

# Partial Bindings of Components - Any Harm?\*

Jiri Adamek<sup>1</sup>, Frantisek Plasil<sup>1,2</sup>

<sup>1</sup>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>,  
phone: +420 2 2191 4266, fax: +420 2 2191 4323

<sup>2</sup>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 [19], 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].*

## 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 [16] to OMG CCM [18] and Fractal [7], composition of an application is based on ties of the components' interfaces. In many component models the specification of components includes their formal behavior specification allowing for an automatic checking of composition errors as well as selected properties of the composed application. E.g., a behavior specification written in CSP [23] is an integral part of

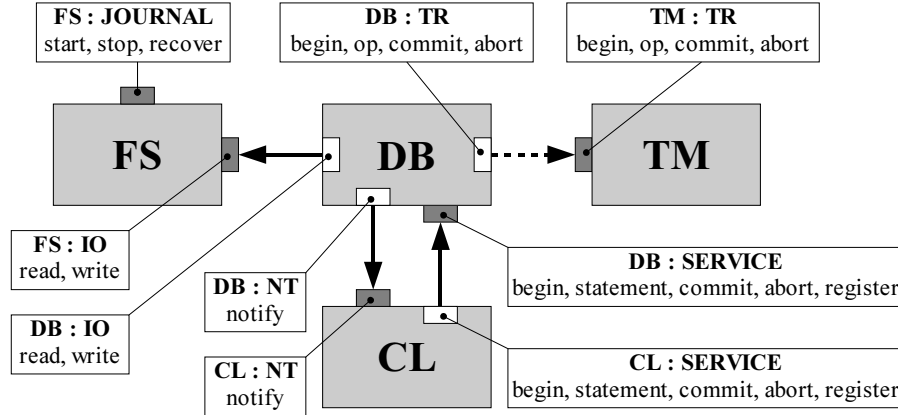
Wright [5], while Darwin/TRACTA [11] employs FSP behavior specification which is based on CSP and CCS [17] (but works with finite state spaces). In this paper, we focus on our SOFA component model [24], featuring behavior protocols [19] supported by verification tools.

Typically, a component A has *provides interfaces* as the reification of the services it offers, as well as *requires interfaces* indicating what services of external components it needs to utilize. In implementation-oriented view, an A's requires interface *ia* reflects the need of getting in A a reference to another component B (to a provides interface *ib* in B). If this is the case, we say that *ia* is *bound* to *ib*. Binding is a special case of interface *tie*, which includes also provides-provides and requires-requires ties, both being an abstraction for forwarding references when 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 component models in which one can face this phenomenon (even though none of the models mentions it explicitly), such as Kilim [14], the OMG configuration and deployment framework [9], and Fractal [7]. In particular, participating in the ITEA OSMOSE project [26], we became familiar with the Kilim configuration framework, used for "real-life" large

---

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



**Figure 1: Example of component application**

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 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 write this paper. A detailed analysis of the Kilim component model and its relation to the problem of missing bindings can be found in [2].

## 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 its 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. If partial bindings cause an error, we talk about missing binding(s).

In this context, *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 the operations defined for 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 consent operator [1]. The rest of the paper is devoted to evaluation, related work, and conclusion.

Also, the whole idea is illustrated in the case study provided in an extended version of this paper available as a TR [2].

## 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 where it is instantiated, and its local name. For example, FS:IO is a provides interface of FS, while DB:NT is a requires interface of DB (IO and NT are the local names of these interfaces). 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 we published earlier [1,3,19] and developed within the SOFA project [24]. Also, we explain the semantics of behavior protocol operators via comparison with CCS [17].

There are several reasons why we find behavior protocols to be more suitable for capturing behavior of software components than classical process algebras (such as CCS or CSP). Some of the behavior protocols' benefits (static analysis, readability, decidability) are mentioned at the end of this section; more details can be found in [19].

**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 [17] means (a thorough analysis of behavior protocols' mappings to process algebras is out of the scope of this paper). As meaning of behavior protocols is defined via traces [19], we compare the meaning with the trace semantics of CCS expressions.

