

# Static Checking for Missing Bindings of Components\*

Jiri Adamek, Frantisek Plasil

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

*Academy of Sciences of the Czech Republic  
Institute of Computer Science  
{plasil}@cs.cas.cz  
<http://www.cs.cas.cz>*

## Abstract

*Reuse is one of the key benefits of components. It inherently means that the functionality of a component may be employed only partially. This triggers the issue whether all of the component's interfaces have to be really bound to the other components in its current environment (missing binding problem). Assuming each of the components is equipped by its behavior protocol [12], we show that missing bindings can be statically identified via verification tools, in particular by employing the concept of bad activity error introduced in [1].*

Keywords: software component, behavior specification, behavior protocols, component composition

## 1. Introduction and motivation

### 1.1. Background (composition in component models)

The main reason for composing a software application from well specified components is reuse. A component can be embedded in different applications, potentially being adjusted via wrappers, adapters, etc. In a number of component models, ranging from classical Darwin to OMG CCM and Fractal [5], composition of an application is based on ties of the components' interfaces. In many component models the specification of components includes their formal behavior description allowing for automatic checking of composition errors and selected properties of a composed application (e.g., behavior specification written in CSP is an integral part of Wright, while Darwin/TRACTA employs FSP behavior specification

which is based on CSP and CCS but works with finite state spaces). In this paper, we focus on our SOFA component model [16], featuring behavior protocols [12] supported by verification tools.

Typically, a component A has *provides interfaces* as the reification of the services it provides, as well as *requires interfaces* indicating what services of external components it needs to utilize. In implementation-oriented view, an A's requires interface  $i_a$  reflects the need of getting in A a reference to another component B (to a provides interface  $i_b$  in B). We say that  $i_a$  is *tied* to  $i_b$ . By convention, a tie between a provides and a requires interface is called *binding* (there can be also provides-provides and requires-requires ties to forward references if component nesting is considered).

When reusing a component in a particular system, the need may be to employ only a part of the component's functionality. This inherently triggers the question whether it is necessary to bind all its interfaces to some interfaces of other components, or whether it is possible to leave some of them unbound. The notion of unbound interfaces, which either can (*missing bindings*) or cannot cause errors is not just a theoretical thought: there are several projects that consider this phenomenon, such as Kilim [11] and the OMG configuration and deployment framework [7]. In particular, participating in the ITEA OSMOSE project [17], we became familiar with the Kilim configuration framework, a product of an industrial partner, KELUA, used for "real-life" large Java applications, where bindings are defined "asymmetrically" via interface groups (slots) and "freelance" interfaces: Plugging a component A into a slot SB of a component B means that some of the interfaces in SB are

---

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

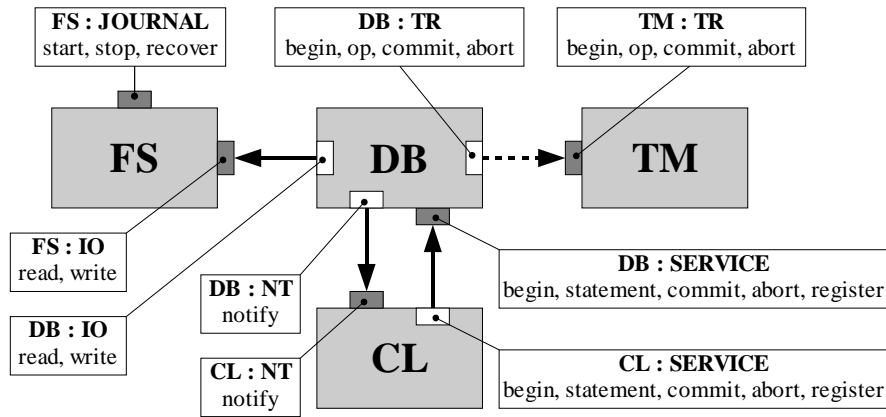


Figure 1: Example of SOFA application

implicitly bound to some freelance interfaces of A (association based on a naming convention). This way some of the freelance interfaces of A and interfaces in SB can remain unbound. An experience with this component model inspired us to writing this paper.

## 1.2. Goal and structure of the paper

The paper aims at addressing the issue mentioned above: Do all interfaces of a component have to be really bound — if some of them are not, does it cause any harm? In general, a component C may require a specific method on its particular provides interface to be called no matter what subset of the functionality is required. In a similar vein, the other components may expect certain reactions on specific requests to C. Thus, whether an interface of a component C has to be bound to an appropriate interface of another component (part of the *environment* of C) or not, depends on the behavior of C and its environment. In other words, C may feature *partial bindings* to its environment.

Here *behavior* is an abstraction of (1) the way the services provided by C can be used (the permitted sequences of method calls on provides interfaces), (2) the way C uses services provided by its environment (the permitted sequences of method calls via requires interfaces), and (3) the “interplay” of (1) and (2) (the permitted interleaving of the sequences (1) and (2)).

The goal of this paper is to show that the issue above can be resolved statically via operations upon behavior protocols. The remaining part of the paper is organized as follows. Sect. 2 provides an example “justifying” partial bindings and reviews the key concepts of behavior protocols important for the rest of the text. In Sect. 3 we show how missing bindings can be identified via the concept of consent operator. The idea is illustrated in the case study (Kilim focused) provided in Sect. 4. The rest of the paper is devoted to evaluation, related work, and conclusion.

