Behavior Protocols and Components

Frantisek Plasil[1,2], Stanislav Visnovsky[1], Miloslav Besta[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,besta}@nenya.ms.mff.cuni.cz
http://nenya.ms.mff.cuni.cz

[2] Academy of Sciences of the Czech Republic
Institute of Computer Science
Pod vodarenskou vezi 2
180 00 Prague 8
Czech Republic
{plasil, besta}@uivt.cas.cz
http://www.uivt.cas.cz

Abstract

In this paper we enhance the SOFA Component Description Language with a semantic description of a component's functionality. There are two key requirements this description aims to address: First, for the design purpose, it should ensure correct composition of the nested architectural abstractions; second, it should be easy-to-read so that an average user can identify a component with the correct semantics for the purposes of component trading. The semantic description in SOFA expresses the behavior of the component in terms of behavior protocols using a notation similar to regular expressions which is easy-to-read, and which grants guarantees about required and provided services. The behavior protocols are used on three levels: interface, frame, and architecture. One of the key achievements of this paper is that it defines a protocol conformance relation where the component designer can statically verify that the frame protocol adheres to requirements of the interface protocols, and that the architecture protocol adheres to the requirements of the frame and interface protocols.

Keywords: behavior protocols, protocol conformance, architecture description language, component-based programming

This work is 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), and MLC Systeme, Ratingen, Germany.

1 Introduction

It is widely accepted that in the very near future a majority of software applications will be composed from reusable software components and there will be a market of such components. One of the cornerstones of successful development of component-based applications and component trading is the possibility to describe the functionality of components and thus to distinguish among different components providing the same set of interfaces and consequently to choose the most suitable implementation. Such a description should be quite abstract in order to provide freedom for implementation. On the other hand, it should be detailed enough to allow application designers to get to know how a particular component should be used and at the same time to prescribe to component implementors the way in which a component should be implemented. Moreover, the description should be sufficiently precise in order to allow for automatic checking of correctness of component use and binding. Needless to say, such a description should be easy to understand for programmers and simple to write for designers.

1.1 Objects and protocols

An object interface definition can be considered as a service definition. The limitations of the standard service definition in many programming languages are that they do not take into account availability and responsiveness of services. As stated in [8], the sequences of requests that an object is capable of servicing constitute the object's protocol. A typical way [2, 3, 7, 8, 9, 11, 28, 32, 33] 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 state transition system, e.g. [8, 29], (2) via a parser accepting the valid request sequences of the protocol, e.g. [7], (3) as a regular-like expression generating the valid request sequences [15].

As there is no generally accepted terminology as to the meaning of "protocol," we will follow in this paper the convention that protocol means the regular-like expression mentioned in the approach (3).

The origin of protocols is in path expressions [1] which specify synchronization of procedures executed in parallel. The ANSA computational model [11] modified the concept of path expressions in order to control access to object interfaces. Procol [2, 3] might serve as an example of an object language in which protocols are used to describe both access synchronization and availability of object's service. In all these synchronization schemes, checking the compliance of calls to an object with its protocol was expected to be done at run-time.

However, in [8] it is proposed to specify object protocols as an integral part of an object interface. As emphasized in [9], rather than simply raising exceptions when protocols are violated, it would be desirable to statically validate clients' conformance to protocols and to determine automatically if a protocol can be formally viewed as a "subtype" of another one. Moreover, in [8], a subtyping relationship on regular types is defined which allows to statically determine whether one protocol can be replaced by another one. Satisfaction of client's expectations is formalized as request satisfiability, in which the sets of accepted and a set of rejected requests are defined.

The Fusion method [28] comes up with the idea to describe not only the effect of each system operation in terms of the state change it causes, but also by the output events it sends. Thus the protocols in Fusion (syntactically based on regular expressions) express also the output events as a reaction on input events (method invocations). Compliance with a protocol is expected to be checked at run-time.

1.2 Components and protocols

Recently, the component oriented program design has drawn a lot of attention, mainly because components are more suitable for rapid development of applications than objects. Usually, a component is considered to be a static abstraction serving as an element of a component framework. It can be viewed as a black-box entity which provides and/or requires a set of services (via interfaces). The components

can be arranged to form another component. This allows for building hierarchies by establishing connections between required and provided services.

In order to achieve a high level of safety and robustness, it is desirable to describe semantics of components. Many papers have been published about the description of semantics and about composition of components based on such description. Mostly, they are based on formal specification of method behavior, e.g. VDM method [26], RAISE method [25], and on process algebra (calculi) specifications, e.g. CSP [6], CCS [5], Wright [12]. Although many advantages of these methods have been claimed, it is not an easy task to employ these rigorous semantical methods routinely.

Among the approaches based on applying the idea of object protocols to components belong [12] and [29]. The protocol idea described in [29] is based on cooperating pairs of typed interfaces. When interfaces of two components are bound together, i.e., they communicate bidirectionally, the interfaces are said to engage in a collaboration. From a component's point of view, a collaboration specification consists of two parts: (1) Interface signature part which describes the set of messages that can be exchanged between the components. (2) Protocol part which describes the collaboration by a set of sequencing constraints based on a transition system as mentioned in Section 1.1.

The Wright architecture description language [12] allows a more sophisticated specification of a component's semantics based on CSP [6]. Being enhanced to accommodate the distinction between incoming events and outgoing events, the Wright's variant of CSP fits well for describing component communication. A component's behavior is specified on two levels of abstraction: (1) interfaces, and (2) communication among interfaces. In both cases the specification takes the form of a CSP process. Using this specification, it is possible to check the behavior of a component architecture for consistency, i.e., to check if the behavior specified for an interface is consistent with the specification of communication among all interfaces.

1.3 Challenges, the goal of the paper

None of the approaches mentioned in Section 1.2 are based on describing protocol in a form similar to regular expressions which is very easy to read. Readability of protocols is a very important issue, e.g., with respect to the trading of components. Moreover, none of these approaches address a step-by-step development of a component's protocol during the design process of the component. As this process is typically based on refinement, it would be very helpful to verify the matching of the abstractions which evolve resp. are created step-by-step during the design.

The goal of this paper is to address these two issues (readability and step-by-step development support). To reflect the goal, the paper is organized as follows. In Section 2 we provide an overview of the SOFA component model which will serve as a proof-of-the-concept base. Section 3 introduces behavior protocols as the underlying model of component communication. The key contribution of the paper is provided in Section 4 which shows how behavior protocols can be associated with the SOFA architecture description language called CDL [21]. Moreover, this section points out how well the idea of step-by-step protocol refinement fits into the SOFA component model where refinement-based component design is supported by providing black-box view and grey-box view on a component as a part of the component's type definition. Section 5 illustrates use of behavior protocols on a case study example. Section 6 is devoted to evaluation and open issues. Related work is discussed in Section 7 and Section 8 concludes the paper by summarizing key achievements.

2 SOFA Components

2.1 Component model

In SOFA[16], an application is viewed as a hierarchy of nested software components. In analogy with the classical concept of an object as an instance of a class, we introduce a *software component* (*component* for short) as an instance of a *component template* (*template* for short). In principle, "template" can be interpreted as "component type".

In general, a component template is a pair <template frame, template architecture>. The template frame (*frame* for short) of a template T defines the set of individual interfaces any component which is an instance of T will possess. Basically, the frame of T reflects a black-box view on T. Interfaces are defined in the SOFA CDL language. In a template frame, similar to many other ADLs (e.g., Darwin [14]), an interface (type) can be instantiated as a *provides-interface* or a *requires-interface*. In principle, a provides-interface of T specifies a service each instance of T will offer to its clients. Similarly, a requires-interface of T specifies a service any instance of T will need (call) from an external component.

