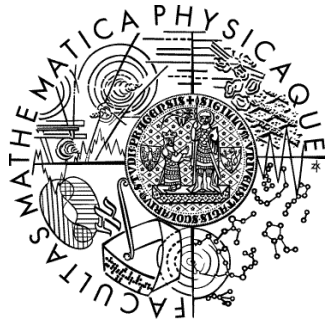
CHARLES UNIVERSITY IN PRAGUE
FACULTY OF MATHEMATICS AND PHYSICS

DOCTORAL THESIS



Jiří Adámek

Behavior Composition in Component Systems

Department of Software Engineering
Advisor: Prof. Ing. František Plášil, DrSc.

Abstract

Title:	Behavior Composition in Component Systems
Author:	RNDr. Jiří Adámek
	email: adamek@nenya.ms.mff.cuni.cz
	phone: +420 2 2191 4236
Department:	Department of Software Engineering
	Faculty of Mathematics and Physics
	Charles University in Prague, Czech Republic
Advisor:	Prof. Ing. František Plášil, DrSc.
	email: plasil@nenya.ms.mff.cuni.cz
	phone: +420 2 2191 4266
Mailing address (both Author and Advisor):	
	Dept. of SW Engineering, Charles University in Prague
	Malostranské nám. 25
	118 00 Prague, Czech Republic
WWW:	http://nenya.ms.mff.cuni.cz
This thesis:	http://nenya.ms.mff.cuni.cz/~adamek/phd-thesis/

Abstract

In order to formally verify a component application, it is suitable to structure the formal specification of its behavior according to the application architecture. Such an approach eases the maintenance of the specification and allows utilizing efficient verification algorithms that are based on decomposition of the application into several communicating parts. How those parts cooperate is formally described via an operation that is called behavior composition.

In this thesis we claim that in current software component systems behavior composition has typically two drawbacks: (1) it lacks support for composition error detection and (2) it does not address the problem of unbounded parallelism specification. While detection of composition errors allows checking design inconsistencies at a design time, unbounded parallelism specification is necessary for precise formal description of reentrant components that are used in practice very often. Therefore we introduce two new concepts – the consent operator and the behavior templates – in order to address both the issues (1) and (2). Our solutions are demonstrated on the SOFA component model [35], behavior protocols [32] are used as a behavior specification language.

Keywords: software components, formal verification, behavior composition, behavior protocols

Acknowledgments

I would like to thank all those who supported me in my doctoral study and the work on my thesis. I very appreciate the help and counseling received from my advisor Prof. František Plášil. For the various help they provided me, I also thank my colleagues; a particular thank goes to Tomáš Bureš, Petr Hnětynka, Pavel Ježek, Jan Kofroň, Martin Mach, Vladimír Mencl, and Pavel Parízek.

My thanks also go to the institutions that provided financial support for my research work. Through my doctoral study, my work was partially supported by the Grant Agency of the Czech Republic projects 201/03/0911 and 102/03/0672, by the Czech Academy of Sciences project 1ET400300504, and the ITEA project OSMOSE.

Last but not least, I am in debt to my parents and Zuzka, whose support and patience made this work possible.

Table of contents

Chapter 1

Introduction

In the last several years, software industry starts to take advantage of the software component systems. Building an application from components (produced either by the application developer or by third parties) shortens the development time, allows better code reuse, and eases the implementation of various non-functional features - e.g. dynamic update or load balancing. The idea to build applications from components comes from other industry branches (e.g. mechanical and electrical engineering).

The process of a component application development is supported by a *component system* - a set of software tools and libraries that allow to perform all the stages of the *component life-cycle*, i.e. to specify, design, implement, package, distribute, deploy and run component applications [37]. The development process of a component application differs from development of non-component software. In particular, we distinguish the specification, design, and implementation of a component and the specification, design, and implementation of a component application. Those two tasks are often made by different developers or teams in different companies. A single component is typically used in several applications.

A *component model* is a conceptual framework covering the abstractions used for component and component application design, as well as the relations between those abstractions. A component system then can be seen as a realization of a particular component model.

The component models can be divided into two categories: *flat* and *hierarchical*. In a flat component model, a component is implemented directly in an underlying programming language. The examples of flat component models include COM/DCOM [29], CCM [31], and EJB [36]. If the component model is hierarchical, a component may be either composed of several smaller *subcomponents* (such a component is called a *composite component*) or implemented directly (a *primitive component*). The hierarchical component models include e.g. Darwin [26], Wright [5], Fractal [11], or SOFA [35]. In this thesis, we focus on hierarchical component models, as they are more general - flat component models can be seen as their special case. In particular, various aspects of formal behavior specification and behavior composition addressed by the thesis are more complex in hierarchical component models.

An important feature of software component systems is the fact that they both require and simplify the usage of formal methods. Formal methods are required to describe and verify properties of the software components, and to prove that the components fit well together. This is important especially when third party components are used and the application under development is very complex. The usage of formal methods is simplified, as the complexity of many verification problems decreases substantially if the application under verification can be divided into several smaller subsystems communicating in a well defined manner. In case of a software component application such a division is done implicitly by design.

The area of formal methods is relatively wide. In this thesis we direct our attention to the methods that allow to verify various properties of a component application *behavior* in an automatized way (i.e. by the tools). Behavior of a component application is typically determined by all possible *executions*, an execution consists of *events*. How the events are combined to form an execution, what the types of events are, whether the events are atomic or not – this all depends on the choice of a particular formal method. Most of formal methods use a *labeled transition system* [10] or its modification to model the behavior.

A labeled transition system (or its modification) is very suitable to be processed by the tools. Also, being a simple mathematical structure, it allows to formulate rigorous proofs on various properties of the represented behavior in a concise way. However, it is not feasible as a primary language used by developers, as it is not easy to express a complex behavior as a labeled transition system (nevertheless it is always provably possible). Therefore, a typical approach is to employ a high-level *specification language*, whose constructs reflect the concepts the developers are familiar with. A *behavior specification* written in this language is automatically translated into a labeled transition system in order to verify the behavior. Specification languages, that can be automatically translated into labeled transition systems, include Petri nets, process algebras [10], behavior protocols [32], or FSP [25].

Behavior specification of a single component is a suitable subject of the classic verification tasks - *equivalence checking* [30] and *model checking* [15]. The purpose of equivalence checking is to determine whether two specifications describe equivalent behavior (with respect to a given behavior equivalence). Several behavior equivalences are known to be useful, differing in the level of detail on which the specifications have to correspond with each other in order to be equivalent. Model checking means finding out whether a given property (expressed typically in a temporal logic or via a finite state machine) holds for a given behavior specification. Another important verification problem, specific to software components, is *behavior compliance checking* [32]: behavior of a component B is compliant with the behavior of a component A, if B can replace A in any environment without causing any problems. The notion of behavior compliance can be generalized so that B is not a single component, but a component architecture. There are two motivations for introducing behavior compliance. (1) If a component is replaced by a new version, the behavior of the new one should be compliant with the behavior of the old one.

(2) Providing that for each component in a hierarchical component application a behavior specification is given, there exist two specifications for any composite component: the one that is given explicitly, and the *aggregate behavior* of its subcomponents (determined by their behavior specifications). If the design is consistent, the aggregate behavior should be compliant with the one that is given explicitly.

Typically, behavior of a component is specified manually by the component designer. In order to verify the behavior properties of a component application, it is often necessary to *compose* the behavior specifications of the components in order to obtain behavior specification of a composite component, a group of components, or the whole application (an aggregate behavior). This operation is called *behavior composition* and it is typically performed automatically (supported by a tool).

There are two tasks that require behavior specification of an entity that is bigger than a single component (and therefore make behavior composition important): (1) model checking of the behavior of the entity itself and (2) checking compliance between two specifications of the same entity on two different levels of abstraction. The first task appears e.g. as model checking of the whole application or its part (in both flat and hierarchical component models) or as model checking of the behavior of the components belonging to a single level in the component hierarchy (in hierarchical component models). The second task is specific to hierarchical component models and has the form of checking compliance of the aggregate behavior of a composite component with the explicitly given behavior.

There are many ways to perform behavior composition; we provide their overview in Chapter 2. The goal of this thesis is to analyze the approaches to behavior composition employed by known component models, to identify the problems of those approaches and to propose a solution of the problems. To present the solution, we use the SOFA component model [35] and behavior protocols [32].

The goals of the thesis are described in detail in Sect. 3.2. Here, we just roughly outline the main results:

1. The current component models employ such a form of behavior composition that only correct aggregate behavior is determined. We claim that it is also necessary to identify errors that result from composition of incompatible behaviors (*composition errors*). We have identified four types of composition errors: *bad activity*, *no activity*, *divergence* and *unbound requires error*. Detection of composition errors can be seen as a special kind of model checking aggregate behavior, where the property is fixed (i.e. common for all component applications). In a hierarchical component application, the composition errors can be detected with the granularity of one level in the component hierarchy.

2. The current component models do not allow to formally specify behavior of a reentrant component, as the underlying verification tools are based on manipulation

with *finite state specifications*. The problem is that it is often impossible to specify the behavior of a reentrant component by a finite state specification at the component design time when the behavior of the *component environment* (the other components in the application) is unknown. Therefore, we propose to specify the behavior of a reentrant component by a *behavior template* (at the component design time), which is automatically transformed to a finite state specification at the time the behavior composition is performed (at the application/architecture design time). The template transformation becomes an integral part of behavior composition, as in earlier stages of the design/verification process it cannot be performed due to the lack of information.

This thesis is conceived as a collection of commented papers. It includes four papers [AP05], [AP04a], [AP04b], and [Ada06] (all other papers cited in this thesis are referenced by numbers, e.g. [1]). [AP05] was published in an international scientific journal, [AP04a] and [AP04b] were published in proceedings of international conferences. [Ada06] was accepted for publication in proceedings of an international conference.

[AP05] Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance and Evolution: Research and Practice 17(5), pp. 363-377, Sep 2005

[AP04a] Adamek, J., Plasil, F.: Partial Bindings of Components - any Harm?, Presented at the SACT 2004 Workshop, Busan, Korea (held in conjunction with the APSEC 2004 conference), and published in the Proceedings of APSEC 2004, IEEE Computer Society, ISBN 0-7695-2245-9, pp. 632-639, Nov 2004

[AP04b] Adamek, J., Plasil, F.: Erroneous Architecture is a Relative Concept, in Proceedings of Software Engineering and Applications (SEA) conference, Cambridge, MA, USA, published by ACTA Press, ISBN 0-88986-425-X, pp. 715-720, Nov 2004

[Ada06] Adamek, J.: Addressing Unbounded Parallelism in Verification of Software Components, Accepted for publication in proceedings of the Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2006), Las Vegas, Nevada, USA, to be published by IEEE Computer Society, Jun 2006

The structure of the thesis is as follows: in Chapter 2 we show how the problem of behavior composition is solved by current component models and also how it is solved in the area of formal methods in general. In Chapter 3 we formulate the problem statement and the goals of the thesis. The Chapters 4-7 contain the papers [AP05], [AP04a], [AP04b], and [Ada06], respectively, with additional comments. In Chapter 8 we evaluate the contribution of the papers, while Chapter 9 concludes the thesis.

Chapter 2

Background & related work

In this chapter, we present an overview of the behavior specification languages and verification methods that are employed by current component systems. In particular, we show what form of behavior composition is used in those languages. We also briefly discuss the related component models.

In Sect. 2.1 we describe the SOFA component model and the SOFA component definition language. In Sect. 2.2 we introduce behavior protocols. In Sect. 2.3 we discuss other approaches to component behavior specification and verification. In Sect. 2.4 we present classic approaches to behavior specification and verification that are not component-oriented, but still utilize behavior composition. We conclude the related work by a brief summary in Sect. 2.5.

SOFA and behavior protocols are introduced in distinct sections as behavior protocols were successfully used as a behavior specification language not only for SOFA [32], but also for the Fractal component model [1], and therefore we consider them to be a topic that is independent on SOFA. As the contributions presented in this thesis are drawn up as extensions of behavior protocols, mainly in the context of the SOFA component model, both SOFA and behavior protocols are key for presentation of those contributions. SOFA and behavior protocols are therefore presented in this chapter in more detail than other component models and specification languages. Also, we use SOFA to present the concepts and terminology that are common for all component models.

2.1. SOFA

In this section, we provide a description of the SOFA component model and the SOFA component definition language (CDL).

2.1.1. SOFA component model

The goal of the SOFA project (SOFA = SOFtware Appliances) [35] is to provide a software developer with methods and tools for building applications from software *components*. In this section, we describe the SOFA *component model*: it defines what a

SOFA component consists of and how SOFA components are combined to form a *component application*.

2.1.2. Running example

We illustrate the concepts of the SOFA component model (and also the concepts of behavior protocols, presented in Sect. 2.2) on the component application from Fig. 2. The application consists of a database (`DB`), a client (`CL`), that uses the database, a filesystem (`FS`), that supports encrypted files and is used as a store for the data managed by `DB`, and a cryptographic component (`CR`), that provides encryption and decryption functionality used by `FS`. Using different implementations of `CR`, the data are encrypted/decrypted using different algorithms. The database consists of two components: the query optimizer (`QO`), transforming the queries in such a way that their processing is efficient, and the database engine (`EN`), that does the processing.

2.1.3. Software connectors

In SOFA, a component is a unit of distribution and deployment: the components can be deployed on different computers, communicating over a network. Communication between components (either local or remote) is implemented by software *connectors* . A connector

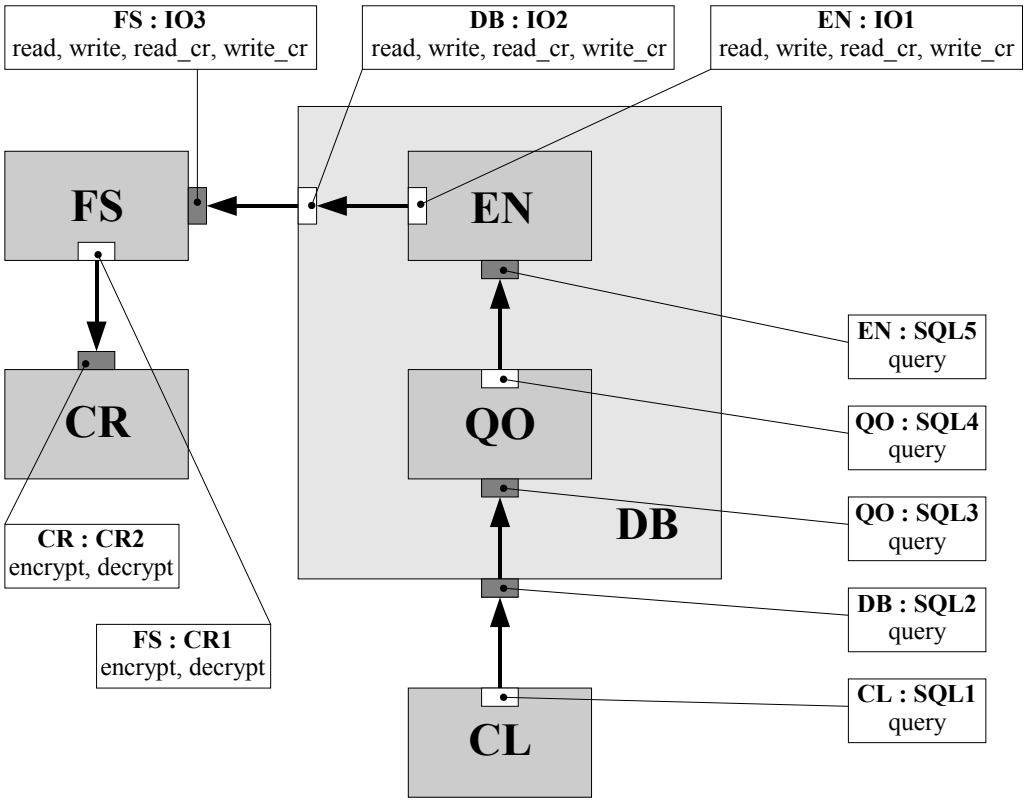


Figure 2: Example of a SOFA component application

provides independence of the components on a concrete type of communication middleware, as well as additional communication services (e.g. encryption). Software connectors are out of scope of this thesis: we refer to [13] for further reading.

2.1.4. Component nesting

SOFA also supports *component nesting*: a component can be either implemented in an underlying language (e.g. Java) – in this case, we call it a *primitive component* – or it can be composed of several (direct) *subcomponents* – then it is called a *composite component*. Each of those subcomponents can also be a composite component. Thus, an application in SOFA is formed by a component hierarchy. If two components A, B are (direct) subcomponents of a component C, we say that A and B are on the same *level of nesting*. If S is a (direct) subcomponent of C, we say that C is the (direct) *supercomponent* of C.

On Fig. 2, there is an example of component application consisting of six components: `CL`, `DB`, `FS`, `CR`, `QO`, and `EN` (detailed description of the application can be found in Sect. 2.1.2). Here, `QO` and `EN` are direct subcomponents of `DB` (and they are on the same level of nesting). Also, `CL`, `DB`, `FS`, and `CR` are on the same level of nesting – they are (direct) subcomponents of a virtual component that corresponds to the whole application.

We say that S is an *indirect subcomponent* of C, if there exists a sequence of components $S_0, S_1, ..., S_n$ such that $S_0 = C$, $S_n = S$, and for all $i \in \{0, ..., n-1\}$ the component S_{i+1} is a direct subcomponent of S_i. If S is an indirect subcomponent of C, we say that C is an *indirect supercomponent* of S.

2.1.5. Interfaces

In SOFA, components communicate via *interfaces*. There are two kinds of interfaces: *provided interfaces* and *required interfaces*. A provided interface is very similar to an object interface from object-oriented languages: it defines a list of methods, that are provided by a component (i.e. other components can call those methods) and, from the design point of view, they form a logical group. As well as an object in object oriented languages can implement several interfaces, a SOFA component can employ several provided interfaces. If a component C features a provided interface P, the methods defined by P are either implemented by C (if C is primitive), or they are implemented by a (possibly indirect) subcomponent of C (if C is composite).

A required interface of a component C defines a logical group of methods, that have to be provided by another component in order to C's implementation can work: C's implementation calls those methods. In the area of object oriented languages, the concept closest to a required interface is a reference to another object. While in an object oriented language a dependence on another object (a reference) is visible only in the code (this information is not present in any of the object's interfaces), in SOFA (and other component models) a dependence on methods implemented by another component is

explicitly denoted by a required interface. A component can employ several required interfaces.

On Fig. 2, provided interfaces are denoted by small dark boxes, while required interfaces are denoted by small white boxes. Each interface caption on Fig. 2 shows the name of the component that employs the interface, the local name of the interface (i.e. in the context given by the component - interfaces of different components can have the same local names) and the list of the methods that are provided or required. E.g., the `EN` component employs one provided interface (`SQL5`) and one required interface (`IO1`). The presence of the `IO1` interface expresses the fact that the implementation of `EN` calls the methods `read`, `write`, `read_cr`, `write_cr`, that have to be provided by another component.

2.1.6. Interface ties

If a component C employs a required interface R, it is necessary to provide C with an implementation of the methods that are defined by R. In SOFA, it is done by an *association* of R with a provided interface P of a primitive component D such that P defines the methods that are defined by R. As D is primitive, the methods are implemented by D. This association has the form of a sequence of *interface ties*.

An interface tie connects two interfaces. An interface tie is oriented: it goes from an interface I to an interface J such that J defines the methods that are defined by I. Such an interface tie expresses the following fact: any time a method m is called on I, the call is forwarded to J (J certainly defines m). Either m is implemented by the component that employs J, or the call is forwarded by another interface tie. On Fig. 2 the interface ties are depicted by the arrows (the direction of the arrow shows the orientation of the tie).

To refer to the interface ties, we use the following convention: a tie going from an interface I of a component C to an interface J of a component D is denoted as <C:I-D:J>.

There are three kinds of interface ties: a *binding*, a *delegation*, and a *subsumption*. A binding goes from a required interface of a component C to a provided interface of a component D such that C and D are on the same level of nesting. A delegation goes from a provided interface of a composite component C to a provided interface of a direct subcomponent of C. A subsumption goes from a required interface of a component S to a required interface of the direct supercomponent of S.

On Fig. 2, `<DB:SQL2-QO:SQL3>` is a delegation, `<EN:IO1-DB:IO2>` is a subsumption, and all other interface ties are bindings. Note that e.g. `<CL:SQL1-DB:SQL2>` and `<QO:SQL4-EN:SQL5>` are bindings on different levels of nesting.

Now, we can show an example of the association of a required interface with an appropriate provided interface of a primitive component. On Fig. 2, for `CL:SQL1` such an association is realized by the following sequence of interface ties: `<CL:SQL1-DB:SQL2>`, `<DB:SQL2-QO:SQL3>`. For `EN:IO1` it is the sequence `<EN:IO1-DB:IO2>`, `<DB:IO2-`

FS:IO3>. Note that for DB:IO2 it is the sequence <DB:IO2-FS:IO3> (a suffix of the previous one).