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  $P \mid_{\emptyset} \mid_{\emptyset} Q$ , where  $P, Q$  are CCS processes equivalent to protocols  $P, Q$  and  $\{\}$  denotes empty set (the  $\mid_M \mid_N$  operator, where  $M, N$  are sets of events, is not a basic CCS operator; it can be found in [17]). 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.  $X = 0 + a^\uparrow.\bar{a}^\downarrow.X$ , 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

$$\begin{aligned} \text{Prot}_{\text{FS}} = & \\ & (?FS:IO.read + ?FS:IO.write)^* | \\ & (?FS:JOURNAL.start ; ?FS:JOURNAL.recover^* ; \\ & ?FS:JOURNAL.stop)^* \end{aligned}$$

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  $?x.y\{<\text{some actions}>; !x.y\}$ , 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 [19] 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 [19], 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  $Prot_G$  can be constructed from the group protocols of  $G_1$  and  $G_2$  via the consent operator:  $Prot_G = Prot_{G_1} \nabla_S Prot_{G_2}$  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 [19]). 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  $CCS_{G_1}, CCS_{G_2}$  describe the same behaviors as the group protocols  $Prot_{G_1}, Prot_{G_2}$ . Provided S is the set of all events occurring in the communication between  $G_1$  and  $G_2$ , the meaning of  $Prot_{G_1} \nabla_S Prot_{G_2}$  is equivalent to  $(CCS_{G_1} \mid CCS_{G_2}) \setminus S$  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  $Prot_{G_1} \nabla_S Prot_{G_2}$  arbitrarily interleaved.

We demonstrate the semantics of  $\nabla_S$  on the following example (formal definition of  $\nabla_S$  is provided in [1,2]). 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):

$$((Prot_{CL} \nabla_{S1} Prot_{DB}) \nabla_{S2} Prot_{TM}) \nabla_{S3} Prot_{FS}$$

For instance, the set S1 of all the events occurring on the bindings between DB and CL is

$$S1 = \{ \langle CL:SERVICE-DB:SERVICE \rangle.begin!, \langle CL:SERVICE-DB:SERVICE \rangle.begin!, \langle CL:SERVICE-DB:SERVICE \rangle.commit!, \langle CL:SERVICE-DB:SERVICE \rangle.commit!, \langle CL:SERVICE-DB:SERVICE \rangle.abort!, \langle CL:SERVICE-DB:SERVICE \rangle.abort!, \langle CL:SERVICE-DB:SERVICE \rangle.statement!, \langle CL:SERVICE-DB:SERVICE \rangle.statement!, \langle CL:SERVICE-DB:SERVICE \rangle.register!, \langle CL:SERVICE-DB:SERVICE \rangle.register!, \langle DB:NT-CL:NT \rangle.notify!, \langle DB:NT-CL:NT \rangle.notify! \}.$$

Here,  $\langle CL:SERVICE-DB:SERVICE \rangle.begin!$  stands for a call request for `begin` emitted by CL:SERVICE and absorbed by DB:SERVICE. The set S2 contains the events on the bindings of the CL-DB group and TM (S3 is constructed in a similar way). For illustration, assuming the frame protocol of CL takes the form