## 2. Background

### 2.1. Example

**Setting.** Consider an application consisting of four components: DB (database server), CL (client), TM (transaction manager), and FS (filesystem) as illustrated in Fig. 1 where provides/requires interfaces are depicted as small dark/white rectangles and each interface is labeled by the name of the component which features it and its local name. For example, FS:IO is a provides interface of FS, while DB:NT is a requires interface of DB. Also, the methods of each interface are listed, e.g., DB:SERVICE features *begin*, *commit*, and *abort* for managing transactions, *statement* for performing database operations and *register* - registering for a callback on CL:NT (*notify*) if a particular data item gets modified. FS provides the IO interface (with *read* and *write*) and JOURNAL interface. Finally, TM externally manages transactions for DB: the *begin*, *commit* and *abort* methods are propagated from DB:SERVICE to TM:TR; every *statement* call on DB:SERVICE within a transaction is reported to TM via calling *op* on TM:TR. Note that each requires interface contains the same methods as the provides interface it is bound to.

**Unbound interfaces.** In Fig. 1, the FS component is capable of providing services through FS:IO and FS:JOURNAL. In the application considered, the provides interface FS:JOURNAL remains unbound – this is correct assuming the designer of FS specified FS:JOURNAL as optional (suppose FS can serve as a non-journaling file system as well). In a similar vein, if DB:TR were bound to TM:TR (dashed line), DB could employ the transaction manager TM. Assuming the client CL does not use the methods controlling the transaction processing on CL:SERVICE, there is no need for binding the requires interface DB:TR to TM:TR for correct functionality of the application. Thus, the bottom line is that some of the

provides and/or requires interfaces (FS:JOURNAL, DB:TR) may remain unbound depending upon the behavior expected both by the components (FS, DB) and their environment.

## 2.2. Behavior protocols - basics

In the rest of Sect. 2, we review with some simplifications the key concepts of behavior protocols [1, 2, 12] we developed within the SOFA project [16].

**Basic concepts.** Behavior protocols are expressions describing behavior at various levels of granularity (e.g. component, a group of components) by determining the possible (finite) sequences of events. A finite sequence of events (called a trace) reflects a run of the observed entity. Typically, an event is either a *request* for a method call, or its *response*. For a method  $m$ , a request for a call of  $m$  is denoted as  $m\uparrow$  and the response as  $m\downarrow$ . Emitting (!) and absorbing (?) a request (or response) is distinguished to allow modeling calls on provides and requires interfaces. For instance, the fact that  $C$  absorbs and executes a call of a method  $m$  via its provides interface  $PI$  is described as absorbing a request followed by emitting a response:  $?C:PI.m\uparrow ; !C:PI.m\downarrow$  where  $?C:PI.m\uparrow$  and  $!C:PI.m\downarrow$  are *event tokens* and  $;$  denotes sequencing. By convention, for such a sequence we use the abbreviation  $?C:PI.m$ . In a similar vein,  $!C:RI.m$  means  $C$  calls a method  $m$  through its requires interface  $RI$  (emitting a request followed by absorbing a response).

**Operators.** Behavior protocols are expressions built from event tokens, operators, and abbreviations. In addition to the standard regular expression operators “ $;$ ” (sequencing), “ $+$ ” (alternative), “ $*$ ” (repetition), new operators are defined to enhance expressiveness. These include “ $|$ ” for parallel execution and “ $\nabla$ ” (consent) for specific form of composition as explained in Sect. 2.3 and 2.4.

**Behavior protocols ver. CCS.** To make it easier to comprehend behavior protocols for those familiar with process algebras, we briefly outline the semantics of the basic operators of behavior protocols via the CCS means (a thorough analysis of behavior protocols’ mappings to process algebras is out of the scope of this paper). Supposing  $a\uparrow$  and  $a\downarrow$  are the CCS names that correspond to request and response of an event  $a$ , and by mapping  $?a\uparrow$  to  $a\uparrow$  and  $!a\downarrow$  to  $\bar{a}\downarrow$ , we can express the meaning of the basic operators of behavior protocols in CCS as follows: The “ $+$ ” operator directly corresponds to “ $+$ ” of CCS, i.e. the protocol  $?a\uparrow + ?b\uparrow$  is equivalent to the CCS process  $a\uparrow.0 + b\uparrow.0$  (where  $0$  denotes inactive process). In a similar vein, sequencing can be expressed via the prefix operator “ $.$ ”, e.g.  $?a\uparrow ; !a\downarrow$  as  $a\uparrow.\bar{a}\downarrow.0$ . The parallel operator “ $|$ ” provides interleaving of traces like “ $|$ ” of CCS does, but “ $|$ ” does not do any synchronization. Therefore, a protocol of the form  $P | Q$  has to be expressed as  $\mathbf{P} \parallel \mathbf{Q}$

$\mathbf{Q}$ , where  $\mathbf{P}, \mathbf{Q}$  are CCS processes equivalent to protocols  $P, Q$  and  $\{\}$  denotes empty set. The only basic operator which cannot be exactly mapped to CCS is “ $*$ ”. For example,  $R = (?a\uparrow;!a\downarrow)^*$  means calling the method  $a$  repeatedly finite number of times, while the CCS expression with the meaning “closest” to  $R$ , i.e.  $\mathbf{X} = \mathbf{0} + a\uparrow.\bar{a}\downarrow.\mathbf{X}$ , specifies both finite and infinite sequences of calls of  $a$ .

As an aside, behavior protocols do not support value-passing (events with parameters) and explicitly denoted internal state; however, it would not be difficult to add these features, still preserving the finiteness of the state space (in a similar way as FSP does).