In general, the association is a sequence $<C_1:I_1\text{-}C_2:I_2>$, $<C_2:I_2\text{-}C_3:I_3>$, ..., $<C_{n-1}:I_{n-1}\text{-}C_n:I_n>$ of interface ties. In the sequence, there is exactly one binding (let it be $<C_j:I_j\text{-}C_{j+1}:I_{j+1}>$ for some j). All predecessors of $<C_j:I_j\text{-}C_{j+1}:I_{j+1}>$ in the sequence are subsumptions, all successors of $<C_j:I_j\text{-}C_{j+1}:I_{j+1}>$ in the sequence are delegations. There may be no subsumption in the sequence, as well as there may be no delegation.

If each required interface is connected to another interface (by binding or subsumption) and each provided interface of a composite component is connected by a delegation to a provided interface of a subcomponent, then every required interface R is associated with a provided interface of a primitive component (and therefore, for every method call on R there is an implementation of the method that processes the call).

2.1.7. Interface type, frame, and architecture

In order to allow reuse of the components, we define three important concepts: *interface type*, *frame*, and *architecture*.

Two interfaces have the same type if they define the same list of methods. The type of an interface is orthogonal to the fact whether the interface is provided or required. Thus, a provided interface and a required interface can have the same type.

Recalling the interface ties from Sect. 2.1.6, we can now reformulate the condition of the same method lists for interfaces connected by an interface tie: a tie connects two interfaces of the same type.

The frame of a component C is defined by all the interfaces of C that are externally visible (i.e., the interfaces of C's subcomponents are not included). For every interface, the following information is included: the local name of the interface, its type, and the fact whether it is provided or required.

On Fig. 2, the frame of the DB component consists of the description of the DB:SQL2 and DB:IO2 interfaces, while the frame of QO consists of the description of QO:SQL3 and QO:SQL4.

The frame of a component represents so-called *black-box view* of the component: it gives the information on the externally visible properties of the component, but not on the internal structure.

Architecture A of a composite component (a *composite architecture*) C is defined by the frame F of C, the frames of C's direct subcomponents, all the bindings between C's direct subcomponents, and the delegations and subsumptions connecting interfaces of F with the interfaces of C's direct subcomponents. We say that A *implements* F.

Architecture A' of a primitive component C' (a *primitive architecture*) is defined only by the frame F' of C' (as C' does have neither subcomponents nor bindings between subcomponents etc.). As in the previous case, we say that A' implements F'.

On Fig. 2, the architecture of `DB` consists of the frame of `DB`, the frames of `EN` and `QO`, the binding `<QO:SQL4-EN:SQL5>`, the delegation `<DB:SQL2-QO:SQL3>`, and the subsumption `<EN:IO1-DB:IO2>`. The architectures of `FS`, `CL`, `EN`, `CR`, and `QO` are primitive.

The components on the top of a component hierarchy form so called *system architecture*. The frame implemented by this architecture is implicit (and contains no interfaces). On Fig. 2, the system architecture consists of the frames of `FS`, `CR`, `DB`, and `CL`, and the bindings `<CL:SQL1-DB:SQL2>`, `<DB:IO2-FS:IO3>`, `<FS:CR1-CR:CR2>`. In any system architecture neither delegations nor subsumptions are present.

An architecture represents information about a component's structure, but not the whole information. For a composed component, it describes the internals only on the first level of nesting. The structure of deeper levels is unspecified. We say that the architecture provides a *gray-box view* of the component.

Within the context of a given architecture A, an interface tie is identified by a *tie name*. For a binding, the tie name follows the convention for referring to interface ties described above: it has the form <C:I-D:J>, where C, D are names of two subcomponents (in A), and I, J are the names of external interfaces of C, D, respectively. For a delegation, the tie name has the form <I-D:J>, where I is a provided interface of the frame implemented by A, D is a subcomponent in A and J is a provided interface of D. For a subsumption, it has the form <C:I-J>, where I is a required interface of a subcomponent C (in A) and J is a required interface of the frame implemented by A.

2.1.8. SOFA Component definition language (CDL) and CDD descriptors

The SOFA *Component Definition language* (CDL) and the SOFA *CDD descriptors* provide a means of an incremental, top-down design of a component application. First, interface types, frames, and architectures are specified using CDL. The CDL sources are processed by the *CDL compiler*, which stores the information on the interfaces, frames and architectures into a *component repository*. Thus, in the component repository only partial descriptions of the components are stored.

A complete description of a component C has the form of a CDD descriptor. It refers to the data stored in the component repository (following the fact that a complete description of C is based on an architecture that was previously stored into the repository) and to CDD descriptors of C's subcomponents. While an architecture of C contains only the frames of C's direct subcomponents (and also the interfaces ties on the appropriate level of nesting), the CDD descriptor of C defines complete descriptions of those subcomponents. Thus, a

component application in SOFA is specified by a set of CDD descriptors with mutual references respecting the supercomponent-subcomponent relation.

Although it is possible to write a CDD descriptor manually, a tool is distributed with SOFA that can generate most of its parts automatically, if a component architecture (stored in a component repository) is given. The developer only adds a version of the CDD descriptor (in order to distinguish between different the CDD descriptors based on the same component architecture). Also, the code skeletons for primitive components are generated automatically in SOFA.

The information from the CDD descriptors (including the implementation of the primitive components) is stored in an *implementation repository*. Any component from the repository can be then reused in many different component applications.

We illustrate the basics of CDL and CDD descriptors on the component application from Fig. 2. First, the interface types are specified in CDL. An interface type specification consists of method signatures. A method signature specifies the return type, the method identifier, and for each parameter the type, the name, and whether it is passed from the caller to the callee (an in-parameter), from the callee to the caller (an out-parameter), or in both directions (an inout-parameter):

```
interface SQLInterface {
   void query(in string query, out Table data);
};

interface IOInterface {
   void read(in FileDescriptor fd, in long offset,
             in long size, out Buffer data);
   void write(in FileDescriptor fd, in long offset,
              in long size, in Buffer data);
   void read_cr(in FileDescriptor fd, in long offset,
                in long size, out Buffer data, String key);
   void write_cr(in FileDescriptor fd, in long offset,
                 in long size, in Buffer data, String key);
};

interface CRInterface {
   void encrypt(in Buffer data_in, out Buffer data_out,
                String key);
   void decrypt(in Buffer data_in, out Buffer data_out,
                String key);
};
```

Then the frames are described. A frame description consists of two parts: the list of provided interfaces and the list of required interfaces. Each list consists of pairs of the form interface type - interface name. In this example, the name of a frame consist of the name of the component that uses the frame and the suffix `Frame`. It is possible as in the application from Fig. 2 every frame is instantiated only once (i.e. there is no pair of components with the same frame). However, in general it is not necessarily true:

```
frame ClientFrame {
   requires:
      SQLInterface SQL1;
}

frame DatabaseFrame {
   provides:
      SQLInterface SQL2;
   requires:
      IOInterface IO2;
};

frame FileSystemFrame {
   provides:
      IOInterface IO3;
   requires:
      CRInterface CR1;
}

frame CryptoFrame {
   provides:
      CRInterface CR2;
};

frame QueryOptimizerFrame {
   provides:
      SQLInterface SQL3;
   requires:
      SQLInterface SQL4;
}

frame DatabaseEngineFrame {
   provides:
      SQLInterface SQL5;
   requires:
      IOInterface IO1;
};
```

Finally, the architecture descriptions are written in CDL. The description of a composite architecture consists of the name of the architecture (in this example an architecture name consists of the component name and the suffix `Architecture`), the name of the frame implemented by the architecture (after the keyword `implements`), descriptions of the *frame instances* (one frame instance for each subcomponent in the architecture), and descriptions of the interface ties.

A frame instance description consists of the keyword `inst`, the name of a frame (defined above), and the name of the frame instance. A frame instance is an abstraction of a subcomponent (it abstracts from the subcomponent internals). Therefore, by convention we use the same name for a subcomponent and for the corresponding frame instance.

A tie description consists of the keyword `bind`, `delegate`, or `subsume` (depending of the kind of the tie) and the names of two interfaces, that are connected by the tie. An interface name here consists of a frame instance name (defined within the architecture description) and an interface name (defined in the appropriate frame description):

```
architecture DatabaseArchitecture
implements DatabaseFrame {

   inst DatabaseEngineFrame EN;
   inst QueryOptimizerFrame QO;

   delegate SQL2 to QO:SQL3;
   bind QO:SQL4 to EN:SQL5;
   subsume EN:IO1 to IO2;
};
```

A primitive architecture description is very simple, as it contains neither frame instances nor interface ties. The fact that the architecture is primitive is denoted by the keyword `primitive`:

```
architecture QueryOptimizerArchitecture
implements QueryOptimizerFrame primitive;

architecture DatabaseEngineArchitecture
implements DatabaseEngineFrame primitive;

architecture ClientArchitecture
implements ClientFrame primitive;

architecture FileSystemArchitecture
implements FileSystemFrame primitive;

architecture CryptoArchitecture
implements CryptoFrame primitive;
```

A system architecture (the top-level of the component hierarchy) is described in the same way as any composite architecture. It refers to `::SOFA::libs::Application` as to the implemented frame:

```
system architecture ApplicationArchitecture
implements ::SOFA::libs::Application {

   inst DatabaseFrame DB;
   inst ClientFrame CL;
   inst FileSystemFrame FS;
   Inst CryptoFrame CR;

   bind CL:SQL1 to DB:SQL2;
   bind DB:IO2 to FS:IO3;
   bind FS:CR1 to CR:CR2;
}
```

To employ the frames and architectures in a component application, it is necessary to write the CDD descriptors. In bold we show the parts of descriptors that are written by a developer, the rest is generated automatically. This is the CDD descriptor for the `DB` component from Fig. 2 (unimportant generated parts of the descriptor are omitted) :

```
<?xml version="1.0" encoding="UTF-8"?>
<sofa_system>
<architecture_ref>DatabaseArchitecture</architecture_ref>
<frame_ref>Database</frame_ref>
<version>0.0.1</version>
...
<component_ref inst="EN" arch="DatabaseEngineArchitecture"
version="0.0.1"/>
<component_ref inst="QO" arch="QueryOptimizerArchitecture"
version="0.0.1"/>
</sofa_system>
```

The descriptor of `DB` is uniquely identified by the name of the architecture (`DatabaseArchitecture`) and the version (`0.0.1`). In the same way the referenced CDD descriptors of the subcomponents are identified. The CDD descriptor describing the virtual component corresponding to the whole application is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<sofa_system>
<architecture_ref>ApplicationArchitecture</architecture_ref>
<frame_ref>::SOFA::libs::Application</frame_ref>
<version>0.0.1</version>
...
<component_ref inst="DB" arch="DatabaseArchitecture"
version="0.0.1"/>
<component_ref inst="CL" arch="ClientArchitecture"
version="0.0.1"/>
<component_ref inst="FS" arch="FileSystemArchitecture"
version="0.0.1"/>
<component_ref inst="CR" arch="CryptoArchitecture"
version="0.0.1"/>
</sofa_system>
```

We omit the other CDD descriptors – they describe primitive components and they are therefore very simple. In fact, the developer fills in only the version number, the rest of the information is automatically generated.

As this thesis addresses the problem of component behavior description, we do not show all the details of a SOFA application development. More information on SOFA, the CDL language, CDD descriptors, and writing the code of the primitive components can be found in [35].

2.2. Behavior protocols

As a part of SOFA, we have defined a formal model of component *behavior* [32] (based on the concept of atomic *events*) and *behavior protocols* [32] - a language for specifying the behavior. Our notion of behavior is very close to the *trace semantics* of the labeled transition systems [10], while behavior protocols can be seen as a special case of *process algebras* [10].

Although originally defined as a behavior specification language for SOFA, behavior protocols were also successfully used in the context of the Fractal component model [1].

2.2.1. Events and event tokens

As stated in Sect. 2.1, SOFA components communicate via interfaces and interface ties. Such a communication during a single run of a component application can be viewed as a sequence of *method calls*; in more detail, we can distinguish a *request* and a *response* - events that occur at the beginning and at the end of a method invocation. Requests and responses are called *events*. The events are *atomic* - i.e. an event is indivisible and at the same time only one event can proceed. We abstract from the physical time (also called real-time in the context of formal verification). It means that we model neither the duration of an event, nor the time interval between two subsequent events.

To denote events, we first introduce a *method name*, identifying a method. It can be either short or long. A *short method name* identifies a method within the context of a frame; it consists of a local interface name and a method identifier (defined within the interface). A *long method name* identifies a method in the context of an architecture; it consists of a tie name and a method identifier (defined within the interfaces on both ends of the tie). E.g., the `query` method of the `SQL3` interface of the `QO` component from Fig. 2 is (in the context of the frame of `QO`) identified by the following short method name: `SQL3.query`. Within the architecture of DB, the same method is identified by the following long method name: `<SQL2-QO:SQL3>.query`. This long method name also identifies the `query` method on `SQL2`. It is quite natural, as a call on `SQL2` is immediately forwarded to `SQL3` - we consider the original and the forwarded call as being the same (this also holds not only for the call as a whole, but also for the request and the response forming the call).

If m is a method name (either short or long), then a request for a call of m (including passing the in- and inout-parameters) is denoted by m↑, while the response (including passing the return value and the out- and inout-parameters) is denoted by m↓. The strings "m↑", "m↓" are called *event names*.

While the event name identifies an event uniquely, an *event token* denotes an event from the point of view of a concrete component. Thus, for a single event there exist three different event tokens describing the event. If an event occurs on one of the interfaces forming the frame of a component C (an *external event*), it is either *emitted* by C (either

a request on a required interface, or a response on a provided interface) – then, from the point of view of C, it is denoted by an event token !m↑ or !m↓ (where m↑, m↓ are the appropriate event names) – or the event is *absorbed by* C (either a request on a provided interface, or a response on a required interface) – then, from the point of view of C, it is denoted by an event token ?m↑ or ?m↓. If an event occurs on an external interface of one of C's direct subcomponents, it is an *internal event* from the point of view of C and it is denoted by an event token τm↑ or τm↓. To denote an internal event, a long method name has to be used. To denote an external event, either short or long method name can be used.

From the point of view of the `QO` component's frame (Fig. 2), the call of the `query` method on the `SQL4` interface consists of a request emission and a response absorption. Using short method names it is denoted as the sequence: `!SQL4.query↑`, `?SQL4.query↓`. From the point of view of the `EN`'s frame, the same call is denoted as absorption of a request and emission of a response (on `SQL5`): `?SQL5.query↑`, `!SQL.query↓`. Finally, from the point of view of the `DB`'s architecture, it consists of two internal events (long method names are used here): `τ<QO:SQL4-ENSQL5>.query↑`, `τ<QO:SQL4-ENSQL5>.query↓`.

The same event can be denoted using both the short method name (the frame's point of view) and the long method name (the architecture's point of view). E.g., a request for a `query` call on `SQL2` is denoted as `?SQL2.query↑` from the point of `DB`'s frame and as `?<SQL2-QO:SQL3>.query↑` from the point of view of `DB`'s architecture.

2.2.2. Traces and behavior

A run of a component system is modeled as a trace: a finite sequence of event tokens. The run can be modeled either from the point of view of a given frame (short method names are used in the trace), or from the point of view of a given architecture (long method names are used). A trace denotes only the information on the method calls (requests and responses): other aspects of the computation (the manipulation with the internal data structures of the primitive components, the concrete parameters and the return values, etc.) are not included.

The set of all possible traces forms the *behavior* (of a frame or an architecture). From the point of view of the language and automata theory [20], the behavior is a language over the alphabet consisting of all event tokens (with either short or long method names, depending on the context).

To illustrate the concepts of traces and behavior, let the `DB` component from Fig. 2 accept either one or two calls of `query` on `SQL2` (for simplicity, we suppose here that no methods are called on `IO2`). From the point of view of the `DB`'s frame, such a behavior is specified by the following set of (two) traces:

```
{
<?SQL2.query↑, !SQL2.query↓>,
```

```
<?SQL2.query↑, !SQL2.query↓, ?SQL2.query↑, !SQL2.query↓>
}
```

By convention, we use the symbols "<", ">" to denote both traces and tie names. From the point of view of the DB's architecture, the same behavior is specified by the following set:

```
{
< ?<SQL2-QO:SQL>.query↑, !<SQL2-QO:SQL3>.query↓ >,
< ?<SQL2-QO:SQL>.query↑, !<SQL2-QO:SQL3>.query↓,
  ?<SQL2-QO:SQL>.query↑, !<SQL2-QO:SQL3>.query↓ >
}
```

2.2.3. Protocols

Behavior protocols (protocols for short) are a behavior specification language based on regular expressions [20], similar to process algebras [10]. A protocol is an expression that specifies behavior of a component, a group of components, or the whole application. Formally, a protocol P specifies a language (behavior) L(P). We say that L(P) is *generated* by P. A protocol consists of event tokens and *operators*. If, for a trace *s* and a protocol P, $s \in L(P)$ holds, we say that *s* is generated by P.

If e is an event token, then e is a protocol representing the language containing only one trace: L(e) = {<e>}. NULL is the protocol representing an empty set (of traces): L(NULL) = {}. There are three *basic operators* for behavior protocols (the same as basic operators for regular expressions): sequencing (;), alternative (+), and repetition (*). Let P and Q be protocols:

$L(P ; Q) = \{<e_1, ..., e_m, f_1, f_2, ..., f_n> : <e_1, ..., e_m> \in L(P) \text{ and } <f_1, ..., f_n> \in L(Q) \}$
$L(P + Q) = L(P) \cup L(Q)$
$L(P^*) = L(NULL) + L(P) + L(P ; P) + L(P ; P ; P) + \ldots$

Informally, sequencing (P; Q) results in the set of all traces formed by concatenation of a trace generated by P and a trace generated by Q. Alternative (P + Q) stands for the set of all traces which are generated either by P or by Q. Finally, repetition (P*) expresses arbitrary (but finite) number of repetitions of the behavior, that is specified by P.

We provide several examples (a, b, x, y are event tokens):

L((a + b) ; (x + y)) = {<a, x>, <a, y>, <b, x>, <b, y>}
L((a*) ; b) = {<b>, <a, b>, <a, a, b>, <a, a, a, b>, …}
L((a + b)*) = {<>, <a>, <b>, <a, a>, <a, b>, <b, a>, <b, b>, … }

We also define *advanced operators* (P, Q are protocols):

P | Q ... and-parallel; an arbitrary interleaving of traces generated by P and Q
P || Q ... or-parallel; stands for P + Q + (P | Q)

The "|" operator is useful for representing parallel execution of two protocols; two traces generated by these protocols (one from each) are arbitrary interleaved, but the mutual order of event tokens from every of the traces is preserved. Look at the following example: in all traces generated by (a; b) | (x; y), a, is before b and x is before y.

L((a; b) | (x; y)) = {<a, b, x, y>, <a, x, b, y>, <a, x, y, b>,
<x, a, b, y>, <x, a, y, b>, <x, y, a, b>}

In order to formally express behavior composition and to allow formal reasoning on the behavior specified by the protocols, the following operators are defined (P, Q are protocols, T is a set of event tokens, E is a set of events):

P / T ... *restriction*: the event tokens that are not elements of G are omitted from the traces in L(P)

P $\sqcap_E$ Q ... *composition*: an arbitrary interleaving of traces generated by P and Q (i.e. this operator is similar to |), but any event token of the form ?e, in a trace generated by P, where e $\in$ E, is merged with a token of the form !e in a trace generated by Q, resulting in an internal event τe. Similarly, !e in a trace generated by P is merged with ?e generated by Q - the operator is symmetric. For an event f $\notin$E, the tokens are not merged.

P |T| Q ... *adjustment*: this operator is similar to $\sqcap$, but T here is a set of event tokens (not events), and not complementary, but the same tokens are merged

Finally, we define *abbreviations* (P is a protocol, m is an event name):

?m{P} ... nested incoming call; stands for ?m ; P; !m
?m ... simple incoming call; stands for ?m ; !m
!m ... simple outgoing call; stands for !m ; ?m

More information on behavior protocol operators and abbreviations can be found in [32].

2.2.4. Types of protocols and behavior composition

In SOFA, there are three types of behavior protocols: *interface protocols* (describing behavior on interfaces), *frame protocols* (describing behavior on frames) and *architecture protocols* (describing behavior of architectures). We mentioned interface protocols just for completeness, for the topic of this thesis only frame protocols and architecture protocols are important.

In frame protocols, short method names are used, while in the architecture protocols long method names are used. The reason is that a frame protocol specifies events on a frame of a single component, while in an architecture protocol events of several components may appear (and it is therefore necessary to explicitly identify a component that emits or absorbs a particular event).

Another important difference between frame protocols and architecture protocols is the fact that frame protocols are written by a developer in CDL files, while architecture protocols are constructed automatically. This construction is formalized via the composition operator ⊓ and realized by the Behavior Protocol Checker – a tool developed for verification of behavior protocols [35]. Also, this construction represents behavior composition in the context of behavior protocols.

2.2.5. Running example - behavior protocols

In this section, we show the behavior protocols for the DB, QO, and EN components from Fig. 2. The DB component expects SQL queries on its SQL2 interface and (as a consequence) calls I/O operations on its IO2 interface. This behavior is specified by the following frame protocol:

$Prot_{DB}$ = (?SQL2.query{ (!IO2.read + !IO2.write)* })*

Now, we specify the behavior of the subcomponents of DB. While the behavior of EN is (from the point of view of behavior protocols that abstract from data) the same as the behavior of DB itself, QO repeats very simple action - on a call on SQL3 it reacts by a single call on SQL4 (i.e. it receives a query, optimizes it and delegates it to EN):

$Prot_{EN}$ = (?SQL5.query{ (!IO1.read + !IO1.write)* })*
$Prot_{QO}$ = (?SQL3.query{ !SQL4.query })*

The architecture protocol that specifies the behavior of DB's architecture can be constructed from $Prot_{EN}$ and $Prot_{QO}$ by replacing short method names by the long method names and using the composition operator $⊓_E$. The set E is formed by the names of all events that occur on the bindings between $Prot_{EN}$ and $Prot_{QO}$:

```
Prot_EN-QO =
    (?<QO:SQL4-EN:SQL5>.query{
       (!<EN:IO1-IO2>.read + !<EN:IO1-IO2>.write)*
    })*
    ⊓_E
    (?<SQL2-QO:SQL3>.query{ !<QO:SQL4-EN:SQL5>.query })*
=
    (?<SQL2-QO:SQL3>.query{
       τ<QO:SQL4-EN:SQL5>.query{
          (!<EN:IO1-IO2>.read + !<EN:IO1-IO2>.write)*
       }
    })*
```

Here, E = { <QO:SQL4-EN:SQL5>.query↑, <QO:SQL4-EN:SQL5>.query↓ }.

2.2.6. Behavior protocols in CDL

Interface protocols and frame protocols are written in CDL by a developer. Architecture protocols are generated by CDL compiler automatically: for each non-primitive component type, architecture protocol is generated from frame protocols of the subcomponent frames. The following CDL code fraction shows how interface and frame protocols are embedded into CDL:

```
interface SQLInterface {
   void query(in string query, out Table data);
   protocol: query*
};

frame QueryOptimizerFrame {
   provides:
      SQLInterface SQL3;
   requires:
      SQLInterface SQL4;
   protocol:
      (?SQL3.query{!SQL4.query})*
};
```

2.2.7. Behavior compliance

In SOFA (or in any component system that employs behavior protocols) for any composite component there exist both the frame protocol (specified by the developer) and the architecture protocol (automatically generated by the CDL compiler). It is desirable to automatically compare those two protocols in order to ensure that the design of a component application is consistent. The relation that formally addresses such a comparison is called *behavior compliance*.

Informally, the architecture protocol $Prot_A$ is compliant with the frame protocol $Prot_F$ if (1) all method calls that are absorbed according to $Prot_F$ are also absorbed according to $Prot_A$ and (2) only those method calls that are emitted according to $Prot_F$ may be emitted according to $Prot_A$. This means that $Prot_A$ is compliant with $Prot_F$ if $Prot_A$ "provides more and requires less" than $Prot_F$. The formal definition of behavior compliance can be found in [32].

The behavior compliance is automatically tested by the Behavior Protocol Checker - a tool that was implemented within the SOFA project [35]. The Behavior Protocol Checker is also integrated with the CDL compiler, so that compliance is automatically tested for all composite component types defined by the developer.

2.3. Other approaches to component behavior specification

2.3.1. Parameterized synchronized networks of labeled transition systems

In [9,8], the authors use *parameterized synchronized networks of labeled transition systems* (pNets) to specify behavior of Fractal [11] components. pNets were designed to address behavior of a component with a controller (implementing non-functional aspects of the component) and dynamic behavior of components (binding/unbinding the interfaces, adding/removing/updating subcomponents at run time). They also support modeling of data structures. First, we shortly introduce the parts of the Fractal component model that are relevant from the point of view of behavior specification and verification. Then, we show how the behavior is specified using pNets.

In Fractal, a component consists of a *controller* and a *content*. If the component is *composite*, the content consists of finite number of *subcomponents*. If the component is *primitive*, the content remains empty. The motivation for introducing the controller is addressing the non-functional behavior, and in particular those kinds of non-functional behavior which are specific to component systems.

The controller can have *external* and *internal* interfaces: the component interacts with its environment via the operations on its external interfaces, while the internal interfaces are accessible from the subcomponents.

There are two kinds of interfaces: *server* and *client*. As well as a provided and a required interface in SOFA (respectively), a server interface is defined by a set of methods which are implemented by the component, while a client interface is defined by a list of methods which have to be provided by another component.

A client interface and a server interface are connected by a *binding*. A *normal binding* connects two external interfaces (of two components which have common enclosing component), an *export binding* connects an internal client interface of a component and an external server interface of its subcomponent, while an import binding connects an internal server interface of a component and an external client interface of its subcomponent.

From this description, the main difference between SOFA and Fractal component model is obvious: in SOFA, an interface of a component C (either provided or required) is both external and internal (i.e., it is accessible for both the environment of C and the subcomponents of C), while in Fractal an interface has only "a half" of this functionality. In Fractal, a SOFA-like interface could be simulated by a pair of interfaces (one internal and one external) with an additional constraint: if an event is absorbed on the external interface, a corresponding event is emitted on the internal one (and vice versa).

In general, an event on an external interface of a Fractal component can cause an arbitrary number of events on the internal interfaces (or none), similarly for an event on an internal interface. What are the actual reactions "on the other side" depends on the behavior of the controller. The controller is composed of several parts; every part is responsible for one *non-functional aspect* and can be controlled via a *control interface*. A non-control interface (either internal or external) is called a *functional interface*.

From the point of view of behavior description, the following non-functional aspects are important [9]:

- *Attribute control*: provides operations to get and set attribute values of the component
- *Binding control*: provides operations to bind and unbind the component interfaces
- *Content control*: provides operations to add and remove subcomponents into/from the component
- *Life cycle control*: provides operations to stop and start the component, as well as to get its current status

Behavior of a component employing the full functionality of the controller is very complex. To ease the development and maintenance of Fractal applications, the authors of Fractal put several constraints on the interactions between the controller (non-functional aspects of a component) and the subcomponents (taken from [9]):

- Content and binding control operations are only possible when the component is stopped
- When started, a component can emit or accept invocations
- When stopped, a component do not emit invocations and must accept invocations through control interfaces; whether or not an invocation to a functional interface is possible is undefined

Even more constraints are put by the authors of [9], so that the behavior can be modeled by pNets. The most important constraints include the following:

- The start/stop operations are recursive, i.e. they affect the component and each of the subcomponents
- Functional operations cannot fire control operations
- For functional calls on a composite component, the controller is only a forwarder between external and internal functional interfaces. This implies that there is exactly one internal interface for each external (functional) interface of a composite component.

If all those constraints are applied, the behavior of a component can be viewed as a sequence of *phases*. There are four kinds of phases [9]:

- *Deployment*: the building phase of a component. The component's content (its

subcomponents) is defined, the initial sequence of control operations is performed. This phase typically ends with a recursive start operation.
- *Running phase*: only functional operations can occur here.
- *Non-structural reconfiguration*: life-cycle and binding control operations (start, stop, bind, unbind)
- *Structural reconfiguration*: adding, removing, or updating components

pNets are based on two concepts: *parameterized labeled transition systems* (pLTS) and *hierarchical networks of communicating systems* (*synchronization networks*) [7].

The concept of pLTS is based on the ideas published in [23]. A pLTS is similar to a common labeled transition system – i.e. it consists of a set of states and a set of labeled transitions among those states. In addition, with a state in pLTS a finite set of *variables* is associated and any transition in pLTS is structured – it consists of a *guard*, a *parameterized action* and a *set of assignments*.

A guard is a boolean expression over the variables associated with a state *s* the transition is going out of. The parameterized action represents either a non-observable action in the modeled system, a local action of a process, or an emission/acceptance of a remote method call (between different processes), including the parameters. Finally, the set of assignments define the value of the variables associated with a state *s'* the transition comes in: on the left side, a variable associated with *s'* appears, on the right side an expression over the variables of *s* appears.

The semantics of a transition in a pLTS is the following: a transition from a state *s* into a state *s'* can be performed if the guard evaluates to true (the values of variable associated with *s* are used to evaluate the guard). If this is the case, the parameterized action is performed, the state changes to *s'* and the values of the variables associated with *s'* are defined using the set of assignments.

A (remote) method call in a pLTS is represented by a single parameterized action (i.e. by a single transition). As a transition in a pLTS is atomic (as well as in a common labeled transition system), parallel method calls cannot be modeled using a pLTS.

A pNet is parameterized behavior specification, where the parameters are pLTSes. A pNet consists of a (fixed) pLTS called *transducer* and a set of *holes (formal parameters)*. With a hole a set of parameterized actions (a *sort*) is associated; only a pLTS that employs (a subset of) those parameterized actions can be used as an actual parameter for the hole.

An action of the transducer consists of a *global action* and a vector of *local actions*, one for each hole. A local action is either an *idle* action, or an element of the sort associated with the appropriate hole.

The definition of a pNet reflects the structure of hierarchical component systems. If a composite Fractal component C is modeled using a pNet, the holes are parameterized by

the pLTSes specifying the behavior of C's subcomponents and the parts of C's controller (responsible for the non-functional aspects). The global actions represent the method calls on the external interfaces of C, while the local actions represent the method calls on the external interfaces of C's subcomponents, or the events in C's controller (including the changes in C's architecture).

The semantics of a transition of the transducer is the following: if the global action occurs, a *subsystem* (a subcomponent or a part of the controller) either performs the given local action, or waits (if the idle action is used). This way, the transducer defines the cooperation of the subsystems and the association of the external method calls with the internal events.

For given actual parameters (pLTSes), the behavior of the pNet is given by parallel composition of the actual parameters and the transducer, resulting in new pLTS. This pLTS can be used as an actual parameter for a pNet specifying behavior of a component on the adjacent level of the component hierarchy. This way, pNets are used to specify behavior of hierarchical Fractal components. In fact, such an operation comprehending the transducer and the actual parameters represents behavior composition of pNets.

Using the formalism described above, the authors model behavior of Fractal applications, including the behavior of the controllers (i.e. both functional and nonfunctional behavior). Using the modal μ-calculus model checking [22], various properties of the modeled behavior can be automatically checked, including non-presence of method calls on unbound interfaces, successfulness of the component deployment, or the functional temporal properties defined by the developer (i.e. such properties that regard only the calls on functional interfaces).

A tool was developed that automatically generates the part of the model that refers to the nonfunctional behavior (the call on the controllers) from the description of a component system. In order to generate the whole model (by constructing the product of the generated model for the nonfunctional behavior and the model for the functional behavior that is written by a developer) the CADP toolset [17] is used. This toolset is also used to minimize the resulting model and to perform model checking on it.

2.3.2. Tracta

Tracta [18] is a verification framework allowing to analyze behavior of a distributed system specified as a hierarchy of components by automatic verification tools. As an underlying architecture description language, the authors of Tracta use Darwin [26].

A Darwin application consists of components; a component can be either *composite* (consisting of several *subcomponents*) or *primitive* (having no subcomponents). A component communicates with its environment via *portals* (interface instances). With every portal, a set of communication *events* is associated. Portals are connected through *bindings*.

In Tracta, each primitive component is equipped with a behavior description, which has the form of a labeled transition system (LTS) [10] or an expression in FSP [25] (a process algebra), from which the appropriate LTS can be automatically generated by a tool. Communication events defined by the portals are used as the labels in such an LTS.

An LTS representing the behavior of a composite component (or the whole application) is computed as a parallel composition of the LTSes of the subcomponents. The parallel composition is defined so that it addresses both independent computation of distinct components and the communication among the components.

As the parallel composition if designed as a binary operation, behavior of a composite component consisting of more than two subcomponents, or even the behavior of a composite component whose subcomponents are also composite, is computed in several steps. In order to optimize the time and memory needed for such a computation, after each step the LTS (the intermediate result) is minimized. The minimization is based on the week bisimulation relation [10]. Parallel composition with minimization of intermediate results represents behavior composition in Tracta.

A developer defines only the behavior of the primitive components. In this fact the Tracta approach differs from e.g. SOFA (Sect. 2.1), where behavior of all components (on all levels of nesting) is specified. Therefore, it is not possible to automatically compare the behavior of a composite component specified manually by the developer with the behavior computed by the parallel composition of the subcomponent specifications. Instead, Tracta provide the developer with the tools for model checking of the model resulting from the parallel composition.

There are two kinds of properties that can be subjects to model checking in Tracta: *safety properties* and *liveness properties*. A safety property is expressed as an LTS specifying all *acceptable* finite behaviors of the application under verification. If the LTS specifying the application behavior contains any finite trace that is not covered by the property LTS, the property is considered to be violated. In such a case, the verification tool reports the trace (a counterexample). Differing from safety properties, a liveness property reasons about infinite traces and is expressed as a Büchi automaton [15]. An application satisfies the property, if the automaton accepts all infinite executions of the system (expressed by infinite traces of the LTS describing the system behavior). If this is not the case, a counterexample is generated by the verification tool. The Büchi automaton can be either given directly, or automatically generated from an LTL formula [15].

2.3.3. Parameterized contracts

In [33], the authors introduce the concept of *parameterized contracts*. A parameterized contract is a mapping between provided functionality of a component and required functionality of the component. Using this mapping, it is possible to find out what functionality the component is able to provide in a given environment (as the parameters

of the environment determine which provisions of the component can be satisfied) or what are the requirements of the component if a given functionality has to be provided.

Such a mapping is very useful when a developer designs an application, finding the components with a required functionality in a repository. Typically, the components provide more functionality than is needed in the application. Using the mapping, it is possible to minimize also the requirements of the components in order to provide only the functionality required by the application specification and (recursively) to minimize the number of the components to build the application.

In [33], the RADL component model based on Darwin [26] is used. As the contract analysis is done on per-component basis, the component nesting is not important; also, each component is considered to have a single *provides-interface* (formed by the list of all provided methods) and a single *requires-interface* (the list of the required methods).

A parameterized contract (the mapping between provided and required functionality of a component) is derived automatically from behavior specification of the component, consisting of a *provides finite state machine* (P-FSM) and a set of *method requires finite state machines* (SE-FSM) – one for each method provided by the component. While P-FSM specifies all valid sequences of the provided method calls, a SE-FSM specifies all sequences of the required method calls, by which the component may react on the call of a particular provided method.

In addition, from the P-FSM and the SE-FSMs of a component it is possible to generate the *requires finite state machine* (R-FSM), specifying all possible sequences of the required method calls – such an operation represents a special kind of behavior composition. P-FSM and R-FSM are then used for the *substitutability checks*, answering the question whether a component can replace another one without violating the functionality of the whole application, and the *interoperability checks*, finding out whether given connected components cooperate correctly, i.e. whether the R-FSM of a component C is compatible with the P-FSM of another component C uses.

2.3.4. Component-interaction automata

In [39] the authors define *component-interaction automata* as a language for component behavior specification, that is independent on both component model and the verification tool that is used to check the properties of the specification. It is designed as to be general enough so that for any component model (and the associated architecture description language) a tool can be developed that transforms an ADL description into a system of component-interaction automata. On the other hand, the language is simple enough so that the existing verification tools (e.g. model checkers) can be used to verify properties of the models expressed in it (potentially after a necessary transformation).

For such an intermediate specification language behavior composition is a key concept: it cannot stick with a fixed method of behavior composition, as the universe of potential

component communication styles (influencing the method of behavior composition) is very huge. Therefore, component-interaction automata are defined as non-deterministic finite state automata, where the label associated with a transition in the automaton is a triple (A, *a*, B): A is the identifier of a component that emits an event (or the "_" symbol), *a* is the name of the event, and B is the name of the component that absorbs the event (or, again, the "_" symbol). Depending on whether and how the "_" symbol is used, we diversify three kind of events: inputs (of the form (_, *a*, B)), outputs (of the form (A, *a*, _)) and internal events (of the form (A, *a*, B)).

As the identifiers of the communicating components are stated in the model explicitly from the beginning, and the names of the components are not mixed with the names of the events (e.g. the *a* event emitted by the A component is distinguished in the model from the *a* event emitted by the B component), the behavior composition can be very flexible, reflecting any particular component communication style. In order to compose behavior specified by automata M_1, M_2, a product automaton M is constructed. In M, any transition is either adopted from M_1 or M_2, or it is a new internal event created as a merge of an input event from M_1 and an output event from M_2 (or vice versa). The process of behavior composition is also flexible in the question of hiding the input and output events from which a new internal event is created: the original events may be both visible and invisible in M, depending on the communication style that is modeled by the composition. The composition of two automata M_1, M_2 described here can be generalized to an arbitrary finite number of automata.

Once an automaton specifying behavior of a whole component application is constructed using the behavior composition, the desired properties of the model can be specified in a temporal logic and verified by an appropriate model checker. The authors evaluated their approach by experiments with the DiVinE model checker [3], whose input language is based on automata.

2.3.5. Wright

The goal of Wright [5] is to provide a formal basis for modeling *interactions* in software architectures, supported by verification tools. Wright is not associated with a concrete component model. Instead, it uses the notions of *components* and *connectors* for description of a general software architecture. In Wright, a component is a well-defined and independent part of an application, while a connector expresses an interaction among components. As the modeling of interactions is crucial here, connectors in Wright are very flexible: they are able to express many known interaction patterns, including procedure call, event passing, shared variables, pipe, and blackboard.

A component has several *ports*. Differing from typical component models, a port is not a classical procedure-call interface; it defines a logical point of an arbitrary interaction between the component and the environment. Similarly, a connector features *roles*. When components and connectors are instantiated, each port is *attached* as a connector role. This way, the components are connected by the connectors. For example, a connector

specifying a Unix pipe has two roles, `Reader` and `Writer`. Ports of two distinct components are attached as `Reader` and `Writer` in order to send the data through the pipe.

A Wright specification consists of three parts: (1) Definition of connector and component types, (2) definition of connector and component instances and (3) definition of attachments (describing connections among component instances via connector instances). A connector type is defined by the list of its roles and a *glue protocol*. Each role definition consists of a name and a *role protocol*. Both role and glue protocols are expressions in a subset of CSP [34]. While a role protocol specifies how a single component (featuring the port that is attached as the role) *should* participate in the interaction, the glue protocol specifies how the roles cooperate among each other. For example, let us recall the Unix pipe and its `Reader` and `Writer` roles: the role protocol for `Writer` specifies a sequence of write operations and a `close` operation, the role protocol for `Reader` specifies a sequence of successful read operations, followed by an event denoting that all data was read (`eof`), while the glue protocols describes the fact that `eof` cannot occur before `close`.

A component type is defined by the list of ports, each consisting of a name and a *port protocol*, specifying how the component *actually* participates in an interaction.

Three properties of Wright specifications are formally defined and can be checked by the tools: *compatibility* (of a port with a role), *deadlock freedom* (of a connector), and whether a connector is *conservative*.

A port is compatible with a role, if its behavior (specified by the port protocol) is substitutable for the role behavior in such a way that the rest of the connector interaction can't detect that the role behavior has been replaced by the port behavior. The formal definition of compatibility is based on the refinement relationship of CSP processes [34], which is extended in order to be not so strict and to support wider connector reuse.

A deadlock occurs when two (or more) components can wait in the middle of an interaction, each expecting the other to take some action that will never happen. Formally, deadlock freedom definition is based on *failures* of CSP processes [34].

A connector is *conservative*, if the behavior described by the glue protocol is consistent with the behaviors described by the role protocols (i.e., the glue protocol does not add any traces to those that are expressible as interleaving of the role traces). A connector is *well-formed* if it is both deadlock free and conservative. How the terms of compatibility, deadlock freedom, and conservative connectors relate to each other is described in the following theorem [5]: if a connector is well-formed and compatible ports are attached to all of its roles, then the resulting interaction (which is obtained by replacing each role protocol by the corresponding port protocol) is deadlock free. This theorem tells that the deadlock freedom can be checked locally (for each component type), and it is preserved for any compatible instantiation. It means that behavior composition (represented by the

replacement of the role protocols by the port protocols) is present only as a theoretical concept. It is not performed by the tool, as the deadlock freedom of the aggregate behavior is implied by the behavior properties that can be verified locally, without actually performing behavior composition.

For automated verification of compatibility, deadlock freedom, and connector conservativeness the FDR model checker [16] is used. FDR checks properties of finite state CSP processes. Therefore, only a subset of CSP specifying finite-state processes is supported by Wright. An input of FDR (a CSP process) is generated from port protocols, role protocols and glue protocols given by a Wright specification.

In order to capture behavior that cannot be specified by a finite-state CSP process, a glue specification can be extended by *trace specifications*. Recalling the example with Unix pipe, such an extension can express e.g. the fact that the pipe behavior complies with the FIFO discipline. However, deadlock freedom and conservativeness of such a specification cannot be verified in an automated way: the designer has to prove the properties manually.

Wright is focused on interactions. However, the interactions are verified separately. In a real system, two interactions can influence each other if a component participates in both of those interactions. Moreover, the influence can be complex, comprehending more interactions and components. All those aspects of the component/connector behavior are invisible in Wright, as the ports of a component are modeled as independent: any dependencies between interactions caused by the component are suppressed in the model. To eliminate this flaw, it would be necessary to equip each component with a glue protocol (currently, only connectors have glue protocols) and to redesign the properties to be checked in order to cover complex dependencies between interactions.

2.4. Non-component behavior specification and verification

2.4.1. Model checking

Model checking [15] is a formal verification method that allows to decide whether a given model (expressing behavior of a system, typically as a finite state Kripke structure) has a given property (typically expressed in a temporal logic).

As model checking was originally developed as a method for verification of hardware, the potential of coping with systems that consist of several parts executing in parallel and communicating with each other was researched. This inherently involves also pursuing behavior composition. The main outcome of this research were the optimizations of the model checking algorithms addressing the state explosion problem [15] and allowing to verify even very complex systems. Those optimizations are based on Binary Decision Diagrams [12] and their modifications (e.g. [14]). In model checking of communication protocols, distributed algorithms, cryptographic algorithms, or non-component software in general the component centric view (and the associated behavior composition) is present

only implicitly, mainly to take advantage of the optimization methods developed for hardware.

2.4.2. Process algebras

Process algebras [10] were developed to formally specify behavior of complex systems of communicating components (in a general sense) via expressions consisting of event/action labels and operators. Process algebras naturally address behavior composition via composition operators. Behavior protocols introduced in Sect. 2.2 can be seen a special case of process algebra, and also the composition operator of behavior protocols is very similar to composition operators from the famous process-algebraic specification languages - CCS [30] and CSP [34].

The semantics of process algebras, i.e. the behavior expressed via process-algebraic expressions, is defined in the terms of labeled transition systems [10]: oriented graphs with labeled edges, where the nodes denote the states of the modeled system and labeled edges (called transitions) denote events/actions in the system. Behavior composition is then typically defined in the terms of cartesian products of the labeled transition systems, where the transitions denoting the same communication event/action from the point of view of distinct components are merged in the resulting product. In addition, only reachable states of the product are considered.

2.4.3. Automata-based specification languages

In the area of formal specification and verification, several methods based on finite automata [20] were developed. In model checking, finite automata can be used for specification of temporal properties, either explicitly [18], or implicitly, as an intermediate language [19]. However, the ability of finite automata to be used as a language for behavior specification is much more important. The formal methods utilizing this ability include the component-interaction automata [39] described in Sect. 2.3.4 (finite automata are used here for behavior specification of software components), and also interface automata [4] and I/O automata [24] (behavior specification of general, i.e. non-component systems).

A finite automaton is typically determined by a set of states, some of which are denoted as accepting, and a transition relation, expressing whether a state *t* is directly reachable from another state *s*, and if it is, what event/action such a transition denotes. Finite automata can be classified according to several criteria: they can be either deterministic or non-deterministic [20], they can specify either finite or infinite traces (the automata specifying infinite traces are called Büchi automata [15]), etc.