$$Prot_{CL} = (!CL:SERVICE.statement + !CL:SERVICE.register + ?CL:NT.notify)*$$

the result of the first composition is

```

ProtCL ∇S1 ProtDB =
(
  τ<CL:SERVICE-DB:SERVICE>.statement{
    (!DB:IO.read + !DB:IO.write)*
  } +
  τ<CL:SERVICE-DB:SERVICE>.register +
  τ<DB:NT-CL:NT>.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  $\text{Prot}_{DB}$  describing transactional processing was “silently eliminated” in the result of the composition  $\text{Prot}_{CL} \nabla_{S1} \text{Prot}_{DB}$ , as the client ( $\text{Prot}_{CL}$ ) 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, or a 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 was

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

$\text{Prot}_{CL}' \nabla_{S1} \text{Prot}_{DB}$  would result in a bad activity error, since  $\text{Prot}_{DB}$  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  $\epsilon$ <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 the 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  $\text{Prot}_{CL}''$  and  $\text{Prot}_{DB}$  would result in a no activity error (denoted by the event token  $\epsilon\emptyset$ ):

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

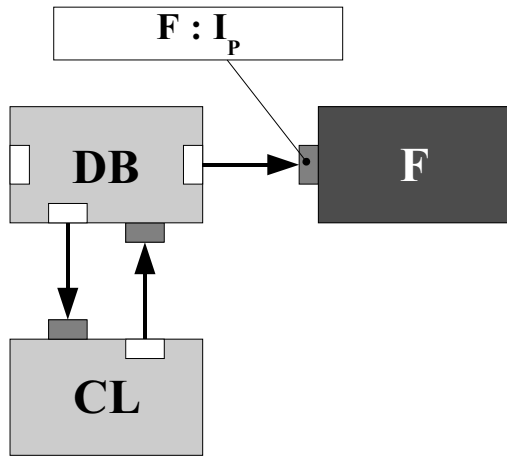
Here, a no activity error occurs because none of the components CL and DB can emit an event (even though DB is able to absorb (alternatively) the `DB:SERVICE.statement↑`, `DB:SERVICE.commit↑`, and `DB:SERVICE.abort↑` events) and one of the components (DB in this case) is not able to stop (as it has to 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  $n \geq 2$  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 (*missing binding problem*).

**Undefined bindings and 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: we say that an interface has *undefined binding in a group G*, if (i) the tie of the interface



**Figure 2: Usage of a feigned component F**

is defined within a group  $G'$  which contains  $G$ , (ii) the tie of the interface is defined for the parent component of  $G$  (if component nesting is considered), or (iii) the interface is not tied at all (*unbound interface*).

**Unbound requires interface.** If a requires interface  $I_R$  of a component  $C$  is unbound and an event  $e$  on  $I_R$  is emitted during a run, we say  $e$  causes an *unbound requires error*. This is an extension to the concept of composition errors as defined in [1]. In this section, we show how to transform the problem of checking for unbound requires errors to the problem of checking for bad activity errors (done via the consent operator, defined in [1]).

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  $I_R$  in  $G$  is bound to a provides interface  $I_p$  of  $F$ , having the same type as  $I_R$ .  $F$  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 ( $\text{Prot}_F = \text{NULL}$ ),  $F$  does not absorb any events. Therefore, every unbound requires error on any  $I_R$  results in a bad activity error on a binding to  $F$  captured by the following protocol (let  $\text{Prot}_G$  be the group protocol of  $G$ ):

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

We illustrate the conversion on the following example: Assume again the group DB-CL (Fig. 1). If the frame protocol of CL was

$$\text{Prot}_{\text{CL}} = \text{!CL:SERVICE.begin}; \text{!CL:SERVICE.statement}; \text{!CL:SERVICE.commit}$$

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

$$\begin{aligned} \text{Prot}_{\text{CL-DB}} &= \text{Prot}_{\text{CL}} \nabla_{S_1} \text{Prot}_{\text{DB}} = \\ &\tau\langle\text{CL:SERVICE-DB:SERVICE}\rangle.\text{begin} \{ \\ &\quad \text{!DB:TR.begin} \\ &\}; \\ &\tau\langle\text{CL:SERVICE-DB:SERVICE}\rangle.\text{statement} \{ \\ &\quad (\text{!DB:IO.read} + \text{!DB:IO.write} + \text{!DB:TR.op})^* \\ &\}; \\ &\tau\langle\text{CL:SERVICE-DB:SERVICE}\rangle.\text{commit} \{ \\ &\quad \text{!DB:TR.commit} \\ &\} \end{aligned}$$

Since the DB:TR interface is unbound, it is important to identify an unbound requires error which could occur in CL-DB. To do so, we introduce  $F$  with a provides interface  $F:I_p$  (corresponding to DB:TR) and bind DB:TR to  $F:I_p$  (Fig. 2). The resulting protocol  $\text{Prot}_{\text{CL-DB-F}}$  is

$$\begin{aligned} \text{Prot}_{\text{CL-DB-F}} &= \text{Prot}_{\text{CL-DB}} \nabla_S \text{Prot}_F = \text{Prot}_{\text{CL-DB}} \nabla_S \text{NULL} = \\ &\tau\langle\text{CL:SERVICE-DB:SERVICE}\rangle.\text{begin!}; \\ &\epsilon\langle\text{DB:TR-F:I}_p\rangle.\text{begin!} \end{aligned}$$

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  $\epsilon\langle\text{DB:TR-F:I}_p\rangle.\text{begin!}$ . Here  $S$  contains all the events on the new binding, i.e.

$$\begin{aligned} S = \{ \\ &\langle\text{DB:TR-F:I}_p\rangle.\text{begin!}, \quad \langle\text{DB:TR-F:I}_p\rangle.\text{begin!}, \\ &\langle\text{DB:TR-F:I}_p\rangle.\text{op!}, \quad \langle\text{DB:TR-F:I}_p\rangle.\text{op!}, \\ &\langle\text{DB:TR-F:I}_p\rangle.\text{commit!}, \quad \langle\text{DB:TR-F:I}_p\rangle.\text{commit!}, \\ &\langle\text{DB:TR-F:I}_p\rangle.\text{abort!}, \quad \langle\text{DB:TR-F:I}_p\rangle.\text{abort!} \\ &\}. \end{aligned}$$

**Unbound provides interfaces.** Since, from the point of view of a component  $C$ , it is not observable whether a method on its provides interface  $I_p$  is not called because  $I_p$  is unbound, or because the other component bound to  $I_p$  behaves incorrectly, an unbound provides interface can cause just the composition errors already defined in [1] - bad activity and no activity.

**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. Evaluation and Related Work

**Evaluation.** As missing bindings might be a result of a serious design flaw, it is very important to detect them statically at an early design stage - this is a key purpose of applying behavior protocols.

*Tools.* To detect missing bindings by a supporting tool, we are currently working on a new version of protocol checker stemming from [15], which implements detection of

