Behavior Protocols for Software Components

Frantisek Plasil[1,2], Stanislav Visnovsky[1,2]

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

[2]*Academy of Sciences of the Czech Rep.*
Institute of Computer Science
{plasil, visnovsky}@cs.cas.cz, http://www.cs.cas.cz

Abstract

In this paper, we propose a means to enhance an architecture description language with a description of component behavior. A notation used for this purpose should be able to express the "interplay" on the component's interfaces and reflect step-by-step refinement of the component's specification during its design. In addition, the notation should be easy to comprehend and allow for formal reasoning about the correctness of the specification refinement and also about the correctness of an implementation in terms of whether it adheres to the specification. Targeting all these requirements together, the paper proposes to employ behavior protocols which are based on a notation similar to regular expressions. As a proof of the concept, the behavior protocols are used in the SOFA architecture description language at three levels: interface, frame, and architecture. Key achievements of this paper include the definitions of bounded component behavior and protocol conformance relation. Using these concepts, the designer can verify the adherence of a component's implementation to its specification at run time, while the correctness of refining the specification can be verified at design time.

Keywords: behavior protocols, component-based programming, software architecture

1. Introduction

In the near future, the majority of software applications will be composed from reusable, potentially off-the-shelf software components. One of the cornerstones of successful component trading and usage is the possibility to describe their functionality in terms of both internal and external communication taking place through the component interfaces. Such a description should be sufficiently precise in order to allow for automatic checking of correctness of component composition and component use, while easy to comprehend for application programmers and simple to write for component designers. From this perspective, one of the current concerns with components is that the usual signature-based interface definitions do not describe the component communication precisely enough. The need for such a definition is reflected in efforts of the object-oriented programming community, e.g., in [3, 16, 24, 26].

1.1. Objects and protocols

An object interface definition can be considered as a service definition. As stated in [15], the sequences of requests (method calls) that an object is capable of servicing constitute the object's *protocol*, a specification of which should be an integral part of the object's interface definition(s). A typical way [3, 7, 15, 16, 24, 28] to express the object's protocol is to model it as a finite state machine. There are three basic approaches to specify such a machine: (1) directly as a state transition system, e.g. [15, 24, 28], (2) via a parser accepting the valid request sequences, e.g. [7], (3) as a regular-like expression generating the valid request sequences, e.g. [3, 19]. The protocols originate in path expressions [5] which specify synchronization of procedures executed in parallel. Procol [3] may serve as an example of an object language in which protocols are used to describe both the access synchronization of method calls and the methods' availability for servicing requests.

In most of the approaches mentioned above, particularly when addressing synchronization, checking the compliance of the calls to an object with its protocol is expected to be done at run-time. As emphasized in [16], rather than simply raising exceptions when protocols are violated, it is desirable to statically validate clients' conformance with protocols and determine automatically if a protocol can be formally viewed as a "subtype" of another one. In a similar vein, a subtyping relationship on regular types is defined in [15] which allows to statically determine whether a protocol can be replaced by another one.

1.2. Components and protocols

Recently, component-oriented program design has drawn a lot of attention, mainly because components provide a higher level of design abstractions than objects [27]. Usually, a component can be viewed as a black-box entity which provides and/or requires a set of services (accessed through interfaces). Components can be composed together by binding required to provided services to form a higher-level component. Typically, components and their compositions can be specified in ADL (Architecture Description Language); an overview of ADLs can be found in [13].

With respect to describing component communication, the approaches based on applying the idea of object protocols to components include [1, 2, 6, 8, 9, 28]. A number of papers express a component protocol via a process algebra, e.g. CSP in [1, 2], FSP in [8, 9] and

π-calculus in [6]. The protocol idea outlined in [28] is based on cooperating pairs of typed interfaces (collaborations). A collaboration description includes a protocol described as a set of sequencing constraints based on a transition system. Another option of how to describe a component behavior is to employ the UML collaboration, interaction, and state diagrams [29]. These diagrams are primarily used for a semi-formal description of object behavior, but the notation can be used for describing component behavior as well.

For employing a behavior description of a component, the notation used has to strongly support description of the "call interplay" on the component's several interfaces, and to reflect step-by-step specification and refinement of the component during its design. Most of the approaches mentioned above employ a black-box view of a component (no visible internals) together with a white-box view (all internals visible). However, to support a step-by-step specification, it is reasonable to include also specification of a "grey-box" view of the component (selected details visible only). Naturally, the notation chosen for behavior description should be easy to comprehend and allow for formal reasoning about the specified behavior. This might also imply the need for a solution to inherent obstacles such as the state-explosion problem and decidability of specific properties related to behavior description, e.g., correctness of an implementation in terms of adhering to its specification, as well as specification refinement correctness.

1.3. The goals and structure of the paper

The goal of this paper is to address the following issues presented in Section 1.2: (1) development of an easy-to-read notation for behavior specification, (2) clear support for behavior specification refinement in ADL, (3) support for formal reasoning about the adherence of a component's implementation to the behavior specification of the component, as well as about the correctness of behavior specification refinement, (4) dealing with the potential decidability and state explosion problems.

To reflect the goals, the paper is organized as follows. In Section 2, we provide an overview of the SOFA component model [18] which will serve as a proof-of-the-concept base. Section 3 introduces behavior protocols and the underlying model of communication. Section 4 shows how behavior protocols can be associated with the SOFA architecture description language (CDL), while the protocol conformance relation is defined in Section 5. Further, in Section 6, seamless integration of the idea of step-by-step protocol refinement into the SOFA component model is illustrated. Section 7 is devoted to evaluation and open issues. Related work is discussed in Section 8. Section 9 concludes the paper by summarizing key achievements.

2. SOFA Components

2.1. Component model and component lifecycle

The SOFA (Software Appliances) project [18, 25] targets the issue of composing applications from components which can be deployed over a network. In the SOFA component model, an application is viewed as a hierarchy of nested software components. Analogously with the classical concept of object being an instance of a class, we introduce *software component* (*component* for short) as an instance of a component *template*. In principle, "template" can be interpreted as "component type".

A template T is a pair <F, A> where F is a template *frame*, and A is a template *architecture*. The frame F defines the set of individual interfaces any component which is an instance of T will possess. The interfaces are instances of interface types. In F, an interface can be instantiated as a *provides-interface* or a *requires-interface* (this concept is typical for most ADLs, e.g. [13]). Basically, the frame F reflects the black-box view on T. To support versioning, the frame F can be implemented by more than one architecture. An architecture A describes the structure of an implementation version of F by (1) instantiating direct subcomponents of A (those on the adjacent level of component nesting, *subcomponents* of A for short), and by (2) specifying the subcomponents' interconnections via interface ties. Basically, the architecture A reflects a particular grey-box view on the template T.

There are four kinds of interface ties: (a) *binding* of a requires-interface to a provides-interface between two subcomponents, (b) *delegating* from a provides-interface of F to a subcomponent's provides-interface, (c) *subsuming* from a subcomponent's requires-interface to a requires-interface of F, (d) *exempting* an interface of a subcomponent from any ties (the

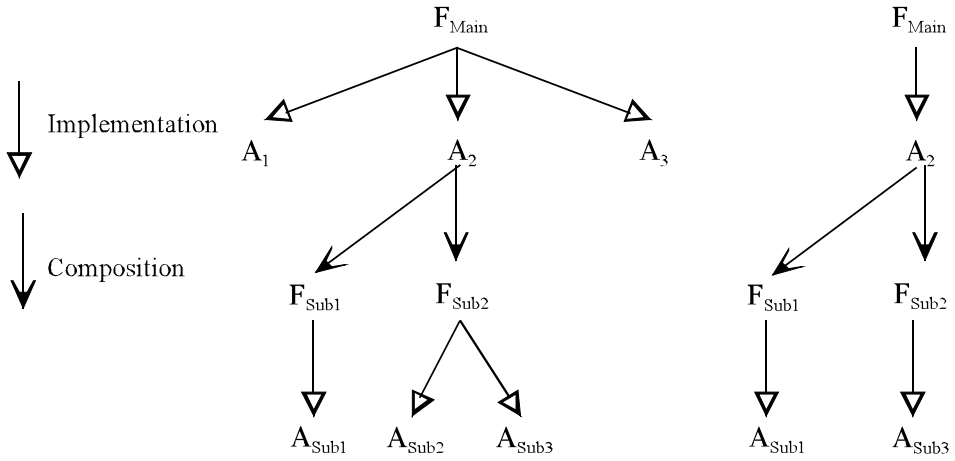

Figure 1. Development tree of application and assembled application

interface is not employed in A). An architecture can also be specified as *primitive*, which means that there are no subcomponents and its structure/implementation will be provided in an underlying implementation language, out of the scope of the component model.

A component's lifecycle is characterized by (potentially repeated) sequence of design time and run time phases. In a more detailed view, a design time phase is composed of the following design stages: development and provision, assembly, and deployment.

At the development and provision stage, a component is specified by its frame and potentially several architectures, each of them being a design version of the frame as illustrated in Figure 1. For instance, the frame F_{Main} is implemented by three different architectures: A_1, A_2, and A_3. While A_1 and A_3 are primitive, A_2 is composed of two subcomponents Sub_1 and Sub_2; these subcomponents are visible in A_2 only at the level of their frames F_{Sub1}, F_{Sub2}. It is important to emphasize that the actual specification of an architecture A is always based on the frames of A's subcomponents (and not on the

architecture of those subcomponents). Reflecting top-down design, the specification of an application is factored this way into a of alternating layers frame – architecture – frame – …, forming a tree with nodes alternately of the "frame" and "architecture" types. As an aside, by clear separation of the levels of revealing architectural details, it also allows for easy replacement of a subcomponent by another one [18].

At the assembly stage of design time, the executable form of an application/component is determined by selecting an implementation architecture for each frame. In Figure 1 it means reducing the tree in such a way that each frame node has only one child architecture node as presented on the right-hand side of Figure 1. This process starts at F_{Main} by choosing one particular template $<F_{Main},A_i>$. If A_i is not primitive, the selection is applied recursively to all frames involved in A_i. Consequently, the executable form of the application/component is primarily based on all the primitive architectures involved recursively in the reduced subtree of A_i.

The executable is then deployed during the deployment stage, when the component run time configuration has to be devised. In particular, this includes distribution over computer network nodes and setting some of the component property parameters. At the end of the deployment, the component is ready to run.

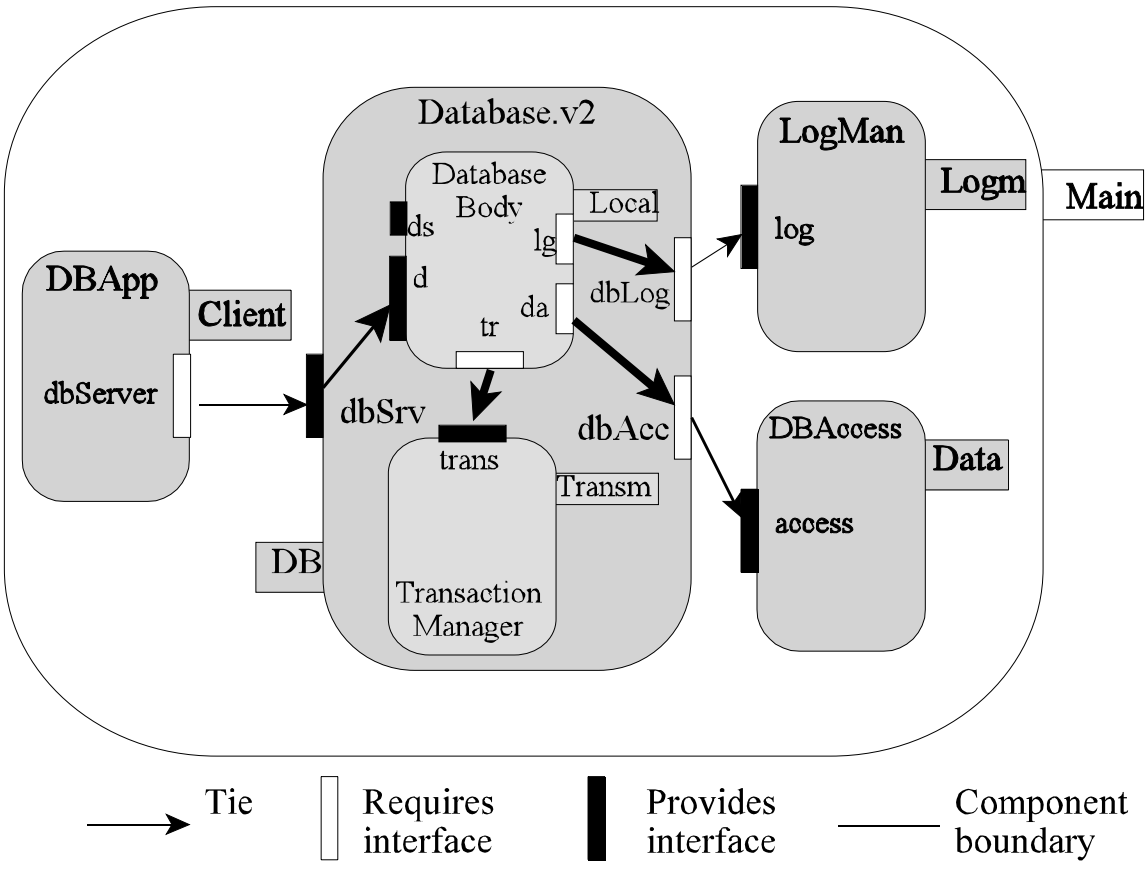

Figure 2. Database architecture

2.2. CDL specification language

Based on CORBA IDL, SOFA CDL (component definition language) is the means to specify components in SOFA. The syntax of CDL is provided in full in [25]. In this paper, we demonstrate the use of CDL just on a simple example.

