Enhancing Component Specification by Behavior Description - the SOFA Experience*

Frantisek Plasil

Charles University, Faculty of Mathematics and Physics
Department of Software Engineering, Distributed Systems Research Group
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
plasil@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz

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

ABSTRACT
This paper aims at sharing with the reader the experience with behavior protocols - the component behavior description introduced in our SOFA component model and emphasize the key research challenges we have faced in this respect during its 6 years existence and development. In particular, this includes the issue of finding the "right" semantics of fulfilling the behavior contract in terms of both horizontal (client-service) and vertical (nesting) cooperation of components. The contribution of the paper is that it brings our findings published "incrementally" under one umbrella and articulates verbally what was elsewhere captured in an exact, formal way.

KEYWORDS
Behavior subtyping, component behavior, behavior contract

1. Introduction - component model

Components are believed to be an important part of emerging advanced software technologies, though the most spread, industrial component models are based on simple, low-level granularity components (COM/DCOM [16] and EJB [27]). Ironically, the idea of higher-level granularity component models, including the classic Polylith [23], Darwin/Tracta [15], Wright [5], and relatively newer CCM [14] and Fractal [8] has been still waiting for full commercial exploitation. All of these higher-level models are based on a very similar idea that we will illustrate on our experimental component model SOFA [22, 25] (which is, as well as FRACTAL, available at www.object.org). An example of a SOFA application is on Fig.1. A component features interfaces, each of them being either *provides* (black rectangle) or *requires* (white rectangle); TIRQuery has two provides and one requires interfaces, while Controller has only a single requires interface. Obviously, a provides interface defines, as, e.g., in Java, a set of services (methods) callable through the interface, while a requires interface is an abstraction of a reference to another interface.

Components can be nested: TIRQuerybory and TIR are internal to the composed component TIRQuery. In general, interfaces are tied (represented as arrows in Fig.1); more specifically, a provides interface is bound to requires interface (as the one of TIRQueryBody to the one of TIR). When nesting is applied, a tie can be delegation (direction "down" in a provides - provides tie), or subsumption (direction "up" in a requires - requires tie); for example TIRQueryBody is tied this way

*The work was partially supported by the Grant Agency of the Czech Republic (project number 201/03/0911); the results have been applied in the OSMOSE/ITEA project.

to its parent component TIRQuery). Needless to say, the "actual code" resides in primitive (non-composed) components.

Another important feature of some component models are connectors (small rectangles drown over the interface ties in Fig.1). In SOFA, semi- automatically generated at deployment time, they serve to support transparent distribution of a component-based application. Moreover, recently, connectors were employed to achieve interoperability between Fractal and SOFA [30].

Application design is supported in the following way:

(a) In top-down design/refinement from scratch, a required functionality (like TIRQuery) is first

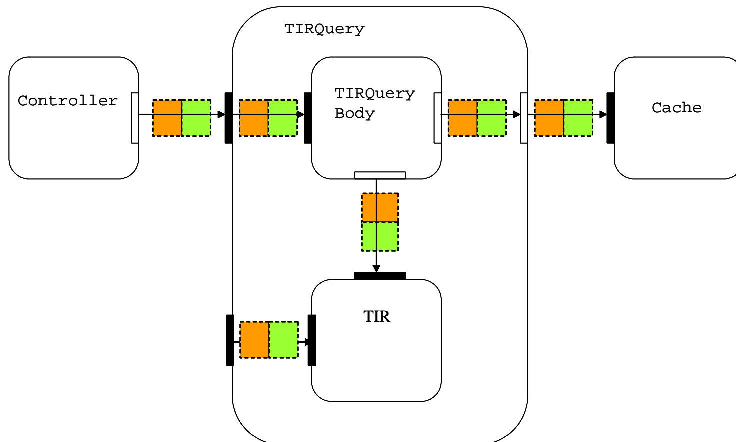

Figure 1

viewed by the environment as a black box component, *frame* in SOFA terminology. Later, by refinement, the component's *architecture* is elaborated - it is formed by the frame, bindings and external ties of the components at the first level of nesting (TIRQuerybody and TIR), and, eventually, *setup* is finalized (the architecture is placed into to the higher-level frame by turning the external ties into delegations and subsumptions.

(b) When reuse is to be applied, a potentially useful framework is identified as a composition of bound components at the level of their frames, and its generic parameters are set. Its resulting architecture is than placed into the required frame, what yields a setup. This way architecture can serve in different frames (be part of different setups).

