*, Frantisek Plasil

TR 02/10

Technical Report Series
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF NEW HAMPSHIRE
Durham, New Hampshire 03824

* Charles University, Department of Software Engineering, Prague, Czech Republic

Contents

Abstract	3
1. Introduction	4
1.1. Component models - basic ideas	4
1.2. Describing behavior of components	5
2. Two flaws of Behavior Protocols: Bindings and faulty architectures	6
2.1. Classical way to address bindings and related problems	6
2.2. Goals and structure of the paper	7
3. Capturing errors resulting from incorrect composition	8
4. Benefits of the consent operator	10
4.1. Faulty architecture is a relative concept	10
4.2. Dynamic updates	11
5. Evaluation and Related Work	13
6. Conclusion and Future Work	14
References	14

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 an 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 an erroneous trace. There are two key benefits of the consent operator: (1) faulty software architecture is a relative concept - erroneous traces can be avoided by the way the composed component is utilized in a concrete environment, (2) it can be statically determined via behavior protocol, whether the atomicity of a dynamic update of a component is implicitly guaranteed 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 [11] and EJB [21], to higher-level granularity component models where Polyolith[8], Darwin/Tracta[12], Wright[2,3] belong to the “classics”, while CCM[14] and Fractal[4] 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 elaborated in our research team [15,20,9]; intending to keep its description here as simple as possible, we refer the reader to [15] 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 on the arrow expressing the tie indicates that a is the method to be called on 2 by Alpha via 1. In a similar vein, in Fig 2, there are two bindings of Alpha and Beta. Here, through 1 the component Alpha calls a or b on 3, and

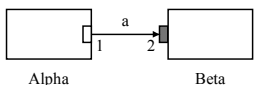
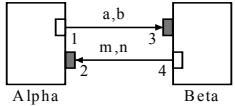
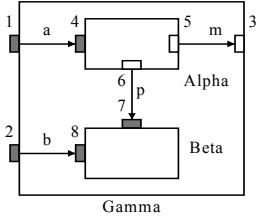
a)		$\text{Prot}_{\text{Alpha}} = !a;!a!;!a!$ $\text{Prot}_{\text{Beta}} = ?a=?a!;!a!$
b)		<p>I) $\text{Prot}_{\text{Alpha}} = !a\{?m + ?n\}$ $\text{Prot}_{\text{Beta}} = ?a\{!m\} + ?b\{!n\}$</p> <p>II) $\text{Prot}'_{\text{Alpha}} = !a\{?m\} + !b\{?n\}$ $\text{Prot}'_{\text{Beta}} = ?a\{!m + !n\} + ?b\{!n\}$</p>
c)		$\text{Prot}_{\text{Alpha}} = ?a;!p;!m$ $\text{Prot}_{\text{Beta}} = ?b;!p$ <p>I) $\text{Prot}_{\text{Gamma}} = ?b;!a;!m$</p> <p>II) $\text{Prot}'_{\text{Gamma}} = ?a;!b;!m$</p>

Figure 1. Examples of component systems and behavior protocols

through 4, Beta calls m resp. n on 2 of Alpha. In 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 *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 [16,17], 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

Models formally describing behavior of (not only software) components observe a set of primitive actions/communicated 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 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 are typically chosen as a part of tokens. To the transition systems designed to model software component behavior belong:

CSP [19], employed, e.g., in the Wright architecture description language [2,3], 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.

TRACTA/FSP [6,7], part of the Darwin ADL [12], uses system of recursive equations as well, but the set of all allowed operators is restricted so that regularity of traces is guaranteed; thus, the equations define a finite automaton accepting the desired traces.

UML [22] 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, [10]) is intended for behavior description of components in a specific settings where component is a web service provider. Related

¹Many 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.

transaction 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 [16] we model the behavior of component as traces capturing events (a *request* and a *response* which can compose a method call) on component interfaces. For an event name $a \in \text{EventNames}$, request and response are denoted as $a\uparrow, a\downarrow$ (*events*). Let the component Alpha from fig. 1a) calls 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 a response occurs inside of a composed component (as an internal event), the corresponding event token takes the form τa (*internal event*). By convention, the set of all event tokens processed (emitted and/or absorbed) by a component A forms its alphabet $S_A \subseteq \text{ACT}$. The *behavior* L_A of a component A is the set of all possible traces produced by A , forming a language $L_A \subseteq S_A^* \subseteq \text{Act}^*$ (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 like expression, a behavior protocol employs in addition to concatenation ($;$), alternative ($+$) and finite sequencing ($*$) also composition (\prod_X , Sect. 2.1), interleaving/shuffle of traces (\updownarrow), and the abbreviations: $?m = (?m\uparrow;!m\downarrow)$, $!m = (!m\uparrow;?m\downarrow)$, $\tau m = (\tau m\uparrow, \tau m\downarrow)$, $?m\{P\} = (?m\uparrow;P;!m\downarrow)$, $\tau m\{P\} = (\tau m\uparrow;P;\tau m\downarrow)$, P is a protocol.