Let us imagine a component designed as a very simple database server with some add-on functionality. The component provides the *Insert*, *Delete*, and *Query* operations for inserting and removing records from the database, and querying the contents of the database. In support of its functionality, the component employs a low-level database server component and a component allowing for logging messages to a log file (Figure 2). These components publish their services by means of provides-interfaces; the underlying database server provides the interface of the *IDatabaseAccess* type, and the logging component's interface is of the *ILogging* type. Similarly, the database server component provides its services via an instance of the *IDBServer* interface type.

Interface type definitions are expressed via the CDL interface construct specifying an interface type as a set of method signatures as follows:

```
interface IDBServer {
    void Insert(in string key, in string data);
    void Delete(in string key);
    void Query(in string query, out string data);
};

interface ILogging {
    void LogEvent(in string event);
    void ClearLog();
};
```

```
interface IDatabaseAccess {
    void Open();
    void Insert(in string key, in string data);
    void Delete(in string key);
    void Query(in string query, out string
    data);
    int GetTrModel();
    void SetTrModel(int model);
    void Close();
};
```

After the necessary interface types have been specified, the black-box view of the proposed component can be designed. In CDL, this is done by means of the frame construct which encapsulates instances of the provides-interface and requires-interfaces in the way illustrated in the *Database* frame below.

```
frame Database {
    provides:
        IDBServer dbSrv;
    requires:
        IDatabaseAccess dbAcc;
        ILogging dbLog;
};

architecture Database version v2
    inst TransactionManager Transm;
    inst DatabaseBody Local;
    bind Local:tr to Transm:trans;
    exempt Local.ds;
    subsume Local:lg to dbLog;
    subsume Local:da to dbAcc;
    delegate dbSrv to Local:d;
};
```

```
interface ICfgDatabase {
    int GetTrModel();
    void SetTrModel(int model);
};

frame DatabaseBody {
    provides:
        IDBServer d;
        ICfgDatabase ds;
    requires:
        IDatabaseAccess da;
        ILogging lg;
        ITransaction tr;
};
```

The internals of the proposed components are specified via the CDL architecture construct. In the example above, the *Database* architecture version *v2* illustrates how subcomponents are instantiated and how their ties are specified (distinguishing bind, subsume, delegate, and exempt ties). Here, two subcomponents *Transm* and *Local* are instantiated, each of them being specified at the abstraction level of its frame (the respective architectures of these subcomponents will be specified at the application assembly time). Notice how *Local*'s interfaces are tied to the interfaces of the *Database* frame and to the *Transm* subcomponent. Moreover, the architecture specification reveals that *ds*, one of the *DatabaseBody*'s interfaces, is not bound to any subcomponent nor the *DatabaseBody* frame interface; this means that *ds* will never be engaged in component communication.

3. Behavior protocols

In this section, we provide a formal model aiming at describing the behavior of software components. We abstract from a particular component model and most of the ADL-dependent details such as name spaces, typing rules, etc. Being focused on fundamental principles, we base our model on the abstract component - "agent" concept, where interface ties of components became connections among agents, method calls on interfaces turn into events on connections, and a component's behavior is modeled via the event sequences (traces) on the connections of the agent representing the component. The behavior can be approximated and represented by regular expression-like "protocols" introduced in this section. Relations defined upon these protocols will allow us later on to reason about component cooperation statically at assembly time and dynamically at run time.

3.1. Model of communication

An *agent* is a computational entity handling sequences of events. As to handling events, agents can emit events, absorb events, and process internal events. This handling is fully determined by the *implementation* of the agent. Agents communicate via peer-to-peer bi-directional connections transmitting events; an agent can communicate with a finite number of other agents, and two agents can communicate through a finite number of connections.

An agent can be *primitive* or *composed*. A *primitive agent* is an agent which does not contain any other agent and all of its connections are *external*. A composed agent P is constructed by composition of agents A and B in the following way (easily extendable for n agents): The union of the connections of A and B become the connections of P and the events on these connections are handled by both P and A and/or B jointly. The connections through which A and B communicate with each other (external connections from A's and B 's points of view) become the *internal connections* of P. The remaining connections of A and B become external connections of P. The events on the internal connections of P are referred to as *internal events* of P (similar to internal actions τ in [14]). The implementation of P is fully determined by its internal connections and by the implementations of A and B. (A primitive agent is a black-box entity with an implementation defined out of the scope of the model.)

For illustration, let us consider agents A, B, and P and connections C_1, C_2, C_3, and C_4 in Figure 3. Here A and B are primitive and P is composed of A and B. In this context C_1, C_2, and C_3 are the external connections of A, while C_2 and C_4 are the external connections of B.

Further, C_1,C_3, and C_4 are the external connections, and C_2 is the internal connection of P which shares all of them with A and B

Based on composition, every agent is a part of the hierarchy of agents called a *system*. The root of the system is an agent with no external connections. In a system Σ, all agents start event handling after receiving an init signal (broadcasted by an external, intuitively defined, control authority). Similarly, all agents in Σ stop event handling after receiving a stop signal from this control authority. The event handling of all the agents in Σ between the init and stop signals constitute a *run* of Σ.

We assume that an agent cannot handle more than one event at the same time, there is no connection delay, and an agent emits an event only if its counterpart on the connection is prepared to accept it (emitting and absorbing a particular event is done as one atomic action). Thus, in a specific run of a system Σ, the activity of its agent A on a set of connections V is observed as the finite sequence of events which A handles on V in the run. By convention, such a sequence of event handling is represented as a *trace of A on V*; the trace is a sequence of *event tokens*, each of them standing for handling of exactly one event. The set of all possible sequences of event handling of A on V in any run is referred to as the *behavior* of A on V in Σ.

With the intention to reason about reusability of modeled components, it is desirable to formally capture the behavior of A in any run and in any system. To support this aim, we pretend all the potential neighboring agents of A–those connected to A's external connection–form the *environment* E of the agent A; in other words, the external connections represent the interface between A and E. Abstracting from specific neighboring agents, we simply presume that there is predefined contract between A and E as to how they can exchange events. In a properly designed system, the contract is respected, and A has to absorb the events emitted by E and vice versa–E has to absorb the events emitted by A.

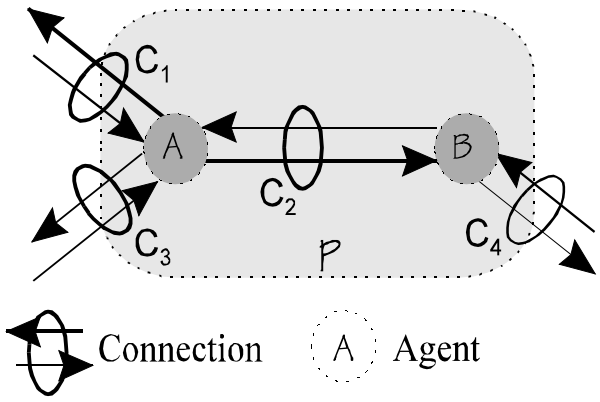

Figure 3. Composition of agents

To be able to formulate the contract at a higher level of abstraction, we assume that every trace representing a particular communication between an agent A and its environment E is an interleaving of two logical parts *provision* and *requirement*. This reflects the idea that A provides (offers) services to E (and E can submit a requirement to A to handle same of the services–it can choose a provision). Similarly, A can require some services from E. Since the agents (and their implementation) forming E are not explicitly known, we pretend that E makes the choice, "dictates", which of the provisions offered by A it will require, while A chooses the requirement it will demand from E depending, typically, upon the particular provision chosen by E.

To distinguish provisions and requirements in the traces of an agent A, we define the *provision alphabet* S_{prov} and *requirement alphabet* S_{req} of A as the sets of event tokens used for representing event handling on the external connections of A (those on internal connections form the *internal alphabet* S_{int} of A). For simplicity we assume $S_{prov} \cap S_{req} = \varnothing$, and define $S_{prov} \cup S_{req}$ as the *external alphabet* S_{ext} of A.

In the example in Figure 3, the event tokens representing the event handling on the connections C_1, C_3 and C_4 comprise P's external alphabet S_{ext}, and those on C_2 the internal alphabet S_{int}. To illustrate the idea of provision and requirement alphabets, one can define, e.g., that the event tokens representing P's event handling on C_1, C_4 comprise S_{prov} and on C_3 comprise S_{req}.

3.2. Behavior as a language

We assume that the set of all event names *EventNames* is composed of pairs syntactically written as <connection_name>.< local_event_name>, where connection_name is from the single global name space GNS and local_event_name from the local event name space LNS_{cn} of a connection_name cn. By convention, for any alphabet S considered in the text holds $S \subseteq \{!,?,\tau\} \times GNS \times \cup LNS_{cn} \times \{\uparrow, \downarrow\}$, and, therefore, every action token is syntactically written as <event prefix><connection_name> .<local_event_name><event suffix>. The event prefix (one of the symbols *!*, *?*, resp. τ), expresses whether an event is emitted, absorbed, resp. internal. To support modeling of specific events like the request resp. response part of a remote method call, an event suffix can be employed. In this text we use the symbol $\uparrow$ resp. $\downarrow$ to denote a *request* and *response* as the event suffix . Thus, a pair <event name, event suffix> reflects an *event* while <event prefix> reflects "the end" of the connection the event is viewed from. The traces of an agent A on V are words over its alphabet S (from S*); the behavior of A on V is represented as the set of these traces - the *language of* A *on V*. By L_A we denote the language of A on all its connections (V comprises all A's connections).[1] For simplicity, we will understand by L_A also the behavior of A.

To capture a provision in a trace t, we say t is with *provision* p if $t/S_{prov} = p$, and the set L_A/S_{prov} forms the *provisions* of A. (Here, t/S denotes the restriction of a trace t to a set of event tokens S, i.e., the event tokens not in S are omitted in the resulting trace.) In a similar vein, we say that t is with *requirement* r if $t/S_{req} = r$ and L_A/S_{req} forms the *requirements* of A.

Let us assume the language of the agent A in Figure 3 contains only two traces: $L_A = \{ <?C_1.a\uparrow, !C_1.a\downarrow, ?C_2.x\uparrow, !C_2.x\downarrow, !C_3.b\uparrow, ?C_3.b\downarrow>, <?C_1.a\uparrow, ?C_2.x\uparrow, !C_1.a\downarrow, !C_2.x\downarrow> \}$. Here, the < resp. > symbol denotes beginning resp. end of a trace. Since C_1 and C_3 are external connections of both A and P and C_2 is an external connection of A but internal of P, $<?C_1.a\uparrow, !C_1.a\downarrow, \tau C_2.x\uparrow, \tau C_2.x\downarrow, !C_3.b\uparrow, ?C_3.b\downarrow>$ is a trace of P. As the event tokens on C_1, C_4 comprise S_{prov} and on those on C_3 comprise S_{req} of P, the trace is with the provision $<?C_1.a\uparrow, !C_1.a\downarrow>$ and with the requirement $<!C_3.b\uparrow, ?C_3.b\downarrow>$.

---

[1] Usually, we will express the language of A on some set V via restricting L_A to the subset of its alphabet reflecting V

3.3. Behavior protocols – the basics

The language of an agent is typically not as simple as in the example above. In fact it can be an infinite and even unrestricted [30] language in general. A challenge is to find a finite notation for a definition of such a language; the notation should be simple enough to be easily applied in component ADL specifications and manipulated by automated tools. With the aim to employ a notation simpler than any of those used in related work such as CSP [23], TRACTA [8, 9], Rapide [12], the approach we choose is to approximate the behavior by a regular language which can be expressed by a behavior protocol introduced below (regularity of behavior protocols is justified in [21]).

A *behavior protocol* (*protocol* for short) Prot over an alphabet[2] S is a regular-like expression which (syntactically) generates a set of traces over S–the language L(Prot). The simplest behavior protocol is an event token or the NULL symbol (empty trace). A behavior protocol is constructed in a way similar to a regular expression and can use the operators and abbreviations listed below. The basic operators are those of regular expressions [30]. The enhanced operators provide a notation for describing concurrency resp. communication hiding and represent well known operations of shuffle resp. restriction of a language [30]. Finally, we define the composed operators as a special purpose operators. In principle, the semantics of the adjustment operator $|_T|$ is inspired by the generalized parallel operator defined in CSP [23], while the semantics of the composition operator $\sqcap_X$ by the parallel composition in CCS [14]. For an exact definition of these operators, we refer the reader to Appendix A.

Operators (A, B, denotes a protocol; m, e an event name)

Basic operators (defined in classical regular expressions)

A ; B — *sequencing*; the set of traces formed by concatenation of a trace generated by A and a trace generated by B,

A + B — *alternative*; the set of traces which are generated either by A or by B,

A* — *repetition*; equivalent to NULL + A + (A;A) + (A;A;A) + … where A is repeated any finite number of times.

Enhanced operators

A | B — *and-parallel;* an arbitrary interleaving of event tokens of traces generated by A and B,

A || B — *or-parallel*; stands for A + B + (A | B)

A / G — *restriction*; the event tokens not in a set G are omitted from the traces of L(A).

