Bounding Component Behavior via Protocols

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

[1]*Charles University, Prague*
Department of Software Engineering
{plasil,visnovsky,besta}@nenya.ms.mff.cuni.cz
http://nenya.ms.mff.cuni.cz

[2]*Academy of Sciences of the Czech Rep.*
Institute of Computer Science
{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, it should ensure correct composition of the nested architectural abstractions (for design purposes); 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 comprehend. The behavior protocols are used on three levels: interface, frame, and architecture. The key achievements of this paper include the definition of the protocol conformance relation. Using this relation, the designer can in most cases statically verify that the frame protocol adheres to the requirements of the interface protocols, and that the architecture protocol adheres to the requirements of the frame and interface protocols.

1. Introduction

It is widely accepted that in the very 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 use, but 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 [2, 10, 14, 15].

1.1. Objects and protocols

An object interface definition can be considered as a service definition. As stated in [9], the sequences of requests 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 [2, 4, 9, 10, 17, 14] 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. [9, 14, 17], (2) via a parser accepting the valid request sequences, e.g. [4], (3) as a regular-like expression generating the valid request sequences, e.g. [2, 12]. The protocols originate in path expressions [3] which specify synchronization of procedures executed in parallel. Procol [2] might serve as an example of an object language in which protocols are used to describe both the access synchronization and the availability of an 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. As emphasized in [10], rather than simply raising exceptions when protocols are violated, it would be desirable to statically validate clients' conformance with protocols and to 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 [9] which allows to statically determine whether a protocol can be replaced by another one.

1.2. Components and protocols

Recently, the component-oriented program design has drawn a lot of attention, mainly because components provide a higher level of design abstractions than objects. 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, forming a framework resp. a higher-level component.

With respect to describing component communication, the approaches based on applying the idea of object protocols to components include [1] and [17]. In [1], the protocol is expressed via a set of recursive CSP-based equations. The protocol idea outlined in [17] is based on cooperating pairs of typed interfaces (collaborations). A collaboration description includes the protocol described as a set of sequencing constraints based on a transition system.

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. Moreover, none of these approaches address a step-by-step development of a component's protocol during the design process of the component. Thus, the goal of this paper is to address these two issues: protocol readability and support for step-by-step protocol refinement.

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 and the underlying model of 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 (CDL), and in Section 5, where the protocol conformance relation is defined. Moreover, these sections outline seamless fitting of the idea of step-by-step protocol refinement into the SOFA component model, where refinement-based component design is supported by providing both black-box view and grey-box view on a component as a part of the component's type definition. Section 6 is devoted to evaluation and open issues. Related work is discussed in Section 7. Section 8 concludes the paper by summarizing key achievements.

2. SOFA Components

2.1. Component model

The SOFA (Software Appliances) project [11] 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. Analogous with the classical concept of an object being 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".

A 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 of T. Interfaces are defined in the SOFA CDL language. In a template frame, similarly to many other architecture description languages (ADLs), an interface (type) can be instantiated as a *provides-interface* or a *requires-interface*.

The *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 adjacent 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 underlying implementation language, out of the scope of the architecture specification. When an architecture is not primitive, the nested components are viewed on the level of their frames.

2.2. CDL specification language

The specification of a SOFA component is written in the SOFA component definition language (CDL), which is based on CORBA IDL. The complete syntax of CDL is given in [7]. 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 provide the *Insert, Delete*, and *Query* operations for inserting and removing records from the database, and querying the contents of the database. 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. For this purpose, CDL includes the interface construct which specifies the interface type as a set of method signatures. Our interfaces 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);
};

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

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

```
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;
    subsume Local:lg to dbLog;
    subsume Local:da to dbAcc;
    delegate dbSrv to Local:d;
};
```