To address behavior compliance throughout a hierarchy of component nesting, a frame protocol describes external communication of a component A at the level of its frame, while 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 $\tau a\{\tau x\}$ (generated automatically from these frame protocols). As explained in [16], 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.

2. Two flaws of Behavior Protocols: Bindings and faulty architectures

2.1. Classical way to address 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 \prod_X for languages $L_1, L_2 \subseteq \text{Act}^*$ and $X \subseteq \text{EventNames}$ was introduced in behavior protocols [16]:

$L_1 \prod_X L_2$ is the set of traces, each formed as 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 $\text{Prot}_{\text{Alpha}}$ and $\text{Prot}_{\text{Beta}}$ from Fig. 1b), i.e. $\text{Prot}_{\text{Alpha}} = !a\{?m + ?n\}$ and $\text{Prot}_{\text{Beta}} = ?a\{!m\} + ?b\{!n\}$. The combined behavior of Alpha and Beta as the result of their binding is:

$$\text{Prot}_{\text{Alpha}} \prod_X \text{Prot}_{\text{Beta}} = \tau a\{\tau m\}, X = \{a\uparrow, a\downarrow, b\uparrow, b\downarrow, m\uparrow, m\downarrow, n\uparrow, n\downarrow\}.$$

Note that X is formed by all the events from the interface bindings between Alpha and Beta. Moreover, here Alpha and Beta behave correctly in the sense that every request or response emitted (such as $!a\uparrow$ or $!a\downarrow$) is accepted by a counterpart ($?a\uparrow$ or $?a\downarrow$). However, this is not always the case - consider $\text{Prot}'_{\text{Alpha}}$ and $\text{Prot}'_{\text{Beta}}$ from Fig.1b), i.e.: $\text{Prot}'_{\text{Alpha}} = !a\{?m\} + !b\{?n\}$ and $\text{Prot}'_{\text{Beta}} = ?a\{!m + !n\} + ?b\{!n\}$. After Alpha emits $!a\uparrow$ (accepted by Beta), Beta can emit $!m\uparrow$ or $!n\uparrow$. According to $\text{Prot}'_{\text{Alpha}}$, $!m\uparrow$ would be accepted, while $!n\uparrow$ would not. The bottom line is that Beta can attempt to call a method which is not permitted to be invoked on Alpha in a particular situation. This is an example of *bad activity*. However, such a (faulty) trace is omitted in the language constructed via \prod : $\text{Prot}'_{\text{Alpha}} \prod_X \text{Prot}'_{\text{Beta}} = \tau a\{\tau m\} + \tau b\{\tau n\}$. This problem originates in the CCS parallel composition operator (being an inspiration for \prod_X), where it is not defined who the originator of complementary events $?a$ and $!a$ is. However, when modeling a procedure call a , the event $!a$ is what starts the communications. Bad activity is analyzed in Sect.3 and so are other types of faulty behavior, *no activity* and *divergency*.

Consider now the architecture composed of components Alpha and Beta (Fig.1c) where $\text{Prot}_{\text{Alpha}} = ?a; !p; !m$ and $\text{Prot}_{\text{Beta}} = ?b; ?p$. The combined behavior of Alpha and Beta contains, e.g., the trace $?a\uparrow, ?b\uparrow, !a\downarrow, !b\downarrow, \tau p\uparrow, \tau p\downarrow, !m\uparrow, ?m\downarrow$. It contains also $?a\uparrow, ?b\uparrow, !a\downarrow, !p\uparrow$ though, which is faulty, as the communication “hangs up”, since there is no counterpart $?p\uparrow$. However, applying this architecture in the component Gamma with the frame protocol $\text{Prot}_{\text{Gamma}} = ?b; ?a; !m$ (Fig.1c), eliminates occurrence of the faulty trace. (The only trace of Alpha and Beta nested in Gamma is $?b\uparrow, !b\downarrow, ?a\uparrow, !a\downarrow, \tau p\uparrow, \tau p\downarrow, !m\uparrow, ?m\downarrow$. On the contrary, assuming the protocol $\text{Prot}'_{\text{Gamma}} = ?a; ?b; !m$, the behavior of Gamma would include the faulty trace. The lesson is that an architecture may fail working in one environment (be considered faulty in this environment), while being fully correct in another environment. Thus the issue is to formally capture this relativity of “faulty architecture” (addressed in Sect.4.1).