**Frame protocol.** Advantageously, a behavior protocol can easily express the permitted interplay of method calls on the interfaces of a component (*frame protocol*). For example, a frame protocol of FS may take the form

```
ProtFS =
( ?FS:IO.read + ?FS:IO.write ) * |
( ?FS:JOURNAL.start ; ?FS:JOURNAL.recover * ;
  ?FS:JOURNAL.stop ) *
```

i.e. FS absorbs a sequence of `read` and `write` calls on FS:IO and (in parallel) on FS:JOURNAL a `start` call, followed by a sequence of `recover` calls and a `stop` call (both parallel scenarios can repeat a finite number of times). The frame protocol of DB is more complex as it captures the interplay on its four interfaces:

```
ProtDB =
(
  ?DB:SERVICE.statement { (!DB:IO.read + !DB:IO.write)* }
  +
  (
    ?DB:SERVICE.begin { !DB:TR.begin }
    ;
    ?DB:SERVICE.statement {
      (!DB:IO.read + !DB:IO.write + !DB:TR.op)*
    }
    ;
    (
      ?DB:SERVICE.commit { !DB:TR.commit }
      +
      ?DB:SERVICE.abort { !DB:TR.abort }
    )
  )
  +
  ?DB:SERVICE.register
  +
  !DB:NT.notify
)*
```

This illustrates another important abbreviation:  $?x.y\{<\text{some actions}>\}$  stands for  $?x.y\{<\text{some actions}>; !x.y\downarrow\}$ , e.g.  $?DB:SERVICE.commit\{!DB:TR.commit\}$  means that DB will react inside the execution of a `commit` called on SERVICE by calling `commit` on its requires interface TR.

**Key benefits - static analysis, readability, decidability.** One of the key benefits of behavior protocols is the ability to

provide the information needed for static analysis of component behavior (at design time). In [12] we showed how behavior compliance of the neighboring layers of components can be statically verified via behavior protocols, while in [1] we discussed how incompatible behavior of the components cooperating at the same layer can be statically detected. The latter option is briefly reviewed in Sect. 2.3 and 2.4. As emphasized in [12], we consider behavior protocols reminding regular expressions more readable than a process algebra's notation, and their expressive power strong enough to reasonably approximate behavior of components. Moreover, they always lead to finite state spaces (in contrast to CCS) and all the compliance relations are decidable.

### 2.3. Consent operator and group protocol

To support development process of a composed component it is very important to express the behavior of a *group* of components. Via behavior protocols this is captured by a *group protocol*. Considering a component group  $G$  composed of (disjoint) subgroups  $G_1, G_2$ , the group protocol  $\text{Prot}_G$  can be constructed from the group protocols of  $G_1$  and  $G_2$  via the consent operator:  $\text{Prot}_G = \text{Prot}_{G_1} \nabla_S \text{Prot}_{G_2}$  where  $S$  is the set of all events related to communication on the bindings between  $G_1$  and  $G_2$ . The group protocol of all the components forming an application (as in Fig. 1) can be incrementally constructed starting with frame protocols of the components (the resulting protocol is called architecture protocol in [12]). From this point of view, a frame protocol is a group protocol associated with a group consisting of just one component. The bottom line is that group protocol is primarily a technical concept, reflecting the partial results of incremental construction of architecture protocol from frame protocols.

Let us assume that the CCS processes  $\mathbf{CCS}_{G_1}, \mathbf{CCS}_{G_2}$  describe the same behaviors as the group protocols  $\text{Prot}_{G_1}, \text{Prot}_{G_2}$ . Provided  $S$  is the set of all events occurring in the communication between  $G_1$  and  $G_2$ , the meaning of  $\text{Prot}_{G_1} \nabla_S \text{Prot}_{G_2}$  is equivalent to  $(\mathbf{CCS}_{G_1} \mid \mathbf{CCS}_{G_2}) \setminus S$  in CCS except for the fact that  $\nabla_S$  in addition identifies composition errors as described in Sect. 2.4. In both cases, the key principle is that the composed subjects (group protocols, processes in CCS) are synchronized on dual events from  $S$  (i.e.  $?e, !e$  in behavior protocols,  $e, \bar{e}$  in CCS for  $e \in S$ ) resulting in *internal events* ( $\tau e$  in behavior protocols,  $\tau$  in CCS). The events which are not elements of  $S$  are in the result of  $\text{Prot}_{G_1} \nabla_S \text{Prot}_{G_2}$  arbitrarily interleaved.

We demonstrate the semantics of  $\nabla_S$  on our example (a full formal definition is provided in the Appendix). To get the architecture protocol of the application from Fig. 1, consent has to be applied three times (as it is commutative and associative, the order of composition is not important):

$$((\text{Prot}_{\text{CL}} \nabla_{S_1} \text{Prot}_{\text{DB}}) \nabla_{S_2} \text{Prot}_{\text{TM}}) \nabla_{S_3} \text{Prot}_{\text{FS}}.$$

For instance, the set  $S_1$  of all the events occurring on the bindings between DB and CL is

$$S_1 = \{ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{begin!}, \\ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{begin!}, \\ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{commit!}, \\ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{commit!}, \\ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{abort!}, \\ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{abort!}, \\ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{statement!}, \\ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{statement!}, \\ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{register!}, \\ \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{register!}, \\ \langle \text{DB}:\text{NT-CL}:\text{NT} \rangle.\text{notify!}, \\ \langle \text{DB}:\text{NT-CL}:\text{NT} \rangle.\text{notify!} \}.$$

Here,  $\langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{begin!}$  stands for a call request for `begin` emitted by `CL:SERVICE` and absorbed by `DB:SERVICE`. The set  $S_2$  contains the events on the bindings of the CL-DB group and TM ( $S_3$  is constructed in a similar way). For illustration, assuming the frame protocol of CL takes the form

$$\text{Prot}_{\text{CL}} = \\ ( !\text{CL}:\text{SERVICE}.\text{statement} + !\text{CL}:\text{SERVICE}.\text{register} + \\ ?\text{CL}:\text{NT}.\text{notify} )^*$$

the result of the first composition is

$$\text{Prot}_{\text{CL}} \nabla_{S_1} \text{Prot}_{\text{DB}} = \\ ( \\ \tau \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{statement} \{ \\ (!\text{DB}:\text{IO}.\text{read} + !\text{DB}:\text{IO}.\text{write})^* \\ \} + \\ \tau \langle \text{CL}:\text{SERVICE-DB}:\text{SERVICE} \rangle.\text{register} + \\ \tau \langle \text{DB}:\text{NT-CL}:\text{NT} \rangle.\text{notify} \\ )^*$$

Here, the  $\tau$  symbol denotes an internal event - activity inside the CL-DB group. The result indicates that `!DB:IO.read` and `!DB:IO.write` take place only as parts of a `statement` call execution (which is an internal activity of the group CL-DB).