Template architecture (*architecture* for short) describes the structure of a concrete version of the corresponding template frame implementation by instantiating direct subcomponents (those on the first level of component nesting) and by specifying the necessary component interconnections via interface ties. There are three kinds of interface ties. (1) Binding of a requires-interface to a provides-interface. (2) Delegating from a provides-interface to a nested component's provides-interface. (3) Subsuming of a subcomponent's requires-interface to a requires-interface. Basically, the architecture of T reflects a grey-box view on T. The architecture can be specified as *primitive* which means that there are no subcomponents and the template frame implementation will be given in an underlaying implementation language out of the scope of the architecture specification[1]. When an architecture is not primitive, it is not necessary to recursively specify ties of the nested components because the nested components are again instances of templates, each of them containing its architecture. Consequently the SOFA component is recursively defined up to the finest level of granularity.

Specific to the SOFA component model is the concept of updating. Updating can be done even at run-time if the updated component follows the rules of the DCUP architecture [17].

2.2 CDL specification language

The specification of SOFA components is done by the component definition language (CDL) , which is based on CORBA IDL. The complete syntax of CDL is given in [21]. Here, we just demonstrate CDL on a simple example.

Let us imagine we need to create a component which will serve as a very simple database server (Figure 1). Such a component should furnish its clients with the *Insert*, *Delete*, and *Query* operations for inserting and removing records from the database, and querying the contents of the database. For this purpose CDL includes the *interface* construct, which specifies the interface type as a set of method signatures. Our interface can be specified 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);
};
```

The database server will access the underlying database via the *IDatabaseAccess* interface type and will use the *ILogging* interface to log invocations of the provided operations.

```
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);
    void Close();
};
```

[1] In the [16] SOFA publication, an architecture specification can contain both subcomponents and local implementation objects. We believe that introducing the primitive architecture concept separates the design and implementation concepts more clearly. Also, in [16] architecture names can contain indication of a component provider; for simplicity we do not use the option.

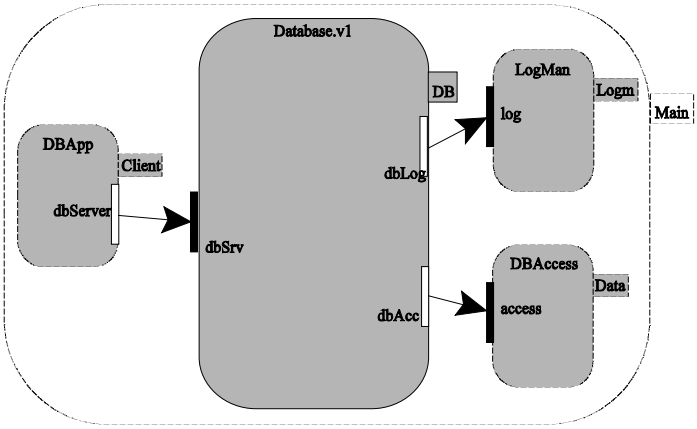

Figure 1

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

The frame specification contains declarations of provides-interfaces and/or requires-interfaces. Thus, the frame *Database* representing the intended simple database server is specified by the *frame* construct:

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

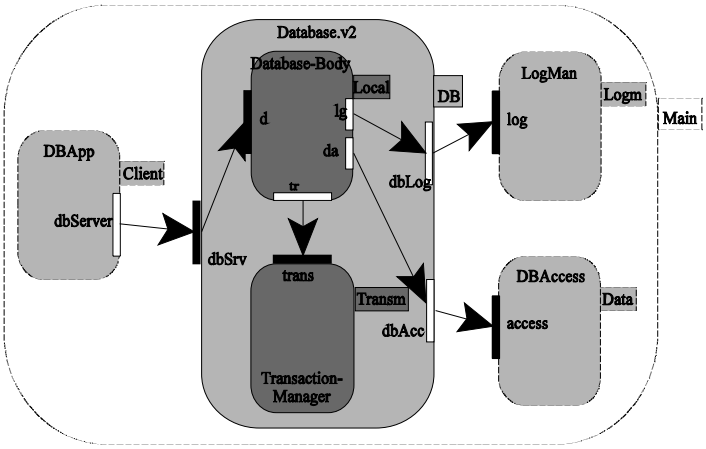

Figure 2

The template architecture of *Database* could be specified in many different ways. The first possibility is to declare it as primitive (no nested component as on Figure 1):

```
architecture Database version v1 {
    primitive;
};
```

Creating another template, the *Database* frame can be associated with a more structured architecture such as the one in Figure 2 specified as

```
architecture Database version v2 {
    inst TransactionManager Transm;
    inst DatabaseBody Local;

