Electronic Notes in Theoretical Computer Science 82 No. 5 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 12 pages

On the coherence of component protocols

Andrés Farías [1] Yann-Gaël Guéhéneuc [2,3]

École des Mines de Nantes
4, rue Alfred Kastler – BP 20 823
44 307 Nantes Cedex 3
France

Abstract

Component-based programming promises to ease the construction of large-scale applications. The construction of applications using components relies on the notion of interfaces. However, the notion of interfaces provided by current component models is restricted: In particular, it does not include behavioral information to define the protocols of the components: Sequences of service requests. The lack of behavioral information limits our trust in components: Security, reuse, and quality relate directly on this missing information. In this paper, we consider the problem of verifying if a component implementation respects the protocol specified during its design. First, we define a notion of coherence between protocols and an algorithm to verify the coherence between two protocols. Then, we describe an algorithm to extract the protocol of a component from its source code. Finally, we present a tool that enables the static verification and enforcement of the notion of coherence.

1 Introduction

Component-oriented programming promises to simplify the construction of large-scale applications by using components off the shelf. The construction of applications using components relies on the notion of *interfaces*. Interfaces describes the services components offer and require: Their syntax and semantics.

The developpers of components specify these interfaces during the design phase. However, the specifications of the interfaces cannot be always easily translated into the components source code. Therefore, developpers, designers, and users of components must use specialized tools to *verify* that the implementations of components comply with their specified interfaces.

[1] E-mail: afarias@emn.fr
[2] E-mail: guehene@emn.fr
[3] This work is partly funded by Object Technology International, Inc. – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

In particular, developers must ensure that their implementations of components respect allowed sequences of service requests. Sequences of service requests determine the order in which components can request or accept services from their collaborators. Sequences of service requests define the *protocols* of the components.

Protocols are not explicit in most industrial component models. Work proposed to *integrate* protocols in academic component models [5,16,19,12,20] has produced interesting results.

In [5], we defined protocols using *finite state machines* (FSM) [8]. We developed a set of operators to manage FSM-based protocols. This set of operators is interesting because it verifies a property of substituability. The property of substituability ensures that compositions of components still verify the protocols of the composed components.

We aim at integrating these results with industrial component models. The integration of explicit protocols with an industrial component model (as with any academic component model) should results in *advanced security*, *better reuse*, and *improved quality* of component-based applications [4].

However, the integration of protocols in an industrial component model raises *new* problems which have no answer yet. In this paper, we discuss the problem of coherence between protocols specifying the components behavior, at the design level, and the sequences of service requests that components really perform.

Introductory example

We consider a simple example: A chat server for the publication of messages among clients. The architecture of the chat server is shown in Figure 1. The `ChatServer` component offers services for clients to login and to logout, to publish messages, and to search for already published messages. This component delegates the responsibility of some of its services to collaborators: Services `login()` and `logout()` call the `login()` and `logout()` services provided by the `Login` component, for example.

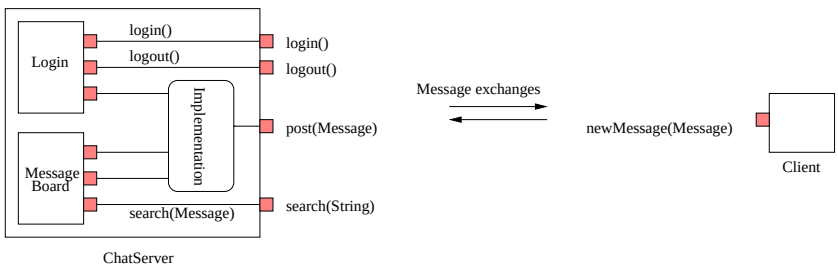


Fig. 1. Architecture of the chat server component

At run-time, the availability of the `ChatServer` services depends on its state and on the states of its clients. Clients can publish messages only if they are already logged in: They can publish messages by calling *first* the `login()`

service and *then* the `post(Message)` service. The protocol of the chat server defines these constraints on sequences of service requests.

Question

Then, the following question arises naturally: *How to ensure the coherence between the protocol defined at the design-level and the sequences of service requests followed by a component?* For examples, how do we ensure that the implementation of a `Client` respects the protocol defined by the designers of the chat server? If we change the implementation of the `Login` component, how do we verify that it still follows the protocol defined at the design-level?