2.2. Goals and structure of the paper

The paper has three key goals: (1) To analyze the problem that a composition of components can result in faulty behavior. Therefore, in Sect 3, it introduces the concepts of *erroneous trace* and a new operator, *consent*, for component behavior composition. (2) To show (Sect.4.1) that faulty architecture is a relative concept (i.e. faulty architecture is not necessarily “bad”- an architecture can produce an erroneous trace only if used in specific way by its environment (frame). In principle, the paper here aims at proposing a new perception of “proper behavior/design” of a component system based on the idea that faulty traces should be considered in behavior description via protocols, since their presence may be tolerated depending on the way the component is used. (3) To show (Sect. 4.2) that the consent operator can be advantageously used for a static verification whether an update of a component A can take place in the situation specified in a behavior protocol

with respect to the behavior of A's actual environment. A short evaluation and related work is provided in Sect.5 while conclusion and future work forms Sect. 6.

3. Capturing errors resulting from incorrect composition

Let A, B be components with 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 connections between A, B. In this section, we show all types of faulty computations of C which may be a result of 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* of t to P (denoted $\text{Trace}_p(t)$) is the trace processed by P while Q processes t³. Thus, if C has processed a sequence of event tokens t, the subcomponent A resp. B have processed the sequence $\text{Trace}_A(t)$ resp. $\text{Trace}_B(t)$; 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 $\text{Trace}_A(t) \in L_A$; in a similar vein, B can stop after C has processed t if $\text{Trace}_B(t) \in L_B$. Note that C can stop after processing t iff both A and B can stop.

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

To capture computations with errors, we introduce *error tokens* $\epsilon n \uparrow, \epsilon n \downarrow$ ($n \in \text{EventNames}$), $\epsilon \emptyset$ and $\epsilon \infty$ and *erroneous traces* of the form $w \langle e \rangle$, where w is a trace formed of non-error event tokens ($w \in ((\text{ACT} \setminus \text{ErrorTokens})^*)^4$, and e at the end of the trace stands for the error token reflecting the type of the error occurred. In case of a no-continuation error, the error occurs at the end of the trace (just where e is placed). In the case of divergency, a erroneous trace is a finite prefix followed by $\epsilon \infty$ to represent an infinite continuation.

In case of a *bad activity* A tries to emit $!n \uparrow$ or $!n \downarrow$ ($n \in \text{EventNames}$), 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 error token of the form $\epsilon n \uparrow$ or $\epsilon n \downarrow$. For example, composition of the protocols P_{Alpha} and P_{Beta} from Sect. 2.1 ($!a\{?m\} + !b\{?n\} \nabla_X (?a\{!m + !n\} + ?b\{!n\})$, $X = \{a \uparrow, a \downarrow, b \uparrow, b \downarrow, m \uparrow, m \downarrow, n \uparrow, n \downarrow\}$, results in $\tau a\{\tau m\} + \tau a \uparrow; \epsilon n \uparrow + \tau b\{\tau n\}$. In general, the *bad activity set* $BA(L_A, L_B, X)$, of erroneous traces, where A tries to emit an event which cannot be accepted by B, is defined as follows:

$$BA(L_A, L_B, X) = \{w \langle \epsilon e \rangle : (\exists u)(\exists v) (u \langle !e \rangle \in \text{Prefix}(L_A) \ \& \ v \in \text{Prefix}(L_B) \\ \& \ v \langle ?e \rangle \notin \text{Prefix}(L_B) \ \& \ w \in (u \prod_X v) \ \& \ e \in X) \}.$$

² In this section, $\text{Prefix}(L) = \{u : (\exists v) uv \in L\}$, $\text{Tokens}_\tau(E) = \{\tau e : e \in E\}$, $\text{Tokens}_!(S) = \{!e : e \in E\}$, $\text{Tokens}_?(E) = \{?e : e \in E\}$, 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. $\text{Trace}_p(t)$ is obtained from t by deleting all of its tokens which are not elements of the set $\text{Tokens}_\tau(Y) \cup S_p$ 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 $\text{ErrorTokens} = \{\epsilon \emptyset, \epsilon \infty\} \cup \{\epsilon n \uparrow : n \in \text{EventNames}\} \cup \{\epsilon n \downarrow : n \in \text{EventNames}\}$

In case of *no activity*, non of A and B is able to emit an event, and exactly one of A and B cannot stop. For example, if the behavior protocols on Fig. 1b) were defined as $P_{\text{Alpha}} = (?m; ?n)$, $P_{\text{Beta}} = (!m; ?a)$, their composition would yield the (only) trace: $\langle \tau m \uparrow; \tau m \downarrow; \varepsilon \emptyset \rangle$. Formally, the *no activity set* $NA(L_A, L_B, X)$ is defined as follows:

$$NA(L_A, L_B, X) = \{ w \langle \varepsilon \emptyset \rangle : (\exists u)(\exists v) (u \in \text{Prefix}(L_A) \ \& \ v \in \text{Prefix}(L_B) \ \& \ (u \notin L_A \vee v \notin L_B) \ \& \ w \in (u \prod_X v) \ \& \ (\forall t \in (S_A \cup S_B) \setminus \text{Tokens}_\tau(X)) (u \langle t \rangle \notin \text{Prefix}(L_A) \ \& \ v \langle t \rangle \notin \text{Prefix}(L_B))) \}.$$