The key abstraction (interface, frame, architecture) are explicitly reflected in SOFA CDL (architecture description language of SOFA). In Fig.2, we see how the frame `TIRQueryBody` is defined (notice the instances of interfaces), while Fig.3 illustrates the way the architecture (and setup) of `TIRQuery` is specified - among other details, the instances of two internal components are worth to agnize. Finally, and most importantly with respect to the purpose of this paper, these examples also illustrate that each of the abstractions in CDL is accompanied by behavior specification, behavior protocol, introduced in SOFA [20][21] . Even though, as emphasized a long time ago [18], "the value of component behavior is priceless, specially when it is formally, completely and accurately described", the current status is far from claiming a silver bullet has been identified (Sect. 3). From this perspective, this paper aims at giving an insight into the evolution of behavior protocols, with the goal to provide a concise overview of their features, properties, and benefits (published so far "incrementally" in [20, 21, 1, 3, 2, 4]). The rest of the paper is structured as follows: Section 2 contains an overview of the key concepts of behavior protocols and gives a whole picture view of the their potentials. Related work is discussed in Sect. 3; Sect. 4 is a conclusion.

```
interface TIRAccessInterface {
  void init();
  string queryName(in string name);
  id querryFull (in string name);
  void finish();
  protocol:
    init; (queryName + queryFull)*;
    finish
};

frame TIRQueryBody {
  provides:
    TIRQueryInterface query;
  requires:
    TIRAccessInterface tir;
    CacheInterface cache;
  protocol:
   !tir.init;
   (?query.queryName{ !cache.get;
          (!tir.query+NULL) }
    +
    ?query.queryFull {!tir.query}
    )*
    ; !tir.finish
};
```

Figure 2

```
frame TIRQuery {
  ...
   protocol ...
}


architecture CUNI TIRQuery
     implements TIRQuery {

  inst TIRQueryBody body;
  inst TIR tir;

  bind body:tir to tir:access;
  delegate query to body:query;
  subsume body:cache to cache;
   -- protocol: auto generated
}
```

Figure 3

2. Behavior Protocols

2.1. Basics

Behavior protocol [21] is an expression describing a set of traces (sequences of events). When applied to components, an event is an abstraction of issuing a method call or response to a call. For example (Fig. 2), a call of `queryName` on the interface `query` of `TIRQuerybody` is captured as `querry.querryName↑`, a response to the call as `querry.queryName↓`. Every event is *emitted* by a component and *accepted* by another component. Calling `queryName` via the interface `querry` is seen as emitting `querry.querryName↑` by `Controller`(denoted by an *event token* of the form `!Controller:query.queryName↑`from the perspective of `Controller`); at the same time `query.queryName↑` is accepted by `TIRQuery` (denoted as `?TIRQuery:querry.querryName↑` from the perspective of `TIRQuery`).

The operators employed in behavior protocols are: ";" means *sequencing*, "+" *alternative* choice, "*" a finite *repetition*, and "|" means *parallel interleaving* of the traces generated by the operands. For example, the protocol in the frame `TIRQueryBody` means: At the beginning of any run `TIRQueryBody` issues an `init` call through its requires interface `tir`, then, alternatively, accepts a call of `queryName` or `queryFull` on its provides interface `query` (this is repeated a finite number of times). Finally, it issues a call of `finish` on the requires interface `tir`. Easy to comprehend is the convention that events are denoted in a generic form; "full-fledged" event tokens are determined by the actual context the protocol is applied in. Moreover, to increase readability, handful abbreviations are used, such as `?a{...}` standing for `?a↑; ... ;!a↓`. This way, advantageously, a reaction of component to a call `a` can be efficiently expressed; e.g.`!cache.get; (!tir.queryFull + NULL)` will happen during a `queryName` call on the interface `query` of `TIRQueryBody`.

2.2. Benefits

Behavior protocols yield benefits both at the design and run time. The former are based on the concepts of frame and architecture protocols. While a frame protocol is explicitly stated in CDL (such as in the TIRQuerybody frame in Fig. 2), an architecture protocols is constructed (by a CDL compiler) as a parallel composition of the frame protocols of the components involved in an architecture. To illustrate the idea, consider the architecture from Fig. 3. Its architecture protocol is constructed as `(TIRQueryBody_frame_protocol) ⊓ (TIR_frame_protocol)`, where ⊓ is the parallel composition (some systematic, technical modification of event identification is necessary [21]).