This article is structured as follows: In section 2, we present our general model of components with explicit protocols. In section 3, we define a notion of coherence between the protocols of a component specified at design time and its implementation, and we present a tool for extracting the protocol from code source to describe interface-level protocols and verifying coherence between protocols. In section 4, we present related work. Finally, we conclude and we give some directions for future work.

2 A component model with explicit protocols

In this section, we present our model of components with explicit protocols and describe the life cycle of components and how problems of coherence arise.

2.1 Component model

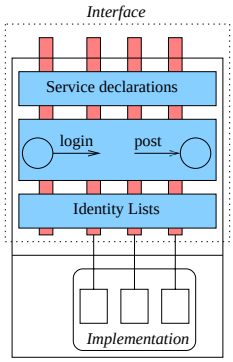
We base our study of protocols on a component model with explicit protocols [5]. We consider a component as a logical unit providing an interface consisting of a set of services (methods), a protocol, and a set of lists of collaborator identities.

Informally (a formal description is presented in [5]), the semantics of such an interface is defined as follows: The set of services corresponds to services offered by the component to its collaborators; The protocol defines all the valid sequences of service requests between a component and its collaborators, using a FSM.

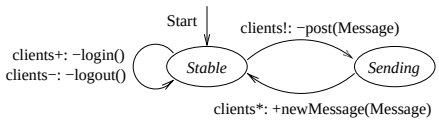
In the FSM, a transition between two states is labelled with the name of a service, a direction specifying if the transition corresponds to a service request *to* the component or *from* the component, and (optionally) constraints on the identities of the collaborators involved in the service requests.

The list of collaborators restricts the interactions among components depending on their identities and types. Types of the collaborators are set at implementation-time, identities of the collaborators are set and may change at run-time. We use four types of identity constraints to restrict the service requests and to manage the identity lists: $l+$, to add the identity of a component; $l-$, to remove an identity; $l!$, to restrict the call or reception of a

service to a collaborator in list l; And, $l*$, to broadcast a request to all the collaborators in list l.



(a) The `ChatServer` component structure.



(b) A subset of the `ChatServer` component protocol.

Fig. 2. Structure and protocol of the `ChatServer` component.

Figure 2(a) shows the structure of the `ChatServer` component. The `ChatServer` structure includes: The `login()`service; A protocol defining constraints on the order in which services `login()` and `post()` are called; And, a list of collaborators of the `ChatServer` component containing the identities of the clients currently logged in.

Figure 2(b) presents a subset of the protocol of the `ChatServer` component. The initial *Stable* state has three transitions representing the services provided by the component (prefixed by '`-`') for adding a client (`-login()`), for deleting a client (`-logout()`), and for posting a message (`-post()`). Transitions labelled with identity constraints add clients (*clients*+) or delete clients (*clients*−), and verify that messages are published only by clients that are already logged in (*clients*!). When a client successfully publishes a message, the protocol transits to the *Sending* state and the server broadcasts the message to every connected client (*clients*∗, with `+newMessage(Message)`).

2.2 Life cycle of a component

The life cycle of a component has five main phases: Design, implementation, assembly, deployment, and run-time [10,2]. Component specifications are defined during the design phase. The specifications expressed as protocols are integrated in the component implementation, during the implementation phase.

We call the protocol specifying the behavior of a given component, at the design-level, a *design-level protocol* ($\mathcal{D}-$LP). At the implementation level, a component follows sequences of service requests that we denote as the component *implementation-level protocol* ($\mathcal{I}-$LP).

A fundamental hypothesis of component-based programming is that component implementations respect the behavioral constraints defined during the design phase. In a component model with explicit protocols, the component implementation must comply with its given $\mathcal{D}-$LP: The $\mathcal{I}-$LP must be coherent with the $\mathcal{D}-$LP.

It is possible that the implementation of a component does not respect the protocol defined at the design level. We imagine two scenarios where such an incoherence can take place. In the first scenario, designers have correctly defined the desired behavior of a component but, because of the natural complexity of design specifications and implementations, developers made errors when implementing the component. In a second scenario, designers, who defined the $\mathcal{D}-$LP, took wrong design decisions that developers noticed and corrected at the implementation level, thus implementing a component which $\mathcal{I}-$LP is different from the given $\mathcal{D}-$LP.

We argue that it is of high importance to qualify (and to correct, if necessary) the coherence between $\mathcal{D}-$LP and $\mathcal{I}-$LP for security, reuse, and quality reasons. In the next section, we introduce of notion of coherence between protocols and present algorithms to extract the $\mathcal{I}-$LP of a component and to compare the $\mathcal{I}-$LP against the $\mathcal{D}-$LP.