For finite automata, behavior composition is typically defined via a cartesian product of the state sets. The particular automata-based methods may differ in the way the transition relations of the composed automata are merged together, specifying different

synchronization and communication styles between the subsystems specified via the automata.

2.4.4. UML 2.0

As a part of UML 2.0 [38], four graphical languages for behavior specification are defined, namely: interaction diagrams, activity diagrams, state machines, and their specialization, protocol state machines. None of those languages, in the form presented in the UML 2.0 specification, allows behavior composition: interaction diagrams lack an explicit and simple notation for behavior composition (although, in principle, behavior composition is possible via complex merging of two interaction diagrams); activity diagrams, state machines, and protocol state machines lack formal definition of their semantics that would be unambiguous [27].

On the other hand, UML provides a very powerful instrument: profiles. A profile allows to restrict (and hereby precise) any UML construct. As activity diagrams, state machines, and protocol state machines are based on finite automata [20] and Petri nets, it is not much complicated to complete their definition so that the semantics is unique (this approach is e.g. used in [28] to precise the semantics of protocol state machines). Behavior composition is then a natural operation: for activity diagrams, that stem from finite automata, it is performed as a cartesian product of state spaces (see Sect. 2.4.3), for state machines and protocol state machines, that stem from Petri nets, it is performed via connecting two state machines and adding a shared place for each communication event between them.

2.5. Behavior composition - summary

In all presented component systems (and also in the presented non-component specification languages), behavior is modeled by finite automata (for details see e.g. Sect. 2.4.3) or finite labeled transition systems (Sect. 2.4.2). Although the Wright specification language (Sect. 2.3.5) allows to specify infinite state behavior, such a specification cannot by verified automatically by a tool.

Behavior composition is either defined via specification rewriting (Wright – Sect. 2.3.5, parameterized contracts – Sect. 2.3.3), via Petri net transformation (state machines – Sect. 2.4.4), or as the cartesian product of state spaces (automata or labeled transition systems) with modification of the resulting transitions following the communication style of the specified system (the rest of the cases).

Chapter 3

Problem statement and addressing the goals

3.1. Problem statement

In Chapter 2, we have described current component models that employ formal behavior specification and their approach to behavior composition. We claim that all those component models lack support for *composition error detection* and do not address the problem of *unbounded parallelism specification*. In this section, we roughly describe those two concepts, explain why they are important and show how they relate to behavior composition.

3.1.1. Composition errors

Composition error basics. Typically, during behavior composition the correct behavior of a composite component is determined. We propose to detect also *composition errors*: the errors that are caused by composition of components that are not able to cooperate in a correct way. None of the current component systems utilizing formal behavior description takes advantage of this option. As a consequence, many composition errors that could be detected in principle are ignored at all, as the information on such errors is typically lost during behavior composition (and therefore the errors cannot be identified by model checking the resulting aggregate behavior).

A composition error indicates a design problem that is able to cause malfunction of the component application. Even if the problem does not become evident at a runtime as a malfunction, still there is a design inconsistence in the application, that could e.g. make the maintenance of the application difficult. In either case, ignoring the composition error brings problems into the development process.

Erroneous component architectures. There are two basic approaches to formal behavior specification of hierarchical component systems: either all components have associated a behavior specification written by a developer (e.g. SOFA with behavior protocols, Sect. 2.1 and Sect. 2.2), or the manually written specification is associated only with primitive components (e.g. Tracta, Sect. 2.3.2). In the first case, for a composite component there exist two specifications: the manually written one and the specification describing the aggregate behavior, computed automatically from the behavior specifications of the

subcomponents. Typically (e.g. in SOFA) there is a tool that checks the compliance between those two specifications.

However, the manually written specification not only expresses how the component should behave: it also says how the component may be used (what is the expected behavior of its environment E). Considering the manually written specification as to be a constraint, it is possible that a part of the behavior described in the aggregate behavior specification becomes irrelevant when the composite component is put into the environment E. In particular, some of the composition errors identified during the behavior composition have no effect under those constraints (the operations causing the errors may not occur).

Therefore, it is important not only to check composition errors for a given composite component (component architecture), but also to take into account the expected behavior of the environment: erroneous component architecture (i.e. an architecture in whose behavior a composition error can occur) is a relative concept. It is necessary to count with this fact when designing a tool for composition error detection.

3.1.2. Finite state specification of unbounded parallelism

Any behavior specification language is always a compromise between the expressive power (in order to sufficiently wide class of systems can be specified by the language) and the existence of efficient algorithms for verification of the behavior described in the language (so that the appropriate verification tools can be developed). Most of the languages for component behavior specification are finite state, i.e. each behavior specified in such languages can be equivalently specified by a finite labeled transition system [10] or a finite state machine [20]. Although verification algorithms for such languages have typically nontrivial time and memory complexity (because of the state explosion problem [15]), still most of the verification problems are decidable for those languages.

The weakness of the finite state specification languages is the fact that they are not able to specify every behavior of real-life software components. The non-expressible behavior includes access to data structures with unbounded data domains, unbounded recursion, or *unbounded parallelism*: the situation when arbitrary (but still finite) number of method calls may be accepted or emitted by a component in parallel. Modeling of unbounded parallelism is very important: while data structures and recursive method calls are often hidden as implementation details and do not influence the “black box” view of the component behavior, the specification of parallel method calls on the component’s interfaces is necessary to distinguish between reentrant and non-reentrant components, restricting substantially the way the component may interact with the environment.

To specify precisely the behavior of a reentrant component while not throwing over the benefits of the finite specification languages, it is necessary to distinguish between the behavior specification at the component design time (when the component is developed), and the specification at the architecture design time (when the component is instantiated

in an architecture/as a part of a composite component). While at the component design time a reentrant component has to be considered as practicing unbounded parallelism, at the architecture design time the upper bound to the number of parallel calls is often determined by the behavior of the other components in the architecture. Therefore, it is possible (at least for some component architectures) to specify the behavior of the whole architecture by a finite state model (in a finite state specification language), even if the behaviors of (one or more) components in the architecture have to be specified by infinite state models when considered separately. This is called *finite state specification of unbounded parallelism*. To our knowledge, none of the component systems allows to identify the architectures for which finite state specification of unbounded parallelism is possible and to automatically compute the resulting finite state models, that could by analyzed using the common verification tools.

3.1.3. How the issues relate to behavior composition

Both detection of composition errors and finite state specification of unbounded parallelism are closely related to behavior composition. It is natural to detect the composition errors during behavior composition, as once the composition is accomplished, the information needed for the detection is typically lost. Moreover, it is convenient to let the algorithm for behavior composition share certain information with the algorithm for composition error detection so that many duplicate operations are performed only once.

Similarly, finite state specification of unbounded parallelism relates to behavior composition: to determine the upper bound of the number of parallel calls for a reentrant component, the behavior specification of the other components is needed. In other words, the information needed for the upper bound computation is the same as the information requested as the input for a behavior composition – behavior specification of all components in a given architecture. Therefore, it is natural to consider the upper bound computation and the subsequent finite state model generation (for the architectures with components practicing unbounded parallelism) as a part of behavior composition.

3.2. Goals of the thesis

The general goal of the thesis is to address the problems of composition error detection and finite state specification of unbounded parallelism introduced in Sect. 3.1. This is the list of detailed goals:

(G1) To identify what types of composition errors occur in component systems.

(G2) For each type of composition error identified within (G1), to propose a technique of checking for it.

(G3) To propose a method of composition error checking inside a composite component in the context of the environment the component is put into. I.e., only the relevant

composition errors (those that are able to occur when the component is put into the environment) are detected.

(G4) To propose a language for specification of unbounded parallelism at a component design time.

(G5) To determine the conditions under which the behavior of a composite component (a component architecture) can be expressed by a finite state specification even if one or more subcomponents practice unbounded parallelism.

(G6) To propose an algorithm that automatically generates a finite state behavior specification for a composite component (component architecture) where one or more subcomponents practice unbounded parallelism, providing that the architecture satisfies the conditions determined in (G5) and the unbounded parallelism is specified using the language proposed in (G4).

The solutions of the problems will be demonstrated on the SOFA component model [35], behavior protocols [32] will be used as a behavior specification language.

3.3. Addressing the goals

In this section we show how the goals from Sect. 3.2 are addressed by the papers [AP05] (Chapter 4), [AP04a] (Chapter 5), [AP04b] (Chapter 6), and [Ada06] (Chapter 7).

The goals (G1) and (G2) are addressed in the papers [AP05], [AP04a]. In [AP05] we introduce three types of composition errors – bad activity, no activity, and divergence – including the technique of checking for those composition errors, formalized as the *consent operator* for behavior protocols. In [AP04a], we introduce the fourth type of composition errors - the unbound requires errors. The problem of checking of this type of errors is reduced to the checking of bad activity errors by transformation of the component architecture.

The goal (G3) is addressed in [AP04b]. Here, a method of composition error detection in the context of a given environment is proposed. The method is based on the usage of the consent operator, the behavior of the environment is expressed by so called *inverted frame protocol*, which is automatically generated.

The goals (G4), (G5), (G6) are addressed in [Ada06]. Here, we introduce *behavior templates* based on behavior protocols, that are able to specify unbounded parallelism at the component design time (addressing (G4)). At the architecture design time, those behavior templates are transformed into common behavior protocols, that are finite state. For a given component architecture we define the *dependence graph* that expresses dependencies among the transformations of all behavior templates associated with the components in the architecture. The properties of the dependence graph determine whether

all the transformations can be performed in a way respecting all the dependencies. In other words, those properties determine whether the behavior of the component architecture can be specified using the finite state specification. We show for what types of dependence graphs (component architectures) the template transformations are possible (addressing (G5)). Finally, we show how the transformations are performed (addressing (G6)).

The methods addressing the goals (G2) and (G3) were implemented in the Behavior Protocol Checker, a tool which is available for the download on the website of the SOFA project [35]. However, this implementation is not a work of the author of this thesis. The method addressing (G6) has not been implemented yet. The goals (G1), (G4), (G5) are theoretical and form the intermediate steps to the solutions of (G2), (G3), (G6) – see Sect. 3.2.

Chapter 4

Component Composition Errors and Update Atomicity: Static Analysis

This chapter contains the paper

[AP05] Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance and Evolution: Research and Practice 17(5), Sep 2005

Summary. In this paper, we introduce the concept of a *composition error* in order to formally describe the problems that occur when software components with incompatible behavior are composed together. We extend the *composition operator* denoting behavior composition for behavior protocols (defined in [32]) so that it produces not only the traces describing the correct behavior, but also the traces containing the composition errors (so called *erroneous traces*). The new operator is called the *consent operator*.

We identify three types of composition errors: *bad activity*, *no activity*, and *divergence*. A bad activity occurs when a component A tries to emit an event on its required interface R and the counterpart (a component B) is not able to absorb the event. By counterpart we mean the component that features a provided interface P to which R is (directly or indirectly) connected. The ability of B to absorb the event is determined by the frame protocol of B and the state of B, i.e. the sequence of events that occurred on the interfaces of B before A tried to emit the event.

A no activity error means that no component in a given architecture (within a composed component) is able to progress in the computation, and at the same time, at least one of those components has not finished its task yet. Such a situation occurs e.g. when one or more components wait for a method call on their required interfaces, but non of the components in the architecture is able to emit such a call.

Divergence is defined as an infinite run of a component architecture (a composed component) – the components never stop to communicate.

The behavior composition with detection of composition errors is formally represented by the consent operator. As behavior is modeled as a language (see Sect. 2.2.2), the consent operator has three parameters: two languages representing behavior of two groups of

components, and the set of all events that may occur on the bindings between those two groups of components. In the result of the behavior composition (a new language representing the aggregate behavior of both groups of components) a composition error is represented by an error trace: a sequence of classical event tokens from Sect. 2.2.1 (denoting correct communication among the components) and ending with an *error token*, denoting a concrete type of a composition error (bad activity, no activity, or divergence).

The consent operator is commutative and associative. Therefore, it is not important in which order the behavior of components in a given architecture is composed.

In the Section 4 of the paper, "Dynamic updates", we apply the concept of bad activity error in order to detect update atomicity for dynamically updated components. Here, update atomicity means that during a dynamic update of a component C (i.e. replacement of C's implementation by a new version at the run-time), no other component tries to call any method on the provided interfaces of C. Update atomicity is important to preserve consistency of C's data structures.

Comments. It this paper, we provide a general form of the consent operator definition, based on languages [20], in order to the operator can be used to compose arbitrarily complex behavior. However, for practical purposes, i.e. for construction of a tool that checks for composition errors, a definition based on finite automata is more suitable (although it addresses only finite state behavior). Such an automata-based definition of the consent operator can be found in [2].

The main reason for defining divergence error is the fact that our semantic model for behavior protocols is based on finite traces and infinite computation is not directly representable within this model. On the other hand, it is not desirable to ignore the possibility of an infinite computation (such an information is valuable for the developer), especially due to the fact that it can be detected statically from the behavior protocols. Therefore, we decided to include the infinite computation into the result of a behavior composition as a divergence error.

Component Composition Errors and Update Atomicity: Static Analysis*

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

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

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

Abstract

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

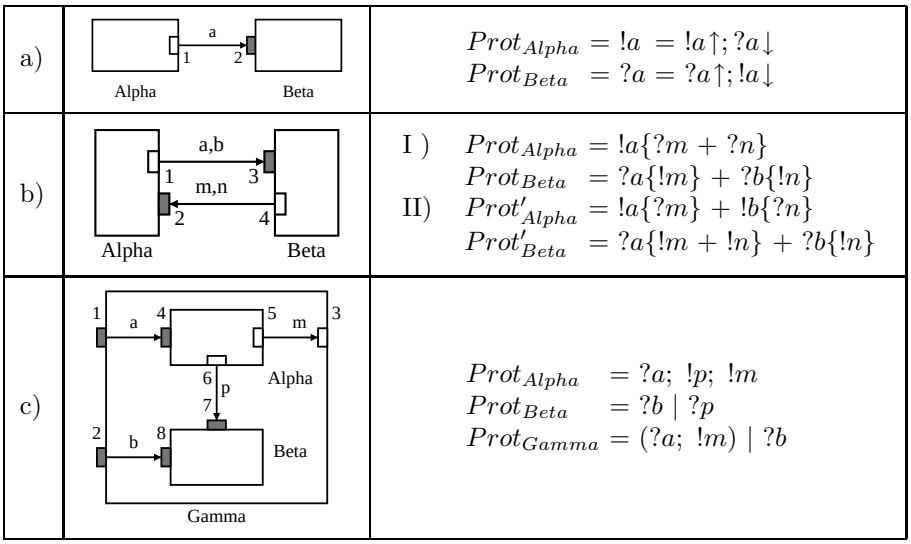


Figure 1: Examples of component systems and frame protocols

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.

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

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 is 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... . 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* $UN = \{\pi, \pi_1, \pi_2, ...\}$, $UN \subseteq EN$.

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 versaa), 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

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

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

sequence of event tokens t, the subcomponent A resp. B have processed the sequence $Trace_A(t)$ resp. $Trace_B(t)$. We say that a component P *can stop* after it has processed a trace t if $t \in L_P$. Thus, A can stop after C has processed t if $Trace_A(t) \in L_A$; in a similar vein, B can stop after C has processed t if $Trace_B(t) \in L_B$. Note that C can stop after processing t iff both A and B can stop.

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

To capture computations with errors, we introduce *error tokens* $\varepsilon n\uparrow$, $\varepsilon n\downarrow$, $\varepsilon\oslash$ and $\varepsilon\propto$, $(n \in EN)$, and *erroneous traces* of the form $w{<}e{>}$, where w is a trace formed of non-error event tokens ($w \in ((ACT \backslash ErrorTokens)^*)$[4] and e at the end of the trace stands for the error token reflecting the type of the error occurred. In a case of a no-continuation error, the error occurs at the end of the trace (just where e is located). In a case of divergency, an erroneous trace is a finite prefix followed by $\varepsilon\propto$ to represent an infinite continuation.

In a case of *bad activity*, A tries to emit $!n\uparrow$ or $!n\downarrow$ $(n \in EN)$, but B at the other side of the respective binding is not ready to accept (to issue $?n\uparrow$ resp. $?n\downarrow$), i.e. no suitable trace is defined in B's behavior. A bad activity is expressed by an error token of the form $\varepsilon n\uparrow$ or $\varepsilon n\downarrow$. For example, the composition of P_{Alpha} and P_{Beta} from Sect. 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:

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

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

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

In a case of *divergence*, A and B can emit events, but after each event, at least one of the components cannot stop. Such computation can be formally captured by an infinite meta-trace of the form wt, $w \in T^*$, $t \in T^\infty$, $T = Tokens_\tau(X) \cup (S_A \backslash\ (Tokens_?(X) \cup Tokens_!(X))) \cup (S_B \backslash\ (Tokens_?(X) \cup Tokens_!(X)))$. Here, w is a (finite) correct prefix, i.e. w is also a prefix of a non-erroneous trace. Until the whole w is processed, it would be always possible to chose another path of computation such that C could stop. The

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

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 $w\beta < \varepsilon \propto >$, where β is a finite prefix of t such that one of A, B can stop after C has processed $w\beta$, but the other cannot stop. For example, let the frame protocols in Fig. 1b) be: $P_{Alpha} = (!a; (?m; !a)^*)$, $P_{Beta} = (?a; !m)^*$. Their composition results into the infinite meta-trace $<\tau a\uparrow;\ \tau a\downarrow;\ \tau m\uparrow;\ \tau m\downarrow;\ \tau a\uparrow;\ \tau a\downarrow;\ \tau m\uparrow;\ \tau m\downarrow;\ ...>$. It is represented by the set of erroneous traces of the form $\{<\tau a\uparrow;\ \tau a\downarrow;\ \varepsilon\propto>,\ <\tau a\uparrow;\ \tau a\downarrow;\ \tau m\uparrow;\ \tau m\downarrow;\ \varepsilon\propto>,\ <\tau a\uparrow;\ \tau a\downarrow;\ \tau m\uparrow;\ \tau m\downarrow;\ \tau a\uparrow;\ \tau a\downarrow;\ \varepsilon\propto>,\ ...\}$. Formally, the *divergence set* $DIV(L_A, L_B, X)$, containing the erroneous traces of an infinite activities such that A can stop (and B cannot), is defined as follows:

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

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

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:

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

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

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

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 $\varepsilon\oslash$. Then, in every step of any computation, C can always stop or process an event.

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

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 $tp <?\pi\uparrow>$ (or $tp <?\pi_n\uparrow>$) 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 updated 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 too 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 $F_1, ..., F_n$ with frame protocols $Prot_{F_1}, ..., Prot_{F_n}$. The proposed method verifies update atomicity in two steps: (1) *Local atomicity* is tested for every frame protocol $Prot_{F_1}, ..., Prot_{F_n}$: 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 $Prot_{F_1}, ..., Prot_{F_n}$ 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 $n \in EN \setminus UN$.

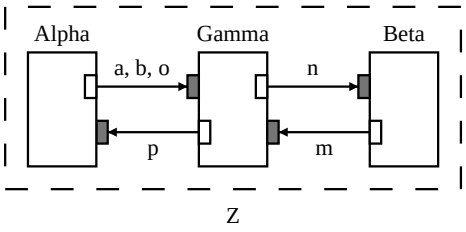


Figure 2: Updating the *Alpha* component

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

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

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

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

where $X = \{a\uparrow, a\downarrow, b\uparrow, b\downarrow, o\uparrow, o\downarrow, p\uparrow, p\downarrow\}$ and $Y = \{m\uparrow, m\downarrow, n\uparrow, n\downarrow\}$. From $Prot_Z$ we see that: 1) Atomicity of π_1 is ensured. 2) During the update π_1, *Beta* and *Gamma* can communicate, not violating the atomicity of π_1. (Ex. 2): Let *Beta* have the frame protocol $NULL$ (i.e., it does nothing) and let $Prot_{Alpha} = ((!a;?\pi_1\uparrow;!\pi_1\downarrow) + (!b;?\pi_2\uparrow;!\pi_2\downarrow));(!o\,|?p)$, $Prot_{Gamma} = (?a;?o;!p) + (?b;(?o \mid !p))$, 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 $w{<}\tau\pi_1\uparrow;\varepsilon e{>}$ for an event e and $w \in ACT^*$). 2) The atomicity of π_2 is not ensured. This fact is indicated by erroneous traces in the resulting behavior (error token occurs always after $\tau\pi_2\uparrow$). 3) All erroneous traces are caused by violating the atomicity of an update (thus no erroneous trace contains $\tau\pi_2\downarrow$). Note, however, when the atomicity of π_2 was ensured by another mechanism (interface locking, etc.), *Alpha* could be updated successfully.

Our update mechanism allows to do unanticipated changes in the structure of a component system: at the design stage, the architecture A_{i+1} updating a C/A_i is unknown. The only part of the C's design which reflects the possibility of an update is the C's frame protocol where the π tokens are to be stated — however, this is not a crucial problem: 1) Even though the update moments are

given by the behavior protocol, an update 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 $\|_X$ in CSP, provided the hiding operator $\setminus X$ is also applied. The main difference is that in CSP the communication is symmetric, i.e. in $P \|_X Q$ the events in P and Q denoted by the same token are synchronized, and a failure occurs when an event from X appears just in one of the processes P 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, Garlan 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 Polylith.* 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.

Chapter 5

Partial Bindings of Components - any Harm?

This chapter contains the paper

[AP04a] Adamek, J., Plasil, F.: Partial Bindings of Components - any Harm?, Presented at the SACT 2004 Workshop, Busan, Korea and published in the Proceedings of APSEC 2004, IEEE Computer Society, ISBN 0-7695-2245-9, pp. 632-639, Nov 2004

Summary. According to the original SOFA component model, any required interface has to be connected (potentially indirectly) to a provided interface of the appropriate type (while a provided interface may remain unconnected). Motivation for such a rule is straightforward: an unconnected required interface means that the functionality needed for correct functioning of the component that employs the interface is not available, while an unconnected provided interface only means that a part of the functionality provided by the component is not used.

In this paper we analyze the issue in depth and show that sometimes it may be useful (and quite safe) not to connect a required interface, e.g. when only a part of the component's functionality is used (i.e. also some of the provided interfaces remain unconnected), and as a consequence the component "requires less". On the other hand, an unconnected provided interface can cause bad activity and no activity errors.

We show that an unconnected required interface may cause problems only when the component tries to call a method on it (such a situation is called an *unbound requires error*). Therefore, we propose a method of checking whether an unbound requires error can occur in a given component architecture (composite component), providing that the behavior of each component is specified by a behavior protocol. The unbound requires error poses the forth kind of composition error (in addition to bad activity, no activity, and divergence).

The bad activity and no activity errors caused by an unconnected provided interface can be checked using the approach introduced in [AP05].

Comments. In this paper, we show how the problem of checking for an unbound requires error can be transformed to the problem of checking for a bad activity error (which has been solved in [AP05]) in an artificial component architecture that is constructed from the

given architecture according to a set of rules (the rules are presented in the paper in detail). We chose this approach in order to the text of the paper is clear and readable. However, in practice (i.e. in the implementation of the Behavior Protocol Checker), no artificial architecture is constructed. Instead, identification of unbound requires errors becomes a part of the behavior composition. Such an approach is formalized as an extension of the consent operator, as described in [2].

Partial Bindings of Components - Any Harm?*

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

[1]*Charles University, Faculty of Mathematics and Physics*
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{adamek,plasil}@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz,
phone: +420 2 2191 4266, fax: +420 2 2191 4323

[2]*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

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

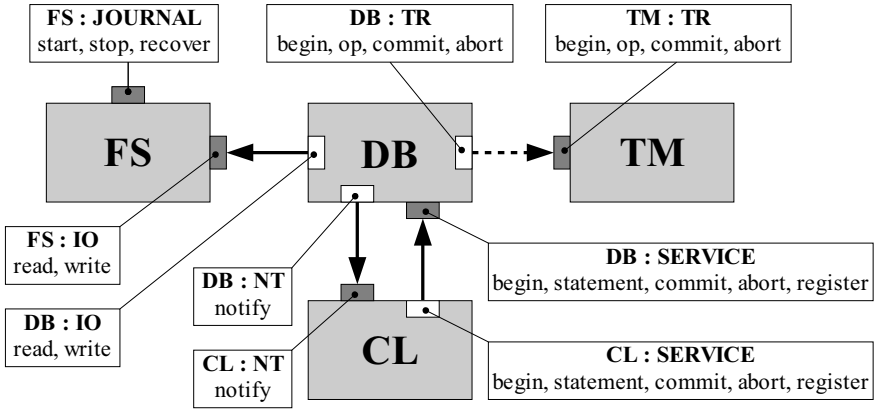


Figure 1: Example of component application

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

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 an 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↑ and the response as m↓. 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↑ ; !C:PI.m↓ where ?C:PI.m↑ and !C:PI.m↓ 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 "∇" (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 $\mathbf{a}\uparrow$ and $\mathbf{a}\downarrow$ are the CCS names that correspond to request and response of an event a, and by mapping ?a↑ to $\mathbf{a}\uparrow$ and !a↓ to $\overline{\mathbf{a}}\downarrow$, 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↑ + ?b↑ is equivalent to the CCS process $\mathbf{a}\uparrow.\mathbf{0} + \mathbf{b}\uparrow.\mathbf{0}$ (where $\mathbf{0}$ denotes inactive process). In a similar vein, sequencing can be expressed via the prefix operator **"."**, e.g. ?a↑ ; !a↓ as $\mathbf{a}\uparrow.\overline{\mathbf{a}}\downarrow.\mathbf{0}$. 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 $\mathbf{P} \,|\!|_{\{\}}\, \mathbf{Q}$, where $\mathbf{P}$, $\mathbf{Q}$ are CCS processes equivalent to protocols P, Q and $\{\}$ denotes empty set (the $|\!|_{\mathbf{N}}$ 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↑;!a↓)* means calling the method a repeatedly finite number of times, while the CCS expression with the meaning "closest" to R, i.e. $\mathbf{X} = \mathbf{0} + \mathbf{a}\uparrow.\overline{\mathbf{a}}\downarrow.\mathbf{X}$, 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