Composed operators

$A \sqcap_X B$ — *composition*; an arbitrary interleaving of event tokens of traces from A and B except for when an event e from the set of events X is absorbed in a trace generated by A and emitted in a trace of B (or vice versa), then the handling of e is expressed as an internal event in resulting trace,

$A |_T| B$ — *adjustment*; an arbitrary interleaving of the event tokens of all pairs of traces $<\alpha, \beta>$ (α generated by A and β generated by B) except for the event tokens from a set

[2] We always assume that the alphabet of a protocol does not contain any useless symbols

T which represent “synchronization points” in a pair (they have to appear in α and β in the same order, otherwise adjustment of the pair does not produce anything). The event tokens from T are in the resulting trace only once and their original order is preserved.

Abbreviations

?m{α} *nested incoming call*; stands for ?m↑ ; α ; !m↓

?m *simple incoming call;* stands for ?m↑ ; !m↓

!m *simple outgoing call*; stands for !m↑ ; ?m↓

Precedence of operators

1. (highest) repetition (*), restriction (/), 2. sequencing (;), 3. and-parallel (|), or-parallel (||), 4. alternative (+), 5. (lowest) composition ($\sqcap_X$), adjustment ($|_T|$) .

To demonstrate behavior protocols as a tool for language generation, let us consider the protocol *?a; (!p + !q); !b*. It contains event tokens *?a,!b,!p,!q* and the operators *;* and +. (In the examples here, we omit event suffixes ↑ and ↓ and connection names for simplicity.) The protocol generates traces, which start with *?a*, followed by *!p* or *!q* and finish with *!b* and therefore the generated language is {<*?a, !q, !b*>, <*?a, !p, !b*>}.

Consider the protocol *?a ; (!p + !q) ; !b || ?x*. The event *x* occurs in parallel with the behavior on the left hand side of ||. The generated language includes for instance the traces <*?a, !p, !b*> and <*?a,?x, !q, !b*> (notice that *x* does not have to be necessarily present). Now, we enhance the protocol by a adding the composition operator, e.g., *?a ; (!p + !q) ; !b || ?x* $\sqcap_S$ *(?q; ?r)*, where S contains *p* and *q*. In the language generated, there cannot be a trace including a *p* event, since the right operand of $\sqcap_S$ requires *q* (and no *p*) to appear. However, for traces of the left operand of composition where *q* is present, in the resulting traces is *q* expressed via an internal event as in <*?a,?x,* τ*q, !b, ?r*> or <*?a,?x,* τ*q, ?r, !b*> (*!b, ?r* can be arbitrary interleaved).

To illustrate the adjustment operator $|_T|$, consider *T = {?x,!y}* and protocols *?a;?x;?b;!y;!c* and *?x;!d;!y;!c* (*?a,?b,!c,!d,?x,!y* are event tokens) generating the languages *{<?a,?x,?b,!y,!c>}* resp. *{<?x,!d,!y,!c>}*. The language generated by a protocol *(?a;?x;?b;!y;!c)* $|_T|$ *(?x;!d;!y;!c)* is constructed as follows: Since both protocol generate a trace containing the sequence <*?x,!y*> of the event tokens from *T*, the resulting language comprises only traces, which are a interleaving of <*?a,?x,?b,!y,!c*> and <*?x,!d,!y,!c*>, where ?*x* and *!y* are “synchronization points”. A resulting traces begin by interleaving all sub-sequences of the original traces before the occurrence of *?x*, i.e, <*?a*> and <>; resulting in <*?a*... It is followed by *?x* (<*?a,?x*,...) and then there is interleaving of the rest of the original traces up till the next occurrence of *!y*, i.e, <*?b*> and <*!d*>, resulting in <*?a,?x,?b,!d*,.. and <*?a,?x,!d,?b*.... They are followed by *y* and finished by interleaving of the rest of the original traces <*!c*> and <*!c*>, resulting in the traces <*?a,?x,?b,!d,!c,!c*> and <*?a,?x,!d,?b,!c,!c*>. Note, that in case of protocol *(?a;?x;?b;!y;!c)* $|_T|$ *(?x;!d;!c)* the language generated is empty, since there is no pair of traces generated by left and right operand where the traces agree on the “synchronization points”, i.e., the sub-sequences formed by the event tokens from *T* only.

We utilize the adjustment operator for comparison of behavior of agents (components) in the following way: If the protocol B in A $|_T|$ B comprises only event tokens from T, it can be seen as an obligation for the protocol A in the sense that A should handle the event tokens

from T in the same way as B does. Since a trace α of B comprises event tokens from T only and a trace β of A contains these event tokens, either α $|_T|$ β generates β, or it does not yield any trace. As a result, L(A $|_T|$ B) ⊆ L(A). For example, from *(?a;?x;?b; (!y;!c+NULL))* $|_T|$ *(?x;!y)*, we yield the trace *<?a,?x,?b,!y,!c>* containing *?x* and *!y* and eliminating *<?a,?x,?b>* containing *?x* only.

To illustrate how a component behavior can be specified, consider the component *Local* from Figure 2 is an agent and its ties are the agent's connections. Let us limit ourselves to the set of event names *{trans.Begin, trans.Commit, trans.Abort, dbAcc.Insert}*. To express that the component (agent) does two sequential successful transactions, we can write *!trans.Begin↑; !dbAcc.Insert↑, !trans.Commit↑; !trans.Begin↑ ; !dbAcc.Insert↑; !trans.Commit↑*.

The composition operator is suitable for expressing joint behavior of components communicating via bound interfaces. For example, should there be a transaction manager component *Transm* (again viewed as an agent) with the behavior specified as *(?trans.Begin; (?trans.Commit + ?trans.Abort))** communicating with *Local* via the *trans* tie, their joint behavior can be described using the composition operator as *!trans.Begin↑; !dbAcc.Insert↑;!trans.Commit↑; !trans.Begin↑ ; !dbAcc.Insert↑;!trans.Commit↑* $\sqcap_X$ *(?trans.Begin↑; (?trans.Commit↑ + ?trans.Abort↑))**, where X is composed of the events on the *trans* connection, i.e. *{trans.Begin↑, trans.Commit↑, trans.Abort↑}*. Since the events on *trans* will be exhibited as internal events in the only trace generated by this protocol, they will be prefixed by *τ*, as *< τtrans.Begin↑, !dbAcc.Insert↑, τtrans.Commit↑, τtrans.Begin↑, τdbAcc.Insert↑, τtrans.Commit↑>*.

3.4. Protocol-based system design

In this section, we address the issue what kind of role can behavior protocols play in the design of an agent hierarchy (of a system).

Here we assume that a typical design step is elaboration/refinement of an agent, and this step takes the form of a replacement of an old agent B by a new, more elaborated agent. A meaningful replacement assumes not only taking over external connections but also a behavior similarity of A and B. As the criterium for a meaningful replacement we choose the requirement that the agent's environment should not "notice" the change (the refinement does not "too much"modify the behavior expected by the environment). Addressing this requirements for external, black-box behavior view of an agent, we introduce *substitutability* in Section 3.4.1. Further, during elaboration, it is important to reason on similarity of parts of A's resp. B's behavior (e.g. on a subset of connections, including some of internal ones). In general, we address this need by capturing the situation that the behavior of A does not differ "too much" from the behavior of B as *behavior compliance* (Section 3.4.2), substitutability becoming a special case of it. There are two additional important special cases of behavior compliance:

(1) Design of agent hierarchy should be based on behavior specification (behavior protocols). Introduction of *model agent* of a protocol (behaving exactly as the protocol) in Section 3.4.3, allows to abstract from any particular implementation of an agent and to judge behavior similarity directly on protocols via *protocol compliance* (Section 3.4.2).

(2) At some point of the design process a model agent has to be replaced by a "real" agent (really implemented); thus there is the need to ask its behavior to be "reasonably similar" to the specification - its behavior should be *bounded* by the model agents' protocol (Section 3.4.3).

3.4.1. Agent substitution [3]

Let us consider an agent A to replace another agent B in its environment E (A being put into B's environment E by taking over all of its external connections).

As a natural requirement, we assume that if the same event token e is in both alphabets S_A and S_B, it should be in the same "part", i.e., if e is in provision (resp. requirement resp. internal) alphabet of A then e is also in the provision (resp. requirement resp. internal) of B and vice versa. Such an alphabets S_A and S_B are called *harmonious*.

In E, A has to render the provisions of B as they can be chosen by E; therefore we assume $S_{B,prov} \subseteq S_{A,prov}$ and ask (1) $L_B/S_{B,prov} \subseteq L_A/S_{A,prov}$. For a chosen provision p of B, the environment E expects a trace from $L_B/S_{B,ext}$ with the provision p. Using the adjustment operator the way described in Section 3.3, we can formally capture the traces of A on its external connections with the provisions equal to those of B as $L_B/S_{B,prov} |S_{A,prov}| L_A/S_{A,ext}$. By asking the traces in $L_B/S_{B,prov} |S_{A,prov}| L_A/S_{A,ext}$. to be also in $L_B/S_{B,ext}$ (i.e. (2) $L_B/S_{B,prov} |S_{A,prov}| L_A/S_{A,ext} \subseteq L_B/S_{B,ext}$) we do not allow A to issue any requirement for a given provision p which could not have been issued also by B (assuming $S_{A,req} \subseteq S_{B,req}$). Therefore, if both (1) and (2) hold, then, for every provision p of B, there exists at least one trace of A with the provision p. In this case, we conclude A can replace B, since E can dictate the same provisions it could for B, and will get requirements which could have been issued by B. In summary:

Definition: Let B be an agent in an environment E, L_B its behavior and S_B its alphabet. Also, let A be another agent, L_A its behavior and S_A its alphabet (harmonious with S_B) such that $S_{B,prov} \subseteq S_{A,prov}$ and $S_{A,req} \subseteq S_{B,req}$.We say B is *substitutable in E by A* (or B can be *substituted* in E by A) if

(1) $L_B/S_{B,prov} \subseteq L_A/S_{A,prov}$

(2) $L_B/S_{B,prov} |S_{A,prov}| L_A/S_{A,ext} \subseteq L_B/S_{B,ext}$.

To illustrate the concept, let us assume an agent B with its alphabet $S = S_{prov} \cup S_{req}$, where S_{prov} = {*?dbSrv.Insert↑, !dbSrv.Insert↓*} and S_{req} = { *!dbAcc.Query↑, !dbAcc.Insert↑*}, in an environment E. Further assume L_B = {<*?dbSrv.Insert↑, !dbAcc.Query↑, !dbSrv.Insert↓*>, <*?dbSrv.Insert↑, !dbAcc.Insert↑, !dbSrv.Insert↓*>}. If A is another agent with the alphabet S and L_A = { <*?dbSrv.Insert↑, !dbAcc.Query↑, !dbSrv.Insert↓*>} then B is substitutable by A in E: The provisions of A and B are the same (i.e., $L_B/S_{prov} = L_A/S_{prov}$ = {<*?dbSrv.Insert↑, !dbSrv.Insert↓*>}). The sets of traces with the provision <*?dbSrv.Insert↑, !dbSrv.Insert↓*> of A resp. B are L_A resp. L_B themselves, and, therefore, the inclusion in condition (2) above holds. In other words, the traces of A with this provision contain requirements interleaved with the provision exactly the same way as they are in the traces of B. However, if L_A contained only the trace <*?dbSrv.Insert↑, !dbAcc.Query↑, !dbAcc.Query↑, !dbSrv.Insert↓*>,

---

[3] Here and in the rest of Section 3 we assume any alphabet S of an agent or a protocol is composed of an external alphabet S_{ext} (formed by provision and requirement alphabets S_{prov} and S_{req}) and internal alphabet S_{int}.

B would not be substitutable by A, since the requirement <*!dbAcc.Query*↑, *!dbAcc.Query*↑> is not a requirement of B in none of its traces with the provision <*?dbSrv.Insert*↑, *!dbSrv.Insert*↓>.

Note, that it would be misleading to deal with provisions and requirements separately by asking (1') $L_B/S_{prov} \subseteq L_A/S_{prov}$ and (2') $L_A/S_{req} \subseteq L_B/S_{req}$ (i.e. A should provide at least the same as B, while requiring not more than B). Then, for instance, assuming S_{prov} = {c,d}, S_{reg} = {x,y}, L_A= {<c,x,d,y>} and L_B = {<c,d,x,y>}, B would be substitutable by A. In other words, as to substitutability, the interleaving of provisions and requirements encountered in B's traces has to be preserved in A's traces.

It is important to apply adjustment on $S_{A,prov}$ (and not on $S_{B,prov}$) in order to consider all the provisions of A, and in particular eliminate the cases when a provision of B is a substring of a provision of A. For instance, if $S_{A,prov}$={a,b}, $S_{B,prov}$= {b} and L_A = {<a,b> , <b>} while L_B ={<b>} only, <a,b> is a resulting trace of the adjustment on $S_{B,prov}$. On the contrary, if the adjustment is on $S_{A,prov}$, the only resulting trace is <b> as desired.

3.4.2. Behavior and protocol compliance

A flexible way to capture the desired partial view on the behavior of agents A and B can be achieved by restricting L_A and L_B to a "partial-alphabet" S. Naturally, the selection of S has to "fit" both the alphabets of A and B, in particular it should be harmonious with them. Generalizing the thoughts on behavior similarity of agents A and B from Section 3.4.1, we will again consider L_A/S to be similar to (*compliant with*) L_B/S if the provisions of L_A contain all the provisions of L_B, and if the traces in L_A/S with the provisions of L_B/S_{prov} are in L_B/S:

Definition: Let A resp. B be agents with harmonious alphabets S_A resp. S_B. Let S be another alphabet (harmonious with S_A resp. S_B) such that $S_{prov} \subseteq S_{A,prov} \cup S_{B,prov}$, $S_{req} \subseteq S_{A,req} \cup S_{B,req}$, $S_{int} \subseteq S_{A,int} \cup S_{B,int}$. The behavior L_A of A *is compliant with* the behavior L_B of B on S if

(1') $L_B/S_{prov} \subseteq L_A/S_{prov}$
(2') $L_B/S_{prov}\ |S_{prov}|\ L_A/S \subseteq L_B/S$.