3 Coherence between $\mathcal{D}-$LP and $\mathcal{I}-$LP

We introduce a notion of coherence between protocols. The notion of coherence allows to determine if two protocols, despite structural differences, define the same sequences of service requests. We present an algorithm for statically verifying the coherence between a $\mathcal{D}-$LP and an $\mathcal{I}-$LP. Then, we introduce a mechanism to extract a $\mathcal{I}-$LP from a component implementation. Finally, we propose a methodology for modifying the implementation of a component so that it obeys a given $\mathcal{D}-$LP. Figure 3 summarizes our contributions.

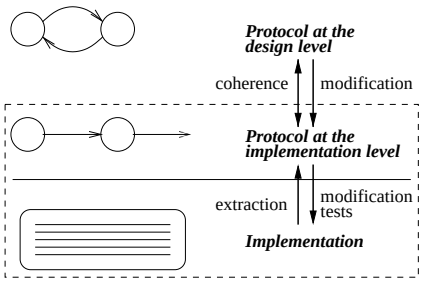


Fig. 3. Extraction, verification between $\mathcal{D}-$LP and $\mathcal{I}-$LP.

3.1 Coherence and its verification

We base our notion of coherence on the notion of substitutability as defined by Nierstrasz [14] and adapt his algorithm for the verification of protocol substitutability to protocol coherence.

Let us consider two protocols p and q. Intuitively, if p can be substituted for protocol q (denoted $p :< q$) then, from the client's point of view, protocol p behaves as protocol q, but not the contrary. Of course, if $p :< q$ and $q :< p$, then p and q behaves identically from its clients' point of view: We say that protocols p and q are *coherent* if and only if $p :< q$ et $q :< p$.

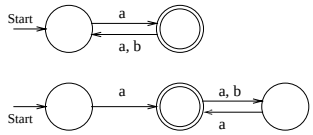


Fig. 4. Two structurally different FSM accepting the same language. These two FSM are coherent.

Formally, substitutability is defined using two notions: *Traces* and *failures* of a protocol (*cf.* [14]). Traces are the possible sequences of service requests that are accepted from a given state. In a non-deterministic FSM, accepting a trace from a given state can lead to different states. Failures are defined as the set of services that the protocol rejects from a state led by accepting a particular trace. Then, p is coherent with q if $failures(p) = failures(q)$ [14]. Figure 4 shows two protocols which are structurally different but coherent. They have the same set of traces (accepting the same set of sequences of services requests) and after accepting a given sequence of services (as for example *abab*), have the same failures (for example *b*).

We present here the algorithm to verify failures equivalence. Starting from the protocols initial states, the algorithms verifies failure sets by iteratively visiting all reachable state until every state has been visited or a difference is found. The following pseudo-code implements such algorithm:

```
if failure(i_p) != failure(i_q)
    STOP
add(i_p, i_q, initials(i_p)) to List    % initial states
While (List is not empty)
    if ((X, Y, R) with R not empty not found in List
        SUCCESS
    else
        add(X', Y', R) reachable from (X, Y, R)
        verify initials between X' and Y'
```

The worst-case complexity of the algorithm is $\mathcal{O}(2^{(n+m)})$ where n and m are the reachable states from the initial state of protocols p and q respectively.

With this algorithm, it is possible to verify the coherence between the $\mathcal{D}-$LP and $\mathcal{I}-$LP. If there is no difference between the failures of the protocols, they are coherent.

3.2 Extraction of $\mathcal{I}-$LP

To compare a $\mathcal{D}-$LP with an $\mathcal{I}-$LP, we need to extract the protocol followed by the component implementation: What the component really does. We perform the extraction by determining the protocol associated with each method of the component.

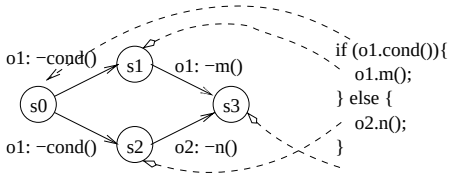


Fig. 5. Protocol corresponding to the *if* instruction.

To extract the protocol of a method, we associate a pre-defined protocol to each kind of instruction. Then, we perform composition operations on the extracted protocols to obtain the component protocol.