```
Prot_FS =
  ( ?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:

```
Prot_DB =
(
  ?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{<some actions>} stands for ?x.y↑ ;<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: $Prot_G = Prot_{G1} \nabla_S Prot_{G2}$ 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 $\mathbf{CCS_{G1}}$, $\mathbf{CCS_{G2}}$ describe the same behaviors as the group protocols $Prot_{G1}$, $Prot_{G2}$. Provided S is the set of all events occurring in the communication between G_1 and G_2, the meaning of $Prot_{G1} \nabla_S Prot_{G2}$ is equivalent to $\mathbf{(CCS_{G1} \mid CCS_{G2}) \setminus S}$ in CCS except for the fact that ∇_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, $\mathbf{e}, \overline{\mathbf{e}}$ in CCS for $e \in S$) resulting in *internal events* (τe in behavior protocols, $\boldsymbol{\tau}$ in CCS). The events which are not elements of S are in the result of $Prot_{G1} \nabla_S Prot_{G2}$ arbitrarily interleaved.

We demonstrate the semantics of ∇_S on the following example (formal definition of ∇_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 = { <CL:SERVICE-DB:SERVICE>.begin↑,
       <CL:SERVICE-DB:SERVICE>.begin↓,
       <CL:SERVICE-DB:SERVICE>.commit↑,
       <CL:SERVICE-DB:SERVICE>.commit↓,
       <CL:SERVICE-DB:SERVICE>.abort↑,
       <CL:SERVICE-DB:SERVICE>.abort↓,
```

```
<CL:SERVICE-DB:SERVICE>.statement↑,
<CL:SERVICE-DB:SERVICE>.statement↓,
<CL:SERVICE-DB:SERVICE>.register↑,
<CL:SERVICE-DB:SERVICE>.register↓,
<DB:NT-CL:NT>.notify↑,
<DB:NT-CL:NT>.notify↓ }.
```

Here, <CL:SERVICE-DB:SERVICE>.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

```
Prot_CL ∇_S1 Prot_DB =
(
  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 τ 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 $Prot_{DB}$ describing transactional processing was "silently eliminated" in the result of the composition $Prot_{CL} \nabla_{S1} Prot_{DB}$, as the client ($Prot_{CL}$) 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

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

$Prot_{CL}' \nabla_{S1} Prot_{DB}$ would result in a bad activity error, since $Prot_{DB}$ does not allow calling commit as the first event of any run:

```
Prot_CL' ∇_S1 Prot_DB = 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 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

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

the composition of $Prot_{CL}''$ and $Prot_{DB}$ would result in a no activity error (denoted by the event token ε∅):

```
Prot_CL'' ∇_S1 Prot_DB =
  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.

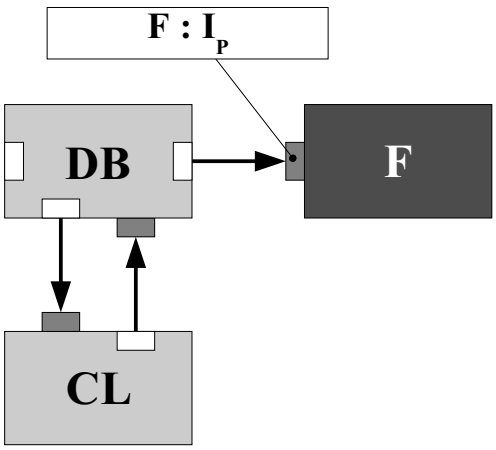


Figure 2: Usage of a feigned component F

The concept of bad activity and no activity errors can be naturally extended to a group of n≥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 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 ($Prot_F$ = 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 $Prot_G$ be the group protocol of G):

$$Prot_{G\text{-}F} = Prot_G \nabla_S Prot_F = Prot_G \nabla_S NULL$$

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

$Prot_{CL}''' =$!CL:SERVICE.`begin` ; !CL:SERVICE.`statement` ;
!CL:SERVICE.`commit`

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

$Prot_{CL\text{-}DB}''' = Prot_{CL}''' \nabla_{S1} Prot_{DB} =$
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, 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 $Prot_{CL\text{-}DB\text{-}F}'''$ is

$Prot_{CL\text{-}DB}''' \nabla_S Prot_F = Prot_{CL\text{-}DB}''' \nabla_S NULL =$
τ<CL:SERVICE-DB:SERVICE>.`begin`↑;
ε<DB:TR-F:I_P>.`begin`↑

Clearly, only the request for the `begin` method (τ<CL:SERVICE-DB:SERVICE>.`begin`↑) is processed, after which an unbound requires error occurs. It is captured by the bad activity error ε<DB:TR-F:I_P>.`begin`↑. Here S contains all the events on the new binding, i.e.

S = {
<DB:TR-F:I_P>.`begin`↑, <DB:TR-F:I_P>.`begin`↓,
<DB:TR-F:I_P>.`op`↑, <DB:TR-F:I_P>.`op`↓,
<DB:TR-F:I_P>.`commit`↑, <DB:TR-F:I_P>.`commit`↓,
<DB:TR-F:I_P>.`abort`↑, <DB:TR-F:I_P>.`abort`↓
}.

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, Garlan 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 Sfotware 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 SNPD'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] OMG Corba Component Model (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

Chapter 6

Erroneous Architecture is a Relative Concept

This chapter contains the paper

[AP04b] Adamek, J., Plasil, F.: Erroneous Architecture is a Relative Concept, in Proceedings of Software Engineering and Applications (SEA) conference, Cambridge, MA, USA, published by ACTA Press, ISBN 0-88986-425-X, pp. 715-720, Nov 2004

Summary. In [AP05] and [AP04a] we have shown how to check for composition errors. The methods introduced there identify all composition errors that can occur during a run of a given composite component (a component architecture) in any environment. In other words, there are no additional assumptions on how the composite component is used by other components.

In this paper, we show that in a particular environment some of the composition errors may not be able to occur, although in another environment they can. As an extreme case, there are composite components (component architectures) that are able to produce errors in one environment, but they are errorless in another one. Therefore, we claim that erroneous architecture is a relative concept and propose how to statically check for composition errors in a given environment.

Contrary to [AP05] and [AP04a], to identify composition errors for a composite component C, now we need not only the behavior specifications of C's subcomponents, but also the specification of C's environment. Provided that with C a frame protocol is associated (that specifies how C behaves towards its environment) we are able to construct so called *inverted frame protocol* for C that specifies how the environment behaves to C. Using this inverted frame protocol as a restriction we identify only the composition errors that are relevant within the given environment.

Comments. In fact, in the paper we solve a problem that is a little more general than described above. We analyze the behavior of *component frameworks*. A component framework is a predefined fragment of component architecture. A developer that wants to reuse the framework defines his or her own components, connects them with the framework (obtaining a complete component architecture) and puts the result into a frame (that can be seen as an abstraction of the environment), creating a *setup*. Here, the behavior of the setup is parameterized by both the components added to the framework by the

developer and the environment. Checking for composition errors in a (complete) component architecture within a given environment forms a special case of this problem - no components are added, the architecture is just put into the frame.

However, from the point of view of behavior composition, the generalization of the problem to component frameworks does not change the algorithm of composition error checking (either within a given environment or without any restriction). It is not important who and when defines a particular component forming the architecture (a framework author or a developer that reuses the framework). To do the work, the algorithm just requires as input the architecture description and behavior specifications of all the component in the architecture.

ERRONEOUS ARCHITECTURE IS A RELATIVE CONCEPT*

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

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

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

ABSTRACT
The problem this paper addresses is that an architecture formed of software components can contain composition errors (introduced, for instance, as a result of the choice of a framework's parameters). The title "Erroneous architecture is a relative concept" is to emphasize that whether a composition error occurs in an architecture depends on the way the architecture is used in its environment. An important issue is finding a way to possibly statically verify that, for a given setup containing the architecture, no composition errors can occur in any run. The contribution of the paper is bringing an evidence that this can be done by employing behavior protocols and their consent operator.

KEY WORDS
Faulty software architecture, framework, component behavior

1. Introduction and Motivation

1.1. Software Component Background

In addition to the industrial component models [1, 2], which operate with simple, object-granularity level components, there are a number of advanced component models [3, 4], all based on a very similar idea: A component features interfaces, each of them being either *provides* or *requires*. (Terminology slightly differs, we stick with our SOFA[5] component model inspired by Darwin [6]). A provides interface denotes the services offered by a component. It consists of a set of provided methods; this set determines the *type* of the interface. Provides interfaces are very similar to interfaces in Java. A requires interface R expresses the fact that a component needs to call methods on a provides interface P of another component to work properly (i.e., a requires interface is an abstraction of a reference to another interface). Again, it consists of a set of required methods determining the interface type. A requires interface R has to be connected to a provides interface of the same type. This connection is realized by an interface tie – provision of a (possibly remote) reference in the implementation view.

Many component models allow for component nesting to support top-down design and refinement. Figure 1 shows a *composed component* (`FORECAST`), providing the weather forecast service. It consists of three *subcomponents* (`SWITCH`, `CACHE`, and `ENGINE`). The small dark/white boxes denote provides/requires interfaces. The caption of an interface shows the name of the component that features the interface, local name of the interface and methods which are provided/required. For instance, `CACHE:C` is a provides interface of `CACHE`, its local name is `C` and contains the `get` and `put` methods.

A tie between a requires interface and a provides interface is called *binding* (e.g. `SWITCH:C->CACHE:C` in Fig. 1). If nested components are considered, a tie can also have the form of *delegation* (a call on a provides interface of the parent component is forwarded to a provides interface of a subcomponent), e.g. `FORECAST:F1->SWITCH:F1` in Fig. 1, or *subsuming* (a call on a requires interface of a subcomponent is forwarded to a requires interface of the parent component), not employed in the setting in Fig. 1.

1.2. Running example - settings

We will illustrate all the concepts used in this paper on the example from Fig.1. As mentioned in Sect 1.1, the `FORECAST` component provides weather forecast for a given region at a given time (the region and time are passed as

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

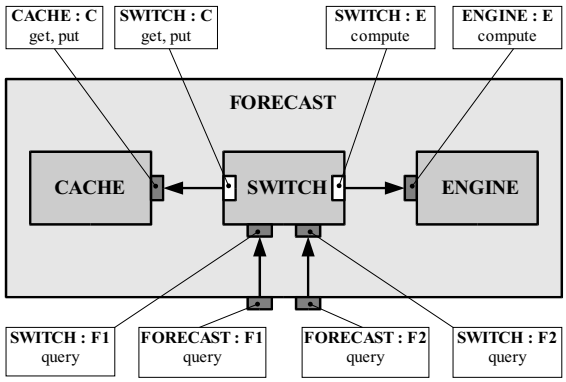


Figure 1: A composed component (FORECAST), providing the weather forecast service, consisting of three subcomponents (CACHE, SWITCH, and ENGINE)

parameters to the query method from the FORECAST:F1 or the FORECAST:F2 interfaces). The functionality of FORECAST:F1 and FORECAST:F2 is the same, the interface is duplicated to provide the service to two clients – we do not consider more clients to keep the example simple; for the same reason, we assume that each of those interfaces can be used by at most one thread at a time.

A forecast is the result of a time-demanding computation stemming from a complex mathematical model. As the forecast for a particular time and region can be requested repeatedly, it makes sense to cache the results of computations. Therefore, when query is called on FORECAST:F1 or FORECAST:F2 (and the call is delegated to SWITCH:F1 or SWITCH:F2 respectively), the SWITCH component first asks CACHE, whether the forecast for given parameters has been computed recently (the get method). If this is the case, CACHE returns the result immediately, which is consequently returned to the caller of query. If the requested forecast has not been computed yet, SWITCH calls compute on ENGINE:E to get one, stores it into CACHE (put) and finally returns it to the caller of query.

Now, let us focus on reentrance of the subcomponents. As SWITCH just "forwards" the calls and makes a simple decisions based on the answers of CACHE, we will assume that it is reentrant, i.e. the methods on SWITCH:F1 and SWITCH:F2 can be called by two threads simultaneously (one thread on each of these interfaces) without any negative impact on its functionality.

Since ENGINE does a time-consuming numerical computation, calling the compute method by several threads in parallel is feasible. Whether this is really possible depends on the way ENGINE manipulates its internal data structures. However, the reentrancy of ENGINE should not be understood as an implementation detail, as it influences reentrancy of FORECAST:

1) If ENGINE is reentrant (two threads can call the methods on ENGINE:E concurrently), FORECAST is reentrant as well (because SWITCH and CACHE are also reentrant).

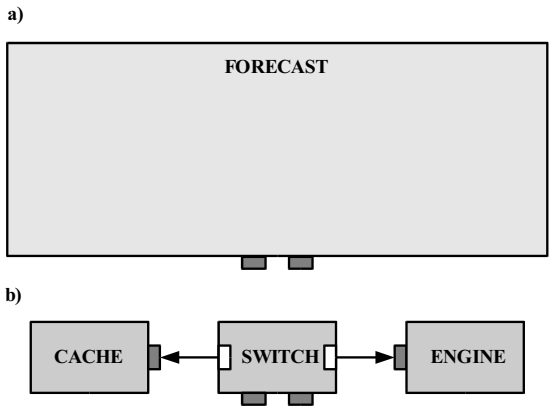


Figure 2: a) the frame of the FORECAST component; b) the architecture of the FORECAST component

2) If ENGINE is not reentrant, concurrent usage of ENGINE:E results in a corruption of its internal data structures. Note that calling query on FORECAST:F1 and FORECAST:F1 simultaneously can eventually result in a concurrent call of compute on ENGINE:E (in case none of the two requested forecasts have been previously computed). Therefore FORECAST is also not reentrant.

1.3. Software Architectures and Frameworks

The *setup* of cooperating components from Fig.1 is ready to communicate with an environment on its external connections (via ties to FORECAST:F1, FORECAST:F2). As an aside, since we have considered alternatives of the component ENGINE - a final setup can be considered as an instance of a component-based framework where ENGINE is a parameter. A particular choice of such a parameter can influence the way some other components in the setup need to communicate, which consequently can be OK for some environments but for others can mean an erroneous behavior in the setup. Obviously, a behavior specification could help analyze the problem.

To identify at which level and how behavior is to be specified it is advantageous to recall the key two approaches to component employment:

(a) In top-down design/refinement from scratch, the required functionality (like weather forecast) is first viewed by the environment as a black box component, *frame* in our terminology (Fig.2a). Later, by refinement, the component's *architecture* is elaborated (Fig.2b), and, eventually, setup is finalized. In [7] we showed that behavior protocols can be specified for both frame and architecture and their compliance verified by a tool.

(b) When reuse is considered, a potentially useful framework is identified and its parameters determined. The resulting architecture is than placed into the required frame, which finally yields a setup. An intuition-based bottom line is that (i) an architecture can serve in different frames (be

part of different setups), (ii) the choice of actual parameters in a framework can cause communication errors in the resulting architecture. In [8] we showed that composition errors in an architecture can be statically detected assuming the behavior of its subcomponents is specified by means of behavior protocols.

1.4. Goal and Structure of the Paper

The message of Sect. 1.2 and 1.3 is twofold:
- it is beneficial to specify behavior,
- even though there are communication errors in an architecture, they can be avoided in a particular environment (abstracted as a frame).

From this perspective, the problem this paper focuses on reads: An architecture can contain composition errors (e.g., as the result of a choice of a framework's parameters). Whether a composition error occurs in a run of this architecture depends on the way the architecture is used in its environment, i.e. erroneous architecture is a relative concept. The key issue is how to statically verify that, for a given setup containing the architecture, no composition errors can occur in any run. The goal is to show that this is possible by employing behavior protocols and their consent operator in particular. The rest of the paper is structured as follows: Section 2 overviews the key concepts of behavior protocols, while Sect.3 presents the key contribution - shows how to check whether an architecture contains composition errors in a model environment. Related work is discussed in Sect.4; Sect.5 provides a conclusion and draws a picture of future work.

2. Behavior Protocols

2.1. Basics

To analyze the behavior of components and frameworks, a formal specification language has to be employed. In this paper, we use *behavior protocols* [7, 8, 9], which were developed as a part of our SOFA component model [5]. As discussed in [7], behavior protocols reminding regular expressions are more readable than a process algebra's notation, and their expressive power is strong enough to reasonably approximate behavior of components. In contrast to CCS [10], they always lead to finite state spaces and the compliance/equivalence relations are decidable.

A key concept behind the behavior protocols is an *event* – an abstraction of issuing a method call, a response to a call, or a general event not associated with a particular method. For instance, a call of `compute` on `ENGINE:E` is captured as `ENGINE:E.compute↑`, a response to the call as `ENGINE:E.compute↓`. Every event is *emitted* by a component and *accepted* by another component. Calling `compute` via `SWITCH:E` is seen as emitting `SWITCH:E.compute↑` by `SWITCH` (denoted as `!SWITCH:E.compute↑` from the perspective of `SWITCH`); at the same time `ENGINE:E.compute↑` is accepted by `ENGINE` (denoted as `?ENGINE:E.compute↑`). Above `!SWITCH:E.compute↑` and `?ENGINE:E.compute↑` are examples of *event tokens*.

Event tokens are primitive operands of a *behavior protocol* - an expression which specifies complex behavior as the set of desired sequences of events (traces). The operators employed in behavior protocols are: ";" which means *sequencing* of the traces described by the operands; "+" (*alternative*) stands for alternative choice of the traces, "*" denotes finite *repetition*, and "|" means *parallel interleaving* of the traces.

Let us illustrate the semantics of behavior protocols on the behavior specification of the `SWITCH`. It repeatedly accepts calls to the `query` method on both of its provides interfaces (`SWITCH:F1`, `SWITCH:F2`) in parallel. On every such call `SWITCH` reacts by calling `get` on `SWITCH:C` to find out whether the requested forecast is available in `CACHE`; depending on the finding, `SWITCH` either responds immediately to `query`, or calls `compute` on `SWITCH:E`. After `SWITCH` accepts a response from `compute` on `SWITCH:E`, it calls `put` on `SWITCH:C` to store the result into `CACHE`, and finally responds on `SWITCH:F1` or `SWITCH:F2`.

The behavior of `SWITCH` is specified by the following protocol:

```
Prot_SWITCH =
(
  ?SWITCH:F1.query {
    !SWITCH:C.get ;
    ( NULL +
      (!SWITCH:E.compute ; !SWITCH:C.put)
    )
  }
)* |
(
  ?SWITCH:F2.query {
    !SWITCH:C.get ;
    ( NULL +
      (!SWITCH:E.compute ; !SWITCH:C.put)
    )
  }
)*
```

Here, `!SWITCH:C.get` is an example of (a few) predefined useful abbreviations; it stands for `!SWITCH:C.get↑ ; ?SWITCH:C.get↓`. The abbreviation `?SWITCH:F1.query { !SWITCH:C.get ; ( ... ) }` stands for `?SWITCH:F1.query↑; !SWITCH:C.get ; ( ... ) ; !SWITCH:F1.query↓` – it describes acceptance of a call to the `SWITCH:F1.query` method, including the reactions specified by the protocol in braces, i.e. `!SWITCH:C.get ; ( ... )`. Furthermore, `NULL` stands for "empty" behavior (no event).

As $Prot_{SWITCH}$ specifies the events on the interfaces forming only the frame of `SWITCH`, we call $Prot_{SWITCH}$ the *frame protocol* of `SWITCH`. The frame protocol of `CACHE` is much more simple:

```
Prot_CACHE =
  (?CACHE:C.get + ?CACHE:C.put)* |
  (?CACHE:C.get + ?CACHE:C.put)*
```

In $Prot_{CACHE}$, we used the abbreviation `?CACHE:C.get`, standing for `?CACHE:C.get↑ ; !CACHE:C.get↓`. Finally, we introduce the frame protocols for reentrant and non-reentrant variants of `ENGINE`:

```
Prot_ENGINE-RE =
  ?ENGINE:E.compute* |
  ?ENGINE:E.compute*

Prot_ENGINE-NR =
  ?ENGINE:E.compute*
```

2.2. Group Protocol and Consent Operator

By a behavior protocol, we can describe not only behavior of a single component, but also behavior of a group of components – we talk about a *group protocol*.

Besides event tokens seen in Sect. 2.1 , we use τe for *internal events* – events occurring on a binding in a group `G` (here, the qualification `e` starts with the binding name $<I_R\text{-}I_P>$, where I_R, I_P are the requires and the provides interface, which are bound) – and *error tokens* to denote *composition errors*. As they are described in [8] and [11] in detail, we provide a brief overview only: A *bad activity* (denotation εe) occurs when a component `C` emits an event on its requires interface I_R, which is bound to a provides interface I_P of another component `D`, and `D` is not able to absorb the event on I_P (demanded by its frame protocol). *No activity* (denotation $\varepsilon\varnothing$) means that the components run into a deadlock - no component in the group can emit an event, but there is a component which has not reached its final state. *Divergence* (denotation $\varepsilon\infty$) denotes a situation when the communication of the components in the group never stops. Finally, *unbound requires error* (denotation $\varepsilon\not\rightarrow e$) occurs when a component C emits an event `e` on its requires interface, which is unbound.

As the behavior of every component in a group is specified separately by its frame protocol, we will advantageously construct the group protocol from these frame protocols via a repeated application of the *consent* operator [8]. Technically, the consent operator (denoted as ∇) composes the group protocols $Prot_{G1}$, $Prot_{G2}$ of two (disjoint) component subgroups `G1`, `G2` into the group protocol `Prot` of the group `G` composed of `G1` and `G2`. In addition to $Prot_{G1}$, $Prot_{G2}$, consent also takes the set `S` of all the events on the ties between the groups `G1`, `G2` (*set of synchronizing events*) as a parameter: `Prot` = $Prot_{G1}$ ∇_S $Prot_{G2}$. As a frame protocol can be viewed as a group protocol of a group consisting of a single component, the group protocol of a non-trivial group `G` can be constructed from frame protocols of the components in G step by step by a repeated application of ∇.

Informally, the semantics of the consent operator can be described by the following three rules: (1) The behavior of `G1` and `G2` is synchronized on the events from `S`; (2) the sequences of events, which are not in `S`, are arbitrarily interleaved; (3) Composition errors are identified. The formal definition of ∇ can be found in the [8].

To show how ∇ works, we provide the behavior of the `CACHE-SWITCH` group:

```
Prot_CACHE-SWITCH = Prot_CACHE ∇_S1 Prot_SWITCH =
  (
    ?SWITCH:F1.query {
      τ<SWITCH:C-CACHE:C>.get ;
      ( NULL +
        (!SWITCH:E.compute ;
         τ<SWITCH:C-CACHE:C>.put)
      )
    }
  )* |
  (
    ?SWITCH:F2.query {
      τ<SWITCH:C-CACHE:C>.get ;
      ( NULL +
        (!SWITCH:E.compute ;
         τ<SWITCH:C-CACHE:C>.put)
      )
    }
  )*
```

Here, the abbreviation `τ<SWITCH:C-CACHE:C>.put` stands for `τ<SWITCH:C-CACHE:C>.put↑ ; τ<SWITCH:C-CACHE:C>.put↓` and the set of synchronizing events is

```
S1 = {
  <SWITCH:C-CACHE:C>.get↑
  <SWITCH:C-CACHE:C>.get↓
  <SWITCH:C-CACHE:C>.put↑
  <SWITCH:C-CACHE:C>.put↓
}.
```

To demonstrate how consent identifies composition errors, consider the composition

$Prot_{CACHE\text{-}SWITCH\text{-}ENR} = Prot_{CACHE\text{-}SWITCH}\ \nabla_{S2}\ Prot_{ENGINE\text{-}NR}$

Here, $Prot_{ENGINE\text{-}NR}$ specifies the behavior of the non-reentrant `ENGINE` and `S2` is the set of synchronizing events on the `<SWITCH:E-ENGINE:E>` binding.

As $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ is relatively complex (however the developer does not have to write it by hand, since it is automatically generated), we show just one of the *erroneous traces* described by $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$, (i.e. a trace which ends by a composition error):

```
?SWITCH:F1.query↑ ;
τ<SWITCH:C-CACHE:C>.get↑ ;
τ<SWITCH:C-CACHE:C>.get↓ ;
τ<SWITCH:E-ENGINE:E>.compute↑ ;
?SWITCH:F2.query↑ ;
τ<SWITCH:C-CACHE:C>.get↑ ;
τ<SWITCH:C-CACHE:C>.get↓ ;
ε<SWITCH:E-ENGINE:E>.compute↑
```

This trace corresponds to the following behavior: `query` on `SWITCH:F1` is called, `SWITCH` reacts by a call to `SWITCH:C.get` (which is immediately returned) and by a call to `SWITCH:E.compute`. Then, a call to `query` on `SWITCH:F2` is accepted, what results in another call of

SWITCH:C.get and a trial to call SWITCH:E.compute. However, this results in a bad activity error (ε<SWITCH:E-ENGINE:E>.compute↑), as the non-reentrant ENGINE is not able to accept a request before it responded the previous one.

All the erroneous traces described by $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ are caused by parallel access to ENGINE:E, resulting from simultaneous access to SWITCH:F1 and SWITCH:F2.

3. Checking Composition Errors in a Given Environment

The architecture protocol $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ presented above demonstrates an important fact: although the architecture of FORECAST described by $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ contains composition errors, those errors occur only in special cases - specifically if two threads call query on FORECAST:F1 and FORECAST:F2 in parallel. If we limited the way the methods on the frame of FORECAST are called accordingly - by a frame protocol - the composition errors would not occur. To allow query be called only sequentially, we define the frame protocol of FORECAST as follows:

$Prot_{FORECAST}$ =
(?FORECAST:F1.query + ?FORECAST:F2.query)*

To check whether the architecture contains composition errors when used in the frame with the frame protocol $Prot_{FORECAST}$, we use the following technique:

(1) We *invert* $Prot_{FORECAST}$, simply by replacing all "?" in the protocol by "!" and vice versa. This way, we obtain an *inverted frame protocol* $Prot_{FORECAST}^{-1}$, specifying behavior of a *model environment* of FORECAST – i.e. behavior of a hypothetical component, which, bound to all the interfaces of FORECAST, behaves exactly how it anticipates.

(2) We compose $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ with $Prot_{FORECAST}^{-1}$ to see how the architecture behaves in the model environment, i.e. how it behaves when forming a setup with the FORECAST frame:

$Prot_{FORECAST}^{-1}$ =
(!FORECAST:F1.query + !FORECAST:F2.query)*

$Prot = Prot_{FORECAST}^{-1} \nabla_{S3} Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ =

```
(
  τ<FORECAST:F1-SWITCH:F1>.query {
    τ<SWITCH:C-CACHE:C>.get ;
    ( NULL +
      (τ<SWITCH:E;ENGINE:E>.compute ;
       τ<SWITCH:C-CACHE:C>.put)
    )
  } +
  τ<FORECAST:F2-SWITCH:F2>.query {
    τ<SWITCH:C-CACHE:C>.get ;
    ( NULL +
      (τ<SWITCH:E-ENGINE:E>.compute ;
       τ<SWITCH:C-CACHE:C>.put)
     )
  }
)*
```