    bind Local:tr to Transm:trans;
    subsume Local:lg to dbLog;
    subsume Local:da to dbAcc;
    delegate dbSrv to Local:d;
};
```

This architecture version illustrates the way subcomponents instantiate and how their ties are specified (distinguishing bind, subsume and delegate cases). Notice that the subcomponents are specified only at the abstraction level of their frames. As this approach clearly separates the levels of providing architectural details, it allows for easy replacement of a subcomponent by its new version.

3 Behavior protocols

3.1 Model of communication

In this section we will define a formal model of component communication. The basic idea is to grasp communication among software components as sequences of communication events. To support the client/server roles in communication, an event can be a *request* or a *response* (request and response are atomic and intuitively defined). The computational activity of a component is modeled by the agent concept.

An *agent* is a computational entity handling sequences of events. As to handling an event, an agent can emit and/or absorb an event, or it can perform an internal event. In general, we say that the agent *exhibits* an *action*. An agent cannot handle more than one event at a time. Agents communicate via peer-to-peer external connections. An *external connection* C between agents A and B is the pair of one-way channels denoted $C_{A->B}$ resp. $C_{B->A}$. The channel $C_{A->B}$ sequentially transmits events emitted by A to B. In B, these events are absorbed in the order they were delivered (similarly via $C_{B->A}$, events emitted by B are transmitted from B to A). For simplicity, we assume that (1) there is no delay on C, and (2) A can emit an event only if B is prepared to accept it, so we pretend that emitting, transmitting, and absorbing of an event is done in one atomic action. An agent communicates with a finite number of agents and two agents can be connected by a finite number of connections.

An agent can be *primitive* or *composed*. A composed agent P is constructed by the *composition* of two agents A and B. (We also say that P is the parent of child agents A and B, resp. that A and B are (direct) children of P). The connections of P are the union of the connections of A and B. The (external) connections via which A and B communicate between each other become *internal connections* of P; events on the internal connections of P are referred to as *internal events* of P (in analogy with internal actions τ in [5]). In other words, P inherits all connections of A and B (naturally, this includes both internal and external connections of A and B). However, those external connections serving for communication between A and B become internal in P.

Let C be a connection of an agent A and P the parent of A. By definition, P shares C with A. The events on C are handled by both P and A jointly (in the sense that the event handling done by A is also considered to be done by P).

Based on composition, every agent is a part of a hierarchy of agents called a *system*. The root of a system is an agent with no external connections. In a system S, all agents start to exhibit their actions after receiving an init signal (broadcasted by an external, intuitively defined, control authority). Similarly, all

agents in S stop their activities after receiving a stop signal from this control authority. The period between the init and stop signals determines a *run* of S.

In a run, the activity of an agent A on a set of connections CS is the sequence of actions A exhibits on CS (recall that "actions" mean absorbing/emitting events or performing internal events). By convention a sequence of actions is represented as a *trace on* CS. A trace is a sequence of *action tokens* created as follows. In an action token, the event it represents is uniquely identified by the *event name* followed by the symbol *↑* resp. *↓* (these symbols distinguish request resp. response.) The event name and the request/response symbol form the *event identification*. For example, *a↑* denotes a request event with the name *a* . To express whether an event is emitted or absorbed, we prefix the event identification in an action token by the *!* resp. *?* symbol. Similarly, an internal event performed by the agent is prefixed by τ, e.g., τa . In summary, an *action token* is syntactically formed as a triple <*!* or *?* or τ, event_name, *↑* or *↓* >. For example, if *!a↑,!b↑,?a↓,?c↑* is a trace of A on C (Figure 3), the corresponding trace of B on C should be *?a↑,?b↑,!a↓,!c↑*.

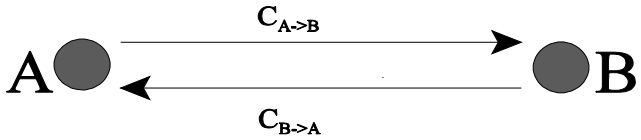

Figure 3

Again, let C be a connection of an agent A and P the parent of A. As mentioned above, the event handling done by A is considered done also by P. Thus, this event handling has to be reflected in traces of both A and P. If C is external both in A and P, the contribution to both traces is the same. (Similarly in the case of C being internal in both A and P). However, if C is external in A but internal in P, the handled events are prefixed by τ in the trace of P and by *?* or *!* in the trace of A .

As to naming, there is a local event namespace associated with every connection. Furthermore, in a system, there is a single, global namespace of connection names. By convention, to achieve unique identification of events in a trace, we use composed names created by qualifying a simple event name by the name of the connection on which the corresponding action is exhibited. For example, the trace *?C.a↑,τB.a↓, !C.a↓* indicates that the event *a↑* is absorbed on the connection C, then the event *a↓* is performed on the internal connection B, and *a↓* is emitted on C.

On a set of connections CS, the set of all A's activities in all possible runs is the *behavior* of A on CS. By convention, the behavior of A on CS is represented as a set of traces—the *language of* A *on* CS (denoted by $L_{A,CS}$).

The *event restriction* of a language L on a set of event names N is a function φ_N: L → L', such that $\varphi_N(\alpha_0 x_1 \alpha_1 x_2 \alpha_2 ... x_n \alpha_n) = x_1 x_2 ... x_n$, where $\alpha_0 x_1 \alpha_1 x_2 \alpha_2 ... x_n \alpha_n \in L$, $\forall i\ x_i \in E_N$, $\alpha_i \in (E_L \setminus E_N)^*$ and E_N is the set of all possible action tokens the identifiers of which are in N and E_L is the set of all action tokens in L. In other words, the restriction is a function that, from every trace of the language L, omits all action token identifiers of which are not in N. The resulting set of words constitutes the language L'.

Let $L_{A,CS}$ be a language of A on CS. The language L on a subset C of the connection set CS is the language $L/C = \varphi_{CE}(L_{A,CS})$, where CE is the set of all event names, identifiers of which are qualified by an identification of connection from C.

The key issue is to find a formal notation allowing to specify $L_{A,CS}$ in a finite way, because this language is typically infinite. Such a notation should be simple enough to be easily included in an ADL language. The approach we choose is to take advantage of the fact that some of the languages (of agents) can be expressed by behavior protocols (Section 3.2). At the same time, a language L which cannot be precisely defined by a behavior protocol can usually be approximated by a "closely relative" language L' reflecting well the abstraction level difference between an ADL specification and implementation in a programming language.

3.2 Syntax and semantics of behavior protocols

Behavior protocol (*protocol* for short) is a regular-like expression, which (syntactically) generates traces. Formally, a behavior protocol is an element of the language generated by the *Behavior Protocol Grammar* BPG. For the full grammar and behavior protocol semantics see Appendix A. The basic element of a behavior protocol is an action token or NULL (for empty protocol). A protocol can use the following operators (listed in descending priority):

$\alpha^\wedge$	reentrancy; it is equivalent to $\alpha \mid \alpha \mid ... \mid \alpha$.
α^*	repetition; it is equivalent to $\alpha \; ; \alpha \; ; ... \; ; \alpha$.
$\alpha \mid \beta$	and-parallel; the traces of α and β are generated in parallel and the resulting trace is an arbitrary interleaving of them.
$\alpha \parallel \beta$	or-parallel; it stands for $\alpha + \beta + \alpha \mid \beta$. The priority of $\parallel$ is equal to the priority of the $\mid$ operator.
$\alpha \; ; \beta$	sequencing; a trace generated by α is followed by a trace generated by β.
$\alpha + \beta$	alternative; the resulting trace is generated alternatively by α or by β.
$\alpha \sqcap \beta$	composition; similar to $\alpha \mid \beta$, however, when in α an event is observed and in β the same event is emitted (and vice-versa), this event is considered internal in the resulting trace.

To simplify specification of method invocation, we introduce these abbreviations:

$?m\{\alpha\}$	nested incoming call; the request for the method m is observed, followed by a trace generated by α and then the response for m is emitted. The curly brackets change the priority in the same way the parentheses do.
?m	simple incoming call; it stands for $?m\downarrow \; ; !m\uparrow$.
!m	simple outgoing call; it stands for $!m\uparrow \; ; ?m\downarrow$.

Given a behavior protocol BP and a set of event identifiers N, *restriction of* BP *on* N (denoted as BP/N) is the behavior protocol constructed from BP by systematic replacement of all action tokens the identifiers of which are not prefixed by element from N by NULL. In the following text L(BP) denotes the language generated by BP.

3.3 Examples

Intuitively, in terms of software components, a behavior protocol can serve for expressing method call ordering semantics. For example, sequencing of method calls can be specified by the ; operator, the alternative calls by the + operator, the parallel calls by the | and the || operators, any number of the sequential invocations by the * operator, and the possibility of reentrant access by the ^ operator. We will illustrate this idea on a few examples. As the first one, let us assume the following Java class declaration:

```
class Half_Monitor {
    void a( ... ) { ... }
    void b( ... ) { ... }
    synchronized void c( ... ) { ... }
    synchronized void d( ... ) { ... }
};
```

The *synchronized* modifier of *c* and *d* specifies that these methods cannot run in parallel. Not considering any waits in c and d, we can describe this semantics by the behavior protocol *?a^ || ?b^ || (?c + ?d)**. The methods *a* and *b* are declared as reentrant and they can be run in parallel (notice the parallel operator is the or-parallel), but the invocation of *c* and *d* must be sequential.

To illustrate the expressive power of behavior protocols, let us discuss a potential call ordering semantics of the interface *IDatabaseAccess* from Section 2.2. The intended use of this interface can be to call the method *Open* first, then do a modification of the database by invocation of *Insert*, *Delete* and *Query*, and finally to finish the work with the database by invoking *Close*. The corresponding protocol can take the

form *?Open ; (?Insert + ?Delete + ?Query)* ; ?Close*. If the methods *Insert*, *Delete* and *Query* were to be designed to handle their parallel execution, we could specify this intention by *?Open ; (?Insert || ?Delete || ?Query) * ; ?Close*. This protocol indicates that parallel run of the methods *Insert*, *Delete* and *Query* is possible, but any two invocations of *Insert* must be done sequentially (the same holds for *Delete* and *Query*). To specify that completely parallel access to these methods is allowed, the reentrancy (^) operator is to be used instead of the repetition (*). Thus, the protocol should take the form *?Open ; (?Insert || ?Delete || ?Query)^ ; ?Close*.

The last example shows how to specify dependencies among the incoming and outgoing method calls. Let us assume the following protocol:

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

At the beginning, the method *dbAcc.Open* is invoked. Then there are three possibilities of incoming invocations—*dbSrv.Insert*, *dbSrv.Delete* and *dbSrv.Query*. If a request call of the *dbSrv.Insert* method is observed, *dbAcc.Insert* and *dbLog.LogEvent* are sequentially invoked an arbitrary number of times and then the *dbSrv.Insert* response is emitted (similarly for *dbSrv.Delete* and *dbSrv.Query*). Finally, *dbAcc.Close* is invoked.

4 Associating behavior protocols and SOFA components

4.1 Agents: components at run-time

Modeling of SOFA components via the concepts introduced in Section 3.1 is straightforward. As agents are principally run-time entities, the central idea is to associate every component with an agent (one-to-one relationship) such that it models the component behavior. Given a component C, we call the agent associated with C the *agent of C*, or simply the *C agent*. In any component C being an instance of T=<F, A>, one can imagine that the C's agent resides "in the middle" of the C component—not in the middle of T. 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).

Communication of components C and C' via a pair of a requires-interface of C and a provides-interface of C' is modeled as a connection *CON* of the C agent and the C' agent. Distinguishing the two kinds of interfaces (provides and requires) can be reflected as employing connections with a "provides" and a "requires" ends in the sense that the emitting and absorbing of events follow this pattern: A method call *m(..)* issued by C on a requires-interface is modeled as the event pair *...!CON. m↑.... ?CON. m↓...* in a trace of the C agent and *...?CON.m↑... !CON.m↓...*in the corresponding trace of the C' agent. A call of a one-way method *ow(..)* is modeled as an event *!ow↑* in a trace of the C agent and *?ow↑* in the corresponding trace of the C' agent. The event namespace of a connection is determined by the interface type used in the corresponding component communication. In principle, the set of method names in the interface type comprises the event namespace (not considering any polymorphism for simplicity here).

A component C which is an instance of T=<F, A> is basically specified by a number of frames involved in the description of A (recursively). As to specifying a connection, it has to be principally reflected in the hierarchy of the architecture specifications. Furthermore, the nesting of components also has to reflect the potential sharing of connections to which the corresponding agents may be exposed. Due to sharing, a connection spans across the architecture hierarchy. To follow the basic philosophy of nesting in CDL specifications, a connection has to be specified also on an incremental description basis. This incremental approach is embodied by tying pairs of interfaces as presented in Section 2. Thus, a connection specification can be seen as a chain of *subsume*, *bind*, and *delegate* clauses spanning across the corresponding hierarchy of architecture specifications.

In Figure 4, *A*, *B*, *C*, *D*, *X*, *Y* are component names, *IA*, *IB*, *IC*, *IX*, *IY* are the names of interface instances (for simplicity instances of the same interface type). The architectures of *A* and *Y* are primitive. The architecture *B* contains the clause *subsume A:IA to IB*, similarly the architecture *C* contains *subsume B:IB to IC*. The interfaces *A:IA*, *B:IB,* and *C:IC* form the subsume chain of the connection. The architecture of *D* contains the clause *bind C:IC to X:IX*, which binds the requires-interface *C:IC* to the provides-interface *X:IX*. In a similar vein, the clause *delegate IX to Y:IY* in the architecture of *X* composes the delegate chain of the connection. Thus, with respect to the composition of agents, the agents of *A*, *B* and *C* share one end of the connection and the agents of *X*, *Y* share the other end. The connection is internal to the *D* agent.

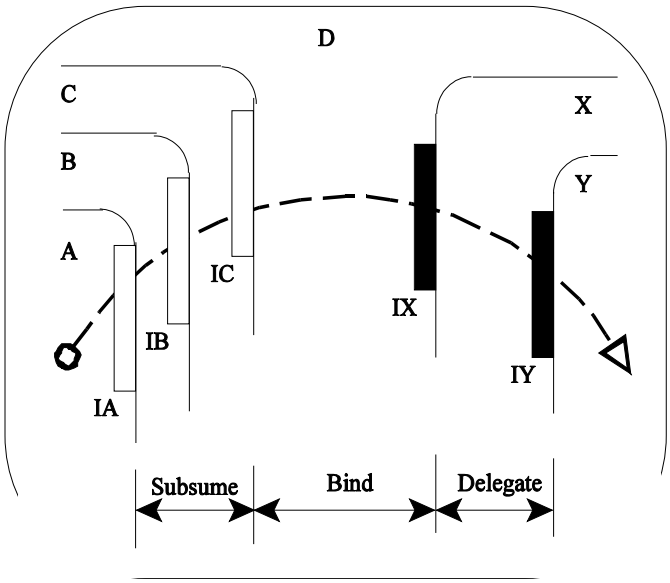

Figure 4

In principle, *D<C:IC→X:IX>* would do for a unique identification of the connection. (Notice that this "context sensitive" style of identification is enforced by the fact that there are no names of bindings in CDL.) Following this approach, however, the information on connection sharing by the subcomponents *A*, *B* and *Y* would be available only from the corresponding CDL specification sources. This kind of information is essential for restricting traces on a particular subset of interfaces. To make it easier to grasp these restrictions, we use a *complete-chain* form of connection identification; for example, *D<A:IA-B:IB-C:IC→X:IX-Y:IY >* would be the complete-chain form name of the connection from Figure 4.4. For each interface I in a system, there is exactly one connection C such that I is a part of C's chain. This connection is referred to as the operative connection of I. Thus, we define a restriction of a behavior protocol BP (resp. a language L) on a set of interfaces SI as the restriction of BP (resp. L) on the set of operative connections which correspond to the interfaces from SI.

4.2 Behavior of components

In terms of components, the root of the system is the outmost component. In a particular run of the system, the "full activity" of a component (the method calls on all its interfaces) is captured by a trace on these interfaces. The trace captures invocation ordering dependencies of method invocations among all subcomponents recursively and expresses invocation dependencies among interfaces of these subcomponents in this particular run. Recall that the set of all activities an agent exhibits on a set of its connections in all possible runs of the system is referred to as the behavior of the agent on this set of the connections.

As a component is modeled by an agent, it is natural to introduce the behavior of a component: The behavior a given component C can exhibit on a set of its interfaces SI is called the *component behavior on* SI. If SI is composed of all the interfaces of C (including recursively the interfaces of subcomponents), we refer to the behavior of C on SI as the *white-box behavior of* C.

To support refinement design process of a SOFA component C, the following three special choices of the set of interfaces SI is to be taken into account: (1) The behavior of C on one of its outmost interfaces I is called the *service behavior of* C on I. The service behavior of C on I captures invocation ordering dependencies among method invocations in I. (2) The behavior of C on its outmost provides-interfaces and requires-interfaces is called the *black-box behavior of* C. The black-box behavior of C captures invocation ordering dependencies among method invocations on interfaces at the frame level of C's description. (3) The behavior of C on the outmost interfaces of C and the outmost interfaces of C's direct subcomponents is called the *grey-box behavior of* C. In this case, the behavior captures dependencies of invocation on interfaces of C and the subcomponents described at the architecture level of C's specification.

4.3 Bounding behavior of components

As emphasized in Section 3.1, agent behavior can be approximated by a behavior protocol. Thus, we could employ behavior protocols to approximate component behavior. The key issue is to define the exact meaning of the "approximation" of the SOFA component behavior. Our answer to this question is based on the idea that a component's behavior on its provides-interfaces can be "richer" than specified by a protocol, while the behavior on its requires-interfaces has to be "narrower" than specified by a protocol. Such an approximation is referred to as "bounding" the behavior of a component; formally:

Let C be a component and P_C resp. R_C the sets of all C's provides-interfaces resp. requires-interfaces (including recursively interfaces of C's subcomponents). We say that the behavior represented as language L_C of the component C on a set of its interfaces SI is *bounded* by a protocol BP if both of the following inclusions hold:

(1) $L(BP)/(P_C \cap SI) \subseteq L_C/(P_C \cap SI)$
(2) $L(BP)/(R_C \cap SI) \supseteq L_C/(R_C \cap SI)$

To support the refinement design process of a SOFA component C, it is a natural step to introduce bounding of service, grey-box, and black-box behaviors of C. The following three definitions capture this idea.

Let BP be a protocol and I an outmost interface of C. If $L(BP) = L(BP/\{I\})$ then BP is a *service protocol of* I. A service protocol bounds the invocation ordering dependencies among the method invocations on a particular interface.

Let BP be a protocol and S the set of all the outmost interfaces of C. If $L(BP) = L(BP/S)$ then BP is a *black-box protocol of* C. A black-box protocol bounds dependencies of invocation among methods of the outmost provides-interfaces and requires-interfaces of C.

Let BP be a protocol and S the set of all the outmost interfaces of C and all the outmost interfaces of C's direct subcomponents. If $L(BP) = L(BP/S)$ then BP is a *grey-box protocol of* C. A grey-box protocol bounds dependencies among interfaces of C and interfaces of its direct subcomponents.

4.4 Protocols in CDL

The most important issue addressed in this paper is how to employ behavior protocols in the design process of software architecture in order to make the specification of the corresponding component-based system more precise. To target this issue, we will associate, at the template/CDL level, every interface, frame, and architecture with a protocol to express how the service, black-box, and grey-box behaviors of components (i.e., instances of templates) should be bounded. Intuitively, these protocols can be seen as the specification of how a particular interface of a template can be used and how instances of the template will behave on their interfaces. In the remaining part of this section we will show how the CDL level protocols are related to the component-level protocols.

At the component level, any protocol in a component contains complete-chain identification of the connections (they reflect the component's ties as illustrated in Figure 4). The complete-chain connection

identification is fully known as late as at the component instantiation time. Therefore, at the CDL specification level, the protocols can contain only a generic identification of (future) connections. Consequently, protocols are specified in their generic form in CDL. In particular, in a protocol in its generic form, the connection names are the interface identifications as declared in the corresponding part of the CDL specification.

The modification of the generic form of a protocol P to the protocol which contains only complete-chain identification is called instantiation of P (the resulting protocol is an *instance* of P). However, because of the step-by-step refinement nature of component design, the knowledge of connection identification is gained also on a step-by-step basis reflecting the nesting of components (Section 4.1). Therefore, each level of a template T's nesting contributes incrementally to the knowledge of a part of the complete chain identification of the connections which are involved in T. This implies step-by-step modification of every protocol P associated with T; we call these intermediate forms of P *semi-instances* of P.

The following three cases of protocol semi-instances should be emphasized: (1) The semi-instance of an interface protocol P to reflect the knowledge gained from a frame F (to reflect in P the identification and the requires/provides role of the interface instances specified in F if P is associated with an interface specified in F). (2) The semi-instance of an interface protocol P to reflect the knowledge gained from an architecture A (to reflect in P the identification and the requires/provides role of the interface instances specified in A if P is associated with an interface specified in A). (3) The semi-instance of a frame protocol P to reflect the knowledge gained from an architecture A (to reflect the ties specified in A and the identification of subcomponents in A if P is associated with F which is declared as a subcomponent of A).

Thus, an interface type specification can include an *interface protocol*. If the protocol is omitted, the implicit protocol is automatically included (all methods are reentrant and can be invoked in parallel). Because a method invocation on a provides-interface resp. requires-interface is prefixed by ? resp. !, and both of them can be instances of the same interface type TI, the interface protocol of TI is in CDL written in its generic form where no prefixing by ? and ! takes place. An easy syntactic transformation can be applied to form the instance of the protocol (the service protocol) for a particular interface. The semi-instance of an interface protocol with respect to a frame resp. architecture is created by prefixing all method names by the interface instance identification and by the corresponding symbol ? or !.

To allow the specification of a black-box behavior, we enhance the frame specification by a *frame protocol*. Since a frame protocol specifies actions upon different interfaces, the names of the actions are qualified by the name of the interface instance the method belongs to and prefixed by ? or !. Method calls can be specified as nested. Typically, invocation of a method on a provides-interface causes some invocations on requires-interfaces. This semantics can be expressed by curly brackets (Section 3.2). To form an instance of the frame protocol (black-box protocol) for a particular component C, we rename the actions in the protocol to reflect the interface ties. The semi-instance of a frame protocol with respect to an architecture A is created by the prefixing method identification by ties specified in A.

In CDL, an architecture specification does not directly contain specification of the protocol to bound a grey-box behavior, instead the *architecture protocol* is generated automatically. For a template T=<F, A>, if A is primitive, the architecture protocol of A equals the frame protocol of F. In the opposite case, A contains some subcomponents. Suppose S_1, S_2, ... , S_N are the subcomponents specified in the architecture A. The architecture protocol of A is the composition ($\sqcap$) of the semi-instances with respect to A. We use the composition operator because the agent of a non-primitive component is composed from the agents of its subcomponents.

4.5 Protocol conformance

The generic protocols specified in a template T=<F, A> constitute just an obligation for all potential implementations of T. At the same time, to ensure that the instances of these protocols will bound the behavior of components based on T, they have to correspond to each other at different levels of abstraction. Intuitively, the architecture protocol of A should follow the design intentions embodied in the frame protocol of F, the interface protocols should comply with the way these interfaces are

employed in the protocols of F and A. The following three definitions related to protocol conformance help perceive this intuition.

Let T1 and T2 be interface types and P_{T1} resp. P_{T2} are the interface protocols associated with T1 resp. T2. We say that *the interface protocol of* T1 *conforms to the interface protocol of* T2 if T1 is a subtype of T2 in the classical sense and $L(P_{T1}) \subseteq L(P_{T2})$.

We say the *frame protocol of* F *conforms to the interface protocols of interfaces of* F if, for every provides-interface P in F with an interface protocol PP, holds $L(PP') \subseteq L(P_F)/\{P\}$, and, for every requires-interface R in F with an interface protocol PR, holds $L(P_F)/\{R\} \subseteq L(PR')$, where PP' (resp. PR') denotes the semi-instance of PP (resp. PR) with respect to F.

Let <F, A> be a template specification. Let P_A be the architecture protocol of A, P_F the frame protocol of F, and PS resp. RS the set of the provides-interfaces resp. requires-interfaces of F (being also the outmost interfaces of A). We say *the architecture protocol of* A *conforms to the frame protocol of* F if $L(P_F)/PS \subseteq L(P_A)/PS$ and $L(P_A)/RS \subseteq L(P_F)/RS$.

Claim: Let a component C be an instance of the template T=<F, A>, P_F the frame protocol of F, P_A the architecture protocol of A, $\{P_i\}$ the set of all provides-interfaces of F, $\{R_j\}$ the set of all requires-interfaces of F. If P_{P_i} is the interface protocol of P_i, and P_{R_j} the interface protocol of R_j, then

$L(P_{P_i}') \subseteq L(F)/P_i \subseteq L(A)/P_i$ for all provides-interfaces P_i in F
$L(P_{R_j}') \supseteq L(F)/R_j \supseteq L(A)/R_j$ for all requires-interfaces R_i in F

This claim can be easily proved by applying the definitions of protocol conformance given above.

Thus, the architecture, frame and interface protocols form a hierarchy: Given a component C, the architectural protocol of C restricted to the outmost interfaces of C has to conform to the frame protocol of C; the frame protocol of C restricted to an interface I of C has to conform to the interface protocol of I. This is an obligation for a correct design of the component template.

5 Case study

In this section, we illustrate the use of protocols on the example from Section 2 by adding suitable interface and frame protocols. We also show how architecture protocols are generated.

5.1 Interface protocol

The first step in enhancing the CDL specification is to associate interfaces with suitable interface protocols. In the *Database* template, the methods of the *dbSrv* interface (of the *IDBServer* type) can be called arbitrary number of times in any order, but not, e.g., in parallel. To bound the behavior on every interface of the *IDBServer* type accordingly, this interface type is to be associated with the interface protocol of the form:

*(Insert + Delete + Query)**

By convention an interface protocol is written as the last part of an interface type definition, such as in