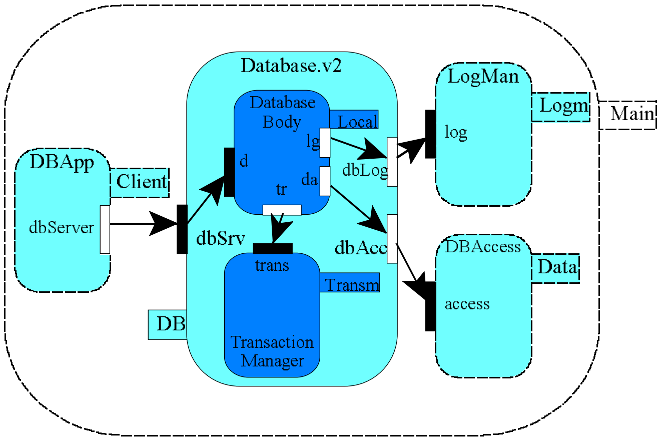

Figure 1. Database architecture

Several versions of the *Database* architecture can be specified. A possibility is to declare a version as *primitive*, i.e., with no nested components. As an alternative, the *Database* architecture version *v2* illustrates how the subcomponents are instantiated and how their ties are specified (distinguishing bind, subsume, and delegate ties). 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, e.g., for easy replacement of a subcomponent by its new version.

3. Behavior protocols

3.1. Model of communication

In this section, we describe the essence of the communication model described in detail in [13]. In this model, an agent is a computational entity handling sequences of events. As to handling events, agents can emit events, absorb events, and perform internal events. In general, we say that an agent *exhibits* some *actions*. Agents communicate via peer-to-peer connections. In principle, an agent can communicate with a finite number of agents, and two agents can communicate by means of a finite number of connections. Moreover, we assume that an agent cannot handle more than one event at a time, there is no connection delay, and an agent can emit an event only if its counterpart is prepared to accept it. We pretend that emitting and absorbing a particular event is done as one atomic action.

By *activity* of an agent A on a set of connections CS we understand the sequence of actions A exhibits on CS. By convention, this sequence is represented by the *trace of* A *on* CS. In general, a trace is a sequence of *action tokens*, each of them representing exactly one action. In an action token, the *event name* is followed by the symbol *↑* resp. *↓* which distinguishes *request* resp. *response*. To express whether an event is emitted, absorbed, resp. is internal, we prefix the event name by the *!*, *?* resp. τ symbol. There is a local event namespace associated with every connection. In order to distinguish among actions exhibited on different connections of the same agent, each event name can be qualified by a connection name. In an application, a single global namespace of connection names also exists.

An agent can be *primitive* or *composed*. A composed agent P is constructed by a *composition* of two agents A and B. The connections of P are the union of the connections of A and B. The connections through which A and B communicate with each other (*external* connections from A's and B's points of view) become *internal connections* of P; events on the internal connections of P are referred to as *internal events* of P (analogous with internal actions τ in [8]). Let C be a connection 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). If C is external both in A and P, the contribution to the corresponding traces of A and P 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.

On a set of connections CS, the set of all possible activities of the agent A 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 $\alpha_0 x_1 \alpha_1 x_2 \alpha_2 ... x_n \alpha_n \in L$, $x_i \in E_N$, $\alpha_i \in (E_L \setminus E_N)^*$, E_N is the set of all possible action tokens the event names 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 which from every trace of the language L omits all action tokens whose event names are not in N. The resulting set of words constitutes the language L'. Furthermore, the restriction of language $L_{A,CS}$ on a subset C of the connection set CS is the language $L_{A,CS}/C = \varphi_{CE}(L_{A,CS})$, where CE is the set of all action tokens, event names of which are qualified by the identification of a connection from C.

The key issue is to find a formal notation able to specify the typically infinite language $L_{A,CS}$ in a finite way. 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 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 an implementation in a programming language.

3.2. Behavior protocols

A *behavior protocol* (*protocol* for short) is a regular-like expression, which (syntactically) generates traces. The basic element of a behavior protocol is an action token or NULL (for empty protocol). A protocol can use the following operators and abbreviations, where α, β denote protocols and m denotes an event name.