```
S3 = {
  <FORECAST:F1-SWITCH:F1>.query↑
  <FORECAST:F1-SWITCH:F1>.query↓
  <FORECAST:F2-SWITCH:F2>.query↑
  <FORECAST:F2-SWITCH:F2>.query↓
}.
```

We just remark that the abbreviation τ<FORECAST:F1-SWITCH:F1>.query{τ<SWITCH:C-CACHE:C>.get ;(...)} stands for τ<FORECAST:F1-SWITCH:F1>.query↑; τ<SWITCH:C-CACHE:C>.get ; (...) ; !τ<FORECAST:F1-SWITCH:F1>.query↓.

In the resulting protocol Prot, there are no composition errors; this complies with what we intuitively expected: limiting the usage of FORECAST by a frame protocol allowing only sequential calls of query would eliminate composition errors. Obviously, the steps (1) and (2) give us a general method to check whether a given architecture is erroneous in the context of a given frame.

4. Related work

Probably the closest to our work is [12] where the problem of identifying a component's behavior errors in all potential environments is addressed. The approach is based on an extension of classical model checking: Checking for a given property of a component yields one of the three following results: (i) the property is preserved in all environments; (ii) the property is violated in all environments; (iii) all the environments in which the property is satisfied are furnished.

In [13], the authors focus on testing interface compatibility. Via interface automata, they check whether there exists an environment in which a given interface (module) works correctly. In addition, they check the errors caused by such method call chains which commence in the component under consideration (e.g., recursive call of a non-reentrant method). Note that, in our approach, the origin (environment or component) is not important for error identification. Contrary to both [12] and [13], choosing a pragmatic view important in practice, we check whether a given component behaves properly in a specific environment; on the other hand it might be interesting to investigate the existence of a "reasonable" environment.

The observation that the behavior of a component depends on the way it interacts with its environment (and vice versa) has been targeted by a number of researchers at different levels of granularity, ranging from dynamically modifiable Usage Policy for a single CORBA object [14], over mandatory calls [15] and the Alloy framework [16] considering cooperation among multiple plugins. To our knowledge, none of them comes up with the idea to relativize the fact that a software architecture contains communication errors. The authors of "predictable assembly" [17] envision a framework for reasoning on assembling of components featuring with specific properties. For behavior specification they consider CSP [18] as an example.

5. Conclusion and future work

We have relativized the fact that a software architecture containing communication errors is erroneous, by showing that for a particular environment, the internal communication errors can be avoided. This observation is important for component architecture reuse. The key instruments allowing to articulate the idea precisely are: (i) The separation of the "frame" and architecture" abstractions allowing the trick with inverted frame to represent a model environment ; (ii) The concept of behavior protocols and, in particular, their composition errors (we introduced in [1]) which capture possible erroneous behavior of cooperating components. Unlike typical process algebras (e.g. CCS [10], CSP [18]), composition errors reflect the inherent asymmetry of a procedure call (caller takes the initiative, while callee is passive). Currently, we are about to finish a new version of protocol checker in our SOFA model and also working on enhancing the Fractal ADL by behavior protocols and making the checker available in Fractal. In the near future we consider identifying "the largest" frame (or "the best environment"), such that all the communication errors in a given architecture would be avoided in it.

6. References

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

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

[3] E. Bruneton, T. Coupaye, J.B. Stefani, Recursive and Dynamic Software Composition with Sharing, *Proc. of Seventh International Workshop on Component-Oriented Programming (WCOP02)*, at ECOOP 2002, Malaga, Spain

[4] OMG Corba Component Model Speci.cation, http://www.omg.org/technology/documents/formal/components.htm

[5] The SOFA project, http://sofa.forge.objectweb.org/

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

[7] F. Plasil, S. Visnovsky, Behavior Protocols for Software Components, *IEEE Transactions on Software Engineering, 28*(11), 2002

[8] J. Adamek, F. Plasil, Component Composition Errors and Update Atomicity: Static Analysis, Accepted for publication in the *Journal of Software Maintenance and Evolution: Research and Practice*, 2004 (also preliminary version available at http://nenya.ms.mff.cuni.cz)

[9] J. Adamek, Static Analysis of Component Systems Using Behavior Protocols, *OOPSLA 2003 Companion*, Anaheim, CA, USA, 2003

[10] R. Milner, *A Calculus of Communicating Systems* (LNCS, Springer-Verlag., 1992)

[11] J. Adamek, F. Plasil, Static Checking for Missing Bindings of Components, Tech. Report No. 2004/3, Dep. of SW Engineering, Charles University, Prague, Mar 2004, http://nenya.ms.mff.cuni.cz

[12] D. Giannakopoulou, C. S. Pasareanu, H. Barringer, Assumption Generation for Software Component Verification, *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, 2002

[13] L. Alfaro, T. A. Henzinger, Interface Automata, *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, ACM Press, 2001, pp. 109-120

[14] W. DePrince Jr., C. Hofmeister, Enforcing a lips Usage Policy for CORBA components, *Porceedings of EUROMICRO'03*, IEEE CS Press, 2003

[15] M. Barnett 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

[16] R. Chatley, S. Eisenbach, J. Magee, Modeling a Framework for Plugins, *Proceedings of the SAVCBS 2003 Workshop*, Helsinki, Finland (http://www.cs.iastate.edu/SAVCBS/)

[17] K. C. Wallnau, Volume III: A Technology for Predictable Assembly from Certifiable Components, CMU/SEI-2003-TR-009

[18] A. W. Roscoe, *The Theory and Practice of Concurrency* (Prentice-Hall, 1998)

Chapter 7

Addressing Unbounded Parallelism in Verification of Software Components

This chapter contains the paper

[Ada06] Adamek, J.: Addressing Unbounded Parallelism in Verification of Software Components, Accepted for publication in proceedings of the Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2006), Las Vegas, Nevada, USA, to be published by IEEE Computer Society, Jun 2006

Summary. In this paper we show how to specify unbounded parallelism via finite state behavior specification. The method is based on the following observation: at the time a reentrant component C is designed unbounded parallelism can be specified only by an infinite state specification. On the other hand, at the time C is instantiated as a part of a composed component (component architecture) A, the behavior of C is limited by the activity of other components in A and it can be therefore often specified via a finite state specification. Therefore, we propose the following scenario: at a design time, the behavior of C is specified using so called *behavior template*. When C is used as a part of A, the behavior template is transformed into common (finite state) frame protocol, that can be used for behavior composition. The transformation reflects the behavior of the other components in A. The behavior templates are based on behavior protocols.

There are two difficulties with regard to the transformation of a behavior template into the frame protocol. First, the behavior of any non-reentrant component in A may not be specified by an arbitrary frame protocol, but only by so called *well-formed* frame protocol (some behavior protocol constructs are forbidden). Also the behavior templates themselves are based on those well-formed protocols. The second difficulty relates to the component architecture A: bindings in A constitute dependencies among the transformations of behavior templates associated with the reentrant components in A; the order of the transformations must respect the dependencies. The dependencies are formalized via *dependence graph*. We show how to perform the transformations (i.e. to generate finite state frame protocols) for any architecture with acyclic dependence graph and also for certain architectures with cyclic dependence graphs (using a technique called *substitute template construction*).

Comments. In the paper, there is an example of a component application employing so called *multiple bindings* – more than one required interface is bound to a single provided interface. Multiple bindings are not allowed in the SOFA component model (but they are allowed in other component models, e.g. Fractal [11]). We use them in order to present a realistic example while not using SOFA connectors (which would be unnecessarily complicated).

To be able to perform behavior composition for component architectures with multiple bindings, it is necessary to transform the frame protocols before the composition (this transformation is different from the transformation of behavior templates). The issue is out of scope of [Ada06] and is presented in detail in [1].

Addressing Unbounded Parallelism in Verification of Software Components*

Jiri Adamek[1,2]

[1]Charles University in Prague, Faculty of Mathematics and Physics
Department of Software Engineering
[2]Academy of Sciences of the Czech Republic
Institute of Computer Science
adamek@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz/~adamek

Abstract

To use verification tools for reliability analysis of a software component, it is desirable to specify the behavior of the component by a finite-state model. This is often impossible at design time if the component practices unbounded parallelism. In that case, the behavior of the component widely depends on the environment the component is instantiated in. Unfortunately, covering all possible environments results in an infinite-state model.

In this paper, we introduce a solution based on the concept of template-to-model transformation: at design time, a developer describes the behavior of the component by a behavior template, which is automatically transformed into a concrete behavior model when the component is instantiated in an environment. As the concrete behavior model is finite-state, it is a suitable input for verification tools.

Keywords: *Software Components, Formal Verification, Unbounded Parallelism, Behavior Protocols*

1. Introduction and Motivation

1.1. Modeling Unbounded Parallelism in Component Systems

Finite-state models are often used in the area of software component formal verification to specify behavior. This is especially true for model checking [7] and related approaches [11,5,14]. The restriction to finite(-state) models has both advantages and disadvantages: the main disadvantage is the fact that not every behavior of a real system can be modeled this way. On the other hand, the verification problems are decidable for finite models, which is not always the case of the infinite ones.

One of the sources of infinity here is *unbounded parallelism*: situation, when a system performs several processes in parallel (where each of those processes in itself is finite-state) and the number of those processes is arbitrary. Although in every concrete run of the system the number of the processes (*level of parallelism*, LOP) is finite, this number typically depends on the inputs from the environment. If there exists no upper bound to LOP, it must be considered to be potentially infinite, causing that the model of the system is also infinite.

In software component systems [13, 6, 10], unbounded parallelism is present very often and has one of the two following forms: (1) LOP practiced by a component depends on the component configuration, the processes are initiated by the component itself (*active unbounded parallelism*). (2) A component, which provides a service, has to process parallel requests from other components; LOP depends on the number of the requests coming in parallel. It is determined by the number and the behavior of the components emitting the requests (*passive unbounded parallelism*). In both cases, the factors determining LOP (component configuration, number and behavior of other components) are typically unknown at the time the component is designed.

The unbounded parallelism in component systems does not cause implementation problems - well known programming techniques as synchronization, request queues, or thread pooling work fine in this situation. The problem occurs when one wants to model such a component: setting down a concrete maximum LOP (in the model) is not suitable because the implementation of the component is

*This work was partially supported by the Czech Academy of Sciences project 1ET400300504.

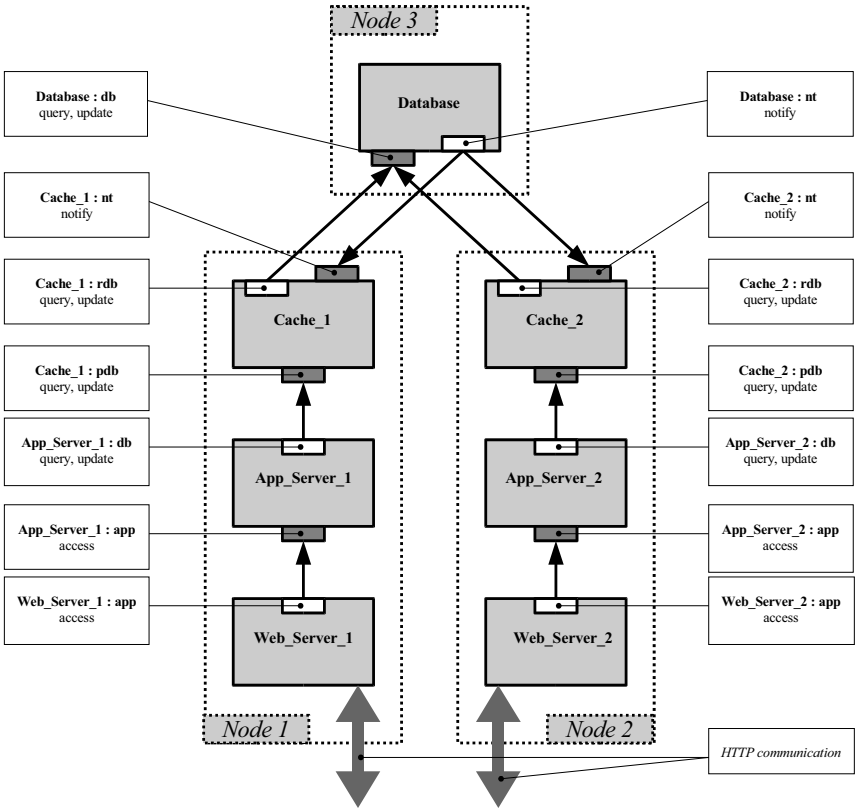


Figure 1: Example of a component application in SOFA

able to handle an arbitrary LOP - model would differ from the implementation in an essential way. On the other hand, describing the unbounded parallelism precisely would result in an infinite state model, which is hard (if not impossible) to verify in an automatized way.

We introduce a compromise solution in this paper: *template-to-model transformation*. It is based on the following fact: at the time the component is designed the LOP is unknown. However, later, when the architecture of a concrete system is designed by instantiating the components and connections among the component interfaces, LOP of each component is typically known. Therefore, we propose to create a *behavior template* at the component-design time (for each component practicing unbounded parallelism) and later, at the architecture-design time, to transform this behavior template into a *concrete behavior model*, depending on the LOP of the component. The concrete behavior model is finite-state and specifies parallelism on the level which is needed in the concrete component architecture. Such a model is suitable as an input of automatized verification tools. Therefore, several methods can be used to analyze the behavior of the architecture, including compliance checking [11], detection of composition errors [1], or general model checking [7].

1.2. Goal and Structure of the Paper

The goal of the paper is to propose a technique of transforming a behavior template of a software component into a concrete finite-state behavior model according to LOP the component practices in the given architecture. Both the behavior template and the concrete behavior model are based on behavior protocols [11] and the whole concept is demonstrated using the SOFA component model [13].

The structure of the paper is as follows: in Sec. 2, we introduce SOFA and Behavior Protocols. In Sec. 3 we define important concepts of maximum incoming and outgoing parallelism. They are utilized in Sec. 4, where we define behavior templates and show how they are transformed into concrete behavior models. In Sec. 5 we evaluate the approach and present the related work, while in Sec. 6 we conclude by future work.

2. SOFA and Behavior Protocols

2.1. SOFA Component Model

In SOFA [13], a component employs *provided* and *required* interfaces. A provided interface (denoted by a small dark box on Fig. 1) is very similar to a common Java interface - it lists the methods which may be called by other components. A required interface (denoted by a small white box) lists methods which are called from the component implementation; they have to be provided by another component. On Fig. 1, every interface has associated a caption showing the interface name (consisting of the global component name and the local name of the interface within the component) and the list of method names. For example, the `App_Server_1` component has one provided interface - `app` (the `access` method) and one required interface - `db` (the `query` and `update` methods).

If a component C has a required interface R, R has to be associated with a provided interface P of another component containing the same methods as listed in R in order to C can work correctly. Such an association is realized by a chain of *interface ties* (arrows on Fig. 1). All calls of the methods of R from C's implementation are then delegated to P.

A direct interface tie between a required interface and an appropriate provided interface is called a *binding*. All interface ties on Fig. 1 are bindings. As SOFA allows to define *nested components* [13], there exist two more kinds of interface ties: *delegation* (from a provided interface of a component C to a provided interface of a *subcomponent* of C) and *subsumption* (from a required interface of a component C to a required interface of the C's *supercomponent*).

Finally, the *frame* of a component C is defined as the description of all C's interfaces (the black-box view of C) and the *architecture* of C as the description of C's internals on the first level of nesting - i.e. frames of C's direct subcomponents and all the appropriate ties. For example, frame of `App_Server_1` consists of the `app` and `db` interfaces. All the components and bindings on Fig. 1 form the top-level architecture (so called *system architecture*). Frames and architectures are defined in ADL-like Component Definition Language (CDL, [13]).

2.2. Running Example

As a running example, we use a simplified information system, providing a web-based user-interface. The three basic building blocks of the system are a database, an application server providing the application and presentation functionality, and a web server communicating with clients (HTML browsers) using the HTTP protocol.

For load balancing, both the application server and the web server are duplicated and deployed on two distinct nodes (Node 1, Node 2), resulting in four components `App_Server_1`, `App_Server_2` and `Web_Server_1`, `Web_Server_2`, while the `Database` component is deployed on its own node (Node 3). This setting is illustrated on Fig. 1. To minimize the inter-node communication, which is time-expensive, the data read from Database are stored in caches (`Cache_1`, `Cache_2`). A cache is fully transparent, i.e. it provides the same interface as `Database`. Although the deployment of the components on the nodes is not a part of the SOFA component model, we provide this information to make certain design decisions clear (duplicity of the servers, presence of the caches).

To keep the example simple, the application and web servers (as well as the caches) are deployed on two nodes only. However, we suppose that the information system works properly in configuration with application servers, web servers, and caches deployed on an arbitrary number of nodes. Therefore, it is necessary that also the behavior templates of those components (presented in Sec. 4.1) are suitable for all those configurations.

The caches are optimized for reading. Data read from the database (the `query` method) are stored in a cache, so that subsequent reads can be processed without communication with the database. All the data written into the cache (`update`) are immediately written into the database. In this case, all other caches are notified that an item in the database changed (`notify`); the caches then invalidate the item, so that next time the item is requested by an application server, it is read directly from the database (and stored in the cache).

The requests from the clients come in parallel. This issue is addressed by the components in different ways. A web-server uses a request-queue and a thread-pool. The number of threads in the pool is defined in the web-server configuration. The application servers are stateless and reentrant: they accept arbitrary number of parallel calls (and generate corresponding number of parallel calls to `Database`). Finally, `Database` is statefull and reentrant: it uses advanced locking strategies to manage parallel queries targeting the same data.

2.3. Behavior Protocols

In [11] the behavior (of a component or a component architecture) is modeled as a language over a finite alphabet (a set of *traces*). A trace is a finite sequence $s = \langle s_1 \ldots s_n \rangle$ of *event tokens* denoting the events: *requests* (denoted by m↑ for a *method name* m) and *responses* (m↓). An event can be either *emitted* (denoted as !m↑ or !m↓) or *absorbed* (?m↑ or ?m↓) by a component. A request and a corresponding

response (for the same method name) form a *method call*. For example, if `Web_Server_1` from Fig. 1 calls the `access` method on its `app` interface, it is described by the trace `<!app.access↑, ?app.access↓>`, if `App_Server_1` accepts a call of `access` on its `app` interface, the appropriate trace is `<?app.access↑, !app.access↓>` - i.e. the request is emitted by the caller, while the response is emitted by the callee.

A *behavior protocol* Prot is a regular-like expression specifying a set of traces (the *behavior*, denoted as L(Prot)). It consists of event tokens, regular operators of *concatenation* (;), *alternative* (+) and *finite repetition* (*) as well as the advanced operator of *parallel execution* (|) and the *abbreviations* denoting emission of a method call (!m = !m↑ ; ?m↓), acceptance of a method call (?m = ?m↑ ; !m↓) or the emission/acceptance of a method call where the method code behaves according to a protocol P: !m{P} = !m↑ ; P ; ?m↓ and ?m{P} = ?m↑ ; P ; !m↓. If a protocol employs only the abbreviations and not the event tokens explicitly denoting the requests and responses, it is called a *well-formed protocol*. NULL stands for an empty protocol (specifying no behavior). Finally, if P is a protocol and n is a positive integer, parallel(P, n) has the same meaning as P | P | ... | P, where P appears n-times.

For example, let the behavior of `Database` from Fig. 1 is specified by the following protocol:

```
Prot_DB = parallel(
    (?db.query + ?db.update{ !nt.notify })*,
    5
)
```

According to `Prot_DB`, `Database` accepts on its `db` interface calls of `query` or `update`. If `update` is called, `Database` calls `notify` on its `nt` interface. `Database` accepts an arbitrary number of the calls. At every moment at most 5 parallel calls on `db` may proceed. `Prot_DB` is well-formed.

`Prot_DB` is an example of a *frame protocol*, as it specifies the events occurring on a frame of a component. An *architecture protocol* (specifying events in a whole architecture) can be obtained by automatic composition of frame protocols of the components in the architecture [11, 1].

As stated in [11], the behavior L(Prot) specified by a protocol Prot is a regular language (i.e. L(Prot) is accepted by a finite-state automaton). Therefore, any behavior protocol (and in particular any frame protocol) specifies a finite-state model.

3. Maximum Incoming and Outgoing Parallelism

The goal of this section is to introduce the concepts of *maximum incoming parallelism* (MIP) and *maximum outgoing parallelism* (MOP). They are employed in the behavior templates, introduced in Sec. 4.1, in order to address the passive unbounded parallelism. MIP (MOP) is the maximum possible number of the calls performed on a given provided (required) interface in parallel. We show how MIP (MOP) can be computed.

3.1. Current and Maximum Outgoing Parallelism

Let Prot be the frame protocol of a component C featuring a required interface R. Let t be a trace specified by Prot (i.e. $t \in L(Prot)$) and let $s = <s_1 \dots s_n>$ be a prefix of t. At the moment the events from s are processed (but before any other event comes) the *current outgoing parallelism* (COP) on R equals to the number of non-responded requests on R, i.e.:

$$COP(R, s) = w_R(s_1) + w_R(s_2) + \dots + w_R(s_n),$$

where $w_R(s_i)$ – the *weight* of the event token s_i with respect to R – is defined in the following way:

$w_R(s_i) = 1$ when s_i is a request on R
$w_R(s_i) = -1$ if s_i is a response on R
$w_R(s_i) = 0$ otherwise, i.e. when s_i denotes an event on an interface different from R

The maximum outgoing parallelism (MOP) on R equals to the maximum possible number of calls which can be performed on R in parallel. It depends on the behavior of C (specified by Prot) and is defined as

$$MOP(R, Prot) = \max \{COP(R, s) : s \text{ is a prefix of } t \text{ and } t \in L(Prot)\}.$$

As an example, let us consider a hypothetical frame protocol `Prot_WS1` of the `Web_Server_1` component from Fig. 1: `Prot_WS1 = parallel(!app.access*, 3)`. Here, MOP(`app`, `Prot_WS1`) = 3.

In [3] a proof can be found that if one confines to the well-formed protocols (defined in Sec. 2.3), MOP is always finite, positive, and can be easily computed from the frame protocol (in the time which is linear in the length of the protocol). The algorithm for MOP computation parses the protocol and applies the equations (1)-(11). Each equation corresponds to one node type in a well-formed protocol parse-tree, the MOP-values on the right side of every

equation are computed recursively (P stands for a provided interface, R, S stand for different required interfaces):

$$MOP(R, !R.x) = 1 \quad (1)$$
$$MOP(R, !S.y) = 0 \quad (2)$$
$$MOP(R, ?P.z) = 0 \quad (3)$$
$$MOP(R, !R.x\{Prot_1\}) = 1 + MOP(R, Prot_1) \quad (4)$$
$$MOP(R, !S.y\{Prot_1\}) = MOP(R, Prot_1) \quad (5)$$
$$MOP(R, ?P.z\{Prot_1\}) = MOP(R, Prot_1) \quad (6)$$

$$MOP(R, Prot_1 + Prot_2) = \\ = \max(MOP(R, Prot_1), MOP(R, Prot_2)) \quad (7)$$
$$MOP(R, Prot_1 \mid Prot_2) = \\ = MOP(R, Prot_1) + MOP(R, Prot_2) \quad (8)$$

$$MOP(R, Prot_1 \,||\, Prot_2) = \\ = MOP(R, Prot_1) + MOP(R, Prot_2) \quad (9)$$

$$MOP(R, Prot_1 ; Prot_2) = \\ = \max(MOP(R, Prot_1), MOP(R, Prot_2)) \quad (10)$$

$$MOP(R, Prot_1^*) = MOP(R, Prot_1) \quad (11)$$

The or-parallel operator (||) is defined in [11]. In the rest of this paper, we assume that all frame protocols are well-formed. This is a reasonable assumption, as a protocol which cannot be (equivalently) rewritten as a well-formed one specifies a non-realistic behavior (the number of requests may be different from the number of responses and the responses may even precede the requests for non-well-formed protocols). In addition, the behavior template transformations presented in Sec. 4.1 take advantage of the algorithm for MOP computation based on equations (1)-(11) that can be applied only to well-formed protocols.

3.2. Maximum Incoming Parallelism

Maximum incoming parallelism (MIP) on a provided interface P is the maximum possible number of calls performed on P in parallel. Let $R_1, ..., R_n$ be all the required interfaces bound to P in a component architecture A, let C_i be the component employing R_i, and let $Prot_i$ be the frame protocol specifying behavior of C_i (for $i <= n$). As the calls emitted by different components may be arbitrarily interleaved, the maximum incoming parallelism on P is defined by the following equation:

$$MIP(P) = \\ MOP(R_1, Prot_1) + MOP(R_2, Prot_2) + ... + MOP(R_n, Prot_n).$$

As well as in the case of MOP, if only well-formed protocols are used to specify behavior, MIP is finite and positive (it follows from the MIP definition and the properties of MOP).

As an example, MIP on the `Database:db` interface from Fig. 1 is:

$$MIP(\texttt{Database:db}) = \\ MOP(\texttt{nt}, \texttt{Prot\_CA1}) + MOP(\texttt{nt}, \texttt{Prot\_CA2}),$$

where `Prot_CA1`, `Prot_CA2` are the frame protocols specifying behavior of `Cache_1`, `Cache_2`, respectively.

4. Template-to-Model Transformation

4.1. Behavior Templates

Let C be a component. If C practices parallelism on the level which is not known at design time, but known at the time C is instantiated in an architecture, the behavior of C is specified using a behavior template. Otherwise, behavior of C is specified using a (well-formed) frame protocol.

A behavior template is very similar to a well-formed frame protocol: the only difference is that a template may contain symbolic values as parameters of the `parallel` construct. When C is instantiated, those symbolic values are replaced by concrete integers and the behavior template becomes a concrete frame protocol. As explicit requests and responses cannot be used in the behavior templates, the generated frame protocol is well-formed. There are two kinds of symbolic values: the *component property* and the *maximum incoming parallelism*.

A component property is used as a symbolic value when C practices active unbounded parallelism, i.e. the LOP of C depends on C's configuration. Recalling the running example, it is the case of web servers. Here, LOP equals to the size of a web server's thread pool. The behavior template of the `Web_Server_1` and `Web_Server_2` components is as follows:

```
Temp_WS =
  parallel(
    !app.access*,
    property threadPoolSize
  )
```

The symbolic value consists of the keyword `property` (defining the kind of the symbolic value) and the property identifier (`threadPoolSize`). Definition of the concrete value of the property (a positive integer) is a part of the architecture description (system architecture description in this case). The values of the property for `Web_Server_1` and `Web_Server_2` may differ.

The second kind of symbolic values is used by a developer when a component C practices passive unbounded

parallelism. LOP of C depends on the maximum number of the requests coming through a provided interface P of C in parallel - i.e. the maximum incoming parallelism (MIP) defined in Sec. 3.2. Therefore, this kind of symbolic value is also called maximum incoming parallelism. In the running example, it is e.g. the case of the application servers. Every application server has to be able to process as many parallel calls as it comes from the appropriate web server through the access interface. Therefore, the behavior template of the `App_Server_1` and `App_Server_2` components is:

```
Temp_AS =
  parallel(
    ?app.access{ (!db.query + db.update)* }*,
    mip app
  )
```

Here, the symbolic value consists of the keyword (`mip`) and the identifier of the provided interface (`app`).

MIP is also used in the behavior template of `Database` (`Temp_DB`) and caches (`Temp_CA`):

```
Temp_DB =
  parallel(
    (?db.query + ?db.update{ !nt.notify })*,
    mip db
  )

