

Component Composition Errors and Update Atomicity: Static Analysis*

Jiri Adamek¹, Frantisek Plasil^{1,2}

¹Charles 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>

²Academy 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 guaranteed thanks to the behavior of its current environment.

1 Introduction

Components have become an important part of software technologies. The existing component models range from simple, low-level granularity components in the industrial standards COM/DCOM [13] and EJB [23], to higher-level granularity component models where Polyolith [7], Darwin/Tracta [11], Wright [2] belong to the "classics", while CCM [16] and Fractal [4] are among the recent ones.

One of the benefits of software components is that they can be used as units of *dynamic update* (the code, i.e. the implementation of a component is replaced

*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).

by a new version at run time). In addition, unless the component is stateless, a transfer between the old and new version of the component state representation is to be done. There are several projects targeting the problem of dynamic updates, ranging from [8] to the recent works on run-time updating of Java classes, such as JDRUMS([3]), DJC([12]), and [20].

Frequently, the component models are enriched by some kind of behavior description, which reflects the dynamic nature of running software. While component models themselves describe the structure (or architecture) of the designed software, a behavior description gives the picture of interactions between parts of the structure, models of internal states of the software parts and the state changes caused by/influencing their communication. The big challenge is to develop a formal notation targeting the relation between communication/state changes and structural changes in dynamic software architectures. However, the problem is so complex, that only a few of formal approaches provide at least partial solution, e.g. [15].

Obviously, 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 [21], employed, e.g., in the Wright architecture description language [2], uses a 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], part of the Darwin ADL [11], uses a 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 [24] defines three packages intended 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. 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).

There are several reasons why to describe behavior formally: (1) Classic

interface descriptions cannot capture the order in which the methods of an interface may/should be called. Behavior description of the interfaces can express these additional constraints, so that the interfaces are better documented. (2) If a component is used as a part of a system, we can view the system as consisting of two (abstract) parts: the component and its environment, formed by all the other components of the system. Supposing both the component and environment are furnished with a behavior description rich enough to not only describe what the component resp. environment does, but also what it expects from its counterpart, we can compare the corresponding parts of the both behavior descriptions by using the methods of *equivalence checking*. A non-equivalence indicates a design error. (3) Using methods of *model checking* we can automatically test if a component-based system has certain properties (e.g. it is deadlock-free, rules of using critical sections are not violated, a final state of the system is always reachable, etc.)

In this paper, we discuss two behavior aspects of component-based systems: *erroneous behavior* and *atomicity of dynamic updates*. By an erroneous behavior we basically understand a behavior violating certain rules, especially the rules of communication among the components of a system. By atomicity of a dynamic update we mean that during the update, there can be no communication between the updated component and the rest of the system.

We analyze both these aspects together by asking: what the types of erroneous behavior are, how they arise; what the relation between erroneous behavior and update atomicity is, and how we can detect erroneous behavior / ensure update atomicity statically, e.g. at compile time, using behavior description of components.

The structure of the paper is as follows: In Sect. 2, to prepare background to propose a solution, we introduce our SOFA component model and *behavior protocols* - component behavior description based on regular languages. In Sect. 3, we analyze the problem of erroneous behavior and propose a solution: An erroneous behavior can be captured by *erroneous traces* generated by our *consent operator*. In Sect. 4 we discuss how update atomicity can be ensured and how the concept of erroneous behavior can help to solve this problem. In Sect. 5 we evaluate our contribution, while Sect. 6 concludes the paper.

2 Component models: bindings, updates and protocols

2.1 Bindings

A *binding* is one of the key concepts of software component systems. It explicitly expresses the fact that a component contains a reference to another component.

To illustrate the basic idea of component binding common to all higher-level models, we will use the SOFA component model (SOFTware Appliances) elaborated in our research team [17, 22, 9]; intending to keep its description here as simple as possible, we refer the reader to [22] 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,