In case of *divergence*, A and B can emit events (no activity excluded), but after every event, at least one of the components cannot stop. Such computation can be formally captured by an infinite meta-trace of the form $w t$, $w \in T^*$, $t \in T^\infty$, $T = \text{Tokens}_\tau(X) \cup (S_A \setminus (\text{Tokens}_\tau(X) \cup \text{Tokens}_\downarrow(X))) \cup (S_B \setminus (\text{Tokens}_\tau(X) \cup \text{Tokens}_\downarrow(X)))$. 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 is still possible to chose another path of the computation after which C can stop. The second part of the meta-trace, 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 an infinite number of all (finite) sequences of the form $w \alpha \langle \varepsilon \infty \rangle$, where α is a finite prefix of t such that one of A, B can stop after C has processed $w \alpha$, but the other cannot. For example, let the frame protocols in Fig. 1b) be: $P_{\text{Alpha}} = (!a; (?m; !a)^*)$, $P_{\text{Beta}} = (?a; !m)^*$. Their composition results into the infinite meta-trace $\langle \tau a \uparrow; \tau a \downarrow; \tau m \uparrow; \tau m \downarrow; \tau a \uparrow; \tau a \downarrow; \tau m \uparrow; \tau m \downarrow; \dots \rangle$. We describe it by the set of erroneous traces of the form $\{ \langle \tau a \uparrow; \tau a \downarrow; \varepsilon \infty \rangle, \langle \tau a \uparrow; \tau a \downarrow; \tau m \uparrow; \tau m \downarrow; \varepsilon \infty \rangle, \langle \tau a \uparrow; \tau a \downarrow; \tau m \uparrow; \tau m \downarrow; \tau a \uparrow; \tau a \downarrow; \varepsilon \infty \rangle, \dots \}$. Formally, the *divergence set* $IA(L_A, L_B, X)$, containing erroneous traces for an infinite activity where A can stop (and B cannot), is defined as follows:

$$IA(L_A, L_B, X) = \{ w \langle \varepsilon \infty \rangle : (\exists u)(\exists v) (u \in L_A \ \& \ v \in \text{Prefix}(L_B) \ \& \ v \notin L_B \ \& \ w \in (u \prod_X v) \ \& \ w \notin \text{Prefix}(\text{NC}(L_A, L_B, X) \cup (L_A \prod_X L_B))) \}, \text{ where}$$

$$\text{NC}(L_A, L_B, X) = \text{NA}(L_A, L_B, X) \cup \text{BA}(L_A, L_B, X) \cup \text{BA}(L_B, L_A, X).^5$$

Now, we can define *consent* operator ∇ for languages L_A, L_B , where X consists of all events from connections between components A, B: $L_A \nabla_X L_B$ produces set of traces, which contains just all traces from $L_A \prod_X L_B$ and all erroneous traces. Formally:

$$L_A \nabla_X L_B = (L_A \prod_X L_B) \cup \text{ER}(L_A, L_B, X), \text{ where}$$

$$\text{ER}(L_A, L_B, X) = \text{NC}(L_A, L_B, X) \cup \text{IA}(L_A, L_B, X) \cup \text{IA}(L_B, L_A, X).^6$$

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 existence of other erroneous traces in the result of $L_A \nabla_X L_B$. It can be proved that the operator ∇ preserves regularity. Although defined on languages, it can be easily extended to protocols, so that: $L(\text{Prot}_A \nabla_X \text{Prot}_B) = L(\text{Prot}_A) \nabla_X L(\text{Prot}_B)$.

⁵ $\text{NC}(L_A, L_B, X)$ stands for the *no continuation set*.

⁶ $\text{ER}(L_A, L_B, X)$ is the set of all erroneous traces. Note that $\text{IA}(L_B, L_A, X)$ captures the divergences when B can stop and A cannot.

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

$$\begin{aligned}
(1a) \quad ?a \prod_X (!m + !a) &= \tau a & (1b) \quad ?a \nabla_X (!m + !a) &= \varepsilon m\uparrow + \tau a \\
(2a) \quad ?a \prod_X (\tau i ; ?m + !a) &= \tau a & (2b) \quad ?a \nabla_X (\tau i ; ?m + !a) &= \tau i ; \varepsilon \emptyset + \tau a \\
(3a) \quad (?a ; !m)^* \prod_X (!a ; (?m ; !a)^* + (!a ; ?m)^*) &= (\tau a ; \tau m)^* \\
(3b) \quad (?a ; !m)^* \nabla_X (!a ; (?m ; !a)^* + (!a ; ?m)^*) &= ((\tau a ; \tau m)^* ; \varepsilon \infty) + (\tau a ; (\tau m ; \tau a)^* ; \varepsilon \infty) + (\tau a ; \tau m)^*
\end{aligned}$$

The following lemma shows that there are no other no continuation errors than 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 $L_A \nabla_X L_B$ contains no erroneous traces ending with $\varepsilon n\uparrow$, $\varepsilon n\downarrow$ (for a method name n) nor $\varepsilon \emptyset$. Then, in every step of any computation, C can always stop or process an event.