```
interface ITransaction {
        void Begin();
        void Commit();
        void Abort();
    protocol:
        ( Begin ; ( Commit + Abort ) )*
};
```

This protocol states the use of this interface for simple transactions. The first method to be called is *Begin* to start a transaction and, subsequently, invocations of either *Commit* or *Abort* are permitted to finish the transaction. This sequence of invocations can be repeated any number of times. Similarly, the

IDatabaseAccess can be enhanced by *(Open ; (Insert + Delete + Query)* ; Close)** as discussed in Section 3.3 and *ILogging* interface protocol can be specified as *(LogEvent + ClearLog)** to allow sequential access to all logging methods.

5.2 Frame protocol

Next, we should specify the frame protocols for *Database*, *TransactionManager*, and *DatabaseBody*. The following example illustrates the employment of a frame protocol on the *Database* frame and the appropriate syntax in CDL:

```
frame Database {
    provides:
        IDBServer dbSrv;
    requires:
        IDatabaseAccess dbAcc;
        ILogging dbLog;
    protocol:
        !dbAcc.Open ;
        (   ?dbSrv.Insert { ( !dbAcc.Insert ; !dbLog.LogEvent )* } +
            ?dbSrvDelete { ( !dbAcc.Delete ; !dbLog.LogEvent )* } +
            ?dbSrvQuery { !dbAcc.Query* }
        )* ;
        !dbAcc.Close
};
```

Here, the need for logging every modification of the database is reflected by specifying nested calls 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*. Every invocation of *dbAcc.Query* is simply forwarded to the underlying database without any logging. This protocol must conform to the interface protocols of *dbSrv*, *dbAcc* and *dbLog*. To verify the conformance, these steps must be taken:

(1) Generate the semi-instance of an interface protocol with respect to the frame protocol. In the following text, to manifest that a protocol is automatically generated, we will make it typographically visible *in this way*. For the *dbLog* interface the semi-instance takes the form:

*(dbLog.LogEvent + dbLog.ClearLog)**

(2) Generate the restriction of the frame protocol to the specified interface, in our example *dbLog*:

(dbLog.LogEvent + dbLog.LogEvent*)**

(3) Verify the inclusion. The *dbLog* interface is a requires-interface. Thus we have to verify if the language generated by the protocol from the step (2) is a subset of the language generated bye the protocol defined in the step (1). However, this is obvious in our case.

A similar process is to be taken for all the interfaces declared in frame *Database*. The specification of the frame *DatabaseBody* might take the form:

```
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 }
        )* ;
        !da.Close
};
```

The last frame to be specified is *TransactionManager*:

```
frame TransactionManager {
    provides:
        ITransaction trans;
    protocol:
    (   ?trans.Begin ; ( ?trans.Commit + ?trans.Abort ) )*
};
```

Since this frame has only one provides-interface, the frame protocol is the same as the interface protocol of the *trans* interface. Of course, we could specify a different protocol, one which conforms to the interface protocol of *trans*.

To illustrate generation of the semi-instance of a frame protocol with respect to an architecture protocol, we generate the semi-instance of the protocol of subcomponent *Local* with respect to the *Database* architecture version *v2.* The semi-instance is created as follows:

For every event we reflect the interface ties by the appropriate renaming of events in the protocol. For example, consider the specification of *Database* version *v2*. Here, the binding *Local:tr to Transm:trans*, subsuming *Local:lg to dbLog*, *Log:da to dbAcc* and delegating *dbSrv to Local:d* yield:

```
!<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:da-dbAcc>.Close
```

5.3 Architecture protocol

To illustrate how an architecture protocol is formed as a composition of frame protocols, let us consider the architecture presented in Figure 2. The *Database* architecture version *v2* contains two subcomponents: *Transm* (instance of *TransactionManager*) and *Local* (instance of *DatabaseBody*). The semi-instance of the frame protocol of *Transm* with respect to *Database* takes the form:

```
(   ?<Local:tr → Transm:trans>.Begin ;
    ( ?<Local:tr → Transm:trans>.Commit + ?<Local:tr → Transm:trans>.Abort )
)*
```

The architecture protocol of *Database* is then:

```
(   ?<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:da-dbAcc>.Close
```

which can be rewritten as follows (notice the internal τ actions in the resulting protocol):

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

Let us now check the conformance of this architecture protocol to the frame protocol of *Database*. Again, we have to create the restrictions and then verify the inclusions. For provides-interfaces (*dbSrv*), the restrictions are for the frame protocol:

```
( ?dbSrv.Insert { } + ?dbSrv.Delete { } + ?dbSrvQuery { } )*
```

and for the architecture protocol:

```
( ?dbSrv.Insert { } + ?dbSrv.Delete { } + ?dbSrv.Query { } )*
```

The equivalence of these protocols is clear. The same verification must be also done for requires-interfaces (*dbLog* and *dbAcc*). The restrictions take the form

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

for the frame. For the architecture it is

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

As we can see, the architecture protocol is on the requires-interfaces is narrower than the frame protocol.

At the end of this section, we will create the semi-instance of the *DB* subcomponent in the *Main* architecture. Thus we reflect the ties and name of an instance:

```
!<Local:da-DB:dbAcc → Data:access>.Open ;
(   ?<Client:dbServer → DB:dbSrv-Local:d>.Insert {
        τDB<Local:tr → Transm:trans>.Begin ;
        !<Local:da-DB:dbAcc → Data:access>.Insert ; !<Local:lg-DB:dbLog → Logm:log>.LogEvent ;
        ( τDB<Local:tr → Transm:trans>.Commit + τDB<Local:tr → Transm:trans>.Abort )
    } +
    ?<Client:dbServer → DB:dbSrv-Local:d>.Delete {
        τDB<Local:tr → Transm:trans>.Begin ;
        !<Local:da-DB:dbAcc → Data:access>.Delete ; !<Local:lg-DB:dbLog → Logm:log>.LogEvent ;
        ( τDB<Local:tr → Transm:trans>.Commit + τDB<Local:tr → Transm:trans>.Abort )
    } +
    ?<Client:dbServer → DB:dbSrv-Local:d>.Query { !<Local:da-DB:dbAcc → Data:access>.Query }
)* ;
!<Local:da-DB:dbAcc → Data:access>.Close
```

6 Evaluation and open issues

The main advantage of behavior protocols is their intuitively easy-to-comprehend notation for description of communication. Protocols are able to describe communication on fairy-grained levels of abstraction in a relatively short and easy-to-read form. The behavior protocols are not designed to be used as a full programming language. For example, they cannot specify any specific number of repetitions, reentrant entries, etc. Therefore, they basically only approximate real traces. Balancing the expressive power and the simplicity of protocols, we believe that the argument of an elegant and easy-to-read notation overweights some loss in expressiveness and thus justifies application of behavior protocols in ADL languages.

The distinction between the request and response parts of method invocation gives a notation strong enough to express special communication situations, such as one-way calls in CORBA [27] which can be modeled as only the request of method invocation without response, and deferred calls (the explicit interleaving of request/response pairs).

Interface protocols are guidelines for using interfaces. They can help to distinguish among different types of services which have a similar interface. Obviously, interface protocols can be used with objects as well as with components. In principle, with respect to a component interface, the information provided by the corresponding interface protocols is redundant as it could be deduced from the component's frame protocol. However, such a deduction is not a simple task. Therefore, we can consider this a "sound" redundancy, since it simplifies the process of component tying, provides "quick" information in component trading, and enhances opportunities for checking of component design correctness. Without this redundancy, however, it would be very hard to reason about the process of component tying. Moreover, interface protocols make a natural separation of the effects which small modifications to the frame protocol induce in the behavior on the frame's interfaces. Without interface protocols, these modifications would cause the necessity of subsequent conformance checking with respect to other components tied to any component based on the frame in question.

The idea of frames helps system designers to build a system from components without detailed knowledge of the components' internals. Here, the notion of the frame protocol allows to define basic relations among interfaces. In addition to being very important for trading, this provides an advantage to designers who, intending to use a component based on a particular corresponding frame, know how the behavior of the component is bounded. Moreover, the implementor of the frame also benefits from having guidelines for an implementation in terms of its expected behavior.

Further, the notion of architecture protocol improves the description of a component's behavior by revealing its behavior at the next level of nesting compared to the corresponding frame protocol. In CDL, we choose not to specify architecture protocol directly because: (1) A lot of technical rewriting is to be done to employ semi-instances of internal frame protocols. (2) To capture the behavior of combined

subcomponents in detail by a hand written protocol can be hard, particularly in case of their mutual dependencies. Our approach is to combine internal frame protocols by the composition operator. This is a general solution which completely eliminates the issue (1), however, it does not capture special properties of architecture, e.g., simple dependencies among internal subcomponents.

By the authors, the fact that interface, frame, and architecture protocols form a hierarchy which allows for reasoning about template design and supports refinement design process is considered to be the key contribution of this paper.

The list of open issues includes:

(1) *Automatized tools for testing protocols conformance.* Based on the experience gained in [20, 22], we are currently investigating two different approaches: (a) Employing term rewriting to capture the idea of structural similarity of protocols, and (b) testing inclusion of languages generated by protocols. Here the main obstacle is the ^ operator which damages the regularity of these languages.

(2) *Guards.* Behavior protocols cannot constrain method invocations, e.g., by the guard abstraction [3]. This implies a question of what kind of predicate is to be allowed in the guard condition. If we considered dynamic checking of protocol conformance, we could use some kind of property (boolean) methods in guards. On the other hand, using guards in this way would not help in the static checking of protocol conformance and could hardly be used for component trading and versioning. Another possibility is to introduce abstract properties (representing a component's internal state) which could be tested in guards. In this case, the issue of static checking is still open.

(3) *Architecture versioning.* The frame and architecture concepts allow for reasoning about the replacement of one architecture by another if their architecture protocols A_{old} and A_{new} conform to a common frame protocol F. This raises an issue of reasoning about a relation between A_{old} and A_{new} (versioning issue). This is particularly important if dynamic updating of components is considered as in SOFA/DCUP.

(4) *Adaptors.* When a component's provides-interface cannot bind to a requires-interface of another component because of small differences in method names, parameters, and/or constraints on ordering of method invocations, it would be valuable to create an adaptor allowing the correct use of the component. There were a few attempts to provide a tool which automatically generates adaptors for components, e.g. [29]. Behavior protocols could help in creating such adaptors by providing information about the ordering of method invocations.

(5) *Updating control.* In DCUP, the moment an update of a component may take place is in full control of the Component-Builder abstraction. Behavior protocols seem to be an ideal means to provide guidelines for Component-Builder, e.g., by including the builder interface call acceptance into the frame protocol. In this way, acceptance of update signals can be easily specified.

(6) *Better conformance architecture – frame.* So far, this conformance is based on separate inclusion of provides and requires restrictions. Via these restrictions, the interplay of calls among provides and requires interfaces in a frame can be lost. An issue is to find another definition of compliance of a frame and architecture protocols overcoming the problem.

7 Related work

Probably the closest to our work is the Wright language [12]. In Wright, the behavior of components is specified as "computation" via a CSP-based notation (a system of recursive equations). In our opinion, regular-like expressions are more readable having expressive power strong enough to reasonably approximate behavior of components. Components in Wright communicate by means of connectors which can be quite complex. Their behavior specification ("glue") is also CSP-based; in fact, glue is very similar to computation. In SOFA, we can simulate connectors by specialized components. In Wright there is no black-box view of component which contains some subcomponents. For this purpose, in our approach, frames with frame protocols are introduced, which we consider very important for refinement-based design of components and for component replacement.

The work on interfaces and protocols [29] is quite similar to our approach in the sense that it describes communication between component interfaces. However, it concentrates only on behavior description related just to a pair of collaborating interfaces. It does not consider specification of a component as whole, and therefore, there are no concepts similar to our frame and architecture protocols. Consequently, the protocol description in [29] can be hardly used for reasoning about component composition, replacement, etc. On the other hand, we found it interesting that it allows for bidirectional communication at one pair of interfaces; as an aside, this can be easily modeled by our communication model. In our opinion, the way we have chosen for expressing component behavior is much easier to apply.

Reuse contracts [38] introduce the idea to specify for each method of an interface the set of internally invoked methods, thus being able to capture invocation dependencies among methods. However, the model presented in this work is limited. As it provides description only at the object level of abstraction, it does not support the component-based approach with more cooperating interfaces. While we can describe ordering of nested method invocations, reuse contracts do not try to express similar information.

The approach proposed in [7] describes complex protocols on objects via a language parser which accepts traces of messages. Disadvantages of this approach include that parsing is done only dynamically and it does not consider any kind of "parser conformance".

In UniCon [39] an application consists from a set of components which can be composed of subcomponents. A possibility to specify communication behavior at the level of connectors (in protocol part of connector specification) is briefly mentioned without any further details.

8 Summary

This paper introduces a novel technique for specification and bounding of component behavior via behavior protocols which take a form similar to regular expressions. Description of component behavior by means of these behavior protocols is precise enough to capture necessary requirements in terms of describing the service provided by a component. It is easy to read and, at the same time, simple to create because its notation is easy to comprehend. Thus, the protocols meet the basic requirements for a practically useful specification. The paper presents a way of their deployment in the SOFA CDL language.