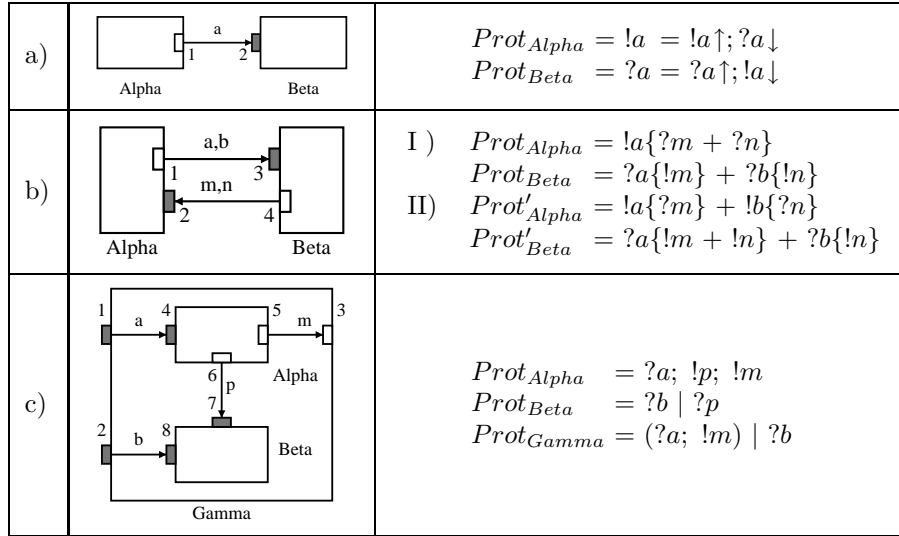


Figure 1: Examples of component systems and frame protocols

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 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 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 [18, 19], the coexistence of the frame and architecture view of a component is very important for refinement-based component design.

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.

2.2 Updates

The basic idea of (*run-time*) *component update* is very simple: The code (implementation) of a component is replaced by a new version. However, several difficulties appear when component update is analyzed in-depth. Mainly, the updated running component has its *state* formed by the values of the internal data structures, contexts of the running threads, opened files, network connections, etc. The state and the contexts have to be reflected in the continuation of the execution of the component, based on the new version of the code - *state transfer* problem [25].

In this paper, we discuss only the problems of thread contexts during an update and illustrate them on the update mechanism for SOFA component model [17, 26], which works as follows: 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 $C/A_1, C/A_2, \dots$. Assuming there is no communication activity between C and its environment (how this is achieved is discussed later), 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 [17, 26].

2.3 Behavior protocols

In [18] 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. 3.1), 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.

In "the middle" of an execution of a component C with a frame protocol P ,

assuming a sequence α of event tokens has been already performed, the set of *all next possible event tokens according to P* is defined as $\{t : \alpha < t \in L(P)\}$. For example, if P is $(?a; ?x) + (?a; ?y)$ and the events corresponding to $?a \uparrow; !a \downarrow$ were already performed, the all possible next events are determined by the set $\{?x \uparrow, ?y \uparrow\}$.

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 . As described in [18], an architecture protocol can be automatically generated from the frame protocols of the subcomponents of A .

For example, all the protocols specified on Fig. 1 are frame protocols. In particular, Fig. 1c specifies the frame protocols of *Alpha* and *Beta*, i.e. $?a; !p; !m$ and $?b|?p$ which means that *Alpha* accepts a call of a , then it calls p and m (in this order). On the contrary, *Beta* accepts calls of b and p in parallel. *Gamma* is composed of *Alpha* and *Beta*, so the architecture protocol of *Gamma* is automatically generated from the frame protocols of *Alpha* and *Beta*: $(?a; \tau p; !m)|?b$. The frame protocol of *Gamma* is defined as $(?a; !m)|?b$.

As explained in [18], 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 [22]) indicate that frame protocols are very important, as they provide more general information on the "interplay" of all the calls on the "component boundary".

A natural way to incorporate the update events in behavior specification of C (extending the approach introduced in [26]) 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. Formally, we introduce the set of *update event names* $UN = \{\pi, \pi_1, \pi_2, \dots\}$, $UN \subseteq EN$.

During the life-cycle of a component C , *dynamic checker* associated with CM_C is monitoring the behavior of C (using the frame protocol), allowing to predict what the next event on C may be.

3 Incorrect composition

3.1 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 Π_X for languages $L_1, L_2 \subseteq Act^*$ and $X \subseteq EN$ was introduced in behavior protocols [18]:

$L_1 \Pi_X L_2$ 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. 3.2 and so are other types of faulty behavior, *no activity* and *divergency*.

3.2 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

²In 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 .

³More 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 $!e \in S_P$).