Temp_CA =
  parallel(
    (
      ?pdb.query{NULL + !rdb.query} +
      ?pdb.update{!rdb.update}
    )*,
    mip pdb
  ) | parallel(?nt.notify*, mip nt)
```

Let us emphasize the fact that the presented behavior templates specify behavior of the components independently on how many nodes are used to host the caches, the application servers and the web servers (i.e. the templates would be suitable even in a different configuration than the one on Fig. 1).

4.2. Behavior Template Transformation - Basics

To do transformation of a behavior template into the concrete behavior model (a frame protocol), it is necessary to replace all the symbolic values in the template by concrete integers. In the case of a component property, it is simple - the concrete value is a part of the architecture description (i.e. defined by the designer). Such a transformation is called a *property-guided transformation*.

If MIP is utilized, the value of MIP is computed first, using the technique presented in Sec. 3.2. Such a transformation is called a *MIP-guided transformation*. As well-formed protocols are used to specify behavior of components (and also the template transformations produce well-formed protocols), the computed MIP is finite and positive (as proven in [3]) and it can be used as the second parameter of the parallel construct.

4.3. Transformation Dependence

For a given component architecture A, it is necessary to transform all behavior templates associated with the components of A in order to obtain behavior specification of the whole architecture.

The property-guided transformations can be performed in an arbitrary order, as they depend only on the property values, which are provided in the architecture description of A. They are performed before any MIP-guided transformation takes place, as MIP depends on the behavior of the components and therefore it can be influenced by the property-guided transformations.

The MIP-guided transformations cannot be performed in an arbitrary order, as they can depend on each other. Let us consider the `App_Server_1` component from Fig. 1: its behavior is specified by the `Temp_AS` behavior template, featuring MIP on the access interface. To perform the transformation of the template, it is necessary to compute the MOP value on `Web_Server_1:access`, as `Web_Server_1:access` is bound to `App_Server_1:access`. The MOP value on `Web_Server_1:access` depends on the (concrete) frame protocol of `Web_Server_1`. However, the behavior of `Web_Server_1` is specified by a template which has to be transformed into a concrete frame protocol first. Therefore, the transformation of the `Web_Server_1` behavior template has to be performed before the transformation of the `App_Server_1` behavior template. We say that there is a *dependence* between the transformations for `Web_Server_1` and `App_Server_1`.

Such dependencies determine the *initial dependence graph* of the architecture - an oriented graph expressing the dependencies between MIP-guided transformations. A node of this graph is a pair of the form (C, Temp), where C is a component and Temp is the template specifying the behavior of C. There is an edge going from (C_1, $Temp_1$) to (C_2, $Temp_2$) if $Temp_2$ employs MIP on an interface P of C_2 and a required interface R of C_1 is bound to P. The graph is called "initial", because it can be changed thenceforth in order to resolve cyclic dependencies (the issue is discussed in Sec. 4.4). The initial dependence graph for the information system from Sec. 2.2 is presented on Fig. 2a).

Given an initial dependence graph without cycles, the order of the MIP-guided transformations can be designated by topological sorting [9] of the graph. If the graph contains

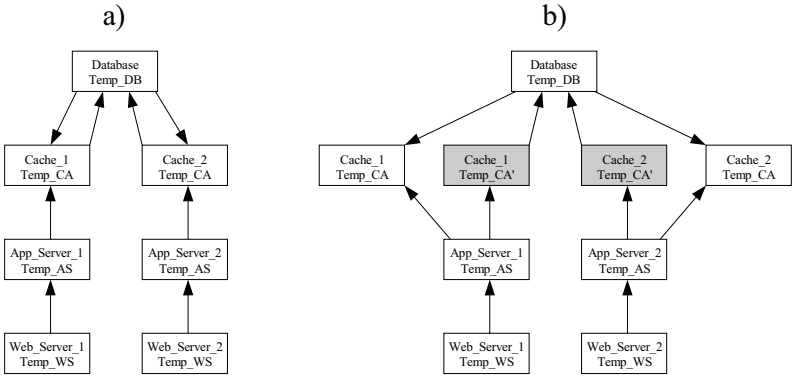


Figure 2: a) The initial dependence graph for the example from Fig. 1. b) The final dependence graph (resolution nodes in grey).

an oriented cycle, there exists no order of the transformations respecting all the dependencies. Therefore, we show in Sec. 4.4 how the cycles can be *resolved*, so that the order of the transformations can be determined.

4.4. Resolution of Cyclic Dependencies

Let us consider the cycle of the length 2 between (`Cache_1`, `Temp_CA`) and (`Database`, `Temp_DB`) on Fig. 2a). In the application from Fig. 1, `Temp_CA` employs MIP on the `nt` interface and `Database:nt` is bound to `Cache_1:nt`, while `Temp_DB` employs MIP on the `db` interface and `Cache_1:rdb` is bound to `Database:db` - this is why the cycle is present in the dependence graph. However, looking at `Temp_CA` in detail, one can see that the MOP on `rdb` in fact is not influenced by the MIP on `nt`, as the parallel construct featuring the MIP (`parallel(?nt.notify*, mip nt)`) does not contain specification of any call on `rdb`. In such a case, we say that (`Cache_1`, `Temp_CA`) is *substitutable*, as by replacing the parallel construct by the `NULL` protocol (which specifies empty behavior, which also does not contain any calls on `rdb`), we obtain a new template `Temp_CA'`, for which MOP on `rdb` is preserved, while (`Cache_1`, `Temp_CA'`) does not depend on (`Database`, `Temp_DB`):

```
Temp_CA' =
  parallel(
    (
      ?pdb.query{NULL + !rdb.query} +
      ?pdb.update{!rdb.update}
    )*,
    mip pdb
  ) | NULL
```

Such a template can be used for the computation of MOP on `Cache_1:rdb` before (`Database`, `Temp_DB`) is transformed. Later, after (`Database`, `Temp_DB`) is transformed, the transformation of the original `Temp_CA` can occur.

This technique is called *substitute template construction*. A substitute template Temp' (for a node (C, Temp)) is less complex than the original template Temp, so that a dependence is eliminated, while the MOP on a given interface is preserved. Temp' is used for MOP computation before the templates depending on Temp are transformed. It is reflected in the dependence graph as an addition of a *resolution node* (C, Temp') and changing the edges so that a cycle is eliminated. Applying this technique repeatedly, the initial dependence graph can be transformed into the *final dependence graph* without cycles. The resolution nodes in the final graph denote the transformation of the substitute templates which do not specify real behavior of components and serve only for the MOP computations.

Using the substitute template construction, the `Temp_CA'` template is generated also for (`Cache_2`, `Temp_CA`). However, in general the substitutions for the same template can differ when the template is used to specify behavior of distinct components. The final dependence graph for the information system from Sec. 2.2 is presented on Fig. 2b).

4.5. Behavior Template Transformation - the Whole Process

In this section we describe the whole process of the behavior template transformation for a component architecture A, including the way the dependencies are addressed.

If there is an oriented cycle in the initial dependence graph (Sect. 4.3) of A and the cycle does not contain a substitutable node, substitute template construction (Sect. 4.4) cannot be used to eliminate the cycle. In such a case,

the templates used to specify behavior of the components in A cannot be transformed. Template transformation for such architectures is a future work.

If elimination of all dependence cycles is possible for A, the order of the MIP-guided transformations can be obtained by topological sorting [9] of the final dependence graph. The algorithm of topological sorting produces a sequence *seq* of the graph nodes such that if there is an edge from a node (C_1, $Temp_1$) to a node (C_2, $Temp_2$) in the graph, (C_1, $Temp_1$) precedes (C_2, $Temp_2$) in *seq*.

First, all property-guided transformations are performed in an arbitrary order, then the MIP-guided transformations are performed in the order determined by *seq* - for any transformation all needed MOP-values are already computed at the time the transformation occurs (as *seq* respects the dependencies). The frame protocols obtained by the transformation of the non-resolution nodes are used as concrete behavior models for the components in A.

The concrete frame protocols for the information system from Sec. 2.2 as well as the MIP- and MOP-values needed for transformation of all the templates can be found in [3].

5. Evaluation and Related Work

Evaluation. The template-to-model transformation allows to produce finite state behavior models (the frame protocols), which specify parallelism exactly on the level needed in a given component architecture. The models can be used as inputs for verification tools, including the tools for compliance checking [11], detection of composition errors [1], or model checking [7].

On the other hand, the behavior templates are independent on the specifics of a concrete architecture and therefore they are feasible as abstractions used at a component design time.

In this paper, the scenario of MIP/MOP computations and template transformations for a top-level architecture (system architecture) was shown. To apply the idea of behavior templates to all architectures, it is necessary to deal with delegations and subsumptions, as they can affect the computation of MOP-values. It can be done by employing the idea of *inverted frame protocol* [2]. However, this extension is out of scope of this paper.

Related work. In [12], behavior of a parallel program is modeled via graphs (modeling data structures of the program) and graph transformations (modeling state-changes of the program). As such models are infinite state, the state space is bounded during model checking - any state that violates a bounding constraint is ignored so that only finite number of the states is processed. Bounding of the state space is also used in [8] for modeling (concurrent) Java programs. Here, it is implemented by putting an upper bound to the number of instances of every Java class. If a program tries to create more instances than allowed by the bounds, it cannot be checked.

In [4], the authors use a model checking algorithm which is able to address even infinite state spaces specified by combination of a finite oriented graph and a context-free grammar. However, not all infinite state spaces can be specified this way. Therefore, the authors combine their approach with techniques based on theorem proving, obtaining an algorithm which does not stop for all inputs.

6. Conclusion and Future Work

In this paper, we presented the technique of template-to-model transformation based on behavior protocols. As a future work, we plan to improve the cyclic dependence resolution presented in Sec. 4.4, so that even the cycles containing no substitutable nodes can be resolved. In addition, we will utilize the template-to-model transformation to model synchronization in component systems.

Finally, we plan to redesign the algorithms for checking behavior compliance [11] and detection of composition errors [1] in order to handle also parameterized behavior specifications. Then we will be able to check the behavior of a given component architecture independently on the concrete values of component properties.

7. References

[1] Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance and Evolution: Research and Practice 17(5), Sep 2005

[2] Adamek, J., Plasil, F.: Erroneous Architecture is a Relative Concept, in Proceedings of Software Engineering and Applications (SEA) conference, Cambridge, MA, USA, published by ACTA Press, Nov 2004

[3] Adamek, J.: Modeling Unbounded Parallelism Using Behavior Protocols, Tech. Report No. 2005/7, Dep. of SW Engineering, Charles University, Prague, Nov 2005, available at http://nenya.ms.mff.cuni.cz/publications/tr-2005-7.pdf

[4] Ball, T., Rajamani, S. K.: Automatically Validating Temporal Safety Properties of Interfaces, SPIN 2001, Workshop on Model Checking of Software, LNCS 2057, May 2001, pp. 103-122.

[5] T. Barros, L. Henrio, E. Madelaine: Behavioural Models for Hierarchical Components, SPIN'05 Workshop, Springer, San Francisco, August 2005 (LNCS 3639)

[6] Bruneton E, Coupaye T, Stefani J. B. The Fractal Composition Framework, The ObjectWeb Consortium, Jun. 2002.

[7] Clarke, E. M., Grumberg, O., Peled, D. A.: Model Checking, MIT Press, 2000

[8] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Robby: Expressing Checkable Properties of Dynamic Systems: The Bandera Specification Language, In International Journal on Software Tools for Technology Transfer, 2002, Volume 4, Number 1

[9] Mehlhorn, K.: Data Structures and Efficient Algorithms, Springer Verlag, EATCS Monographs, 1984

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

[11] Plasil F., Visnovsky, S.: Behavior Protocols for Software Components. Transactions on Software Engineering, IEEE, vol 28, no 11, Nov 2002

[12] Rensink, A., Schmidt, A., Varro, D.: Model Checking Graph Transformations: A Comparison of Two Approaches, Proc. ICGT 2004: Second International Conference on Graph Transformation, Springer 2004

[13] The SOFA project, http://sofa.objectweb.org/

[14] B. Zimmerova, L. Brim, I. Cerna, P. Varekova: Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. Proceedings of SAVCBS 2005, published by Iowa State University, 2005.

Chapter 8

Evaluation

Evaluation of the individual contributions of this thesis is presented in the papers [AP05], [AP04a], [AP04b], [Ada06] (Chapters 4-7). How the goals of the thesis are addressed by the papers is described in Sect. 3.3. Therefore, this chapter has the form of separate comments with regard to the overall contribution of composition error detection and the unbounded parallelism finite state specification, as well as the information on the tool implementation and a case study that were not known at the time the papers [AP05], [AP04a], [AP04b], [Ada06] were published.

Detection of composition errors. Checking for composition errors is very beneficial from both practical and theoretical point of view. From the practical point of view, it is important that a developer of the component application gets the information on potential design flaws very early, i.e. at the initial phase of the development (when the implementation of the components may not exist yet). The problems can therefore be solved at the time when the changes of the application design are relatively easy, resulting in lower cost and shorter time of development.

From the theoretical point of view, detection of composition errors causes that less information is lost during behavior composition. It is important for subsequent formal manipulation with the (automatically constructed) aggregate behavior specification, e.g. model checking [15].

Tool. Detection of all kinds of composition errors presented in this thesis (i.e. bad activity, no activity, divergence, and unbound requires error), including detection of composition errors in the context of a given component architecture, was implemented as a tool. The implementation is not a work of the author of this thesis. The tool (Behavior Protocol Checker) is available for download as a part of the SOFA project [35]. The tool was also successfully ported to the Fractal component model [11] within the CRE project (Component Reliability Extensions of the Fractal Component Model). The details on the CRE project can be found in [1].

Case study. Within the CRE project [1], a case-study was made. The purpose of the case study was to design a prototype Fractal application, to use behavior protocols to formally specify the behavior of the application and then to test the tools for behavior protocol analysis on those behavior protocols.

The purpose of the prototype application is to control wireless internet access on an airport and to manage the payments for the access. The internet access is either paid by a credit card, or it can be for free for an owner of a frequent-flyer card, first/business class air-ticket, etc. The application consists of twenty Fractal components, e.g. a DHCP server, an air-ticket database, a frequent flyer database, and a wrapper communicating with the credit card center. The components forming the architecture of the application are divided into three levels of nesting.

Behavior of each component is specified using a behavior protocol. As a part of the case study, for each composite component the composition errors were detected. As stated in [21], detection of composition errors had helped to identify problems in the application design and also to fine-tune behavior protocols of the primitive components, so that those protocols then provided an important guidance for the (prototype) implementation of the primitive components.

Behavior template transformation and unbounded parallelism. The concept of behavior template transformation was used in [Ada06] to solve the problem of unbounded parallelism finite-state specification for certain kinds of software component architectures. This method allows to model precisely reentrant components that are very important in practice.

From the point of view of behavior specification, software connectors [13] are typically very similar to reentrant components. The purpose of connectors is to provide software components with independence on a concrete type of communication middleware, as well as additional services (encryption, synchronization, etc.). As a connector often provides a link among more than two components (that run in parallel with each other), it has to be reentrant. Therefore, the method proposed in [Ada06] is important also for specification of connector behavior.

Chapter 9

Conclusion and future work

Conclusion. In this thesis, we introduced two formal techniques related to behavior composition in software component systems: detection of composition errors and finite state specification of unbounded parallelism. While the first technique allows to show up design inconsistencies at the beginning of the development cycle and therefore to decrease both time and cost of a component application development, the second one allows to automatically reason on wider class of component behavior than the classical approaches (e.g. common behavior protocols). We have shown how these techniques (that are general in principle) can be applied to the SOFA component model that uses behavior protocols as its behavior specification language.

Future work. As a future work, we plan to extend the semantics of behavior protocols in order to comprise also infinite traces (currently, only traces of finite length are considered). Such an extension inherently involves also reconsideration of composition errors (no activity and divergence in particular).

The behavior templates introduced in [Ada06] allow to specify unbounded parallelism by a finite state specification (a frame protocol), that is automatically generated from a behavior template using the information on a concrete environment. As stated in [Ada06], the frame protocols generated from such behavior templates can be used as an input of all verification tasks for which manually written frame protocols are feasible. One of such tasks is behavior compliance checking [32]. However, in [Ada06] we formulate the process of behavior template transformation only for flat component architectures (while compliance checking makes sense only for nested architectures). In [Ada06] we claim that the concept of behavior template transformation can be extended to nested architectures using the idea of inverted frame protocol [AP04b]. Therefore, as a future work, we plan to design in detail a new template transformation method based on this idea. As an alternative approach, we are also considering formulation of direct behavior template compliance checking in the terms of the parameterized model checking problem [6] and utilizing one of the existing solutions of this problem.

Finally, our future work also comprises the implementation of the behavior template transformations and integrating this implementation into the SOFA development tools, in order to behavior templates can be used by the developers for SOFA component behavior specification as easily as common behavior protocols.

Chapter 10

References (in addition to those mentioned in the papers [AP05], [AP04a], [AP04b], [Ada06])

[1] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, F. Plasil: Component Reliability Extensions for Fractal component model: Architecture/Design manual and User manual, available from http://kraken.cs.cas.cz/ft/public/public_index.phtml

[2] Adamek, J., Plasil, F.: Static Checking for Missing Bindings of Components, Tech. Report No. 2004/3, Dep. of SW Engineering, Charles University, Prague, Mar 2004

[3] J. Barnat, L. Brim, I. Cerna, P. Simecek: DiVinE - The Distributed Verification Environment, PDMC 2005, Lisbon, Portugal, 2005.

[4] L. De Alfaro, T. Henzinger: Interface Automata. Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM Press, 2001, pp. 109-120.

[5] R. Allen, D. Garlan: A Formal Basis for Architectural Connection, A revised version of the paper that appeared in ACM Transactions on Software Engineering and Methodology, July 1997, available at http://www.cs.cmu.edu/~able/publications/wright-tosem97-revision/

[6] K. R. Apt, D. Kozen: Limits for Automatic Verification of Finite-State Concurrent Systems. Inf. Process. Lett. 22(6): 307-309, 1986

[7] A. Arnold. Finite transition systems: Semantics of communicating systems. Prentice-Hall, 1994.

[8] I. Attali, T. Barros, E. Madelaine: Parameterized Specification and Verification of the Chilean Electronic Invoices System, SCCC'04, Arica, Chile, Nov 2004

[9] T. Barros, L. Henrio, E. Madelaine: Behavioural Models for Hierarchical Components, SPIN'05 Workshop, San Francisco, August 2005 (LNCS 3639)

[10] Bergstra J. A., Ponse A., Smolka S.A.: Handbook of Process Algebra, Elsevier 2001

[11] Bruneton E., Coupaye T., Stefani J. B.: The Fractal Component Model, version 2.0-3, the ObjectWeb Consortium, Feb. 2004.

[12] R. E. Bryant: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, C-35(8):677-691.

[13] Bures, T., Plasil, F.: Communication Style Driven Connector Configurations, Extended version of "Scalable Element-Based Connectors", Copyright (C) Springer-Verlag, Berlin, LNCS3026, ISBN 3-540-21975-7, ISSN 0302-9743, pp. 102-116, 2004

[14] G. Ciardo, R. Marmorstein, R. Siminiceanu. Saturation unbound. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), Warsaw, Poland, LNCS 2619, pages 379-393, Apr. 2003. Springer-Verlag.

[15] Clarke, E. M., Grumberg, O., Peled, D. A.: Model Checking, MIT Press, 2000

[16] Failures Divergence Refinement: User Manual and Tutorial. Formal Systems (Europe) Ltd., Oxford, England, 1.2β Edition, October 1992.

[17] H. Garavel, F. Lang, R. Mateescu: An Overview of CADP 2001. European Association for Software Science and Technology (EASST) Newsletter, 4:13-24, August 2002.

[18] Giannakopoulou, D., Kramer, J., and Cheung, S.C. Behaviour Analysis of Distributed Systems Using the Tracta Approach. Journal of Automated Software Engineering, special issue on Automated Analysis of Software, vol.6(1), January 1999. R. Cleaveland and D. Jackson, Eds, Kluwer Academic Publishers.

[19] G. J. Holzmann: The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003

[20] Hopcroft, J.E., Motwani R., Ullman J.D., Rotwani, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 2000

[21] Jezek, P., Kofron, J., Plasil, F.: Model Checking of Component Behavior Specification: A Real Life Experience, Accepted for publication in Proceedings of International Workshop on Formal Aspects of Component Software (FACS'05), Macao, October 24-25, 2005, ENTCS, Oct 2005, available at http://nenya.ms.mff.cuni.cz/publications/JezekKofronPlasil-RealLifeExperience.pdf

[22] D. Kozen. Results on the propositional mu-calculus. Theoretical Computer Science, 27:333-354, 1983.

[23] H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, CONCUR'96, Pisa, Italy, 1006. LNCS 1119.

[24] N. A. Lynch, M. R. Tuttle: Hierarchical correctness proofs for distributed algorithms. In Proceedings of PODC 1987, pages 137-151.

[25] J. Magee, J. Kramer: Concurrency: State Models & Java Programs. Wiley, 1999.

[26] J. Magee, N. Dulay, S. Eisenbach, J. Kramer: Specifying Distributed Software Architectures. Proc. of 5th European Software Engineering Conference (ESEC '95), Sitges, September 1995, LNCS 989, (Springer-Verlag), 1995, 137-153

[27] Mencl, V., Plasil, F., Adamek, J.: Behavior Assembly and Composition of Use Cases - UML 2.0 Perspective, in Proceedings of the Software Engineering (SE) 2005 conference, Feb. 15-17, 2005, Innsbruck, Austria, ISBN 0-88986-466-7, ISSN 1027-2666, pp. 193-201, ACTA Press, Feb 2005

[28] Mencl, V.: Specifying Component Behavior with Port State Machines, Electronic Notes in Theoretical Computer Science, vol. 101C pp. 129-153, Proceedings of the Workshop on the Compositional Verification of UML Models (CVUML, Oct 21, 2003, part of UML 2003), Edited by F. de Boer and M. Bonsangue, ISSN 1571-0661, Elsevier Science, Nov 2004

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

[30] R. Milner: Communication and Concurrency, Prentice Hall, 1995

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

[32] Plasil F, Visnovsky S. Behavior protocols for Software Components. IEEE Transactions on Software Engineering, 28(11), 2002, pp. 1056–1076.

[33] R. H. Reussner, I. Poernomo, H. W. Schmidt, Reasoning about Software Architectures with Contractually Specified Components, in Alejandra Cechich, Mario Piattini and Antonio Vallecillo (Eds.), Component-Based Software Quality: Methods and Techniques, State-of-the-Art Survey, Lecture Notes in Computer Science LNCS 2693, Springer 2003, ISBN 0302-9743, pp. 287-326.

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

[35] The SOFA project, http://sofa.objectweb.org

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

[37] Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, January 2002

[38] Unified Modeling Language (UML), version 2.0, Object Management Group, 2004, http://www.omg.org/

[39] B. Zimmerova, L. Brim, I. Cerna, P. Varekova: Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. Proceedings of SAVCBS 2005.