It should be emphasized that the selection of S includes the choice of both the connections and events on them. As an important case, if S covers all external communication of A and B ($S_{prov} = S_{A,prov}$ and $S_{req} = S_{B,req}$) and the basic assumption of substitutability holds ($S_{B,prov} \subseteq S_{A,prov}$ and $S_{A,reg} \subseteq S_{B,reg}$), then compliance of L_A with L_B implies substitutability of B by A:

Lemma 3.4.1: Let A resp. B be an agent, S_A resp. S_B its alphabet, L_A resp. L_B its behavior. Let S_A and S_B be harmonious and $S_{B,prov} \subseteq S_{A,prov}$ and $S_{A,reg} \subseteq S_{B,reg}$. Let S be an alphabet such that $S_{prov} = S_{A,prov}$, $S_{req} = S_{B,req}$. If L_A is compliant with L_B on S, then B is substitutable by A.

Proof: We show that conditions (1) and (2) of substitutability follow from the conditions (1') and (2') when only the event tokens from S_{ext} are considered: From (1') $L_B/S_{prov} \subseteq L_A/S_{prov}$ it follows that $L_B/S_{A,prov} \subseteq L_A/S_{A,prov}$ and $S_{B,prov} \subseteq S_{A,prov}$; since S_A and S_B are harmonious, it holds that $L_B/S_{B,prov} \subseteq L_A/S_{A,prov}$ (condition (1)). Further, from $L_B/S_{prov}\ |S_{prov}|\ L_A/S \subseteq L_B/S$ it follows that $(L_B/S_{prov}\ |S_{prov}|\ L_A/S)/S_{ext} \subseteq L_B/S/S_{ext}$, which can be simplified as $L_B/S_{prov}\ |S_{prov}|\ L_A/S_{ext} \subseteq L_B/S_{ext}$ because the adjustment is done on S_{prov}, a subset of S_{ext}. Then $L_B/S_{A,prov}\ |S_{A,prov}|\ L_A/(S_{A,prov} \cup S_{B,req}) \subseteq L_B/(S_{A,prov} \cup S_{B,req})$. Again, we can replace $L_B/S_{A,prov}$ by $L_B/S_{B,prov}$. $L_A/(S_{A,prov} \cup S_{B,req}) = L_A/(S_{A,prov} \cup S_{A,req})$, since $S_{A,req} \subseteq S_{B,req}$ and since S_B and S_A are

harmonious. Similarly, $L_B/(S_{A,prov} \cup S_{B,req}) = L_B/(S_{B,prov} \cup S_{B,req})$ since $S_{B,prov} \subseteq S_{A,prov}$, if we assume that S_A is harmonious with S_B. Therefore, $L_B/S_{B,prov} |S_{A,prov}| L_A/S_{A,ext} \subseteq L_B/S_{B,ext}$ (i.e. the condition (2)) holds. □

In order to allow for reasoning on similarity of behavior specified by protocols, we define compliance of behavior protocols for a given alphabet:

Definition: Let $Prot_A$, $Prot_B$ be protocols with harmonious alphabets S_A resp. S_B. Let A and B be agents with the alphabets S_A resp. S_B such that $L(Prot_A) = L_A$ and $L(Prot_B) = L_B$. If the behavior L_A is compliant to the behavior L_B on an alphabet S, we say that $Prot_A$ is *compliant with* $Prot_B$ *on* S.

To illustrate the concept, let us again assume the alphabet $S = S_{prov} \cup S_{req}$, S_{prov} = {*?dbSrv.Insert↑, !dbSrv.Insert↓*}, S_{req} = { *!dbAcc.Query↑, !dbAcc.Insert↑*}. The protocol $Prot_A$ = *?dbSrv.Insert { !dbAcc.Query↑ }* is compliant with $Prot_B$ = *?dbSrv.Insert{ (!dbAcc.Query↑ + !dbAcc.Insert↑)* }* on S, since the provisions[4] of both $L(Prot_A)$ and $L(Prot_B)$ are the same and $Prot_A$ allows to issue only the requirement *dbAcc.Insert↑*, but $Prot_B$ does allow to issue requirements containing any number of *dbAcc.Insert↑* and *dbAcc.Query↑*.

3.4.3. Bounding behavior via protocols, model agents

In a special case, the behavior of an agent A is exactly the same as the behavior specified by a protocol Prot, i.e., $L_A = L(Prot)$. If this holds and if the alphabet of A is the same as the alphabet of Prot, we call A a *model agent* of Prot. It is important to note, that there is a model agent for any behavior protocol Prot. (Because of the regularity of L(Prot), one can always imagine the agent as a finite state machine with the provisions of L(Prot) as input and requirements as output.)

However, in a general case, L_A is non-regular and therefore it cannot be specified by a protocol exactly. In such a case, we approximate the behavior L_A by a protocol Prot (over an alphabet S_P) such that L_A is compliant with L(Prot) on S_P. Of course, the trick of a "close" approximation is two-fold: (a) we maximize the alphabet for considering the compliance of L_A and L(Prot) by asking $S_B = S_P$ in the compliance definition. This way we guarantee that none part of any trace of Prot will be skipped (b) we minimize the difference between L_A and L(Prot) captured by the conditions (1') and (2') by a suitable choice of Prot. A typical part of this choice is the selection of S_P close enough to S_A:

Definition: Let A be an agent with an alphabet S_A, Prot be a protocol over an alphabet S_P, such that $S_{P,prov} \subseteq S_{A,prov}$ and $S_{P,reg} \subseteq S_{A,reg}$ and $S_{P,int} \subseteq S_{A,int}$. We say *behavior of A is bounded by Prot on* S_P if A's behavior is compliant with the behavior of the Prot's model agent on the S_P.

Consequently, the conditions (1') and (2') of the compliance definition can be for behavior bounding rewritten as

(1") $L(Prot)/S_{P,prov} \subseteq L_A/S_{P,prov}$
(2") $L(Prot)/S_{P,prov} |S_{P,prov}| L_A/S_P \subseteq L(Prot)$.

---

[4]For convenience, we introduce the *provisions* resp. *requirements of a language* L over S_{prov} resp. S_{req} as the restriction L/S_{prov} resp. L/S_{req}.

If the behavior of an agent A is bounded by a protocol Prot on S_P, A has to "react" to any provision of L(Prot) by at least one trace from L_A/S_P with the same provision, and, moreover, all such traces have to be in L(Prot) as well. Therefore, if L(Prot) contains more than one trace with a given provision, L_A/S_P can include only some of those traces, i.e. L_A/S_P can be narrower than L(Prot). However, the behavior of A restricted to S_P is "limited" by Prot only in case of the provisions specified by Prot–the condition (1") indicates that there can be a trace the provision of which is not a provision of L(Prot). For such provisions the behavior L_A/S_P is not limited by Prot in any way (*protocol-neutral behavior* of A). Notice that if A is a model agent of Prot, (1) and (2) become equalities and A does not exhibit neither narrower nor Prot-neutral behavior.

As an example of behavior bounding, consider again the protocols and alphabet from Section 3.4.1. Let A be an agent with behavior consisting only of the trace *<?dbSrv.Insert↑, !dbAcc.Query↑, !dbAcc.Insert↑, !dbSrv.Insert↓>*. A's behavior is bounded by the protocol $Prot_B$ = *?dbSrv.Insert{ (!dbAcc.Query↑ + !dbAcc.Insert↑)* }* on S as the trace of L_A is in $L(Prot_B)$ and the only provision *<?dbSrv.Insert↑, !dbSrvInsert↓>* of $L(Prot_B)$ is also the only provision of L_A (thus A does not exhibit any protocol-neutral behavior). Moreover, L_A is narrower than L(Prot), since it does not include, e.g., any trace containing *!dbAcc.Query↑*.

3.4.4. Designing a hierarchy of agents

A system Σ can be specified as a collection of cooperating model agents, each of them with the behavior specified by a certain protocol and an alphabet S reflecting an agreement on the agent's cooperation with its environment in Σ. Top-down design starts by substituting the primitive top model agent TA by another, refined composed model agent RA (with internal agents at the next level of nesting) of a protocol which includes behavior specification of the interplay of these internal agents. Such refinement is recursively repeated until the low-level primitive model agents are specified. As a special case of refinement, a low-level primitive model agent TA is replaced by a "real" primitive agent RA (with "real implementation"), not being subject to a further refinement.

In the design process described above, there are three cases to be considered as a refinement of TA: (1) a primitive model agent TA is substituted by a "real" agent RA, (2) a primitive model agent TA is substituted by a composed model agent RA, (3) a composed agent TA is refined by substituting one of its primitive model subagent RA.

At any level of the hierarchy, we want an (originally primitive model) agent TA with the behavior specified by a protocol $Prot_{TA}$ to be (repeatedly) refined by agent substitution in such a way that the behavior of the modified TA remains bounded by $Prot_{TA}$; i.e., TA's refinement should preserve adherence to the original behavior specification.[5]

In the case (1), we simply ask the behavior of RA to be bounded by TA. As to (2), we can take advantage of the fact that the behavior of RA is specified exactly by a protocol $Prot_{RA}$ (RA is a model agent of this protocol). Here, by asking compliance of $Prot_{TA}$ and $Prot_{RA}$, we can ensure behavior bounding of RA by $Prot_{TA}$:

---

[5] Whenever a substitution of TA by RA is considered, we assume they have the same external alphabets, since, according to Lemma 3.4.1, bounding of RA's behavior by $Prot_{TA}$ implies substitutability of TA by RA provided their external alphabets are equal, i.e. $S_{TAprov} = S_{RAprov}$ and $S_{TAreq} = S_{RAreq}$.

Lemma 3.4.2: Let TA be a primitive model agent of $Prot_{TA}$ with the alphabet S_T. Let RA be a composed model agent of a protocol $Prot_{RA}$ with the alphabet S_R, and $S_{T,ext} = S_{R,ext}$. If $Prot_{RA}$ is compliant with $Prot_{TA}$ on $S_{T,ext}$, resp. $S_{R,ext}$, then RA is bounded by $Prot_{TA}$ on $S_{T,ext}$.

Proof: Employing the compliance of $Prot_{TA}$ and $Prot_{RA}$, it can be easily shown that the conditions (1'') and (2'') for bounding RA by $Prot_{TA}$ are valid since RA is a model agent of $Prot_{RA}$. □

The case (3) is more difficult. Let us assume there is a subagent SA of TA that is to be substituted by a SRA. Similarly to the cases (1) and (2) we ask SRA to be bounded by the protocol of SA. If SA is substituted by SRA, TA will be bounded by $Prot_{TA}$. Firstly, in the lemma, we show this for such a substitution in adjacent levels of an agent hierarchy. Then the theorem 3.4.4 presents the behavior bounding on any level of agent hierarchy:

Lemma 3.4.3: Let A, an agent with S_{ext}, be composed of subagents $Q_1,...,Q_n$. Let Q_k ($1 \leq k \leq n$) be an agent in an environment E determining $S_{ext,k}$. If Q_k can be substituted by another agent R_k in E (R_k's behavior is compliant with the behavior of Q_k on $S_{ext,k}$), then, by such a substitution, A is transformed into the resulting composed agent $\underline{A}$ and $\underline{A}$'s behavior is compliant with A's behavior on S_{ext}.

Proof sketch: For each provision p of L_A, there is an provision p_Q of a trace from $L(Prot_{Q_k})$ such that p contains $p_Q = p/S_{k,ext}$, i.e., the provision event tokens being a part of p (and ignoring internal events of A). Because of substitutability of R_k, p_Q is also an provision of L_{R_k} and as other subagents of A are not modified, the sequence of internal events in p is not modified either. Consequently, p is an provision of $L_{A'}$; therefore condition (1') of behavior compliance holds. Similarly, each provision p of a trace from L_A contains the provision $p_Q = p/S_{k,ext}$ of a trace from $L(Prot_{Q_k})$. From the substitutability of R_k it follows that there must be a trace t in L_{R_k} with provision p_Q such that it is also in L_{Q_k}. The trace t is, therefore, contained in a trace of $L_{\underline{A}}$ with the provision p. Again, because the other subagents are not modified, the internal communication of $\underline{A}$ is not modified either. Therefore, condition (2') of behavior compliance holds. □

By induction this can be generalized for any level of an agent hierarchy:

Theorem 3.4.4: Let A, an agent in Σ, be bounded by a protocol $Prot_A$ on S_{ext}. Let a primitive agent B be a subagent of A at any level of nesting and, at the same time, B be a model agent of a protocol $Prot_B$ with $S_{B,ext}$. If B is substituted by an agent B' and B's behavior is bounded by $Prot_B$ on $S_{B,ext}$, A becomes the agent $\underline{A}$ and $\underline{A}$'s behavior is also bounded by $Prot_A$ on S_{ext}.

Proof sketch: By induction on depth of agent nesting. *Induction Base*: If B is a direct subagent of A, the bounding of B's behavior by $Prot_B$ means compliance of B's behavior with the behavior of B (as B is a model agent of $Prot_B$). According to lemma 3.4.3, A's behavior is compliant with the behavior of A. *Induction step*: Let C be the direct subagent of A and B be a subagent of C at the i-th level of nesting relatively to A ($i>1$). By substituting B by B', the C agent becomes a C' agent with a compliant behavior (according to induction hypothesis). Thus, C can be substituted by C' in A and, according to lemma 3.4.3, A becomes an agent $\underline{A}$ having behavior compliant with A. □

In summary, this is of utmost practical importance: A bottom-up elaboration of a system Σ can be done via substitution of the primitive model agents by primitive agents with a "real

implementation" (provided their behavior is bounded properly), since this substitution induces an implicit elaboration of the agents at higher levels in Σ while preserving bounding of the behavior of these higher-level agents.