Proof sketch: Let C has already processed a trace t , $t_A = \text{Trace}_A(t)$, $t_B = \text{Trace}_B(t)$. 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 $t_A \langle k \rangle \in \text{Prefix}(L_A)$ or $t_B \langle k \rangle \in \text{Prefix}(L_B)$, 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).

4. Benefits of the consent operator

4.1. Faulty architecture is a relative concept

As shown in Sect. 2.1, an architecture A exhibiting some faulty behavior can behave correctly if put into a specific frame F . An erroneous trace in L_A reflects a specific (faulty) sequence of request/response exchanges between A and its environment E (the external components with which A communicates). When A is put into a specific frame F , the frame protocol of F can restrict the way F communicates with E (restricting communication of A with E as well, since calls to F are delegated to calls of A 's components). Thus, not all erroneous traces in L_A apply.

In principle, F 's frame protocol Prot_F represents an agreement between F and E , as to what the “maximal” communication between E and any architecture A implementing F can be. This “maximal” communication can be advantageously modeled by *inverted frame protocol* Prot_F^{-1} , derived from Prot_F by rewriting all symbols ‘?’ to ‘!’ and vice versa. For example, assuming Prot_F contains $?a\uparrow$ (i.e. call of a is accepted), the environment of F can emit $!a\uparrow$, respecting the agreement represented Prot_F . Thus, to capture what part of L_A will be maximally employed when A implements F , we compose A 's architecture protocol Prot_A with Prot_F^{-1} : $L_{AinF} = L(\text{Prot}_A \nabla_X \text{Prot}_F^{-1})$, where X is the set of all events from the F 's interfaces.

Let us assume Prot_A generates some erroneous traces (by definition, Prot_F cannot describe any faulty behavior). There are two ways an erroneous trace $w; \langle e \rangle$ (e is an error token) can appear in L : (1) $w; \langle e \rangle$ “originates” by combining an erroneous trace $u; \langle e \rangle \in L(\text{Prot}_A)$ and

a $v \in L(\text{Prot}_F)$, i.e., $w \in \{u\} \nabla_X \{v\}$). Here, Prot_F is not restrictive enough to filter out erroneous traces in Prot_A . (2) The error e in $w; \langle e \rangle$ is a “new” error (i.e. $w; \langle e \rangle \in u \nabla_X v$, $u \in \text{Prefix}(L(\text{Prot}_A))$, $v \in \text{Prefix}(L(\text{Prot}_F))$, u, v are not erroneous). Here, the error is caused by a “bad” interaction between A and E (modeled by Prot_F^{-1}), i.e. A does not implement F correctly. Note that more erroneous traces of the type (1) and (2) can be the result of a specific pair of u and v ($u \in L(\text{Prot}_A)$, $v \in L(\text{Prot}_F)$, e.g.: $\{\langle \tau i \uparrow; \tau i \downarrow; ?a \uparrow; !a \downarrow; \epsilon n \uparrow \rangle\} \nabla_{\{a \uparrow, a \downarrow\}} \{\langle !a \uparrow; ?a \downarrow \rangle\} = \{\langle \epsilon a \uparrow \rangle, \langle \tau i \uparrow; \epsilon a \uparrow \rangle, \langle \tau i \uparrow; \tau i \downarrow; \tau a \uparrow; \tau a \downarrow; \epsilon n \uparrow \rangle\}$.

If L_{AinF} does not contain an erroneous trace and $L(\text{Prot}_A)$ does, we can conclude that (i) faulty behavior of Prot_A was restricted by F and (ii) no new faulty behavior has appeared because of “pasting” A into F - A implements F successfully, although $L(A)$ itself contains errors. Thus, faulty architecture is a relative concept: Being “faulty” depends on the environment, in which the architecture is executed. The following example illustrates the conclusion.

Let Alpha, Beta on Fig. 1c) have the frame protocols $P_{\text{Alpha}} = (?a;!p;!m)$, $P_{\text{Beta}} = (?b;!p)$. The architecture protocol of Gamma contains erroneous traces: $P_{\text{A-Gamma}} = P_{\text{Alpha}} \nabla_X P_{\text{Beta}} = ?a;\epsilon p \uparrow + (?a!b \uparrow); \epsilon p \uparrow + (?a!b); \tau p; !m$ ($X = \{p \uparrow, p \downarrow\}$). However, assuming the frame protocol of Gamma is $P_{\text{Gamma}} = ?b;!a;!m$, then $P_{\text{Gamma}}^{-1} = !b;!a;!m$ and $L_{\text{AinGamma}} = P_{\text{A-Gamma}} \nabla_X P_{\text{Gamma}}^{-1} = \tau b; \tau a; \tau p; \tau m$, $X = \{a \uparrow, a \downarrow, b \uparrow, b \downarrow, m \uparrow, m \downarrow\}$, not containing any erroneous trace.