Operators	
α^	reentrancy; equivalent to $\alpha \mid \alpha \mid ... \mid \alpha$
α^*	repetition; equivalent to $\alpha ; \alpha ; ... ; \alpha$
$\alpha \mid \beta$	and-parallel; an arbitrary interleaving of traces generated by α and β
$\alpha \parallel \beta$	or-parallel; stands for $\alpha + \beta + \alpha \mid \beta$
$\alpha ; \beta$	sequencing; concatenation of traces generated by α and β
$\alpha + \beta$	alternative; either α or β
$\alpha \sqcap \beta$	composition; similar to $\alpha \mid \beta$ except for when m is absorbed in a trace generated by α and emitted in a trace of β then the simultaneous participation is expressed as internal event

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↓

Intuitively, a behavior protocol can serve for expressing action ordering. For example, if we want to express that an agent emits a request *u* first, then it absorbs any number of requests *x*, *y*, or *z*, and finally a response *v* is emitted, we can describe this behavior by means of a protocol in the form *!u↑ ; (?x↑ + ?y↑ + ?z↑)* ; !v↓*.

4. Associating behavior protocols and SOFA components

4.1. Agents: components at run time

Modeling of SOFA components via agents is straightforward: Every component can be associated 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>, 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).

Distinguishing the two kinds of interfaces (provides and requires) can be reflected as employing connections with a "provides" and a "requires" end in the sense that the emitting and absorbing of events follows this pattern: A method call m(...) issued by the component 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 (CON identifies the connection). 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. In principle, the set of method names in the interface types involved comprises the event namespaces.

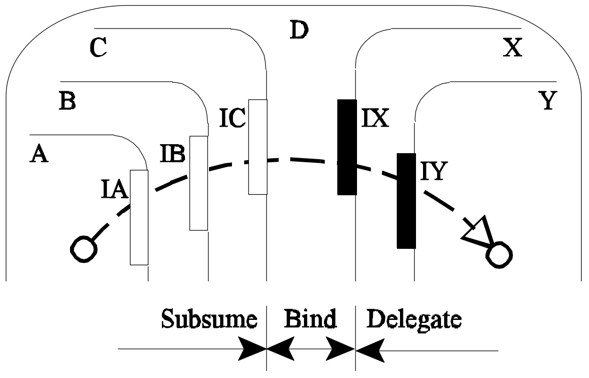

Figure 2. Binding of components

A component C which is an instance of T=<F, A> is basically specified by a set of frame instances involved in the description of A (recursively). To follow the basic philosophy of nesting in CDL specifications, a connection is specified 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, e.g., as depicted on Figure 2. Here, A, B, C, D, X, Y are the component names, IA, IB, IC, IX, IY are the names of the interface instances tied together. Thus, with respect to the composition of agents, the agents of A, B, and C share the "requires" end of the connection and the agents of X, Y share the "provides" end. The connection is internal to the D agent. Technically, the connection is uniquely identified by its *full-chain identification*, taking the form: *D.C.B.A:IA–D.C.B:IB–D.C:IC→D.X:IX–D.X.Y:IY*.

4.2. Bounding behavior of components

As emphasized in Section 3.1, an agent's behavior can be approximated by a behavior protocol. Thus, we could employ the 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 what is specified by a protocol, while the behavior on its requires-interfaces has to be "narrower" than what is 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 (recursively including the 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:

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

Needless to say, a key issue of employing the behavior protocol idea in the component design is to find meaningful sets SI and natural units for which protocols can be easily formed. Obviously, the CDL interface, frame, and architecture concepts are such natural units. We therefore introduce the interface, frame, and architecture protocols in Sections 4.3 – 4.5.

The CDL constructs, however, are inherently generic — at the CDL specification level, the protocols can only contain a generic identification of potential connections. Consequently, the protocols in CDL are specified in a generic form. In particular, the generic connection names are derived from the interface instance identifications as declared in the corresponding CDL frame and architecture constructs.

Because of the step-by-step refinement nature of the component design, the knowledge of the connection identification is also obtained on a step-by-step basis, reflecting the nesting of architectures. Each level of a template T's nesting into another template contributes incrementally to the knowledge of a part of the complete-chain identification of the connections which are involved in T. This implies a step-by-step modification of every protocol P associated with T; we call these intermediate forms of P *semi-instances* of P. After the outmost component is instantiated, each protocol in the component contains a complete-chain identification of the connections; the semi-instance of P becomes an instance of P. It should be emphasized that only an instance of a protocol can bound the behavior of a component. Anyway, bounding the behavior is a run-time issue.