Notice that the consent operator produces only those traces which are realizable. For instance, the part of  $\text{Prot}_{\text{DB}}$  describing transactional processing was “silently eliminated” in the result of the composition  $\text{Prot}_{\text{CL}} \nabla_{S_1} \text{Prot}_{\text{DB}}$ , as the client ( $\text{Prot}_{\text{CL}}$ ) does not use this functionality.

We emphasize that the references between CL and DB components form a cycle (of the length 2). In general, the consent operator can be applied to component groups with reference cycles of an arbitrary length.

### 2.4. Composition errors

Composition errors [1] are abstractions capturing “incompatible”, erroneous behavior of components bound together. Examples of such behavior include calling methods in a way violating the frame protocol of the callee and deadlock. Unlike typical process algebras (e.g. CCS, CSP),

composition errors reflect the inherent asymmetry of a procedure call - a call request emitted through a requires interface (such as !C:RI.m↑) has to be answered by the callee, while ?C:PI.m↑ in a protocol associated with a provides interface PI is just “an advertised willingness of C to take the call” – whether such event really happens is up to the other component bound to PI (caller takes the initiative, while callee is passive).

The consent operator identifies composition errors during the construction of a group protocol. Of the composition errors (fully fledged definitions in [1]), only *bad activity error* and *no activity error* are important for the purpose of this paper – we review the basic idea below.

A **bad activity error** occurs if a component A tries to call a method provided by a component B and the frame protocol of B does not allow for the call at that particular moment. Here we naturally assume that the requires interface of A through which the call is emitted is bound to a provides interface of B. For example, if a group was formed of CL and DB and the frame protocol of CL were

```
ProtCL' = !CL:SERVICE.commit
```

Prot<sub>CL</sub>' ∇<sub>S1</sub> Prot<sub>DB</sub> would result in a bad activity error, since Prot<sub>DB</sub> does not allow calling commit as the first event of any run:

```
ProtCL' ∇S1 ProtDB = ε<CL:SERVICE-DB:SERVICE>.commit↑
```

(an event token of the form ε<the name of the unabsorbed event> denotes a bad activity error). In this particular case, the resulting protocol specifies just a single event as this bad activity error occurs at the beginning of any run. This is a coincidence, since bad activity error is always the last event of a run.

**No activity error.** Considering again composition of two components, no activity error occurs when at least one of them can absorb an event, but none of them can emit an event. For example, if the frame protocol of CL were

```
ProtCL'' = !CL:SERVICE.begin
```

the composition of Prot<sub>CL</sub>'' and Prot<sub>DB</sub> would result in a no activity error (denoted by the event token ε∅):

```
ProtCL'' ∇S1 ProtDB =
τ<CL:SERVICE-DB:SERVICE>.begin{
  !DB:TR.begin
}; ε∅
```

Here, a no activity error occurs because none of the components CL and DB can emit an event (even if DB is able to absorb (alternatively) the DB:SERVICE.statement↑, DB:SERVICE.commit↑, and DB:SERVICE.abort↑ events) and one of the components (DB in this case) is not able to stop (as it has to

process either commit or abort before stopping). Similar to bad activity, a no activity error is the last event of a run.

The concept of bad activity and no activity errors can be naturally extended to a group of components (consisting of more than two components).

### 3. Missing bindings

In Sect. 2, we assumed all the interfaces of a component are bound to appropriate interfaces of other components in the system. In this section, we show how the consent operator can be employed to check whether an unbound interface could cause an error.

**Unbound interfaces.** Let us consider an application consisting of DB, CL, FS, and TM components from Fig. 1, where TM component is not used (since transactional processing is not required by the client CL, the DB:TR interface remains unbound - decided by the designer of the application). Obviously when considering a group of components, such as CL-DB, some of the interfaces have bindings going outside of the group (DB:IO and FS:IO), and some remain unbound by the design decision (DB:TR). In general: If an interface has undefined binding in a group G, it is either unbound, or the binding is defined in a higher level entity (group or parent component if nesting is considered).

**Unbound requires interface.** If a requires interface I<sub>R</sub> of a component C is unbound and an event e on I<sub>R</sub> is emitted during a run, we say e causes an *unbound requires error*.

Unbound requires errors occurring in a group of components G can be converted to bad activity errors by enhancing G by a feigned component F with empty behavior: Every unbound requires interface I<sub>R</sub> in G is bound to a provides interface I<sub>P</sub> of F, having the same type as I<sub>R</sub> (F is defined in such a way that it has all necessary provides interfaces). Let S be the set of all events on such bindings. As the behavior of F is empty (Prot<sub>F</sub> = NULL), F does not absorb any events. Therefore, every unbound requires error on any I<sub>R</sub> (within G, i.e. without F) results in a bad activity error on a binding to F captured by the following protocol (let Prot<sub>G</sub> be the group protocol of G):

```
ProtG-F = ProtG ∇S ProtF = ProtG ∇S NULL
```