4.2. Dynamic updates

In SOFA, a component A can be updated at run-time by another component B, i.e. A is removed from the system and replaced by B. It is assumed B has the same frame as A but possibly different implementation/architecture. A component update has to be *atomic* - i.e. no other component outside A should observe the change. The mechanism of SOFA component update is described in detail in [15,23]. Although the frames of A, B are the same, their frame protocols $\text{Prot}_A, \text{Prot}_B$ can differ (and typically should), assuming a frame specification can be parameterized. In this section, we show how updates are reflected in behavior protocols and how the consent operator can be used to detect violation of atomicity during an update.

There are several ways to ensure atomicity during an update of a component A: 1) The whole system is stopped. 2) All the interfaces of A are locked, so that during the update method calls to A are deferred, as in [15]. 3) By analyzing behavior protocols to test statically whether, during an update, another component could call a method of A. 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 we focus on 3) and propose the following method of static testing of update atomicity via behavior protocols.

To capture updates in behavior protocols, we introduce the set of *update events names* $\text{UpdateNames} \subseteq \text{EventNames}$. In a trace t generated by the frame protocol Prot_A of a component A, an update of A is denoted by the sequence of *update tokens* $? \pi \uparrow, ! \pi \downarrow$ (standing for the beginning and end of an A’s update), $\pi \in \text{UpdateNames}$. The sequence $? \pi \uparrow; ! \pi \downarrow$ has to be always at the end of t , so that there can be at most one occurrence of it

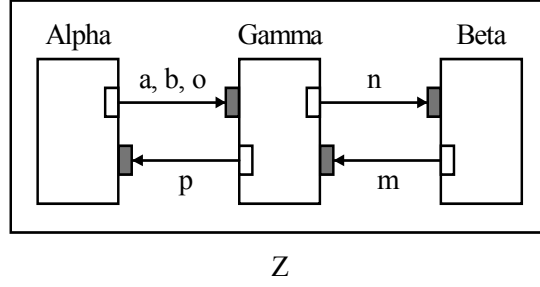


Figure 2. Updating the Alpha component

in t . No other event token is allowed between $?π↑, !π↓$ (i.e., A “should do nothing” during the update). The semantics of $?π↑;!π↓$ is very straightforward: if an update manager M decides to go ahead with the update of A by B , M stops A ’s computation, updates its implementation by B and then starts execution of B .

Assuming $U_A ⊆ L_A = L(Prot_A)$ be the set of all the A ’s traces ending with $?π↑;!π↓$, we call U_A *updating behavior* and $L_A \setminus U_A$ *non-updating behavior* of A . If A is updated by B (with frame protocol $Prot_B$), the resulting component (denoted A/B) has its behavior composed in principle of traces of the form $t;u$, where $t ∈ U_A$ ends by an update $?π↑;!π$, and $u ∈ L_B = L(Prot_B)$ reflects the continuation of the execution after the update. However, to express that $?π↑;!π↓$ was engaged in an internal communication with an update manager, we replace it in $t;u$ by $τπ↑;τπ↓$. Thus, the set of all traces of A/B is $U_A'; L_B$ ⁷, where $U_A' = \{ w; \langle τπ↑; τπ↓ \rangle : w; \langle ?π↑;!π↓ \rangle ∈ U_A \}$. Still, in traces from $U_A'; L_B$, no tokens are allowed between $τπ↑$ and $τπ↓$ for any $π ∈ UpdateNames$. Let the component A/B is a part of an architecture Z . Let A was the only component in Z , able to be updated ($?π↑;!π↓$ in its frame protocol). To describe how A/B behaves within Z , we construct the architecture protocol of Z by using consent operator applied on the behavior of A/B ($U_A'; L_B$) and the behavior of other components in Z (generated by their frame protocols). In the resulting language, there can be two types of erroneous traces: 1) Erroneous traces of the form $w; \langle τπ↑; εn↑ \rangle$ for $π ∈ UpdateNames$ and $n ∈ EventNames \setminus UpdateNames$. Such errors are caused by an attempt to deliver in event $n↑$ to A during the update $π$ - recall that no event tokens between $τπ↑, τπ↓$ are allowed, specially no tokens of the form $?e$ expressing acceptance of events. Here, atomicity of $π$ is not ensured. 2) Erroneous traces of the form $w; \langle k \rangle$, where k is any error token and for every $π ∈ UpdateNames$, either both the tokens $τπ↑, τπ↓$ occur in w , or none of them. Here, the erroneous behavior has nothing common with the update-process itself, however A updated by B doesn’t behave how 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 has been updated by Alpha’. We present two examples of Z ’s behavior after the update: (**Ex. 1**): Let the components have the frame protocols $Prot_{Alpha} = (!a; ?p)^*; ?π1↑;!π1↓$, $Prot_{Beta} = (!m; ?n)^*$, $Prot_{Gamma} = (?a;!p)^* | (?m;!n)^*$, $Prot_{Alpha'} = (!a; ?p)^*$, $π1 ∈ UpdateNames$. Then the architecture protocol of Z is $Prot_{Arch-Z} = ((Prot_U.$