In Σ designed this way, a protocol-neutral behavior of any agent cannot be utilized, as the whole specification of Σ has been based on protocols (and thus model agents) and this inherently excludes a protocol-neutral behavior. Therefore, at a run of Σ, an agent A (which substituted a model agent B of a protocol Prot in B's environment E), gets from E the provisions from $L(Prot)/S_{prov}$ only. This property of agents and their environments in Σ is reflected by the following concept:

Definition: Let Prot be a protocol, A an agent, E its environment and S_{ext} its external alphabet in E. Let E dictate to A provisions from $L(Prot)/S_{prov}$ only. The agent A and the environment E *obey* a protocol Prot if every trace of A in E is in L(Prot).

It can be easily shown that (1) if E dictates to A only the provisions defined by Prot and (2) the behavior of A is bounded by Prot, then A and E obey Prot..

This can be used for run-time checks of a system to identify communication not captured by a protocol: If it si detected that a trace of A is not in L(Prot), then A does not obey Prot and therefore bounding A's behavior by Prot is violated (provided E dictated a "correct" provision).

4. Associating behavior protocols and SOFA components

In this section, as a proof of the concept, we show how the concept of bounded behavior can be embedded into the SOFA component model.

4.1. Components as agents

Modeling of SOFA components via agents is straightforward: Being an instance of T = <F, A>, a component C is associated with a *C agent* (one-to-one relationship). If A is primitive, the C agent is primitive; otherwise, the C agent is the composition of the agents of all subcomponents of C (recursively) as illustrated on Figure 4, employing the same graphical notation as in Figure 2. Here, every interface (and every tie) of a component is mapped as a part of a connection of the component agent; e.g. the binding of interfaces of C_1 and C_2 becomes the external connection CON_5 of the C_1 agent and C_2 agent, and thus the internal connection of C_{parent} (their parent agent). Similarly, the delegation on the provides-interfaces of C_{parent} and C_2 becomes the external connection CON_4 of both C_2 and C_{parent}. In principle, a connection is determined by a chain of the interface ties determined by the subsume, bind, and/or delegate clauses in the CDL specification of component templates used in C. The chain can be used for specification of unique connection names. In Figure 5, for example, D.C.B.A:IA–D.C.B:IB–D.C:IC →D.X:IX–D.X.Y:IY would be the name of the depicted connection (so, by convention, a connection name is a sequence of interface instance names separated by – resp. → to emphasize the kind of their tie, i.e., subsume or delegate, resp. bind). Connection names formed this way comprise the global name space (GNS) of the system (Section 3.1). However, when reasoning about behavior of a component C at its CDL specification level, we may know just a part of a connection, e.g., just one interface instance on it may be known at some point. As a consequence, to reason

about connections in CDL, partial renaming (by name enhancing) might be necessary if a connection is specified at different levels of abstraction granularity.

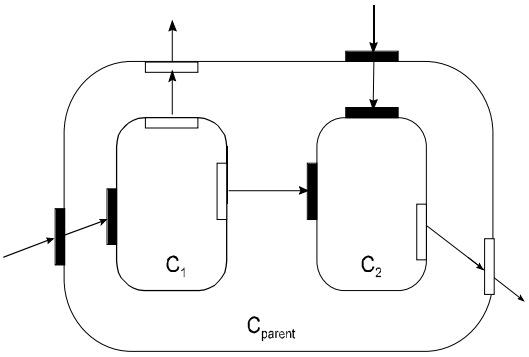

Figure 4a) Nested components

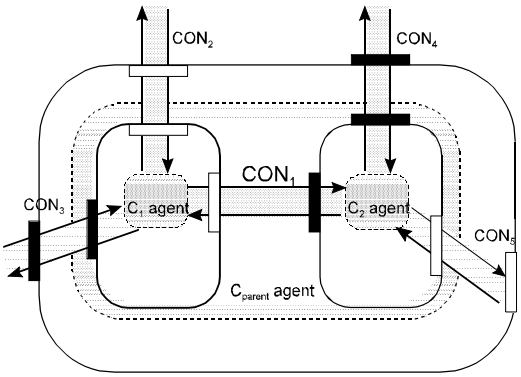

Figure 4b) Nested component as agents

The set of method names declared in the interface types participating in a connection comprise the local event name space of the connection. To reflect the method call style communication of SOFA components, a method call, e.g., m(...) issued by the component C_1 on the requires-interface mapped into the CON_5 connection bound to a provides-interface of the component C_2 (Figure 4), is modeled as the event token pair ..., $!CON_5.m\uparrow$,..., $?CON_5.m\downarrow$,... in a trace of the C_1 agent and ...$?CON_5.m\uparrow$... $!CON_5.m\downarrow$... in the corresponding trace of the C_2, agent. A call of a one-way method ow(...) would be modeled as an event token $!CON_5.ow\uparrow$ in a trace of the C_1 agent and $?CON_5.ow\uparrow$ in the corresponding trace of the C_2 agent. Note that if nested components are connected via a subsume resp. delegate tie (like C_1 and C_{parent} in Figure 4), no such event prefix modification (? → ! resp. ! → ?) takes place.

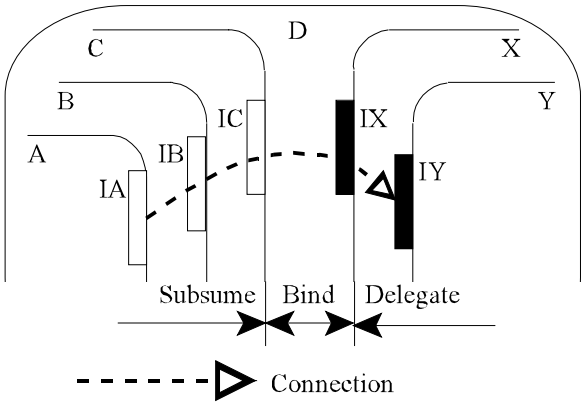

Figure 5. Creating connection name

4.2. Bounding behavior of components

As explained in Section 3, an agent's behavior can be bounded by a behavior protocol. Because of the one-to-one relationship of components and agents, the idea can be applied to components as well. Since the SOFA components are specified in CDL, any behavior protocol designed to bound the behavior of a component should be seamlessly integrated into CDL. For such an integration, the interface, frame, and architecture concepts form the natural abstraction units which can be associated with behavior protocols to bound behavior of a component at different abstraction levels (at different granularity of the component

agent's connections). Thus we introduce the concepts of interface, frame and architecture protocols.

The frame concept corresponds to a primitive agent featuring external connections only, whereas the architecture concept corresponds to a composed agent at the first level of agent nesting (thus considering also the internal connections at this level of nesting). The interface concept is an abstraction to restrict the view of an agent only to one of its connections. Consequently, for the purpose of the introduced protocols, the alphabets associated with these CDL abstractions are those indicated in Table 1. As each of the alphabets is determined implicitly from the related CDL concept, we will simply say that an interface (resp. frame, architecture) protocol bounds the behavior of a component (omitting the "on the ... alphabet" phrase).

From the concept of bounded behavior (Section 3.4.3), it follows that if the behavior of a component C is bounded by a frame protocol, C can provide protocol-neutral behavior on its provides-interfaces and, therefore, C can feature "richer" functionality on provides-interfaces than specified by the frame protocol. Moreover, the behavior of C can be more "strict" on requires-interfaces than the behavior specified by the frame protocol. Similar reasoning can be applied to the architecture and interface protocol concepts as well.

CDL concept	**behavior protocol**	**alphabet S** (the event tokens on:)		
		$\mathbf{S_{ext}}$		$\mathbf{S_{int}}$
		$\mathbf{S_{prov}}$	$\mathbf{S_{req}}$	
frame F	frame protocol	F's provides-interfaces	F's requires-interfaces	⊘
architecture A (of frame F)	architecture protocol	F's provides-interfaces	F's requires-interfaces	interfaces bound in A
provides-interface I_P	interface protocol	I_P	⊘	⊘
requires-interface I_R	interface protocol	⊘	I_R	⊘

Table 1. Associating protocols and CDL concepts.

4.3. Frame protocol

Frame protocol is a behavior protocol specifying the acceptable interplay of method invocations on the provides-interfaces and reactions on the requires-interfaces of the frame (recall that curly brackets can be used to express nesting of the method calls (Section 3.3)). In an event token, the name of an event is qualified by the name of the interface instance the invoked method belongs to, and is prefixed by ? (accepting a call on a provides-interface) or ! (issuing a call on a requires-interface).

```
// Database frame protocol

!dbAcc.Open ;
(   ?dbSrv.Insert { ( !dbAcc.Insert ;
        !dbLog.LogEvent )* }
    +
    ?dbSrvDelete { ( !dbAcc.Delete ;
        !dbLog.LogEvent )* }
    +
    ?dbSrvQuery { !dbAcc.Query* }
)* ;
!dbAcc.Close
```

```
frame DatabaseBody {
    provides: IDBServer d;
    requires: IDatabaseAccess da;
              ILogging lg;
              ITransaction tr;
    protocol:
        !da.Open ;
        (   ?d.Insert { !tr.Begin ; !da.Insert ;
                !lg.LogEvent ; ( !tr.Commit + !tr.Abort ) }
            +
            ?d.Delete {! tr.Begin ; !da.Delete ;
                !lg.LogEvent ; (!tr.Commit + !tr.Abort ) }
            +
            ?d.Query { !da.Query }
            +
            ?ds.SetTrModel { !da.SetTrModel }
            +
            ?ds.GetTrModel { !da.GetTrModel }
        )* ;
        !da.Close
};
```

To illustrate the frame protocol concept and the related CDL syntax, we present the frame protocols of *Database* and *DatabaseBody*. In the *Database* frame protocol, the fact that each modification of the database should be logged is reflected in the following way: inside every *dbSrv.Insert* invocation, any number of *dbAcc.Insert* calls can be executed, and after each of these calls is finished, the modification is logged by invoking *dbLog.LogEvent*. Similarly, as a part of every *dbSrv.Delete* invocation, deleting is logged by *dbLog.LogEvent*.The *DatabaseBody* frame presents how a frame protocol is employed in the CDL specification.

4.4. Architecture protocol

For a template T=<F,A>, *architecture protocol* is a behavior protocol describing the "grey-box" behavior of T. It is based on the frames of the direct subcomponents specified in A. The protocol describes the interplay of the method invocations on the interfaces of F and the outmost interfaces of the subcomponents in A. Our approach is not to specify an architecture protocol in CDL directly, but to generate it by the CDL compiler–by combining the frame protocols of the subcomponents via the composition operator ($\sqcap_X$, Section 3.3). In an architecture protocol, the set X of $\sqcap_X$ is composed of all the events on the interfaces appearing in the bind clauses of the corresponding architecture, and can be inferred from the specification of the architecture.

To illustrate what a generated architecture protocol looks like, consider the *Database* architecture version *v2* which contains two subcomponents: *Transm* (an instance of *TransactionManager*) and *Local* (an instance of *DatabaseBody*). The frame protocols are modified by using enhanced names of interface instances/connections in order to unify their identification. For example, a declaration *bind Local:tr to Transm:trans* results in using <*Local:tr* → *Transm:trans*> instead of *trans* in the *TransactionManager* frame protocol

and instead of *tr* in the *DatabaseBody* frame protocol. After applying the composition operator, the architecture protocol of *Database* version *v2* takes the form:

```
(   ?<Local:tr → Transm:trans>.Begin ;
    ( ?<Local:tr → Transm:trans>.Commit + ?<Local:tr → Transm:trans>.Abort )
) *
⌜⌝
!<Local:da-dbAcc>.Open ;
(   ?<dbSrv-Local:d>.Insert {
        !<Local:tr → Transm:trans>.Begin ; !<Local:da-dbAcc>.Insert ;
        !<Local:lg-dbLog>.LogEvent ;
        ( !<Local:tr → Transm:trans>.Commit + !<Local:tr → Transm:trans>.Abort ) } +
    ?<dbSrv-Local:d>.Delete {
        !<Local:tr → Transm:trans>.Begin ; !<Local:da-dbAcc>.Delete ; <Local:lg-dbLog>.LogEvent ;
        ( !<Local:tr → Transm:trans>.Commit + !<Local:tr → Transm:trans>.Abort ) } +
    ?<dbSrv-Local:d>.Query { !<Local:da-dbAcc>.Query } + ?<Local:ds>.SetTrModel { !<Local:da-
        dbAcc>.SetTrModel } +
    ?<Local:ds>.GetTrModel { !<Local:da-dbAcc>.GetTrModel }
)* ;
!<Local:da-dbAcc>.Close
```

In this example, the X set of $\sqcap_X$ comprises the event names on the interfaces/connection *<Local:tr → Transm:trans>*, i.e., *X = { < Local:tr → Transm:trans >.Begin, <Local:tr → Transm:trans >.Commit, < Local:tr → Transm:trans >.Abort }.*

The automatic generation works for any frame protocols of subcomponents. However, this approach hides (encodes) simple dependencies among internal subcomponents, which when writing the protocol by hand, could result in a more readable form of the architecture protocol, particularly because of not necessarily using the composition operator.