4.3. Interface protocol

To capture the behavior of a service specified by a particular interface type, we enhance the interface type specification by an *interface protocol*. In principle, in an interface protocol associated with a provides (resp. requires) interface, a method invocation is to be prefixed by ? resp. !. In CDL, the interface protocol is written in its generic form, i.e., no ? and ! prefixes are used; the prefixes are automatically added when the corresponding interface type is instantiated.

To illustrate the use of interface protocols, let us consider a protocol for the *IDatabaseAccess* interface from Section 2.2. The intended use of this interface is to call the method *Open* first, then do a modification of the database by invocations of *Insert*, *Delete* and *Query*, and finally to finish the work with the database by invoking *Close*. The corresponding protocol is *Open ; (Insert + Delete + Query)* ; Close*. If the *Insert*, *Delete*, and *Query* methods were to be designed to handle requests in parallel, we could specify this intention by *Open ; (Insert || Delete || Query)* ; Close*. This indicates that parallel execution of the *Insert*, *Delete*, and *Query* methods is possible, but any two invocations of *Insert* must be done sequentially (the same holds for *Delete* and *Query*). To specify that a completely parallel invocation of these methods is allowed, the reentrancy (^) operator is to be used instead of the repetition (*), yielding *Open ; (Insert || Delete || Query)^ ; Close*.

4.4. Frame protocol

To allow the specification of a "black-box behavior" of a component, we enhance the frame specification by including a *frame protocol*. Here, the protocol specifies the acceptable order of method invocations on the provides-interfaces and the expected reactions on the requires-interfaces of the frame. Thus, a name of an action is qualified by the name of the interface instance the invoked method belongs to, and is prefixed by ? or !. Method calls can be specified as nested. This semantics can be expressed by curly brackets (Section 3.2).

For illustration, we present the frame protocols of *Database* and *DatabaseBody*:

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

In the *Database* frame protocol, the fact that each modification of the database should be logged 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*. The specification of the *DatabaseBody* frame illustrates the CDL syntax of employing frame protocol in a frame specification.

4.5. Architecture protocol

For a template T=<F, A>, an architecture protocol specifies a "grey-box" behavior of T's instances. In principle, the architecture protocol is based upon the frames of the direct subcomponents specified in A. The protocol describes the dependencies among the interfaces of F and the outmost interfaces of all subcomponents in A. In CDL, the architecture protocol is not specified directly. Our approach is to generate it, e.g., in a CDL compiler, by combining the semi-instances of the internal frame protocols using the composition operator (⊓). This eliminates the need for what would, in fact, be a manual rewriting and mechanical modification of protocols when creating the semi-instances. However, this approach does not capture special properties of the architecture, e.g., simple dependencies among internal subcomponents.

To illustrate what a generated architecture protocol looks like, let us consider the *Database* architecture version *v2* which contains two subcomponents: *Transm* (instance of *TransactionManager*) and *Local* (instance of *DatabaseBody*). When the frame protocols are modified into the corresponding semi-instances and the composition operator is applied, we obtain the following architecture protocol of *Database*:

```
(    ?<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
```

5. Protocol conformance

5.1. Definition of protocol conformance

The generic protocols specified in a template T=<F, A> constitute an obligation on the part of all potential implementations of T. The protocols of the components tied together within the template have to correspond to each other, i.e., intuitively, 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 protocols of F and A. The definition of protocol conformance reflects this intuition:

Definition: Let T=<F, A> be a template with frame protocol P_F and architecture protocol P_A, and I_1, I_2 be two interfaces with interface protocols P_{I_1} and P_{I_2}. We say that *interface protocol* P_{I_1} *conforms to interface protocol* P_{I_2} iff $L(P_{I_1}) \subseteq L(P_{I_2})$. We say the *frame protocol of* F *conforms to the interface protocols of interfaces of* F iff, for every provides-interface P in F with an interface protocol P_P, $L(P_P') \subseteq L(P_F)/\{P\}$ holds, and for every requires-interface R in F with an interface protocol P_R, $L(P_F)/\{R\} \subseteq L(P_R')$ holds, where P_P' (resp. P_R') denotes the semi-instance of P_P (resp. P_R) with respect to F. We say that *architecture protocol* P_A *conforms to frame protocol* P_F iff $L(P_F')/S_{SET} \subseteq L(P_A)/S_{SET}$ and $L(P_A)/R_{SET} \subseteq L(P_F')/R_{SET}$, where S_{SET} is the set of all F's provides-interfaces and R_{SET} is the set of all F's requires-interfaces and P_F' denotes the semi-instance of P_F with respect to A.

Claim: Let T=<F, A> be a template with frame protocol P_F and architecture protocol P_A, $\{S_i\}$ the set of all F's provides-interfaces, and $\{R_j\}$ the set of all F's requires-interfaces. If P_{S_i} is the interface protocol of S_i and P_{R_j} is the interface protocol of R_j, then $L(P_{S_i}') \subseteq L(P_F')/\{S_i\} \subseteq L(P_A)/\{S_i\}$ for all provides-interfaces S_i in F and $L(P_{R_j}') \supseteq L(P_F')/\{R_j\} \supseteq L(P_A)/\{R_j\}$ for all requires-interfaces R_i in F, where P_{S_i}' resp. P_{R_j}' denotes the semi-instance of P_{S_i} resp. P_{R_j} with respect to A and P_F' denotes the semi-instance of P_F with respect to A.

The definition of protocol conformance ensures that, having a component C as an instance of template T=<F, A>, the architecture, frame, and interface protocols form a hierarchy. The architecture protocol restricted to the frame's interfaces has to conform to the frame protocol and the frame protocol restricted to any frame's interface has to conform to the interface protocol of that interface.

As an example, consider the conformance of the architecture protocol *Database* version *v2* to the corresponding frame protocol. Following the definition above, we have to identify the restrictions and then verify the inclusions. For the requires-interfaces (*dbLog* and *dbAcc*), the restrictions for the frame protocol and for the architecture protocol are shown below. As we can see, the architecture protocol of its requires-interfaces is narrower than the frame protocol. Similarly, the same verification has to be done for the provides-interfaces. For illustration, the restriction here is done on protocols instead of the generated languages. This is not always possible, however, in the cases when we are able to automatize inclusion checking, restriction on languages can be done in an automatized way (Section 5.2).

```
//Database frame protocol restriction

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

```
//Database architecture protocol restriction

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

5.2. Checking of protocol conformance

The most important problem of design-time protocol conformance checking is the complexity of languages generated by behavior protocols (BPLs). The sequence, repetition, and alternative operators define regular languages. Based on the definition of the | and ⊓ operators, each use of these operators in a protocol can be replaced by a finite expression containing only +, ;, and *. Therefore, the languages generated by behavior protocols with no use of the ^ operator are regular and verification of their inclusion as well as a construction of the language restriction based on intersection can be done in an algorithmic way [6]. But the ^ operator damages regularity of BPLs. Even worse, BPLs are not context-free in general case. It can be easily shown that (a;b;c)^ violates the pumping lemma for context-free languages. The identification of the subclass of behavior protocols for which the inclusion verification is possible, e.g., protocols where the reentrant subprotocols are equivalent, is a hot topic in our current research. Also, we are evaluating the justification of employment of the ^ operator.

At run-time, on the contrary, it is always possible to check if the behavior of a given component is bounded by a behavior protocol (no restrictions in terms of ^ as actual number of reentrant "entries" into the protocol is known) The run-time checking is typically based on intercepting the method calls by a "protocol guard". It can also be used for run-time checking of protocol conformance, but this approach is not very useful as the run-time checks cannot prove the conformance, they can only identify any non-conformance encountered.

6. Evaluation and open issues

The main advantage of the behavior protocols is their intuitively easy-to-comprehend notation for description of communication. 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. They 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 can outweigh some loss in expressiveness and justify application of behavior protocols in ADLs.