⁷ $L_1; L_2 = \{ \alpha; \beta : \alpha ∈ L_1, \beta ∈ L_2 \}$

$\text{Prot}_{\text{Alpha}} ; \text{Prot}_{\text{Alpha}'} \nabla_X \text{Prot}_{\text{Gamma}} \nabla_Y \text{Prot}_{\text{Beta}} = ((\tau a; \tau p)^*; \tau \pi 1 \uparrow; \tau \pi 1 \downarrow; (\tau a; \tau p)^*) \mid (\tau m; \tau n)^*$, where $\text{Prot}_{\text{U}'-\text{Alpha}} = (!a; ?p)^*; \tau \pi 1 \uparrow; \tau \pi 1 \downarrow$ generates $U_{\text{Alpha}'}$, $X = \{a \uparrow, a \downarrow, b \uparrow, b \downarrow, o \uparrow, o \downarrow, p \uparrow, p \downarrow\}$ and $Y = \{m \uparrow, m \downarrow, n \uparrow, n \downarrow\}$. From $\text{Prot}_{\text{Arch-Z}}$ we see that: 1) Atomicity of $\pi 1$ is ensured. 2) During the update $\pi 1$, Beta and Gamma can communicate, not violating the atomicity of $\pi 1$. (**Ex. 2**): Let Beta has the frame protocol NULL (i.e., it does nothing) and let $\text{Prot}_{\text{Alpha}} = !a; ?\pi 1 \uparrow; !\pi 1 \downarrow + !b; ?\pi 2 \uparrow; !\pi 2 \downarrow$, $\text{Prot}_{\text{Gamma}} = ?a; ?o; !p + ?b; (?o \mid !p)$, $\text{Prot}_{\text{Alpha}'} = !o; ?p$, where $\pi 1, \pi 2 \in \text{UpdateNames}$. In this case $\text{Prot}_{\text{Arch-Z}} = (\text{Prot}_{\text{U}'-\text{Alpha}} ; \text{Prot}_{\text{Alpha}'}) \nabla_X \text{Prot}_{\text{Gamma}} = \tau a; \tau \pi 1 \uparrow; \tau \pi 1 \downarrow; \tau o; \tau p + \tau b; (\tau \pi 2 \uparrow; \tau \pi 2 \downarrow; (\tau o \mid \tau p) + \tau \pi 2 \uparrow; \epsilon p \uparrow)$, $\text{Prot}_{\text{U}'-\text{Alpha}} = !a; ?\pi 1 \uparrow; !\pi 1 \downarrow + !b; \tau \pi 2 \uparrow; \tau \pi 2 \downarrow$, X is the same as in ex. 1. We see that: 1) The update $\pi 1$ is always atomic (there is no erroneous trace of the form $w; \langle \tau \pi 1 \uparrow; \epsilon e \rangle$ for an event e and $w \in \text{ACT}^*$). 2) The atomicity of $\pi 2$ 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 $\pi 2$ was ensured by another mechanism (interface locking, etc.), Alpha' could replace Alpha successfully.

In case of several successive updates (e.g. A replaced by A/B and later by A/B/C), we can always test the atomicity of a particular update as soon as we obtain a behavior description of the component being updated (this is the sense of “statically” verify via the consent operator whether the atomicity of an update is ensured). For example, to test the atomicity of update from A/B to A/B/C we need the knowledge of the frame protocols of A and B. On the contrary, to test whether the behavior of Z contains errors resulting from its composition (i.e. bad activity (not caused by violating atomicity of an update), no activity, and infinite activity), including the errors caused by C, the knowledge of frame protocol of C is necessary.

5. Evaluation and Related Work

The consent operator. The consent operator provides a means of modeling behavior of composed components (i.e. constructing architecture protocols), including three types of errors: bad activity, no activity, and infinite activity. The approach here is similar to CSP [19] with its failures semantics (capturing unaccepted actions), deadlocks and divergence. CSP provides generalized parallel operator \parallel_X , which combines traces similarly to \prod_X , and hiding operator $\backslash X$, which allows to convert the actions synchronized using \parallel_X into internal actions. The main difference is that in CSP the communication is symmetric, i.e. actions denoted by the same token are synchronized using \parallel_X and a failure occurs when, for the process defined as $P \parallel_X Q$, an action from X is provided just by one of the processes P, Q but not by the other. However, the semantics of a method call is asymmetric - emitting events without absorbing them is a bad activity error, however absorbing without emitting is not - except the case of a deadlock. Thus, modeling of method calls in CSP does not capture this asymmetry. As to CCS [13], the issue persists, since the composition operator \mid 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 \prod_x operator (or, better put, vice versa). In [5], parallel composition of behavior is not considered at all.