unbound requires errors. In addition we work also on integrating the checker into the Fractal component model.

*Scalability.* For simplicity, we illustrated missing bindings on “flat” applications (no component nesting was considered). However, the consent operator works correctly for nested components as well: For a component  $C$  with some subcomponents forming a group  $G$ , the interfaces in  $G$  being tied to interfaces of  $C$  (by provides-provides or requires-requires ties) are handled as if their bindings were undefined in  $G$  but defined in a higher-level group  $G'$ .

*Synchrony/asynchrony.* Although we presented the idea of checking for missing bindings on an example based on synchronous method calls, it is easy to show that it works for asynchronous communication as well (by separating request and response as presented in the original paper on behavior protocols [19]).

*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. We have not intentionally considered such extensions because (i) behavior protocols are designed to reflect a high-level view of a component architecture and its behavior, and (ii) the state space to be handled by a tool tends to be extremely large and require specific techniques (such as BDD and behavior abstractions), and moreover, efficiency is still a bottleneck.

**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 [6] this is addressed as “component mandatory calls”, while in [27] via (component) assembly. In the latter, actual “partial binding” is prohibited - all pins have to be connected. 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 [27]. The authors of [10] 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 [8] 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 [4], 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 [12] 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 [21]. 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 [22]. In [25], the techniques of dependability analysis are applied to component architectures, addressing (among others) the problem of missing bindings. However, as a specification of dependencies among events does not provide all the information provided by a behavior specification, using dependencies an unbound interface can be pessimistically classified as causing unbound requires error, while our algorithm reports (correctly) that no error occurs. In a similar vein, Fractal [7] uses a simple pessimistic approach: All the interfaces not explicitly marked as optional have to be bound, which addresses the missing binding problem at a very coarse granularity and ignores the context of a particular environment.

## 5. 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 [24]. 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).

## 6. 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., Plasil, F., “Static Checking for Missing Bindings of Components”, Tech. Report No. 2004/3, Department of Software Engineering, Charles University, Prague, March 2004, <http://nenya.ms.mff.cuni.cz>
- [3] Adamek, J., “Static Analysis of Component Systems Using Behavior Protocols”, OOPSLA 2003 Companion, Anaheim, CA, USA, Oct 2003
- [4] 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
- [5] Allen R. J., Garland D., “A Formal Basis For Architectural Connection”, *ACM Transactions on Software Engineering and Methodology*, Jul. 1997.
- [6] Barnett, M. et. al., “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>
- [7] 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
- [8] 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/>)
- [9] “Deployment and configuration of Component-based distributed Applications Specification”, OMG Adopted specification, ptc/03-07-08, July 2003
- [10] 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
- [11] 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), pp. 7-35, January 1999.
- [12] 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
- [13] Hopcroft, J.E., Motwani R., Ullman J.D., Rotwani, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 2000
- [14] The Kilim project (<http://kilim.objectweb.org/>)
- [15] Mach, M., Plasil, F., “Addressing State Explosion in Behavior Protocol Verification”, Proceedings of SNP’04, Beijing, China, Jun 2004
- [16] Magee, J., Dulay, N., Eisenbach, S., Kramer, J., “Specifying Distributed Software Architectures”, Proceedings of the 5th European SW Eng. Conference, Barcelona, Spain, 1995.
- [17] Milner, R., *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag.
- [18] O M G C o r b a C o m p o n e n t M o d e l (<http://www.omg.org/technology/documents/formal/components.htm>)
- [19] Plasil, F., Visnovsky, S., “Behavior Protocols for Software Components”, *IEEE Transactions on Software Engineering*, vol. 28, no. 11, Nov 2002
- [20] 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>
- [21] Reussner, R.H., Poernomo, I.H., Schmidt, H.W., “Reasoning about Software Architectures with Contractually Specified Components”, *Component-Based Software Quality: Methods and Techniques*, LNCS 2693, Springer 2003
- [22] 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
- [23] Roscoe A. W., *The Theory and Practice of Concurrency*, Prentice-Hall, 1998.
- [24] The SOFA project, <http://sofa.forge.objectweb.org/>
- [25] Stafford, J.A., Wolf, A.L., “Architecture-level Dependence Analysis for Software Systems”, *Int. Journal of SW Eng. and Knowledge Engineering*, 2000
- [26] The OSMOSE project, <http://www.itea-osmose.org/>
- [27] Wallnau, K.C., “A Technology for Predictable Assembly from Certifiable Components”, CMU/SEI-2003-TR-009