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* $\varepsilon n \uparrow$, $\varepsilon n \downarrow$, $\varepsilon \emptyset$ and $\varepsilon \alpha$, ($n \in EN$), and *erroneous traces* of the form $w \langle e \rangle$, where w is a trace formed of non-error event tokens ($w \in ((ACT \setminus ErrorTokens)^*)^4$ and e at the end of the trace stands for the error token reflecting the type of 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 $\varepsilon \alpha$ 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. 3.1 — $!a\{?m\}+!b\{?n\}$ 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:

$$BA(L_A, L_B, X) = \{ w \langle \varepsilon e \rangle : (\exists u)(\exists v)(u \langle !e \rangle \in Prefix(L_A) \wedge v \in Prefix(L_B) \wedge v \langle ?e \rangle \notin Prefix(L_B) \wedge w \in (u \amalg_X v) \wedge e \in X) \}.$$

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: $\langle \tau m \uparrow; \tau m \downarrow; \varepsilon \emptyset \rangle$. Formally, the *no activity set* $NA(L_A, L_B, X)$ is defined as follows:

$$NA(L_A, L_B, X) = \{ w \langle \varepsilon \emptyset \rangle : (\exists u)(\exists v)(u \in Prefix(L_A) \wedge v \in Prefix(L_B) \wedge (u \notin L_A \vee v \notin L_B) \wedge w \in (u \amalg_X v) \wedge (\forall t \in (S_A \cup S_B) \setminus Tokens_?(X))(u \langle t \rangle \notin Prefix(L_A) \wedge v \langle t \rangle \notin Prefix(L_B))) \}.$$

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 , $w \in T^*$, $t \in T^\infty$, $T = Tokens_\tau(X) \cup (S_A \setminus (Tokens_?(X) \cup Tokens_!(X))) \cup (S_B \setminus (Tokens_?(X) \cup Tokens_!(X)))$. 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

⁴ $ErrorTokens = \{\varepsilon \emptyset, \varepsilon \alpha\} \cup \{\varepsilon n \uparrow : n \in EN\} \cup \{\varepsilon n \downarrow : n \in EN\}$

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 $w\beta <\varepsilon\alpha>$, 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 $\langle \tau a \uparrow; \tau a \downarrow; \tau m \uparrow; \tau m \downarrow; \tau a \uparrow; \tau a \downarrow; \tau m \uparrow; \tau m \downarrow; \dots \rangle$. It is represented by the set of erroneous traces of the form $\{ \langle \tau a \uparrow; \tau a \downarrow; \varepsilon\alpha \rangle, \langle \tau a \uparrow; \tau a \downarrow; \tau m \uparrow; \tau m \downarrow; \varepsilon\alpha \rangle, \langle \tau a \uparrow; \tau a \downarrow; \tau m \uparrow; \tau m \downarrow; \tau a \uparrow; \tau a \downarrow; \varepsilon\alpha \rangle, \dots \}$. 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:

$$DIV(L_A, L_B, X) = \{ w <\varepsilon\alpha> : (\exists u)(\exists v)(u \in L_A \wedge v \in Prefix(L_B) \wedge v \notin L_B \wedge w \in (u \Pi_X v) \wedge w \notin Prefix(NC(L_A, L_B, X) \cup (L_A \Pi_X L_B))) \},$$

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

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:

$$L_A \nabla_X L_B = (L_A \Pi_X L_B) \cup ER(L_A, L_B, X),$$

$$ER(L_A, L_B, X) = NC(L_A, L_B, X) \cup DIV(L_A, L_B, X) \cup DIV(L_B, L_A, X).^6$$

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 $X = \{a \uparrow, a \downarrow, m \uparrow, m \downarrow\}$):

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

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

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.

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 $\varepsilon \emptyset$. 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 , $t_A = \text{Trace}_A(t)$, $t_B = \text{Trace}_B(t)$. 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 Dynamic updates

As we stated in Sect. 2.2, a problem to be solved when designing an update mechanism is to cope with the contexts of threads executing the code of a component. To allow an update of a component C , we have to ensure during the update: (1) *component passivity* — no thread is executing a method of an interface of C ; (2) *update atomicity* — no other component (external to C) in the system calls a method of an interface of C . The motivation of asking for component passivity and update atomicity is that during an update, neither the architecture A_i of C nor C 's state are available. In this section, we discuss how (1) and, in particular, (2) can be ensured/tested.

To address this goal, we allow a designer of a frame protocol to use special update tokens, defining when the component manager can start an update. Thus, component passivity and update atomicity have to be ensured only at the moments corresponding to the appearance of update tokens in the frame protocol, as they determine the moments when an update is possible. More formally: we say that an update of a component C is *possible* after a given trace prefix tp , if $tp \langle ?\pi \uparrow \rangle$ (or $tp \langle ?\pi_n \uparrow \rangle$) is also a prefix of a trace generated by the frame protocol of C .

An update of a component C is controlled by CM_C , which caches update requests until an update is possible. The information on possibility of an update can be acquired either (a) from a dynamic checker (monitoring the communication of C and checking its compliance with C 's frame protocol), or (b) from the component builder CB_{A_i} (associated with the component architecture A_i). Both (a) and (b) have some flaws: A dynamic checker means performance overhead, while asking the component builder needs an extra functionality embodied in it by the component designer.