The construction of the group G-F is described in detail in the Appendix. We illustrate the conversion on the following example: Assume again the group (DB-CL)-TM. If the frame the protocol of CL was

```
ProtCL''' = !CL:SERVICE.begin ; !CL:SERVICE.statement ;
           !CL:SERVICE.commit
```

the behavior of the group CL-DB would be described by

```
ProtCL-DB''' = ProtCL''' ∇S1 ProtDB =
τ<CL:SERVICE-DB:SERVICE>.begin {
  !DB:TR.begin
```

```

};
τ<CL:SERVICE-DB:SERVICE>.statement {
  (!DB:IO.read + !DB:IO.write + !DB:TR.op)*
};
τ<CL:SERVICE-DB:SERVICE>.commit {
  !DB:TR.commit
}

```

Since the DB:TR interface is unbound (Fig. 1), it is important to identify an unbound requires error which could occur in (CL-DB)-TM. To do so, we replace TM by F with a provides interface C:I<sub>p</sub> (corresponding to DB:TR) and bind DB:TR to F:I<sub>p</sub>. The resulting protocol  $\text{Prot}_{\text{CL-DB-F}}$  is

$$\text{Prot}_{\text{CL-DB}} \text{''} \nabla_S \text{Prot}_F = \text{Prot}_{\text{CL-DB}} \text{''} \nabla_S \text{NULL} =$$

```

τ<CL:SERVICE-DB:SERVICE>.begin!;
ε<DB:TR-C:Ip>.begin!

```

Clearly, only the request for the begin method ( $\tau\langle\text{CL:SERVICE-DB:SERVICE}\rangle.\text{begin!}$ ) is processed, after which an unbound requires error occurs. It is captured by the bad activity error  $\epsilon\langle\text{DB:TR-C:I}_p\rangle.\text{begin!}$ . Here S contains all the events on the new binding, i.e.

```

S = {
  <DB:TR-F:Ip>.begin!, <DB:TR-F:Ip>.begin!,
  <DB:TR-F:Ip>.op!, <DB:TR-F:Ip>.op!,
  <DB:TR-F:Ip>.commit!, <DB:TR-F:Ip>.commit!,
  <DB:TR-F:Ip>.abort!, <DB:TR-F:Ip>.abort!
}.

```

**Unbound provides interfaces.** An unbound provides interface can cause just the composition errors already defined in [1] - bad activity and no activity, since, from the point of view of a component C, it is not important whether a method on its provides interface I<sub>p</sub> is not called because I<sub>p</sub> is unbound, or because the other component bound to I<sub>p</sub> behaves incorrectly.

**Missing bindings - summary.** Both an unbound provides and unbound requires interface can cause a composition error. If a component C has an unbound interface, we say that C features partial bindings and if such unbound interface causes a composition error, C has a *missing binding*.

#### 4. Case study - missing bindings in Kilim

We demonstrate the usability of testing missing bindings on the Kilim configuration framework [11]. Kilim is not intended to be a full-featured component framework, but just to provide support for *configuration* - assembling parts of a complex Java application and distributing interface references among these parts. This task is done programmatically, reflecting the specified application structure. The rules of defining this structure are similar to defining a component application in an ADL. On the other hand, Kilim introduces additional constructs, not typically available in other component models (grouping of

interfaces, complex binding definitions). In particular, these additional constructs make testing of missing bindings an urgent issue.

The main difference between classical component models and Kilim is that once the Kilim framework has distributed interface references, the underlying structure of components is not used any more (they were used to control reference distribution); at that point, interface references become standard Java references, ready to be handled in any way the Java language allows. There are no restrictions, e.g. on passing a reference as a parameter to another “component”, etc. Needless to say Kilim does not provide any additional functionality typical for other component models, such as support for component updating, remote calls, connectors, etc.

#### 4.1. Kilim abstractions

The fundamental building block of a Kilim application is *template*. Basically, template is a component type, so that a component is a *template instance*. A template is a composable abstraction, capturing the encapsulated subsystem.

A template has *ports*. The concept of port is very similar to SOFA interface: a port is basically a container storing a Java reference or references (determined by arity of the port). A reference is either created by the template containing the port (*offered port*) or it is a copy of the value stored in an offered port of another template (*required port*). Assigning a copy of an offered port reference to a required port is done by port bindings.

A template can have *slots*. The purpose of a slot is to define a group of ports, which are bound to their counterparts (i.e. an offered port to a required port and vice versa) at the same time. Every port can be either a member of (at most one) slot, or can be defined outside the slots (*freelance interfaces* in Sect. 1.1). Every port has a *name*, which is unique within the slot in which the port is defined (or within the group of ports defined outside slots). Both categories of ports (offered and required) can be defined either in slots or outside of slots. Kilim applications are created by *assembling* template instances. During every assembling step, a slot S of a template A is plugged into a template B (for B, no particular slot is associated with the assembly step). As a result, every port P in S is bound to such a port P' defined in B (outside all B's slots), that has the same name as P. The important fact is that after the assembling step is finished not all ports in S have to be bound (if a counterpart with the same name is not found). Unbound ports can be either bound later by plugging S into another template instance (say C), or can remain unbound. In a similar vein, also some ports defined in B outside B's slots can remain unbound.