4.5. Interface protocol

Interface protocol is a behavior protocol specifying the acceptable order of method invocations on an interface. It is intended to simplify a component design as it represents the behavior of the component on a single interface only. Although the behavior on an interface instantiated in a frame is also reflected in the frame protocol, this redundancy provides support for incremental specification of the component and, in particular, helps check the correctness of the interface ties. In principle, if a protocol is associated with a provides-interface resp. requires-interface, a method invocation is to be prefixed by ?, resp. !. As a particular interface type can be used for instantiating both the provides-interface and requires-interface, the interface protocol associated with the interface type is written in its generic form in CDL, i.e., neither ? nor ! prefixes are included; the prefixes are automatically added when an instance of the interface type is created. Such an instantiation, forming a *provides-protocol* resp. *requires-protocol*, also introduces the interface instance identification in event tokens, as illustrated below.

Consider an interface protocol associated with the *IDatabaseAccess* interface type from Section 2.2. The intended use of this interface type is to call the method *Open* first, then allow for modification of the database by invocations of *Insert*, *Delete*, *Query* or to modify

the configuration of the database by invoking *GetTrModel* and *SetTrModel*. Finally, *Close* should be invoked to finish the work with the database. The corresponding interface protocol can take the form *Open ; (Insert + Delete + Query + GetTrModel + SetTrModel)* ; Close*. As an aside, if the database modification methods were to be designed to handle requests in parallel, we could specify this intention by *Open ; (Insert || Delete || Query || (GetTrModel + SetTrModel))* ; Close*. In this case, the requires-protocol associated with the *dbAcc* instance of the *DB* component from Figure 2 would take the form: *!dbAcc.Open ; (!dbAcc.Insert || !dbAcc.Delete || !dbAcc.Query || (!dbAcc.GetTrModel + !dbAcc.SetTrModel))* ; !dbAcc.Close*.

5. Protocol conformance

5.1. Definition of protocol conformance

Intuitively, in a template T = <F, A>, the architecture protocol of A should follow the design intentions embodied in the frame protocol of F, and the interface protocols of the interfaces in F and A should comply with the way these interfaces are employed in the frame protocol and architecture protocol. Basically, employing protocol compliance (Section 3.4.1) is a natural way to reflect the desired correspondence in behavior description. However, the behavior protocols associated with CDL incorporate connections at different levels of abstraction which imply that reasoning on these protocols may require name unification (Section 4.1). For this purpose, we introduce the notion of *qualification of a protocol* $Prot_X$ with respect to a CDL abstraction Y (denoted as ${}^{Y}Prot_X$) which means that any name of an interface instance/connection in $Prot_X$ associated with the CDL abstraction X is modified (enhanced) to that used in Y for the same interface instance/connection.

There are three basic scenarios to consider with respect protocol compliance in CDL: (1) interface protocol vs. interface protocol, (2) interface protocol vs. frame protocol, (3) frame protocol vs. architecture protocol. To address the fact that direct evaluation of such compliance is not possible (without name unification and potentially some other slight protocol modifications), we introduce the concepts of CDL protocol conformance:

Definition: Let I_1 and I_2 be two interface types with interface protocols P_{I_1} and P_{I_2}, and S the alphabet of I_1 (and I_2 as well). We say that *the interface protocol* P_{I_1} *conforms to the interface protocol* P_{I_2} if the protocol P_{I_2} is compliant with the protocol P_{I_1} on S.

In other words, for a given interface type I and its behavior protocol P_I, another interface protocol P_J conforms to P_I only if it can generate a language which is a subset of the language generated by P_I. Thus, informally, using I as specified in P_J implies not violating the behavior specified by P_I. In this respect, the "direction" of conformance follows the "direction" of the delegate → bind → subsume tie chain. For instance, in Figure 2 the delegation *dbSrv - d* imposes the interface protocol of *dbSrv* to conform to the interface protocol of *d* and the binding *tr → trans* imposes the interface protocol of *tr* to conform to the interface protocol of *trans*.

Definition: Let T = <F, A> be a template with the frame protocol P_F, I_P a provides-interface of F with the provides-protocol P_{I_P}; let S be the alphabet associated with I_P. We say that *the frame protocol* P_F *conforms to the provides-protocol* P_{I_P} if the protocol P_F is compliant with the protocol ${}^{F}P_{I_P}$ on S.

Definition: Let T = <F, A> be a template with the frame protocol P_F, I_R a requires-interface of F with the requires-protocol PI_R; let S be the alphabet associated with I_R. We say that *the frame protocol* P_F *conforms to the requires-protocol* PI_R if ${}^F PI_R$ is compliant with P_F on S.

The intuition behind the definition is that whatever the frame protocol allows to do on a provides-interface I_P, it has to allow it in such a way that a component based on T may exhibit at least the events as specified by the interface protocol of I_P. Similarly, whatever the interface protocol of a requires-interface I_R allows to do, it has to allow it in such a way that a component based on T may exhibit at least the events as specified by the frame protocol on I_R.

Definition: Let T = <F, A> be a template with the frame protocol P_F and the architecture protocol P_A. We say that *the architecture protocol* P_A *conforms to the frame protocol* P_F if P_A is compliant with ${}^A P_F$ on S where S is the alphabet associated with F.

Less formally, this definition expresses that the architecture protocol of A cannot generate traces not allowed by the frame protocol of F, under the assumption that the provides-interfaces known in F are used in A in a way the frame protocol allows for. At the same time, the architecture protocol can be "less demanding" on the requires-interfaces.

In summary, in a correctly designed template T = <F, A>, the architecture protocol P_A conforms to the frame protocol P_F; at the same time, P_F conforms to the provides-protocol PI_P (of a provides-interface I_P) and requires-protocol PI_R (of a requires-interface I_R). Moreover, the definitions of protocol conformance imply (as presented in [21]) that, if conforming, these protocols form a hierarchy.

5.2. Examples

In this section, we present examples of interface vs. frame protocol conformance and frame vs. architecture protocol conformance in the template T = <*Database*, *Database* version *v2*> from Section 2.2.

In T the interface protocol of *dbAcc* being an instance of *IDatabaseAccess* has to conform to the frame protocol of *Database*. Before verifying the protocol conformance, the requires-interface protocol of *dbAcc* at the frame level has to be derived from the interface protocol; it takes the form:

!dbAcc.Open **;** **(** *!dbAcc.Insert* **+** *!dbAcc.Delete* **+** *!dbAcc.Query* **+** *!dbAcc.GetTrModel* **+** *!dbAcc.SetTrModel* **)*** **;** *!dbAcc.Close.*

This protocol (in this section denoted as P_I) has to conform to the frame protocol of *Database*. Even though the compliance relation is based on language restriction, we use in this section protocol restriction instead of language restriction, since they are semantically equivalent and the protocol restriction allows for a direct comparison of protocols. Therefore, the frame protocol restricted to the *dbAcc* interface (denoted as P_F/I in this section) takes the form:

!dbAcc.Open **;** **(** *!dbAcc.Insert* **+** *!dbAcc.Delete* **+** *!dbAcc.Query* **)*** **;** *!dbAcc.Close.*

P_I has to be compliant with P_F/I on the external alphabet S_{ext} = {*!dbAcc.Open*↑, *!dbAcc.Insert*↑, *!dbAcc.Delete*↑, *!dbAcc.Query*↑, *!dbAcc.GetTrModel*↑, *!dbAcc.SetTrModel*↑, *dbAcc.Close*↑, *?dbAcc.Open*↓, *?dbAcc.Insert*↓, *?dbAcc.Delete*↓,

?dbAcc.Query↓, *?dbAcc.GetTrModel*↓, *?dbAcc.SetTrModel*↓, *?dbAcc.Close*↓ }. Recalling the definition of protocol compliance, we have to verify the inclusion $L(P_F/I) \subseteq L(P_I)$ only, since the set of provisions is empty for a requires-protocol (Table 1) and, therefore, (1') holds trivially and (2') is simplified this way. Informally, considering the inclusion above, any trace t from $L(P_F/I)$ starts with invocation of *dbAcc.Open* followed by a sequence of invocations of *dbAcc.Insert, dbAcc.Delete, dbAcc.Query* and the last two event tokens of t represent an invocation of *dbAcc.Close*. As all traces from $L(P_I)$ follow this pattern, we can claim that the frame protocol of *Database* conforms to the requires-protocol of the *dbAcc* interface.

Now, consider an example of the frame vs. architecture protocol conformance. Let P_A denote the architecture protocol of *Database* version *v2* and P_F the frame protocol of the *Database* frame. We have to verify that P_A is compliant with the qualified frame protocol ${}^{A}P_F$ on the alphabet comprising all the event tokens on interfaces of the frame, i.e., S_{prov} comprises all event tokens on <*dbSrv-Local:d*> and <*Local:ds*>, S_{req} comprises all event tokens on <*Local:da-dbAcc*> and <*Local:log-dbLog*> and S_{int} comprises all event tokens on <*Local:tr* → *Transm.transm:trans*>. First, we have to qualify the frame protocol with respect to *Database* version *v2*:

!<Local:da-dbAcc>.Open **;**
(*?<dbSrv-Local:d>.Insert* **{ (** *!<Local:da- dbAcc>.Insert* **;** *<Local:lg-dbLog>.LogEvent* **) * }**
+
?<dbSrv-Local:d>.Delete **{ (** *!<Local:da-dbAcc>.Delete* **;** *<Local:lg-dbLog>.LogEvent* **) * }**
+
?<dbSrv-Local:d>.Query **{ (** *!<Local:da-dbAcc>.Query* **) * }**
) * ;
!<Local:da-dbAcc>.Close

As follows from Section 5.1, P_A is compliant with P_F if (1) $L({}^{A}P_F)/S_{prov} \subseteq L(P_A)/S_{prov}$ and (2) $L({}^{A}P_F)/S_{prov}|S_{prov}|L(P_A)/S_{ext} \subseteq L({}^{A}P_F)/S_{ext}$ hold. Condition (1) can be verified by comparing the following protocols (again, restricted to S_{prov} by the protocol restriction instead of language restriction):

// Database frame protocol restriction

(*?<dbSrv-Local:d>.Insert* **+**
?<dbSrv-Local:d>.Delete **+**
?<dbSrv-Local:d>.Query **) ***

// Database architecture protocol restriction

(*?<dbSrv-Local:d>.Insert* **+**
?<dbSrv-Local:d>.Delete **+**
?<dbSrv-Local:d>.Query **+**
?<Local:ds>.SetTrModel **+**
?<Local:ds>.GetTrModel **) ***

As can be easily shown, the languages generated by these protocols are in the required inclusion. Similarly, the operators of adjustment and restriction in condition (2) imply the protocols below are to be compared for inclusion. As these protocols are "almost identical", since the architecture protocol uses the requires-interfaces/connection <*Local:da-dbAcc*> and <*Local:lg-dbLog*> without repetition compared to the frame protocol, they satisfy (2). Thus, the architecture protocol of *Database* version *v2* conforms to the frame protocol of the *Database* frame.

```
// Database frame protocol

!<Local:da-dbAcc>.Open ;
  ( ?<dbSrv-Local:d>.Insert { ( !<Local:da-
dbAcc>.Insert ; <Local:lg-dbLog>.LogEvent)* }
  +
  ?<dbSrv-Local:d>.Delete { ( !<Local:da-
dbAcc>.Delete ; <Local:lg-dbLog>.LogEvent)* }
  +
  ?<dbSrv-Local:d>.Query { ( !<Local:da-
dbAcc>.Query)* }
) * ;
!<Local:da-dbAcc>.Close
```

```
// Database architecture protocol

!<Local:da-dbAcc>.Open ;
( ?<dbSrv-Local:d>.Insert { !<Local:da-
dbAcc>.Insert ; <Local:lg-dbLog>.LogEvent } +
  ?<dbSrv-Local:d>.Delete { !<Local:da-
dbAcc>.Delete ; <Local:lg-dbLog>.LogEvent } +
  ?<dbSrv-Local:d>.Query { !<Local:da-
dbAcc>.Query }
) * ;
!<Local:da-dbAcc>.Close
```

6. Benefitting from behavior protocols

6.1. Behavior protocols and component lifecycle

Behavior protocols can contribute to the correctness of a component design in as follows: At the assembly design stage, an application is composed as a hierarchy of components; i.e. nested template instances. If these templates were associated with protocols during design in an ADL, as illustrated in Section 4, building up this hierarchy top-down can be viewed as a systematic, top-down composition/refinement of the model agents of these protocols as discussed in general in Section 3.4. Specific to SOFA, a model agent of a frame protocol is replaced by a model agent of an architecture protocol involving some internal frame protocols' agents. These internal agents get further replaced by some architecture protocol agents, etc. Recalling left side of Figure 1, a model agent of the protocol $P_{F_{main}}$ (of the frame F_{Main}) would be replaced by a model agent of the architecture protocol P_{A_2} involving model agents of P_{FSub_1} and P_{FSub_2}. As $ASub_1$ resp. $ASub_3$ are primitive architectures, the model agent of P_{FSub_1} resp. P_{FSub_2} will be substituted by a primitive agent modeling the architecture $ASub_1$ resp. $ASub_3$. As follows from Theorem 3.4.4 provided the frame – architecture – frame ... protocols conformance has been successfully validated in such a model agent hierarchy, the behavior of any component C in the hierarchy is bounded by its frame protocol if the behavior of the primitive components recursively nested in C is bounded by their frame protocols.