There are two models of getting the information on an update possibility from either the dynamic checker or component builder ("the source"). If *pull model* is used, CM_C calls the source when an update is required and by convention an answer is sent when an update is really possible. In *push model*, the source informs CM_C any time an update is possible; CM_C answers if update really starts or not.

4.1 Component passivity

A key problem here is the detection of *internal threads* — the threads which are created during a method call and terminate after the method call has been finished. Even if the creation and termination of such a thread were modeled in the frame protocol (e.g. as a fork and join token), it would not be possible to determine the number of internal threads for a given prefix of a traces statically from the frame protocol (because of the limited expressive power of the regular languages behind behavior protocols).

Thus, component passivity has to be tested at run-time. There are two strategies to do it: (a) Either the implementation of a component C has to ensure component passivity any time an update is possible, or (b) CM_C informs the component builder any time it decides to go ahead with an update (still, this can take place only if the update is possible).

4.2 Update atomicity

A component update has to be *atomic* - i.e. there should be no communication between a component C and its environment while C is being updated. There are several ways to ensure atomicity of an update of C , including: (i) The whole system is stopped. (ii) All the (provides) interfaces of C are locked, so that during the update any method call to C is deferred, as in [17]. (iii) By analyzing behavior protocols, it is statically tested whether another component could call a method of C during the update. The techniques (i) and (ii) worsen performance of the system (stopping all components is too pessimistic, locking means employing a wrapper). This is why we focus on (iii) and propose the following method of static testing of update atomicity via behavior protocols.

4.3 Addressing atomicity via protocols

Using the method described below, we can statically check whether the atomicity of a particular update is ensured (the negative cases will be reflected as erroneous traces). What remains to be done at run time is to make sure that all updates comply with the frame protocol.

Because update atomicity is a matter of communication among the components on a particular level of nesting (the components forming an architecture Z), it is not a property of a single component, but a property of Z . Let Z consists of frames F_1, \dots, F_n with frame protocols $Prot_{F_1}, \dots, Prot_{F_n}$. The proposed method verifies update atomicity in two steps: (1) *Local atomicity* is tested for every frame protocol $Prot_{F_1}, \dots, Prot_{F_n}$: a protocol is locally atomic, if in all traces generated by the protocol, between every pair of corresponding update tokens (i.e. $? \pi \uparrow$ and $! \pi \downarrow$ resp. $? \pi_n \uparrow$ and $! \pi_n \downarrow$) no other tokens occur. (2) If (1) succeeds, the protocols $Prot_{F_1}, \dots, Prot_{F_n}$ are composed together using the consent operator which yields the language of the architecture protocol $Prot_Z$.

If (1) succeeds, there cannot be an accepting token of the form $?e$ between a pair of update tokens. Thus any attempt to call a method on an interface of a frame F_i during an update results in a bad activity (Sect. 3.2) captured by a trace of the form $w < ? \pi \uparrow ; \varepsilon n \uparrow >$ where $w \in ACT^*$, $\pi \in UN$ and $n \in EN \setminus UN$.

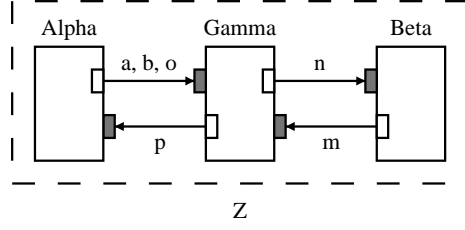


Figure 2: Updating the *Alpha* component

4.4 Two examples of update atomicity verification

In this section, we provide two examples of update atomicity verification, illustrating both update atomicity violation and validity. In addition, we show that if in a frame protocol there are two occurrences of update token pairs, the atomicity can be ensured for one of them and violated for the second.

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 $Prot_{Alpha} = ((!a; ?\pi_1 \uparrow; !\pi_1 \downarrow) + (!b; ?\pi_2 \uparrow; !\pi_2 \downarrow)); (!o \mid ?p)$, $Prot_{Gamma} = (?a; ?o; !p) + (?b; (?o \mid !p))$, where $\pi_1, \pi_2 \in UN$. In this case $Prot_Z = Prot_{Alpha} \nabla_X Prot_{Gamma} = (\tau a; \tau \pi_1 \uparrow; \tau \pi_1 \downarrow; \tau o; \tau p) + (\tau b; ((\tau \pi_2 \uparrow; \tau \pi_2 \downarrow; (\tau o \mid \tau p)) + (\tau \pi_2 \uparrow; \varepsilon p \uparrow)))$, 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 $w \langle \tau \pi_1 \uparrow; \varepsilon e \rangle$ 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 may be demanded 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).