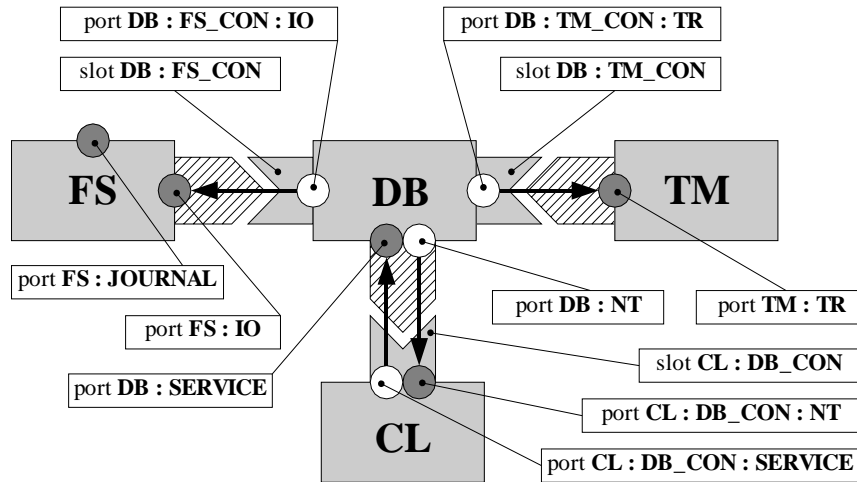


Figure 2: Example of Kilim application

As an example, Fig. 2 shows the SOFA application from Fig. 1 redesigned for the Kilim configuration framework. Again the application consists of four “components” (template instances) - FS, DB, TM, and CL. Instead of the SOFA interfaces, template instances feature offered and required ports (dark and white circles). A slot is captured as a grey polygon, plugging a slot into a template instance as a shaded white polygon, and port binding as a black arrow. Here, the CL:DB\_CON slot is plugged into the DB template instance, and DB:TM\_CON and DB:FS\_CON slots are plugged into TM and FS template instances.

#### 4.2. Missing bindings in Kilim

The concept of slots in Kilim was designed to allow the designer to define bindings at a higher level of granularity than single ports and to support the usage of just a part of a template’s functionality. As we already mentioned in Sect. 2 and in the beginning of Sect. 4, one of the consequences of such a complex approach is that not all ports have to be bound. Clearly, this triggers the demand for a tool testing whether the functionality of a template instance would be corrupted due to a missing binding.

The problem can be solved in the same way as in SOFA, assuming the templates are enriched by behavior protocols. Then unbound requires errors, bad activity, and no activity errors can be statically identified and the resulting information can help fix errors in design.

Although the current version of Kilim does not feature behavior protocols, it is not difficult to add them as well as a corresponding tool into the framework. The structure of Kilim applications is described in XML files, which can be easily extended by a special tag to capture “template protocols”. In the following example we illustrate this option on a part of the Kilim configuration file describing

the CL template. The tag describing its “template protocol” is shown in bold.

```

<template name = "CL_type">
  <slot name = "DB_CON" status = "public">
    <port arity = "1" name = "SERVICE"
      role = "required" status = "public"/>
    <port arity = "1" name = "NT"
      role = "offered" status = "public"/>
  </slot>
  ...
  <!-- Here, the mapping of ports to concrete Java
  classes and interfaces (containing the
  statement, register and notify methods)
  is described -->
  ...
  <protocol>
  (
    !DB_CON:SERVICE.statement +
    !DB_CON:SERVICE.register +
    ?DB_CON:NT.notify
  )*
  </protocol>
</template>

```

#### 5. Evaluation and related work

**Evaluation.** The idea presented in Sect. 3 allows for static (i.e. design time) identification of missing bindings. As they might be a result of a serious design flaw, it is very important to detect them in an early design stage - this is a key purpose of applying behavior protocols.

**Tools.** To employ the idea of detecting missing bindings in a supporting tool, we are currently working on a new version of protocol checker [16]. In the implementation, it is necessary to explicitly designate the events on unbound interfaces. Our approach is to enhance the consent operator by another parameter providing this information. Formally, the composition of group protocols takes the form  $Prot_{G1} \nabla_{S1,S2} Prot_{G2}$  where S1 is the set of events on the bindings between the groups and S2 the set of events on the unbound

interfaces in the resulting group. This way the feigned component is “implicit”; intentionally, we avoided using this view throughout the text since it could blur the basic idea. In addition, we plan to integrate the checker into the Kilim configuration framework [11], to test the idea on real-world applications written in Kilim.

*Scalability.* For simplicity, we illustrated missing bindings on “flat” applications (no component nesting was considered). However, both the consent operator (introduced in [1]) and the technique described in this paper work correctly for nested components as well: For a component  $C$  with some subcomponents forming a group  $G$ , the interfaces in  $G$  bound to interfaces of  $C$  (by delegation or subsuming [12]) are handled as if their bindings were undefined in  $G$  but defined in a higher-level group  $G'$  (i.e. the events on those interfaces never occur in an  $S_1$  nor  $S_2$  parameter of  $\nabla$ ).

*Synchrony/asynchrony.* Although we presented the idea on a component application assuming synchronous method calls, it is suitable for asynchronous communication as well (by separating request and response, behavior protocols allow for modeling message passing, etc.).

*Value passing.* The presented technique of checking for missing bindings is basically orthogonal to the issue of adding value passing (methods with parameters) and explicit internal state (Sect. 2.2) to behavior protocols. However, we have not considered such extensions because the state spaces generated would be too large to be handled by the tools we created.

**Related work.** The intuitively obvious idea that the behavior of a component depends on the way it interacts with other components within its environment (and that the other components may expect certain reactions on specific requests to the component) is reflected in a number of publications. In [4] this is addressed as “component mandatory calls”, while in [18] via (component) assembly. Here, actual “partial binding” is prohibited - all pins have to be connected in [18] terminology. However, our technique of checking for unbound requires errors could be seen as one of the reasoning frameworks if applied to the construction framework defined in [18]. The authors of [8] introduce the concept of (component) usage policy composed of activation policy and interaction policy - the latter is similar to interface protocol in SOFA. Being limited to provided interfaces only, it does not consider binding (proposed as a future work). This way, the missing binding problem could be lowered down to “skipping usage of some methods on a single interface”. The Alloy framework [6] considers cooperation among multiple plugins which inherently involves a decision on missing plugins. This is addressed via “strategies for deciding between different possible bindings to be provided in the form of preference functions written by plugin developers”.