Figure 5 shows the protocol associated to an *if* instruction. Each state of the protocol is associated with an instruction and transitions are labelled with the method called from that state. State s_0, for example, is associated with instruction `o1.cond()` and state s_2 is associated with instruction `o2.n()`. The final state s_3 is not associated with any instruction: It is used when composing this protocol with protocols extracted from other instructions.

We have pre-defined protocols for methods calls, loops (*while*), and *try-catch-finally* expressions. An $\mathcal{I}-$LP for a method is built by composition of these pre-defined building blocks. Suitable composition operators are defined in [5].

We believe that we have a one-to-one correspondence between the component $\mathcal{I}-$LP and its source code. However, we shall formally proof the correspondence between $\mathcal{I}-$LP and source code in a future work.

3.3 Verification of coherence

We use the algorithm defined in sub-section 3.1 to compare the $\mathcal{D}-$LP with the $\mathcal{I}-$LP: To determine the coherence between two protocols. For example, Figure 6 shows a piece of source code of a component (condition *if*), the $\mathcal{I}-$LP extracted from this piece of code and the $\mathcal{D}-$LP specified by the designers. The algorithm detects an incoherence in state e_2 of protocol e with respect to state s_2 of protocol s: Different services are called from these states.

After the verification we have determined if the two protocols, $\mathcal{I}-$LP and $\mathcal{D}-$LP of a given component, are coherent:

- If the two protocols are coherent, the designers and the developers are confident that the component performs (and only performs) the task it was *designed* for. However, coherence between the $\mathcal{I}-$LP and the $\mathcal{D}-$LP only ensures that the component behave as expected by the designers, it does not prevent the component to behave differently than expected by the users.
- If the two protocols are incoherent, the designers and the developers must consider if the incoherences come from the component implementation or from the component design. If the designers prove that the component must behave following its $\mathcal{D}-$LP, then the developers must change the

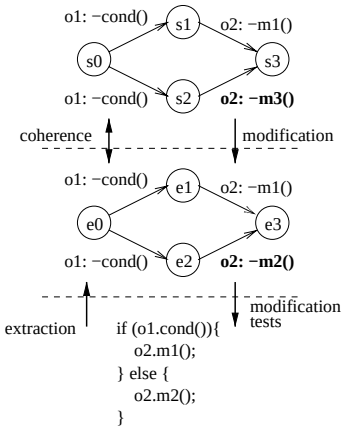


Fig. 6. Protocols and implementations.

component implementation accordingly (for example, in the case of a bug in the implementation). If the developers prove that the component must behave following its $\mathcal{I}-$LP, then the designers must change their design (for example, in the case of unanticipated technological limitations in the component run-time environment).

We now present an algorithm to modify automatically a protocol given another protocol. We use this algorithm to correct automatically a component $\mathcal{I}-$LP according to its $\mathcal{D}-$LP, or a component $\mathcal{D}-$LP following its $\mathcal{I}-$LP.

3.4 Modification of protocols

We modify the structure of an offending protocol $\mathcal{O}$P based on a given protocol $\mathcal{G}$P. We perform protocol modifications using a few primitives, such as adding or deleting a transition, correcting service requests, and adding or modifying identities verification. For each state of the $\mathcal{G}$P, we modify the $\mathcal{O}$P to ensure that the $\mathcal{O}$P has the same failures as the $\mathcal{G}$P. Second, we handle incoherences with respect to collaborator identities. Every incoherence in the $\mathcal{O}$P is corrected by enforcing the identity constraints specified in the $\mathcal{G}$P.

The following pseudo-code describes the algorithm used to modify a protocol according to a given protocol:

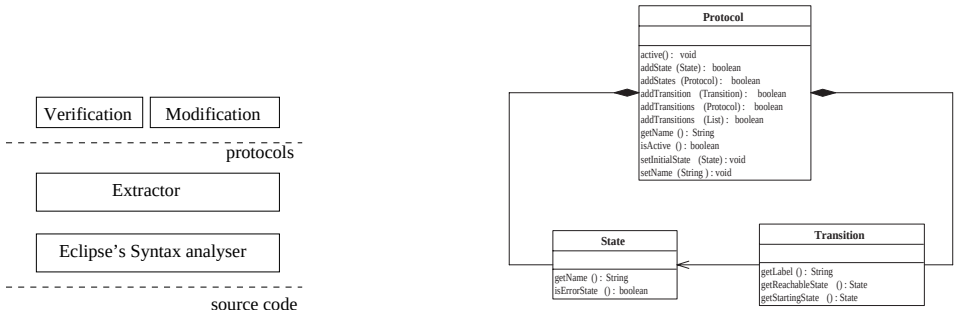
```
while (there is a state to verify)
    if failures != for states reachable by a common trace
        Correct failures of the offending protocol

while (there is a state to verify)
    If identities != for states reachable by a common trace
        Correct identities of the transitions of the offending protocol
```