At design time, i.e. statically, the ability of the tied components to cooperate correctly, in terms of fulfilling the contract determined by each frame protocol, can be checked. This is done(i) vertically, which in principle means looking for an answer to the question "Do the cooperating children do what the parent expects?"; (ii) horizontally, i.e."Do the children cooperate with each other without a conflict?"

Ad (i)) Checking is based on evaluating the relation *compliance* between the architecture and frame protocols; e.g.,

`(TIRQueryBody_frame_protocol) ⊓ (TIR_frame_protocol)`**compl** `TIRQuery_frame_protocol`

is evaluated (**compl** is the compliance relation. Defining a "reasonable" semantics of **compl** is a challenge though. In the course of time, we came up with three variants, now denoted as naive, pragmatic, and consensual. (a) *Naive.* Defined in [20], it is based on the idea of the classical substitutability principle [31] (in terms of the provided operations) and can be briefly articulated by "the architecture can provide more and require less than the frame does". The serious flaw of this approach is that the "best" architecture (i.e. a component replacing frame in principle) is a one which does not require anything. Consequently, we could end up with a system of smarter components that do not cooperate at all. (b) *Pragmatic.* Defined in [21], and based again on the substitutability principle but reflecting the role of required interfaces, it reads: "On a specific input sequence, there should be at lest one reaction of those of the old component, plus there should be the ability to react on all the inputs accepted by the old component. (c) *Consensual.* Defined indirectly in [4], it is based on the idea that the architecture should work well in the same environment as the frame, assuming perfectly, does (again, principle of substitutability). Since the frame basically just mediates the requests of the environment and also the requirements of the architecture, the environment can be bound directly to the architecture. Moreover, the environment can be replaced by a component working exactly as the frame, however, with inverted functionality (requires inverted to provides and vice versa). This way, compliance is transformed to the issue of fulfilment of a horizontal contract (see below).

Ad (ii)) Checking is based on identifying a possible conflict in the cooperating frames (via their interface bindings). Our approach is to employ parallel composition of corresponding frames, enhanced, however, by definition of composition errors (something what the classical process algebras like CCS [17] and CSP [e.g.24]do **not** consider). We have identified a number of composition errors: bad activity, no activity, divergence, unbound requires - for details we refer the reader to [1][3][4]. This approach allows for, e.g., identifying incomplete bindings [3], statically decide on guarantied atomicity of an update[1], claim that faulty architecture is a relative concept[4].

At run time, i.e. dynamically, employing a monitoring or interceptor technique (such as in CORBA [14)], it can be checked whether a running component obeys its frame protocol. At present we are working on an implementation of such a run-time checker.

3. Related work

There have been a huge number of publication on behavior description of software entities. In this section, we focus on the works being most influential and related to our behavior protocols in our view. The issue of "reasonable" replacement has been classically addressed via subtyping; even at present, simple subtyping, based on "subset of interfaces" as in Java and C++, is employed. The observation that "syntactical similarities" are not enough has been addressed after all via the concept of behavior compatibility [31] and behavior subtyping [12]. In the latter, "behavior" is reflected via pre- and postconditions and invariants (subtype preserves the properties of the supertype in terms of values, methods, and invariants and history as a sequence of states) . This method/object focused approach have been followed by several researchers. For instance [13] contains an attempt to enhance it to components; however, as to our "requires part", it suggest to follow the idea of expressing them as additional parameters. In [10], the authors emphasize that for monitoring, one has also, in addition to pre and post-condition errors, to identify hierarchy violations. In our view, one of the key obstacles in applying these approaches to components is that they require an explicit capturing of (object) state - this may be both a very hard-to-achieve and, potentially, limiting decision at an early stage of a component design.

The idea of expressing behavior of an object as a regular process (via traces as sequences of method calls) has been published in [19]. It even considers the role of client calls (in a simple case) via parallel composition. The importance of capturing behavior of components as sequences of events for COTS components (components of the shelf) is emphasized also in [9] where a way of identifying behavior via monitoring experiments is described. There are several component models, where behavior is described via CSP, i.e. via a label transition system with a potentially infinite number of states[5][15]. As discussed in [21], behavior protocol expressions are more readable than process algebras' equation notation, and their expressive power is strong enough to reasonably approximate behavior of components. Moreover, they always lead to finite state spaces and the compliance/equivalence relations are decidable.