In [3], the authors focus on testing interface compatibility. The main difference between their work and

ours is: (1) Via interface automata, they check whether there exists an environment in which a given interface (module) works correctly (“optimistic approach”), while we check whether a given component behaves properly in a specific environment. (2) They focus on the errors caused by the method call chains originating in the component (e.g. a method of an interface indirectly calls itself although not being reentrant), while we address the errors caused by the calls originating both in the environment and the component itself. The problem of identifying component's behavior errors while all potential environments are considered is addressed in [9] via an extension of classical model checking. They propose a model checking algorithm which, given a property, returns one of three possible results: (i) the component satisfies the property in all possible environments; (ii) it violates the property in all environments; (iii) all the environments in which the property is satisfied are characterized.

The idea that only a part of a component's functionality can be used in a specific environment (and, therefore, not all of the component's interfaces have to be bound) can be also found in [14]. However, the authors focus mainly on the reliability analysis (e.g., predicting mean time to failure), while we test whether a failure might occur due to an omitted binding. For “non-cyclic” architectures the problem of missing bindings can be addressed via protocols with counters [15].

## 6. Conclusion

In our view, partial bindings are inherent to component reuse, in particular when components are understood as design or composition units of the whole application (not just as plugins). In this paper, we have presented a simple way to identify missing bindings statically, at the component specification level. The key idea is to equip the component specifications with behavior protocols and apply the consent operator (originally defined in another of our works [1]). There is a tool available to evaluate the consent operation [16]. The proposed method works for any component model which can adopt behavior protocols, and scales well for hierarchical components, provided a behavior protocol is available for every component (at any level of nesting).

## 7. References

- [1] Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis, Accepted for publication in the Journal of Software Maintenance and Evolution: Research and Practice, 2004 (also <http://nenya.ms.mff.cuni.cz>)
- [2] Adamek, J.: Static Analysis of Component Systems Using Behavior Protocols, OOPSLA 2003 Companion, Anaheim, CA, USA, Oct 2003
- [3] Alfaro, L., Henzinger, T. A.: Interface Automata, Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM Press, 2001, pp. 109-120

- [4] Barnett, M. and all: Serious Specification for Composing components. Proceedings of the 6-th ICSE Workshop on Component-Based Software Engineering, Portland, OR, USA. Carnegie-Mellon and Monash University, 2003, <http://www.csse.monash.edu.au/~hws/cgi-bin/CBSE6/Proceedings/proceedings.cgi>
- [5] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. Seventh International Workshop on Component-Oriented Programming (WCOP02), at ECOOP 2002, Malaga, Spain
- [6] Chatley, R., Eisenbach, S., Magee, J.: Modeling a Framework for Plugins. Proceedings of the SAVCBS 2003 Workshop, Helsinki, Finland (<http://www.cs.iastate.edu/SAVCBS/>)
- [7] Deployment and configuration of Component-based distributed Applications Specification. OMG Adopted specification; ptc/03-07-08, July 2003
- [8] DePrince, W. Jr., Hofmeister, C.: Usage Policies for Components Proceedings of the 6-th ICSE Workshop on Component-Based Software Engineering, Portland, OR, USA. Carnegie-Mellon and Monash University, 2003
- [9] Giannakopoulou, D., Pasareanu, C. S., Barringer, H.: Assumption Generation for Software Component Verification, in Proceedings of the 17th IEEE International Conference on Automated Software Engineering, 2002
- [10] Hopcroft, J.E., Motwani R., Ullman J.D., Rotwani: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 2000.
- [11] The Kilim project (<http://kilim.objectweb.org/>)
- [12] Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002
- [13] Plasil, F., Visnovsky, S., Besta, M.: Behavior Protocols, Tech. Report 2000/7, Department of Software Engineering, Charles University, Prague, Oct. 2000, <http://nenya.ms.mff.cuni.cz>
- [14] Reussner, R.H., Poernomo, I.H., Schmidt, H.W.: Reasoning about Software Architectures with Contractually Specified Components. In: Component-Based Software Quality: Methods and Techniques., LNCS 2693, Springer 2003
- [15] Reussner, R.H.: Counter-constraint finite state machines: a new model of resource-bounded component protocols. In: Grosky, W., Plasil, F., (Eds.): Proceedings of SOFSEM, LNCS 2540, Springer, 2002
- [16] The SOFA project, <http://sofa.forge.objectweb.org/>
- [17] The OSMOSE project, <http://www.itea-osmose.org/>
- [18] Wallnau, K.C.: Volume III: A Technology for Predictable Assembly from Certifiable Components. CMU/SEI-2003-TR-009

## Appendix

**Consent operator - formal definition.** In this section we formally define the semantics of the consent operator in the standard terms of automata theory [10]. In [1], we have defined the semantics directly on languages (to not to be restricted to finite state spaces), while here we provide a

definition via automata, which is a bit more readable and less general, but still appropriate for behavior protocols.