If $\mathcal{I}-$LP and $\mathcal{D}-$LP are not coherent, it is possible to correct the error by either modifying the $\mathcal{I}-$LP or modifying the $\mathcal{D}-$LP: We solve the problem of coherence statically by modifying the $\mathcal{O}$P to ensure that service requests always respect the desired order, specified by the $\mathcal{G}$P. Furthermore, we add test that verify the identity of the caller or receiver defined in the protocol according to the interface-level protocol's identity lists.

It is important to remember that we have a one-to-one correspondence, by construction, between a component $\mathcal{I}-$LP and the component source code. The one-to-one correspondence ensures that the component source code will comply with the modified $\mathcal{I}-$LP, after modifying its $\mathcal{I}-$LP according to a given $\mathcal{D}-$LP. We do not either (re)generate the implementation of the component from the modified $\mathcal{I}-$LP(when $\mathcal{I}-$LP is the offending protocol), or replace the $\mathcal{D}-$LP by the $\mathcal{I}-$LP (when $\mathcal{D}-$LP is the offending protocol) because the two protocols can be structurally different, while coherent, and the replacement of one by the other could lead to information loss.

3.5 Tool support for components with explicit protocols

We have developed a tool, CwEP, (Components with Explicit Protocols), supporting the algorithms previously described. With this tool, we can extract the $\mathcal{I}-$LP from a component source code, verify its coherence with respect to a given $\mathcal{D}-$LP. The tool does not support yet $\mathcal{I}-$LP modification, we are currently working on this part of the implementation.



(a) The tool architecture.

(b) The meta-model to describe protocols.

Fig. 7. Structure and protocol of the `ChatServer` component.

CwEP is implemented based on a meta-model to describe protocol and on the compiler technology provided by the Eclipse platform [15].

We develop a meta-model to describe protocols. Figure 7(b) presents the main classes of the meta-model. Instances of class `State` represent the states of the FSM, describing protocols. Instances of class `Transition` represent transitions. Instances of class `Protocol` represent protocols. We also have representations for identity lists.

We use the Java parser provided with this platform to analyze the source code of components and to build abstract syntax trees (AST). Then, we visit each node of the AST and we build the $\mathcal{I}-$LP of the component. The construction of the $\mathcal{I}-$LP from the component AST ensures a one-to-one correspondence between the two. However, we must proof formally this one-to-one correspondence. We are currently implementing the algorithms of coherence verification and protocol modification.

4 Related work

Several works have been proposed to integrate protocols in component interfaces. A first step towards the integration of protocols on component interface is presented in [20]. Software adaptors are defined as bridges between applications working with incompatible component protocols. The research team of Charles University working on the SOFA project [16] study the use of communication protocols to extend and to adapt the object model proposed by Nierstrasz [14] to components. This team implements a prototype supporting protocol specification for components. In this prototype, the execution environment is in charge of controlling the coherence between the components and their associated protocols. But, this model remains academic and, until now, there has been little work to apply it to industrial models [11].

Specification and verification of properties using finite-state verification tools has been largely studied but slowly adopted by practitioners [3]. The transition from the results obtained in academic component models to industrial component models, such as Enterprise JavaBeans [2], Corba [17], and Miscrosoft COM and .NET [9], is scarce. A first step towards a foundation of a generic component model supporting the description of structural and behavioral specifications has been presented in [18].

In the context of security, Proof Carrying Code [13] can be named as a technique used to guarantee that a piece of code will satisfy some properties when executed. There are several kind of component specifications for describing different kind of requirements, as for example: Concurrency, general security properties, sequential constraints.

In the context of agent-based programming, Law-Governed Interaction (LGI) [12] proposes a model based on four principles to describe and to enforce the behavior of interacting agents. LGI is based on a dynamic and decentralized mechanism of message interception. While we want to ensure statically the coherence of a component implementation with respect to its design, LGI enforces dynamic verification of the agents behaviour with respect to given laws.

Several work takcle the adaptation and understanding of programs, based on the coherence of *implicit* protocols. For examples, in the context of configuration and adaptation of component communication, an aspect-oriented approach is used to correct incohenrence [7]; In the context of design patterns, approaches exist to extract design pattern instances from source code code [6,1].