Errors in architecture protocols. Architecture protocols are proposed to be constructed by using the consent operator (instead of \prod_x , as in [16]), which in turn can generate erroneous traces. We have shown that the environment of an architecture is put into can 1) filter out some of the erroneous traces generated by the architecture protocol, and 2) create new erroneous traces. The issue (1) - restriction of an architecture's behavior by the environment - is discussed in [18]. However, this paper does not address faulty behavior; just restriction of the architecture's requirements. The issue (2) is very much related to the compliance of architecture and frame protocols, thoroughly discussed in [16]. Another approach, employing the consent operator for similar purpose, can be found in [1].

Dynamic updates. Using 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 none of the other components can call a method on A during the update, while the rest of the system is running, (performance benefit). Moreover, this technique can also selectively identify which of the several update events in a frame protocol of A would violate atomicity and which would not. To our knowledge there is no a similar work addressing the issue. Naturally, a testing of component update atomicity could be realized via CSP [19] with its failure semantics as well; in general we consider behavior protocols much easier to apply in a real architecture description language than CSP [16].

6. 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 determining behavior of a component architecture in a specific environment and for verifying the atomicity of runtime component updates. As a future work, we intend to focus on: 1) Allowing internal events to appear in frame protocols, because it is important to explicitly state that a component should be engaged in a particular feature of its functionality between handling some of the external communication events. Currently, internal events appear in architecture protocols only, so that the semantics of an architecture protocols differ from the one of the related frame protocol (which in turn provides less information that it could). 2) Enriching behavior protocols by *guards* to explicitly describe the condition under which a particular event is chosen in the case more continuation alternatives exist. Such an enhancement should also address the issue of internal/external choice [19,16]). 3) Analyzing the semantics of stopping of composed components. The question is how each component should contribute to the decision about end of processing and how such a decision should be coordinated. Addressing the issue may even need to modify the consent operator.

References

- [1] J. Adamek, "Enhancing Behavior Protocols", Master Thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2001
- [2] R.J. Allen, A Formal Approach to Software Architecture, doctoral dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997.
- [3] R.J. Allen, D. Garlan, A Formal Basis For Architectural Connection, ACM Transactions on Software Engineering and Methodology, Jul. 1997.
- [4] 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.
- [5] 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
- [6] D. Giannakopoulou, Model Checking for Concurrent Software Architectures, doctoral dissertation, Imperial College, University of London, Jan.1999.
- [7] 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.
- [8] C.R.Hofmeister, J.Atlee and J.Purtilo: Writing Distributed Programs in Polyolith, University of Maryland, November 1990.
- [9] T. Kalibera, P. Tuma, "Distributed Component System Based On Architecture Description: The SOFA Experience", Accepted at DOA 2002, Copyright (C) Springer-Verlag, Oct 2002
- [10] F. Leyman, "Web Services Flow Language (WSFL 1.0)", IBM Software Group, May 2001.
- [11] Microsoft COM Technology, <http://www.microsoft.com/com>
- [12] J. Magee, N. Dulay, S. Eisenbach, J.Kramer, "Specifying Distributed Software Architectures", 5th European Software Engineering Conference, Barcelona, Spain, 1995.
- [13] R. Milner, "A Calculus of Communicating Systems", LNCS 92, Springer-Verlag.
- [14] OMG Corba Component Model Specification, <http://www.omg.org/technology/documents/formal/components.htm>
- [15] 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.
- [16] F. Plasil, S. Visnovsky, "Behavior protocols for Software Components", IEEE Transactions on SW Engineering, 28 (9), 2002.
- [17] F. Plasil, S. Visnovsky, M. Besta, "Bounding Behavior via Protocols," Proceedings of TOOLS USA '99, Santa Barbara, CA, Aug. 1999.
- [18] R. Reussner, "Enhanced Component Interfaces to Support Dynamic Adaption and Extension", HICSS 2001.
- [19] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall, 1998.
- [20] SOFA project, <http://nenya.ms.mff.cuni.cz/thegroup/SOFA/sofa.html>.
- [21] Sun Enterprise Java Beans, <http://java.sun.com/products/ejb>
- [22] UML Resource Page, <http://www.omg.org/technology/uml/index.htm>
- [23] S. Visnovsky, "Modeling Software Components Using Behavior Protocols", doctoral dissertation, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2002