Let us denote the set of traces specified by a protocol  $P$  by  $L(P)$  and the set of traces accepted by a finite automaton  $A$  by  $L(A)$ . Let  $P, Q$  be group protocols specifying behavior of component groups  $G_1, G_2$ . As justified in [13],  $L(P), L(Q)$  are regular languages, and, therefore, there exist nondeterministic finite automata  $A, B$  s.t.  $L(P) = L(A), L(Q) = L(B)$  and the number of the states of both  $A, B$  is minimal. Let  $S$  be a set of events. We define the semantics of the consent operator by construction of a (nondeterministic) finite automaton  $D$  s.t.  $L(D) = L(P \nabla_S Q)$ . The construction is done in two steps: In the first step a (nondeterministic) automaton  $C$  is constructed, which accepts all the traces reflecting correct communication between  $G_1, G_2$  (i.e. the traces containing no composition errors), and also the traces with bad activity and no activity errors. In the second step,  $D$  is constructed in such a way that it accepts all the traces which are accepted by  $C$  and in addition it accepts also the traces expressing *divergence* errors, denoted by the event token  $\varepsilon^\infty$  (we do not discuss divergence errors in this paper as they are not relevant to the problem of incomplete bindings). Basically, if a divergence error occurs in  $P \nabla_S Q$ , the groups  $G_1, G_2$  might never stop their communication.

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

We define  $C = (\text{Act}, Q_C, q_C, F_C, N_C)$ , where  $Q_C = Q_A \times Q_B \cup \{\text{error}\}$ ,  $q_C = (q_A, q_B)$ ,  $F_C = F_A \times F_B \cup \{\text{error}\}$ . A transition is an element of  $N_C$  if and only if it is deduced by one of the following rules:

- $((q_1, q_2), ?e, (q_1', q_2')) \in N_C$  if  $(q_1, ?e, q_1') \in N_A$  &  $e \notin S$
- $((q_1, q_2), ?e, (q_1, q_2')) \in N_C$  if  $(q_2, ?e, q_2') \in N_B$  &  $e \notin S$
- $((q_1, q_2), !e, (q_1', q_2')) \in N_C$  if  $(q_1, !e, q_1') \in N_A$  &  $e \notin S$
- $((q_1, q_2), !e, (q_1, q_2')) \in N_C$  if  $(q_2, !e, q_2') \in N_B$  &  $e \notin S$
- $((q_1, q_2), t, (q_1', q_2')) \in N_C$  if  $(q_1, t, q_1') \in N_A$ , where  $t$  is either  $\tau e, \varepsilon e, \varepsilon \emptyset$  or  $\varepsilon^\infty$
- $((q_1, q_2), t, (q_1, q_2')) \in N_C$  if  $(q_2, t, q_2') \in N_B$ , where  $t$  is either  $\tau e, \varepsilon e, \varepsilon \emptyset$  or  $\varepsilon^\infty$
- $((q_1, q_2), \tau e, (q_1', q_2')) \in N_C$  if  $(q_1, !e, q_1') \in N_A$  &  $(q_2, ?e, q_2') \in N_B$  &  $e \in S$
- $((q_1, q_2), \tau e, (q_1', q_2')) \in N_C$  if  $(q_1, ?e, q_1') \in N_A$  &  $(q_2, !e, q_2') \in N_B$  &  $e \in S$
- $((q_1, q_2), \varepsilon e, \text{error}) \in N_C$  if  $(q_1, !e, q_1') \in N_A$  & (there is no  $q_2' \in Q_B$  s.t.  $(q_2, ?e, q_2') \in N_B$ ) &  $e \in S$
- $((q_1, q_2), \varepsilon e, \text{error}) \in N_C$  if  $(q_2, !e, q_2') \in N_B$  & (there is no  $q_1' \in Q_A$  s.t.  $(q_1, ?e, q_1') \in N_A$ ) &  $e \in S$
- $((q_1, q_2), \varepsilon \emptyset, \text{error}) \in N_C$  if (there is no  $(q_1, t, q_1') \in N_A$  s.t.  $t$  is either  $!e, \tau e, \varepsilon e, \varepsilon \emptyset$  or  $\varepsilon^\infty$ ) & (there is no  $(q_2, t, q_2') \in N_B$  s.t.  $t$  is either  $!e, \tau e, \varepsilon e, \varepsilon \emptyset$  or  $\varepsilon^\infty$ ) &  $(q_1 \notin F_A$  or  $q_2 \notin F_B)$ .

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

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

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

(iii)  $q_1 \in F_A$  or  $q_2 \in F_B$ .

The resulting automaton is defined as  $D = (\text{Act}, Q_C, q_C, F_C, N_D)$ , where

$$N_D = N_C \cup \{ (q, \varepsilon, \text{error}) : q \text{ is a divergence state } \}.$$

### Converting unbound requires errors to bad activity

**errors.** Let  $G$  be a group of components having the group protocol  $\text{Prot}_G$  and let  $R_1, \dots, R_n$  be all the unbound requires interfaces of components in  $G$ . The detection of unbound requires errors is done in four steps.:

(1) A feigned component  $F$  is created: For every  $R_i, 1 \leq i \leq n$ , there is a provides interface  $F:P_i$  corresponding to  $R_i$  in  $G$ . At the same time,  $F$  does not accept any events, nor emits any events, so that its frame protocol is  $\text{Prot}_F = \text{NULL}$ .

(2) Every  $R_i$  is bound to  $F:P_i$ . Let  $S$  be the set of all events on these bindings, i.e.

$$S = \{ \langle R_i-F:P_i \rangle.e \mid e \text{ is an event on } R_i \}$$

At this point, all the originally unbound requires interfaces in  $G$  are bound to the provides interfaces of  $F$ . Note that the events on unbound interfaces are represented in  $S$ , although events on interfaces with undefined bindings which are not unbound (bindings defined in a group  $G'$  including  $G$ ) are not represented in  $S$ .

(3) Using the consent operator, the following protocol is constructed:

$$\text{Prot}_{G-F} = \text{Prot}_G \nabla_S \text{Prot}_F = \text{Prot}_G \nabla_S \text{NULL}.$$

(4) If there is a bad activity error of the form  $\varepsilon \langle R_i-C:P_i \rangle.e$  in a sequence specified by  $\text{Prot}_{G-F}$ , then there is an unbound requires error on the interface  $R_i$ .