5 Conclusions and future work

We return now to the question stated at the beginning of this paper : *How to ensure the coherence between the protocol defined at the design-level and the sequences of service requests followed by a component, during the implementation-phase of the component?*

We proposed a notion of coherence between $\mathcal{D}-$LP and $\mathcal{I}-$LP to compare the specifications made at the design level and the protocol followed by a component implementation. We presented an algorithm to extract a component $\mathcal{I}-$LP from its source code. We showed how to verify the notion of coherence and, finally, we proposed a prototype tool to extract a component $\mathcal{I}-$LP, and to compare it with its $\mathcal{D}-$LP.

There are two direct steps that we want to take next. First, we shall complete our prototype, so it can modify the component source code and protocol definitions. Second, we want to validate our model and algorithms by integrating our framework in the EJB component model. Then, there are several application domains in which we can use protocols. We will apply our approach to detection of behavioral design patterns, to the implementation of a security aspect in an aspect-oriented programming model, and to component specialization.

References

[1] Albin-Amiot, H., P. Cointe, Y.-G. Guéhéneuc and N. Jussien, *Instantiating and detecting design patterns: Putting bits and pieces together*, in: *proceedings of the 16^{th} ASE conference* (2001), pp. 166–173.

[2] DeMichiel, L., L. Yalçinalp and S. Krishnan, "Enterprise JavaBeans^{TM} Specification," SUN Microsystems (2001), version 2.0, Final Release.

[3] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Property specification patterns for finite-state verification*, in: M. Ardis, editor, *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)* (1998), pp. 7–15.

[4] Farias, A., Y.-G. Guéhéneuc and M. Südholt, *Integrating behavioral protocols in enterprise java beans*, in: K. Baclawski and H. Kilov, editors, *Eleventh OOPSLA Workshop on Behavioral Semantics: Serving the Customer*, 2002, pp. 80–89.

[5] Farias, A. and M. Südholt, *On components with explicit protocols satisfying a notion of correctness by construction*, in: *DOA 2002, Distributed Objects and Applications*, Lecture Notes in Computer Science **2519** (2002), pp. 995–1012.

[6] Heuzeroth, D., T. Holl and W. Löwe, *Combining Static and Dynamic Analyses to Detect Interaction Patterns*, in: *Proceedings of the Sixth International Conference on Integrated Design and Process Technology*, 2002.

[7] Heuzeroth, D., W. Löwe, A. Ludwig and U. Aßmann, *Aspect-Oriented Configuration and Adaptation of Component Communication*, in: *GCSE 2001*, number 2186 in LNCS (2001).

[8] Hopcroft, J. E., R. Motwani and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley, 2001, 521 pp.

[9] Löwy, J., "COM and .NET component services," O'Reilly, 2001.

[10] McIlroy, M., *Mass produced software components*, in: *Proceedings of the NATO Conference on Software Engineering* (1968), pp. 138–155.

[11] Menel, V., J. Adámek, A. Buble, P. Hnetynka and S. Visnovsky, *Enhancing EJB component model*, Technical Report 2001/7, Dep. of SW Engineering, Charles University, Prague (2001).

[12] Minsky, N. H. and V. Ungureanu, *Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems*, ACM TOSEM **9** (2000), pp. 273–305.

[13] Necula, G. C., *Proof-carrying code*, in: *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, 1997.

[14] Nierstrasz, O., *Regular types for active objects*, in: O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, Prentice-Hall, 1995 pp. 99–121.

[15] Object Technology International, Inc., *Eclipse platform – A universal tool platform* (2001), available at: `http://www.eclipse.org/`.

[16] Plasil, F. and S. Visnovsky, *Behavior protocols for software components*, in: *IEEE Transactions on Software Engineering* (2002).

[17] Siegel, J., "Corba 3 Fundamentals and Programming," Wiley & Sons, 2000.

[18] Teschke, T. and J. Ritter, *Towards a foundation of component-oriented software reference models*, Lecture Notes in Computer Science **2177** (2001), pp. 70–??

[19] Wydaeghe, B., "PACOSUITE, Component Composition Based on Composition Patterns and Usage Scenarios," Ph.D. thesis, Vrije Universiteit Brussels (2001).

[20] Yellin, D. M. and R. E. Strom, *Protocol specifications and component adaptors*, ACM TOPLAS **19** (1997), pp. 292–333.