5 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 [21]. Also, the operator Π_X combines traces in a similar way as the parallel operator \parallel_X in CSP, provided the hiding operator $\backslash X$ is also applied. The main difference is that in CSP the communication is symmetric, i.e. in $P \parallel_X Q$ 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 and Q . However, we claim that the semantics of a method call is asymmetric - emitting events without absorbing them is an error, but absorbing without emitting is not one (except for the case of a deadlock). Modeling of a method call in CSP fails to capture this asymmetry. As to CCS [14], the issue persists, since the composition operator $|$ allowing to synchronize inputs with outputs by generating internal actions does not address this asymmetry either. 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 local properties of states.

The authors of [5] 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. whether 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 [21] with failures semantics as well; in general, we consider behavior protocols much easier to apply in a real architecture description language than CSP [18].

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 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 require to modify the consent operator. 2) Analyzing different strategies of a component manager CM_C for coping with the internal communication in C during updates.

References

- [1] Adamek J, Plasil F. Behavior Protocols: Tolerating Faulty Architectures and Supporting Dynamic Updates. Technical Report 02/10, Department of Computer Science, University of New Hampshire, 2002.
- [2] Allen R. J, Garland D. A Formal Basis For Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, Jul. 1997.
- [3] Andersson J. A Deployment System for Pervasive Computing. ICSM, 2000.
- [4] Bruneton E, Coupaye T, Stefani J. B. The Fractal Composition Framework. Proposed Final Draft of Interface Specification Version 0.9. The ObjectWeb Consortium, Jun. 2002.
- [5] Farias A, Sudholt M. 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] Giannakopoulou D, Kramer J, Cheung S. C. 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.
- [7] Hofmeister C. R, Atlee J, Purtilo J. *Writing Distributed Programs in Polyolith*. University of Maryland, Nov. 1990.
- [8] Hofmeister C. Dynamic Reconfiguration of Distributed Applications. Ph.D. Thesis, University of Maryland, 1993.
- [9] Kalibera T, Tuma P. Distributed Component System Based On Architecture Description: The SOFA Experience. Proceedings of DOA 2002, Irvine, CA, USA, Springer-Verlag, LNCS 2519.
- [10] Leyman F. *Web Services Flow Language (WSFL 1.0)*, IBM Software Group, May 2001.
- [11] Magee J, Dulay N, Eisenbach S, Kramer J. Specifying Distributed Software Architectures. 5th European Software Engineering Conference, Barcelona, Spain, 1995.
- [12] Malabarba S, Pandey R, Gragg J, Barr E, Barnes J. F. Runtime support for type-safe dynamic Java classes. ECOOP'00, 2000.
- [13] Microsoft COM Technology, <http://www.microsoft.com/com>

- [14] Milner R. *Communication and Concurrency*. Prentice Hall, 1989.
- [15] Milner R. The polyadic pi-calculus: a tutorial. In *Logic and Algebra of Specification*, 203–246, Springer-Verlag, 1993.
- [16] OMG Corba Component Model Specification, <http://www.omg.org/technology/documents/formal/components.htm>
- [17] Plasil F, Balek D, Janecek R. SOFA/DCUP Architecture for Component Trading and Dynamic Updating. Proceedings of the ICCDS '98, Annapolis, IEEE Computer Soc. Press, 1998, pp. 43–52.
- [18] Plasil F, Visnovsky S. Behavior protocols for Software Components. *IEEE Transactions on SW Engineering*, 28(9), 2002.
- [19] Plasil F, Visnovsky S, Besta M. Bounding Behavior via Protocols. Proceedings of TOOLS USA '99, Santa Barbara, CA, Aug. 1999.
- [20] Redmond B, Cahill V. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. ECOOP'02, 2002.
- [21] Roscoe A. W. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
- [22] SOFA project, <http://nenya.ms.mff.cuni.cz/thegroup/SOFA/sofa.html>
- [23] Sun Enterprise Java Beans, <http://java.sun.com/products/ejb>
- [24] UML Resource Page, <http://www.omg.org/technology/uml/index.htm>
- [25] Vandewoude Y, Berbers Y. A Meta-Model Driven Methodology for State Transfer in Component-Oriented Systems. USE 2003, ETAPS, University of Warsaw, Poland.
- [26] Visnovsky S. Modeling Software Components Using Behavior Protocols. PhD Thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2002.