The frame – architecture – frame ... protocols conformance can be advantageously verified beforehand at the development and provision stage by checking the frame-architecture protocol conformance for each template specification. Considering a template T = <F,A>, the architecture protocol P_A of A has to conform to the frame protocol P_F of F; similarly, for every interface I in F, the interface protocol of I has to conform to P_F. Naturally, the interface ties specified in A are also subject of the interface protocol conformance requirement. With respect to design efficiency, the following is the recommended order of protocol conformance verification tests (1) Design of F: interface protocols vs. P_F conformance, (2) Design of A_i : conformance of the interface protocols in ties of the interfaces in A. (3) P_A vs. P_F conformance. Advantageously, these tests can be done algorithmically (Section 6.2).

Since there are no means to verify bounding behavior of a primitive component C by its frame protocol Prot statically, it has to be done at runtime by applying the protocol obeying concept (Section 3.4); this means checking whether in a particular run the component its trace is in L(Prot). If this is not the case, it can be so for one of the following reasons: (1) the provision chosen by C's environment is not in the provisions of L(Prot) so that the violation is caused by an "incorrect" environment; (2) C's behavior violates the bounding by Prot.

To check for violations of obeying of Prot, a parser for L(Prot) – *protocol guard*, a finite state machine in principle – is to be built and furnished with all the events on the component's interfaces (these can be obtained, e.g., via interceptors similarly to [17]). The protocol guard follows the current prefix of the trace and indicates any violation. If, at the end of a run, the protocol guard is not in an accepting state, it indicates a violation, as the prefix parsed so far is not a trace in L(Prot). Otherwise, this run supports the hypothesis that the component and its environment obey the protocol and the behavior of the component is bounded by Prot. Of course, similarly to testing, a run time check cannot ensure "full" correctness of the implementation. However, both the protocol guard construction and the obeying validation can be done algorithmically (Section 6.2), and, in general, at any level of component nesting (not only for primitive components).

6.2. Behavior protocols and finite state machines

As mentioned in Section 3.3, behavior protocols generate regular languages. Because the definition of protocol conformance is based upon inclusion of languages, its verification at the design time can be performed in an algorithmic way, for example via comparing finite state machines [30]. Similarly, the runtime checking whether a trace is in the language of a protocol can be done via an automatic parser (Table 2).

	Run time checks	**Design time verifications**
What	bounded behavior via obeying	protocol conformance
How	protocol guard	CDL compiler
Target	implementation	design

Table 2. Checks and verification

Here, the problem of space and time complexity of the finite state machines (automata) for recognizing languages arises. In general, the classical regular language operators (concatenation, alternative, repetition) do not introduce any exponential growth of the state space of a parsing finite state automaton. However, behavior protocols employ also the and-parallel, composition, and adjustment operators that introduce exponential complexity of the resulting automata which might lead to the state explosion problem. In fact, the composition and adjustment operators "behave" better than the and-parallel operator in terms of the required state space as they comprise synchronization of events, thus reducing the interleaving of traces.

In SOFA, the state explosion problem is targeted by factoring the state space – performing the task of checking the conformance of protocols at multiple levels of abstractions which results in a substantial reduction of the state space to be considered. For example, once the conformance of an architecture protocol P_A to a frame protocol P_F is verified, it is not necessary at higher levels of component nesting to manipulate with (typically more complex) P_A; instead, P_F will do. This way, a typically less complex finite state machine corresponding to P_F is used instead of the more complex one corresponding to P_A.

7. Evaluation and open issues

Model and expressiveness. The agent model of communication provides a formal framework for defining the behavior description of components. A behavior is represented as a set of traces, i.e., sequences of communication events on agents' connections. The concept of connection allows to model interface-based communication for provides-interfaces, requires-interfaces and mixed interfaces (comprising both provided and required methods as in [1, 2,12]). Central to the model is the concept of behavior compliance. It is based on enforced communication meaning that an agent/component has to accept all the provisions chosen by its environment, while allowing to choose its requirements as a reaction.

The behavior compliance forms the basis for capturing (a) agent/component substitutability; (b) component implementation adherence to its specification via behavior bounding (meeting the goal (3)); (c) "reasonable correspondence" of selected protocols via the protocol compliance concept. In principle, (b) and (c) are involved in an elaboration of a component (Section 3.4). Contrary to the classical notion of behavior refinement [4, 23] which asks the result of a refinement to behave more deterministically, refinement in terms of our compliance is different: Considering refinement of a component with provisions and requirements, its behavior should be more deterministic on the requirements only. As to provisions, less deterministic behavior is expected (see also [22]). In addition, the behavior containing provisions added by the refinement is not considered in the compliance evaluation (protocol neutral behavior)..

Addressing the goal (1), the specification of components uses behavior protocols, featuring intuitively easy-to-comprehend notation and regularity of the languages generated by them. As to expressive power, since behavior protocols do not include conditional branching, access to the return value of a method invocation, etc., they only approximate the functionality/algorithm of a component. Balancing the expressive power and the simplicity of behavior protocols, we believe that an elegant and easy-to-read notation can outweigh some loss in algorithmic expressiveness and justify the application of behavior protocols in ADLs. A weakness in expressiveness of behavior protocols is the absence of deterministic/non-deterministic choice as defined in CSP [23]. For example, in the frame protocol *(?P.a;?P.b) + (?P.a;!R.x)* once the component has chosen to issue *!R.x*, an external invocation of *P.b* cannot take place. The problem here is the inability of behavior protocols to pass to the environment the information about a choice made internally. To address the problem, we consider employing a special "signal" event in a protocol, e.g., *(?P.a;!P.S~;?P.b) + (?P.a;!R.x)*. Here, the component announces the possibility of *?P.b* by emitting the signal *!P.S~*. Another option might be employing exceptions. Nevertheless, we

admit the internal/external choice problem could be more easily solved by another equivalence semantics [10] based not only on traces, such as CSP failures [23]. On the other hand, the complete trace semantics employed in behavior protocols does not require any assistance of the component itself at runtime checking, as would be the case when exploiting more advanced semantics.

Contribution to component design. As a proof of the concept, we have applied behavior protocols to the SOFA component model, introducing the interface, frame, and architecture protocols to bound component behavior at different levels of abstraction. To formally capture protocol compatibility in component templates, we introduced the protocol conformance relation based on protocol compliance. It allows for reasoning about template design and supports refinement – the black-box view of a component (frame) can be refined via specifying more elaborated, composed architectures while preserving the frame specification (meeting the goal (2)). It should be emphasized that the architecture protocol vs. frame protocol conformance guarantees that the architecture does on its outmost interfaces (external connections) what was specified by the frame protocol, however, it does not guarantee the communication of internal components on the internal bind ties (internal connections) actually proceeds. In this respect, the conformance of interface protocols on the component binding ties can contribute to avoiding such pathological internal communication.

The concept of protocol-neutral behavior addresses advantageously the case when not all the interfaces of a component are tied in the parent architecture – it is important to allow this when reusing 3rd party/off-the-shelf components.

To decrease the burden of writing trivial protocols, using a default protocol in a CDL specification should be considered. For an interface, the default protocol might specify any order of sequential invocation of all the methods of the interface. For example, *(LogEvent + ClearLog)** might be the default protocol for the *ILogging* interface from Section 2.2. For a frame, the default frame protocol might model a passive, single-threaded component allowing for mutually exclusive calls of the methods on all the provides-interfaces, and, as the reaction on such a call, any sequence of requires-interface method invocations is permitted. For example, a generic frame protocol for the *Database* frame might be *(?dbSrv.Insert{ RP } + ?dbSrv.Delete { RP } + ?dbSrv.Query{ RP })**, where *RP* denotes protocol for any sequence of invocation of any method on the *dbLog* or *dbAcc* requires-interfaces. As an architecture protocol is always generated, there is no need for a default one.

Tools. In the SOFA prototype, a CDL compiler (implemented as a module for Sun Forte/NetBeans for Java IDE) is available [25]. The compiler automatically generates architecture protocols and tests the interface, frame, and architecture protocol conformance. Protocols specified in templates are stored in a template repository for further elaboration, e.g., architecture protocol generation/conformance checks. We also plan to provide a generic protocol guard implementation (parameterized by a protocol) to be hooked to a component wrapper resp. interface interceptor to check obeying of the corresponding protocol(s) and report any violations, e.g., via exceptions.

State explosion. We address it by factoring the protocols' state space via handling protocol conformance/run time checks at different levels of abstraction separately (meeting

the goal (4)). Our largest case study currently available [25] consists of 20 interfaces, 30 frames, and 8 composed architectures. The verification of all the interface vs. frame protocol and all the frame vs. architecture protocol conformance took less than a minute; here, the most demanding conformance test (requiring about 23000 states) took about 8 seconds to finish on an 800MHz Pentium III with 512 MB of memory under Sun JDK 1.3.

Experimentally, our current implementation fails due to memory being exhausted for 12 and-parallel operator instances generating about 1.5 million states. Putting aside the option to increase this number by a more efficient implementation, an issue is what a typical/practical degree of parallel operators in a template is (as there is always a remedy to the problem in an architecture refinement leading to a finer protocol factoring). While experimenting with the case study, we have identified that the use of parallel operators tends to be high if the specification deals with in principle session-oriented interfaces. We believe the problem in fact originates in the SOFA component model, where the structure of a component and its interfaces is static, i.e., not modifiable at run time. Consequently, the session-oriented interfaces (requiring a per client instance), are reflected as a single, fully parallel interface to be accessed in multiple sessions simultaneously .

Future intentions and challenges: (1) We consider introducing a predicate operator for constraining a part of a protocol. This could help to better understand a component's semantics; however it is not clear if such a predicate should be formed of parameters and return values featured in method signatures, or if some abstract properties should be defined to reflect the internal state of the component. (2) Not having considered protocol inheritance, we face the challenge to enhance the sound enrichment technique [19] to reflect protocol conformance. (3) Versioning of architectures is considered in SOFA. The factors for version compatibility decisions could include compliance of architecture protocols. (4) A SOFA component can be updated at run time. An updating-related issue is to express "points of safe updating" in the frame or architecture protocols indicating when a system reconfiguration call is possible. (5) Addressing the internal/external choice problem mentioned above.

8. Related work

Probably the closest to our work is the Wright language [1, 2]. In Wright, the behavior of components and connectors is specified as a "computation", resp. a "glue", via a CSP-based notation (a system of recursive equations). In our opinion, regular-like expressions are more readable and better structured, with their expressive power strong enough to reasonably approximate the behavior of components. In Wright, event-based interfaces, called roles, are mixed in terms of an interface featuring both emitted and absorbed events. Even though our agent model can handle this approach as well, the behavior compliance concept has been designed particularly for the partially enforced communication inherent to distinguishing the provided and required features on an interface. Also, in Wright, a component specification includes explicit behavior specification only for primitive components. As to composed components, their architecture behavior descriptions are generated via the composition operator, bottom-up, making the behavior description fully "white-box" based. Thus, there is no black-box nor grey-box view of the composed component. In our approach, frame and architecture abstractions with corresponding protocols are introduced for this purpose; we consider these features very important for

refinement-based design of components, component updating, and addressing the state explosion problem. Being strictly focused on design time, Wright does not address any behavior checks related to run time.

In TRACTA [8, 9], the behavior of a component is described in FSP (Finite State Processes), a formal vehicle similar to CSP in terms of being a system of recursive equations. A component specification includes explicit behavior specification only for primitive components. For composed components, TRACTA uses the same approach as Wright, i.e., generating an architecture process via the composition operator, bottom-up. Consequently, to verify the behavior of a component, the behavior specification of all the primitive subcomponents is to be taken into account. To tackle the state explosion problem, TRACTA employs the Compositional Reachability Analysis (CRA) to restrict behavior observation at a specific level of architecture description to a particular event subset, e.g. by hiding all internal communication of a component. On the contrary, in our approach, the state explosion is solved via multiple layers of the frame and architecture abstractions by verifying, at each of those layers, only if the architecture protocol conforms to the frame protocol. The resulting behavior description state complexity of both the CRA and SOFA approaches should be very similar. However, in SOFA the behavior of components is known before all the nested primitive subcomponents (recursively) are available/designed. Also, a top-level behavior specification can be alternatively refined into several architectural variants – this cannot be achieved in CRA. In other words, in the CRA approach, there is only the "real behavior" determined by the actual composition of a composed component, while in SOFA, there is both the expected behavior and real behavior of the component and the option to test whether the real behavior corresponds to the expected one (protocol conformance).

The work on interfaces and protocols [28] is similar to our approach in terms of describing communication between component interfaces. It concentrates, however, only on the behavior description related to a single pair of collaborating interfaces. The specification of a component as a whole is not considered; consequently, no concepts similar to our frame and architecture protocols are present. Thus, the protocol description in [28] is less suitable for reasoning about component composition, replacement, etc.

The UML collaboration, interaction, and state diagrams [29] are used for a semi-formal description of object behavior. Nevertheless, collaboration and interaction diagrams can describe only "important" traces of execution so that they cannot be used for a complex description of component behavior. Although state diagrams are, in principle, finite state machines, having thus the same expressive power as regular expressions, there is no support in UML for their combining via a composition operator (similar to $\sqcap$) – a recognized powerful tool for reasoning about the components composed of subcomponents (Wright, TRACTA, etc.). In UML-related ROOM [24], the approach chosen is similar to the collaborating interfaces [28]; however, communication is not limited to a pair of interfaces. Also, a protocol conformance (called role substitutability) is, only briefly, outlined in [24].

Reuse contracts [26] introduce the idea of specifying the set of internally invoked methods for each method of an interface, thus capturing the invocation dependencies among methods. However, the model presented in [26] is limited in the sense that since it provides description only at the object level of abstraction, it does not support any component-based approach featuring more cooperating interfaces.