The problem of behavior errors in all potential environments is addressed in [12] via an enhancement to model checking - all the environments in which a given property is satisfied are provided. In [6], via interface automata, the authors check whether there exists an environment in which a given interface (module) works correctly. In addition, they check the errors caused by faulty method call chains (e.g., recursive call of a non-reentrant method). A nice overview of component modeling focused on behavior is provided in [29]. Our behavior protocols are seen here as "interaction protocols".

4. Conclusion and future work

We provided an overview of the behavior protocols introduced in the framework of the SOFA project. In particular, we emphasized the challenge of finding a reasonable semantics of the behavior compliance and discussed the three ways we have addressed the issue in the course of time. As the most promising looks the approach based of identifying composition errors (capturing possible erroneous behavior of cooperating components). Currently, we have finished a new version of protocol checker in our SOFA model and are also working on enhancing the Fractal ADL by behavior protocols as well as making the checker available in Fractal. In a search for experimental data, we do some reverse engineering of non-trivial Fractal application containing tens of components in the believe that behavior protocols might help identify some hidden errors in large, "real-life" component applications.

Acknowledgment. The author thanks to the other people who contributed to the idea of behavior protocols, namely to Jiri Adamek, Stanislav Visnovsky, Martin Mach, Milos Besta, and Jan Kofron.

References

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

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

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

[4] Adamek, J., Plasil, F.: Erroneous Architecture is a Relative Concept., Accepted for publication in the proceeding of the SEA IASTED conference, Boston Oct. 2004

[5] Allen, R.J., Garlan, D.A.: A Formal Basis for Composing components. ACM Trans. on SW Engineering and methodology. July 1997

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

[7] Barnett,M. and all: Serious Specification for Composing components. Proceedings of the 6-th ICSE Workshop on Component-Based Software Engineering, Portland, OR, USA. Carnegie-Mellon and Monash University, 2003, http://www.csse.monash.edu.au/~hws/cgi-bin/CBSE6/Proceedings/proceedings.cgi

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

[9] Dias, M.S., Richardson,D.J.: Identiying Cause & Effect Relations between Events in Concurrent Evetn-Based Components. Proceedings of Int. Conference on Automated software Engineering (ASE 2002), Edinburgh, Sept. 2002

[10] Findler, r.B., latendresse,M., Felleisen,M.: Behavior contracts and Behavior Subtyping. Proceedins of the ESEC/FSE 2001, Vienna, ACM Press, Aug. 2001.

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

[12] Liskov, B.H., Wing, J.M.: A Behavior Notion of Subtyping. ACM Trans. on Prog. Lang. and Systems, Nov. 1994

[13] Leavens, G.T., Dhara, K.K.: Concepts of Behavior subtyping and a Sketch of their Extension to Component-Based Systems, in Leavens, Sitaraman (eds.): Foundations of Component-Based Systems, Cambridge Univeristy Press, 2000

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

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

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

[17] Milner, R.: A Calculus of Communicating Systems, LNCS 92, Springer-Verlag.

[18] Meyer,B.: On Formalism in Specifications. IEEE Software, 2(1): 6-26, Jan. 1985

[19] Nierstrasz, O.: Regular Types for Active Objects. Proceedings of OOPSLA'93, ACM, 1993.

[20] Plasil, F., Visnovsky, S., and Besta, M.: "Bounding Behavior via Protocols," Proc. TOOLS USA '99, 1999.

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

[22] 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.

[23] Purtilo, J. M.: The Polylith Software Bus. ACM Transactions on Programming Languages and Systems, 16(1), 1994.

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

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

[26] The OSMOSE project, http://www.itea-osmose.org/

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

[28] Soundarajan, N., Fridella, S.: Enriching Behavior Subtyping. TR Dept. of Computer and Information Science OSU Columbus, 1999

[29] Roshandel, R., Medvidovic, N.: Relations Software Component Models: TR UCCSE 2003-504, 2003

[30] Tuma,P.: SOFA OMG Deployment Framework. Presentedat the ObjectWeb Deployment Workshop, CU Prague, July 2003 (slides available at www.objectweb.org)

[31] Wegner, P., Zdonik, S.B.: Inheritance as an Incremental Modification Mechanism. Proceedings of ECOOP 1988, Springer LNCS 322, 1988