In SOFA, the three abstraction levels of protocol employment (i.e., interface, frame, and architecture) significantly support refinement design process, allowing to reason about component behavior with a different level of information hiding. Not revealing any details on the component structure, the interface protocol enhances the description of the service provided or required on an interface. The frame protocol hides the architecture details; it provides behavior information important for component design and trading and, furthermore, it supports seamless component updating. The architecture protocol describes component architecture in more detail in order to provide guidelines for design and implementation purposes.

Interface, frame, and architecture protocols are tied together by the protocol conformance relationship which incorporates the idea of behavior compatibility. The verification of protocol conformance can be done statically, i.e., at design time, allowing for reliable composition of applications. Moreover, having an implementation of a component type, it is also possible (by intercepting method invocations) to check compliance of the real component behavior with the component specification at run-time.

Acknowledgments

The authors of this paper would like to express their special thanks to Petr Tuma for his useful comments. The authors' appreciation goes also to their colleagues Marek Prochazka for drawing their attention to the nested calls problem and Dusan Balek for comments on the communication model.

Appendix A — Behavior protocol grammar

BPG = ({A, B, ... }, N ∪ {↑, ↓, ?, !, +, ;, (,), ^, |, ||, ⊓, {, }}, Rules, S)

where N is a set of identifiers (event names) and Rules are defined as follows:

S → S ⊓ A	S → A		
A → A + B	A → B		
B → B ; C	B → C		
C → C | D	C → C || D	C → D	
D → E^	D → E*		
E → N	E → (S)	E → N { S }	
N → !M	N → ?M	N → M	N → NULL
M → I↓	M → I↑	M→ I	
I → event_identifier			

The following rules specify the semantics of BP operators by the generated language. We denote the language generated by the protocol A as L(A), a, b denote event names, α, β are protocols and $t_\alpha \in L(\alpha)$, $t_\beta \in L(\beta)$ are traces.

Empty protocol (NULL)
The resulting set of traces is empty.
L(NULL) = ∅.

Nested incoming call (?a{α })
Curly braces are an abbreviation ?a{ α } = ?a↑ ; α ; !a↓.
L(?a{ α }) = L(?a↑; α ; !a↓)

Simple incoming call (?event_identifier)
Event names with ? and without the request/response symbol are the abbreviations used for describing the incoming method invocations, in which there are no other events emitted or absorbed; ?a = ?a{} = ?a↑ ; !a↓.
L(?a) = { ?a↑, !a↓ }

Simple outgoing call (!event_identifier)
Event names with ! and without the request/response symbol are the abbreviations used for describing the outgoing method invocations, in which there are no other events emitted or absorbed; !a = !a↑ ; ?a↓.
L(?a) = { !a↑, ?a↓ }

Alternative (α + β)
The communication described by α or by β can be exhibited by an agent.
L(α + β) = L(α) ∪ L(β)

Sequence (α ; β)
A trace generated by α ; β ia a concatenation of α followed by β.
$L(\alpha ; \beta) = \{ t_\alpha, t_\beta ; t_\alpha \in L(\alpha) , t_\beta \in L(\beta) \}$

Repetition (α*)
A traces generated by α is repeated zero or more times.
L(α*) = L(α) ∪ L(α ; α) ∪ L(α ; α ; α) ∪ ...

And-Parallel execution (α | β)
The resulting language consists of all possible interleaving of traces from L(α) and L(β). For example:

a | b = (?a↑ ; !a↓) | (?b↑; !b↓) = (?a↑ ; !a↓ ; ?b↑; !b↓) + (?a↑; ?b↑ ; !a↓; !b↓) + (?a↑ ; ?b↑ ; !b↓ ; !a↓)
+ (?b↑; ?a↑ ; !a↓ ; !b↓) + (?b↑; ?a↑ ; !b↓; !a↓) + (?b↑ ; !b↓ ; ?a↑ ; !a↓).

Or-parallel operator ($\alpha \| \beta$)
The resulting language consists of interleaved traces generated by the protocol α, or β, or both. This is not the case of the | operator which requires traces from both protocols α and β to appear in the resulting trace.
$L(\alpha \| \beta) = L(\alpha + \beta + \alpha | \beta)$

Reentrancy ($\alpha^\wedge$)
The operand is reentrant, same as for $n \geq 0$ parallel α.
$L(\alpha \wedge) = L(\alpha | \alpha | ... | \alpha)$

Composition ($\alpha \sqcap \beta$)
Let $t_1 = A_0 e_1 A_1 e_2 A_2 ... e_n A_n \in L(\alpha)$ and $t_2 = B_0 e_1 B_1 e_2 B_2 ... e_n B_n \in L(\beta)$, S_A is the set of all events used in protocol A, $A_i \in (E_\alpha \backslash (E_\alpha \cap E_\beta))^*$, $B_i \in (E_\beta \backslash (E_\alpha \cap E_\beta))^*$, e_i is an event, which is observed in t_1 and is emitted in t_2, or vice versa. The resulting traces will be $L(A_0^P | B_0^P ; \tau_1 ; A_1 P | B_1^P ; \tau_2 ... \tau_n ; A_n^P | B_N^P)$, where τ_i is the event e_i as internal, and A_i^P denotes the protocol derived from trace A_i by sequencing the events from A_i.

References

[1] Campbell, R. H., Habermann, A. N.: The Specification Of Process Synchronization By Path Expressions. Springer LNCS, Vol. 16, 1974, pp. 89–102.

[2] van den Bos, J., Laffra, C.: PROCOL: A Parallel Object Language with Protocols. In Proceedings of OOPSLA '89, ACM Press, 1989, pp. 95–102.

[3] van den Bos, J., Laffra, C.: PROCOL: A concurrent object-oriented language with protocols delegation and constraints. In Acta Informatica, Springer-Verlag, 1991, pp. 511–538.

[4] Meyer, B.: Object-Oriented Software Construction. Prentice Hall, 1998.

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

[6] Hoare, C. A. R.: Communicating Sequential Processes. Prentice-Hall, 1985.

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

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

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

[10] Liskov, B. H., Wing J. M.: A Behavioral Notion of Subtyping. ACM Press, 1994.

[11] ANSA, Using path expressions as concurrency guards. Technical Report 022.00, Cambridge UK, February 1993.

[12] Allen, R. J.: A Formal Approach to Software Architecture. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997.

[13] Medvidovic, N.: A Classification and Comparison Framework for Software Architecture Description Languages. TR UCI-ICS-97-02, Dept. of Information and Computer Science, University of California, Irvine, 1996.

[14] Magee, J., Dulay, N., Kramer, J.: Regis: A Constructive Development Environment for Distributed Programs. In Distributed Systems Engineering Journal, 1(5), 1994.

[15] Plasil, F., Mikusik, D.: Inheriting Synchronization Protocols via Sound Enrichment Rules. In Proceedings of Joint Modular Programming Languages Conference, Springer LNCS 1204, March 1997.

[16] Plasil, F., Balek, D., Janecek, R., Pospisil, R., Prochazka, M.: SOFAnet and SOFAnode – Basic Functionality. TR 97/12, Dept. of Software Engineering, Charles University, Prague, 1997.

[17] Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP Architecture for Component Trading and Dynamic Updating. In Proceedings of ICCDS '98, Annapolis, IEEE CS, 1998, pp. 43–52.

[18] Plasil, F., Stal, M.: An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. Software Concepts & Tools (vol. 19, no. 1), Springer 1998.

[19] Broy, M., Deimel, A.., Henn, J., Koskimies, K.,Plasil, F., Pomberger, G., Pree, W., Szyperski, C.: What characterizes a (software) component?, Software Concepts & Tools (vol. 19, no. 1), Springer 1998.

[20] Kleindienst,J.,Plasil,F.,Tuma,P.: Lessons Learned from Implementing the CORBA Persistent Object Service. Proceedings of OOPSLA'96. San Jose, CA, ACM Press 1996

[21] Mencl, V.: Component Definition Language, Master thesis, Charles University, Prague, 1998.

[22] Mikusik, D., Stranik, J., Svec, M., Visnovsky, S.: Synchronization protocols for Orbix 2.0, Dept. of Software Engineering, Charles University, Prague, 1998, http://nenya.ms.mff.cuni.cz/~svec.

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

[24] Zaremski, A. M., Wing, J. M.: Specification Matching of Software Components. ACM Transactions on Software Engineering and Methodology, 6(4), 1997.

[25] George, C., Haxthausen, A. E., Hughes, S., Milne, R., Prehn, S., Pedersen, J. S.: The RAISE Development Method. Prentice-Hall, 1995.

[26] Woodman, M., Heal, B.: Introduction to VDM. McGraw-Hill, London, 1993.

[27] OMG 98-07-01, CORBA/IIOP Specification. Revision 2.2, 1998.

[28] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P.: Object-Oriented Development: The Fusion Method. Prentice-Hall, 1994.

[29] Yellin, D. M., Strom, R. E.: Interfaces, Protocols, and the Semi-Automatic Construction Of Software Adaptors. In Proceedings of the OOPSLA ‘94, ACM Press, 1994, pp. 176–190.

[30] Issarny, V., Bidan, C., Saridakis, T.: Characterizing Coordination Architectures According to Their Non-Functional Execution Properties. IRISA/INRIA, 1998, http://www.irisa.fr/solidor/work/aster.html.

[31] Issarny, V., Bidan, C., Saridakis, T.: Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with the Aster Prototype. In Proceedings of ICCDS ‘98, 1998, http://www.irisa.fr/solidor/work/aster.html.

[32] Puntigam, F.: Types for Active Objects Based on Trace Semantics. In Proceedings of the 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, Paris, France, 1996.

[33] Puntigam, F.: Coordination Requirements Expressed in Types for Active Objects. Technical report, Institut für Computersprachen, Technische Universität Wien, Vienna, Austria, 1997.

[34] Ellsberger, J., Hogrefe, D., Sarma, A.: SDL – Formal Object-oriented Language for Communicating Systems. Prentice-Hall, 1997.

[35] Richner, T., Ducasse, S., Wuyts, R.: Understanding Object-Oriented Programs with Declarative Event Analysis. ECOOP ‘98 Workshop on Object-Oriented Reengineering, 1998.

[36] Richner, T.: Describing Framework Architectures: more than Design Patterns. Software Composition Group, Institut für Informatik, Universität Bern, 1998, http://www.iam.unibe.ch/~richner.

[37] Ducasse, S., Richner, T.: Executable Connectors: Towards Reusable Design Elements. In Proceedings of ESEC 97 Workshop of Component-Based Systems, Zurich,1997, http://www.iam.unibe.ch/~ducasse.

[38] Steyaert, P., Lucas, C., Mens, K., D’Hondt, T.: Reuse Contracts: Managing the Evolution of Reusable Assets. In Proceedings of OOPSLA ‘96, ACM SIGPLAN Notices, Vol. 31, No. 10, October 1996, pp. 268–285.

[39] Shaw, M.., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., Zalesnik, G.: Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 314–335.

[40] Lumpe, M., Schneider, J., Nierstrasz, O., Achermann, F.: Towards a formal composition language. In Proceedings of ESEC '97 Workshop of Component-Based Systems, Zurich, 1997, pp. 178–187.

[41] Demeyer, S., Rieger, M., Meijler, T. D., Gelsema, E.: Class Composition for Specifying Framework Design. In Proceedings of ESEC/FSE '97, 1997, http://www.iam.unibe.ch/~demeyer.

[42] Brand, D., Zafiropulo, P.: On Communicating Finine-State Machines. The Journal of the ACM, Vol. 30, No. 2, April 1983, pp. 323–342.

[43] Kobayashi, N., Yonezawa, A.: Type-Theoretic Foundations for Concurrent Object-Oriented Programming. In Proceedings of OOPSLA '94, ACM Press, 1994, pp. 31–45.