In Rapide [12], the behavior of a component (module) interface is specified via executable, pattern-based reactive rules, typically describing the relation between data received and sent. Similarly to Wright, the interfaces are mixed. The underlying formal framework is based on a poset execution model representing an execution as a partial ordered set of events. A behavior can be simulated and the resulting poset checked whether it satisfies the specific constraints (features desired properties). Nevertheless, the authors do not address automated verification of behavior conformance in an architectural hierarchy; instead, they provide the option of mapping a description to another one (such a mapping is to be specified manually).

In [4], a software component is modeled as an I/O function transforming input streams of events to set of possible output streams (modeling non-deterministic behavior). Similar to our notion of language compliance is the refinement of an I/O function. Roughly speaking, a function refines another function if it maps an input stream to less output streams, i.e., it is more deterministic. A special case of refinement, glass-box refinement, is exactly the way we use language compliance in our work, i.e., a primitive component is refined by a composed component. However, our agent model is closer to programming concepts. In particular, our provisions modeling component provides interfaces, inherently contain events both emitted and absorbed, which would have to modeled at a lower level of abstraction - via pairs of input and output streams in [4] (similarly for requirements).

In [22], component behavior is described as a provides-automaton (protocol of the only interface provided by a component) and a set of function-requires-automata (each of them modeling one method of a provides-interface). This way, there is a protocol for the whole provides-interface, but not for the requires-interface of the component. The behavior of the component is derived by inserting function-requires-automata into provides-automaton. The resulting component-requires-automaton is conceptually similar to our frame protocol, but the frame protocol describes the interplay on provides- and requires-interfaces in a form directly visible in the specification. Substitutability and behavior similarity are defined for these automata. However, any situation captured in our model as protocol neutral behavior, violates behavior similarity in [22].

9. Summary

This paper introduces a novel technique for the specification of component behavior via behavior protocols which take a form similar to regular expressions. The description of component behavior by means of behavior protocols is precise enough to capture the necessary requirements in terms of describing the desired ordering of method calls on the component's interfaces. In addition, it is easy to read and apply.

Further, the paper presents a model for the behavior protocol-based description of hierarchical software components. The protocol compliance concept provides formal means for capturing component substitutability and adhering of a component implementation to the behavior specification via a protocol.

As a proof of the concept, behavior protocols are built into the SOFA CDL language as first-class entities. Doing so at three abstraction levels (introducing interface, frame, and architecture protocols) supports the refinement design process, allowing one to reason about component behavior at different levels of information hiding. To target refinement correctness, the interface, frame, and architecture protocols in a particular template are tied

together by the protocol conformance relation based on the protocol compliance concept. The verification of protocol conformance can be done at design time at these three abstraction levels which effectively factors the state space inherent to the verification. Moreover, by intercepting method invocations, it is possible to check whether a particular component implementation obeys the component's behavior specification at run time.

Acknowledgments

First of all, we would like to record a special credit to Milos Besta, a co-author of our joint work [20] who also contributed to an early draft of this paper. Special thanks go to Frank Stomp and Petr Tuma, and other colleagues for valuable comments. Also, Jiri Adamek deserves a special credit for designing a non-trivial case study, and Petr Hnetynka for incorporating the protocol checker into the CDL compiler. Vladimir Mencl contributed substantially to the protocol checker implementation. This work was partially supported by the Grant Agency of the Academy of Sciences of the Czech Republic (project number A2030902), the Grant Agency of the Czech Republic (project number 201/99/0244), the PEPiTA/ITEA project (the Eureka project no. 2033).

References

[1] R.J. Allen, A Formal Approach to Software Architecture, doctoral dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997.

[2] R.J. Allen, D. Garlan, A Formal Basis For Architectural Connection, ACM Transactions on Software Engineering and Methodology, Jul. 1997.

[3] J. van den Bos, C. Laffra, " PROCOL: A Concurrent Object-Oriented Language with Protocols Delegation and Constraints," Acta Informatica, Springer-Verlag, 1991, pp. 511–538.

[4] M. Broy, "A Logical Basis for Modular Software and System Engineering," Proceedings of SOFSEM'98, Springer LNCS Vol. 1521, Nov. 1998.

[5] R.H. Campbell, A.N. Habermann, "The Specification of Process Synchronization by Path Expressions," Springer LNCS, Vol. 16, 1974, pp. 89–102.

[6] C. Canal, E. Pimentel, J.M. Troya, "Compatibility and inheritance in software architectures," Science of Computer Programming, vol. 41, 2001, pp 105-138.

[7] G. Florijn, "Object Protocols as Functional Parsers," Proceedings of the ECOOP '95, Springer LNCS 952, Aug. 1995, pp. 351–373.

[8] D. Giannakopoulou, Model Checking for Concurrent Software Architectures, doctoral dissertation, Imperial College, University of London, Jan.1999.

[9] D. Giannakopoulou, J. Kramer, S.C. Cheung, Analysing the Behaviour of Distributed Systems using Tracta, Journal of Automated Software Engineering, special issue on Automated Analysis of Software, vol. 6(1), Jan. 1999.

[10] R. J. van Glabbeek, Linear Time-Branching Time Spectrum, "CONCUR '90: Theories of Concurrency: Unification and Extension", LNCS vol. 458, Springer Verlag, 1990, pp. 278-297.

[11] J. van Leeuwen (ed), Formal Models and Semantics, Handbook of Theoretical CS, MIT Press, 1990.

[12] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann, Specification and Analysis of System Architecture Using Rapide, IEEE Transactions on SW Engineering, 21(4), 1995.

[13] N. Medvedovic, R.N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, Tech. Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, Feb. 1997.

[14] R. Milner, A Calculus of Communicating Systems, Springer LNCS 92, 1980.

[15] O. Nierstrasz, "Regular Types for Active Objects," Proceedings of the OOPSLA '93, ACM Press, 1993, pp. 1–15.

[16] O. Nierstrasz, T.D. Meijler, "Requirements for a Composition Language," Proceedings of the ECOOP '94, Springer Verlag, LNCS 924, 1995, pp. 147–161.

[17] CORBA/IIOP Specification. Revision 2.4.2, OMG 01-02-01, 2001.

[18] F. Plasil, D. Balek, R. Janecek, "SOFA/DCUP Architecture for Component Trading and Dynamic Updating," Proceedings of the ICCDS '98, Annapolis, IEEE Computer Soc. Press, 1998, pp. 43–52.

[19] F. Plasil, D. Mikusik, “Inheriting Synchronization Protocols via Sound Enrichment Rules,” Proceedings of the Joint Modular Programming Languages Conference, Springer LNCS 1204, Mar. 1997.

[20] F. Plasil, S. Visnovsky, M. Besta, “Bounding Behavior via Protocols,” Proceedings of TOOLS USA ‘99, Santa Barbara, CA, Aug. 1999.

[21] F. Plasil, S. Visnovsky, M. Besta, Behavior Protocols, Tech. Report 2000/7, Department. of Software Engineering, Charles University, Prague, Oct. 2000.

[22] R. Reussner, “Enhanced Component Interfaces to Support Dynamic Adaption and Extension,” Proceedings of HICSS-34, IEEE, Jan. 2001.

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

[24] B. Selic, Protocols and Ports: Reusable Inter-Object Behavior Patterns, ObjecTime Limited, Kanata.

[25] SOFA project, http://nenya.ms.mff.cuni.cz/thegroup/SOFA/sofa.html.

[26] P. Steyaert et al., “Reuse Contracts: Managing the Evolution of Reusable Assets,” Proceedings of the OOPSLA ‘96, ACM SIGPLAN Notices, Vol. 31, No. 10, Oct. 1996, pp. 268–285.

[27] C. Szyperski, Component Software, Beyond Object-Oriented Programming, Addison-Wesley, 1997.

[28] D.M. Yellin, R.E. Strom, “Protocol Specifications and Component Adaptors,” ACM Transactions on Programming Languages and Systems, Vol 19, No. 2, Mar. 1997, pp 292-333.

[29] H.E. Eriksson, M. Penker, UML Toolkit, John Wiley & Sons, 1998.

[30] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

Appendix A: Definition of behavior protocol operators

Here, we formally define the composed behavior protocol operators, i.e., composition ($\sqcap_x$) and adjustment ($|T|$) introduced in Section 3.3. Each of them is overloaded in the sense that they are defined on traces, languages, and protocols.

By convention, *Events(X)* denotes the set of all events based on an event name from X, *Tokens(Y)* denotes the set of all event tokens based on events from Y, and, finally, *Tokens$_\tau$(Y)* denotes the set of all the event tokens which represent emitting or absorbing an event from Y (thus it does not include the event tokens representing internal events).

Definition (*Composition* on traces): Let $\alpha, \beta \in$ ACTs* be traces and X $\subseteq$ Events(EventNames) be a set of events. The *composition* of α and β on X (denoted by $\alpha \sqcap_X \beta$) is the set of traces defined as follows:

$NULL \sqcap_X NULL = \{ NULL \}$

$NULL \sqcap_X \langle p \rangle = \{\}$ for $p \in Tokens_\tau(X)$

$NULL \sqcap_X \langle q \rangle = \{ \langle q \rangle \}$ for $q \notin Tokens_\tau(X)$

$\langle p \rangle \hat{} \alpha \sqcap_X \langle q \rangle \hat{} \beta = \{\langle q \rangle \hat{} \gamma$ where $\gamma \in (\langle p \rangle \hat{} \alpha) \sqcap_X \beta\}$ for $p \in Tokens_\tau(X)$, $q \notin Tokens_\tau(X)$

$\langle q \rangle \hat{} \alpha \sqcap_X \langle r \rangle \hat{} \beta = \{\langle q \rangle \hat{} \gamma$ where $\gamma \in \alpha \sqcap_X \langle r \rangle \hat{} \beta\} \cup \{\langle r \rangle \hat{} \gamma$ where $\gamma \in \langle q \rangle \hat{} \alpha \sqcap_X \beta \}$ for $q, r \notin Tokens_\tau(X)$

$\langle ?x \rangle \hat{} \alpha \sqcap_X \langle !x \rangle \hat{} \beta = \{ \langle \tau x \rangle \hat{} \gamma$ where $\gamma \in \alpha \sqcap_X \beta \}$ for $x \in X$

$\langle ?x \rangle \hat{} \alpha \sqcap_X \langle !y \rangle \hat{} \beta = \{ \}$ for $x, y \in X \wedge x \neq y$

$\langle ?x \rangle \hat{} \alpha \sqcap_X \langle ?y \rangle \hat{} \beta = \{ \}$ for $x, y \in X$

$\langle !x \rangle \hat{} \alpha \sqcap_X \langle !y \rangle \hat{} \beta = \{ \}$ for $x, y \in X$.

Definition (*Composition* on protocols): Let A, B be protocols, X $\subseteq$ Events(EventNames) be a set of events. The *composition* of A and B on X (denoted as $A \sqcap_X B$) is a protocol generating the language $L(A \sqcap_X B) = \cup\{\alpha \sqcap_X \beta \mid \alpha \in L(A) \wedge \beta \in L(B)\}$.

Definition (*Adjustment* on traces): Let $\alpha, \beta \in$ ACTs* be traces, T $\subseteq$ ACTs be a set of event tokens. The *adjustment* of α and β with respect to T (denoted as $\alpha \,|T|\, \beta$) is the set of traces defined as follows:

$NULL \,|T|\, NULL = \{NULL \}$

$NULL \,|T|\, \langle p \rangle = \{ \}$ for $p \in T$

$NULL \,|T|\, \langle q \rangle = \{ \langle q \rangle \}$ for $q \notin T$

$\langle p \rangle \hat{} \alpha \,|T|\, \langle q \rangle \hat{} \beta = \{ \langle q \rangle \hat{} \gamma$ where $\gamma \in \langle p \rangle \hat{} \alpha \,|T|\, \beta \}$ for $p \in T \wedge q \notin T$

$\langle p \rangle \hat{} \alpha \,|T|\, \langle p \rangle \hat{} \beta = \{ \langle p \rangle \hat{} \gamma$ where $\gamma \in \alpha \,|T|\, \beta \}$ for $p \in T$

$\langle p \rangle \hat{} \alpha \,|T|\, \langle p' \rangle \hat{} \beta = \{ \}$ for $p, p' \in T \wedge p \neq p'$

$\langle q \rangle \hat{} \alpha \,|T|\, \langle q' \rangle \hat{} \beta = \{\langle q \rangle \hat{} \gamma$ where $\gamma \in \alpha \,|T|\, \langle q' \rangle \hat{} \beta\} \cup \{\langle q' \rangle \hat{} \gamma$ where $\gamma \in \langle q \rangle \hat{} \alpha \,|T|\, \beta\}$ for $q, q' \notin T$.

Definition (*Adjustment* on languages): Let $L_1 \subseteq$ ACTs*, $L_2 \subseteq$ ACTs* be languages, T $\subseteq$ ACTs be a set of event tokens. The *adjustment of* L_1 and L_2 *with respect to T* (denoted as $L_1 \,|T|\, L_2$) is the language $L_1 \,|T|\, L_2 = \cup\{ \alpha \,|T|\, \beta$ where $\alpha \in L_1 \wedge \beta \in L_2\}$.

Definition (*Adjustment* on protocols): Let A, B be protocols, T $\subseteq$ ACTs be a set of event tokens. The *adjustment of A and B with respect to T* (denoted as $L(A \,|T|\, B)$) is the language $L(A \,|T|\, B) = \cup\{ \alpha \,|T|\, \beta$ where $\alpha \in L(A) \wedge \beta \in L(B)\}$.