Interface protocols are guidelines for using interfaces. They help to distinguish among different types of services which have the same interface signatures. Moreover, they provide information for component trading and enhance opportunities for checking of component design correctness. The idea of frames with frame protocols helps system designers to build a system from components without detailed knowledge of the components' internals. The frame protocol publishes information about communication among interfaces implemented by a component and thus gives advice to implementors about the component implementation. The frame protocols also provide means for reasoning about suitability of components for a specific purpose. Finally, the notion of architecture protocol improves the description of component behavior by revealing behavior at the next level of component nesting. Architecture protocols are not specified explicitly in CDL, but they are rather created by a tool which combines the frame protocols of nested components by means of ⊓.

The fact that the interface, frame, and architecture protocols form a hierarchy implies the possibility of checking compatibility of protocols at different levels of abstraction. To formalize the requirements of such a compatibility, we introduce the protocol conformance relation. There is a tool which can statically decide (in most cases) whether two protocols at different levels of abstraction comply. This allows for reasoning about template design and supports process of design refinement which we consider to be the key contribution of this paper.

The list of open issues includes: (1) Although the tool for checking protocol conformance has been implemented, some problems with the reentrant operator persist. The issue is to better define the conditions under which the reentrant operator can be used in order to preserve the possibility of checking the protocol conformance in an algorithmic way. (2) We consider to use guards for constraining method invocations in protocols. Guards could help to better understand a component's semantics, but it is not clear if the guard predicates should rely on the component methods for their evaluation, or if some abstract properties representing the internal state of the component should be defined. (3) We do not consider the issue of protocol inheritance. At present, we face the challenge to enhance the sound enrichment technique [12] to reflect protocol conformance. (4) Versioning of architectures is considered in SOFA. Compatible architectures could be rated by comparing their architecture protocols. (5) The SOFA components can be updated at run-time. An update can take a place only when a component is in a precisely defined state. The issue is to express "points of updating" in the frame or architecture protocols. (6) The conformance relation between the architecture and frame protocols, as defined in this paper, is based on separate inclusions of the provides and requires restrictions. This way, however, the interplay among the provides and requires interfaces within a frame can be lost. The issue is to find a better 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 [1]. 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 while having an expressive power strong enough to reasonably approximate the 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 a component which contains subcomponents. In our approach, frames with frame protocols are introduced for this purpose, which we consider very important for refinement-based design of components and for component updating.

The work on interfaces and protocols [17] is quite similar to our approach in the sense that it describes communication between component interfaces. It, however, concentrates only on a behavior description related to a single pair of collaborating interfaces. The specification of a component as whole is not considered, and therefore no concepts similar to our frame and architecture protocols are present. Consequently, the protocol description in [17] can hardly be used for reasoning about component composition, replacement, etc. On the other hand, we found it interesting that the description allows for bidirectional communication on a single pair of interfaces; while not supported directly in SOFA, this can be easily modeled by our communication model. In our opinion, the way we have chosen for expressing component behavior is easier to apply. Similar approach is chosen in ROOM [14] where the communication is not limited to a pair of interfaces. Also, a protocol conformance (called role substitutability) is briefly outlined here.

Reuse contracts [15] 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 this work is limited in the sense that since 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 aim at expressing such information.

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. The description of a component behavior by means of behavior protocols is precise enough to capture the necessary requirements in terms of describing the method calls ordering requirements. It is easy to read, and, at the same time, simple to create because the 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 the refinement design process, allowing to reason about component behavior on different levels of information hiding. Without unduly exposing any details of 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, 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, in many cases, allowing for reliable composition of applications. Moreover, in 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. 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), and MLC Systeme, Ratingen, Germany.

References

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

[2] 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.

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

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

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

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

[7] Mencl, V.: Component Definition Language, Master Thesis, Charles University, Prague, 1998.

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

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

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

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

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

[13] Plasil, F., Visnovsky, S., Besta, M.: Behavior Protocols and Components, Tech. report No. 99/2, Dept. of SW Engineering, Charles University, Prague, February 1999.

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

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

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

[17] 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.