ERRONEOUS ARCHITECTURE IS A RELATIVE CONCEPT*

Jiri Adamek[1], Frantisek Plasil[1,2]

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

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

ABSTRACT
The problem this paper addresses is that an architecture formed of software components can contain composition errors (introduced, for instance, as a result of the choice of a framework's parameters). The title "Erroneous architecture is a relative concept" is to emphasize that whether a composition error occurs in an architecture depends on the way the architecture is used in its environment. An important issue is finding a way to possibly statically verify that, for a given setup containing the architecture, no composition errors can occur in any run. The contribution of the paper is bringing an evidence that this can be done by employing behavior protocols and their consent operator.

KEY WORDS
Faulty software architecture, framework, component behavior

1. Introduction and Motivation

1.1. Software Component Background

In addition to the industrial component models [1, 2], which operate with simple, object-granularity level components, there are a number of advanced component models [3, 4], all based on a very similar idea: A component features interfaces, each of them being either *provides* or *requires*. (Terminology slightly differs, we stick with our SOFA[5] component model inspired by Darwin [6]). A provides interface denotes the services offered by a component. It consists of a set of provided methods; this set determines the *type* of the interface. Provides interfaces are very similar to interfaces in Java. A requires interface R expresses the fact that a component needs to call methods on a provides interface P of another component to work properly (i.e., a requires interface is an abstraction of a reference to another interface). Again, it consists of a set of required methods determining the interface type. A requires interface R has to be connected to a provides interface of the same type. This connection is realized by an interface tie – provision of a (possibly remote) reference in the implementation view.

Many component models allow for component nesting to support top-down design and refinement. Figure 1 shows a *composed component* (`FORECAST`), providing the weather forecast service. It consists of three *subcomponents* (`SWITCH`, `CACHE`, and `ENGINE`). The small dark/white boxes denote provides/requires interfaces. The caption of an interface shows the name of the component that features the interface, local name of the interface and methods which are provided/required. For instance, `CACHE:C` is a provides interface of `CACHE`, its local name is `C` and contains the `get` and `put` methods.

A tie between a requires interface and a provides interface is called *binding* (e.g. `SWITCH:C->CACHE:C` in Fig. 1). If nested components are considered, a tie can also have the form of *delegation* (a call on a provides interface of the parent component is forwarded to a provides interface of a subcomponent), e.g. `FORECAST:F1->SWITCH:F1` in Fig. 1, or *subsuming* (a call on a requires interface of a subcomponent is forwarded to a requires interface of the parent component), not employed in the setting in Fig. 1.

1.2. Running example - settings

We will illustrate all the concepts used in this paper on the example from Fig.1. As mentioned in Sect 1.1, the `FORECAST` component provides weather forecast for a given region at a given time (the region and time are passed as parameters to the `query` method from the `FORECAST:F1` or the `FORECAST:F2` interfaces). The functionality of `FORECAST:F1` and `FORECAST:F2` is the same, the interface is duplicated to provide the service to two clients – we do

*The work was partially supported by the Grant Agency of the Czech Republic (project number 201/03/0911).

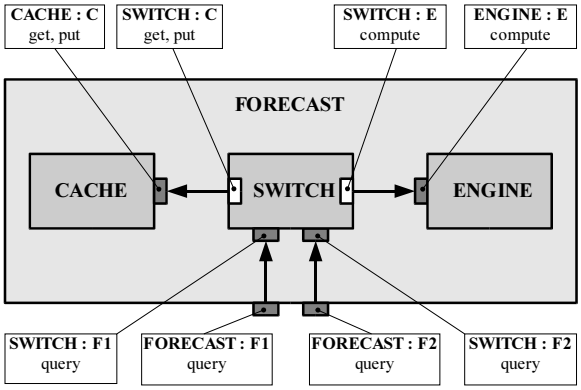

Figure 1: A composed component (FORECAST), providing the weather forecast service, consisting of three subcomponents (CACHE, SWITCH, and ENGINE)

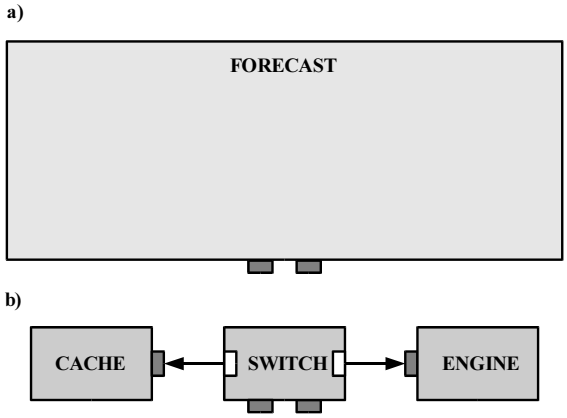

Figure 2: a) the frame of the FORECAST component; b) the architecture of the FORECAST component

not consider more clients to keep the example simple; for the same reason, we assume that each of those interfaces can be used by at most one thread at a time.

A forecast is the result of a time-demanding computation stemming from a complex mathematical model. As the forecast for a particular time and region can be requested repeatedly, it makes sense to cache the results of computations. Therefore, when `query` is called on `FORECAST:F1` or `FORECAST:F2` (and the call is delegated to `SWITCH:F1` or `SWITCH:F2` respectively), the `SWITCH` component first asks `CACHE`, whether the forecast for given parameters has been computed recently (the `get` method). If this is the case, `CACHE` returns the result immediately, which is consequently returned to the caller of `query`. If the requested forecast has not been computed yet, `SWITCH` calls `compute` on `ENGINE:E` to get one, stores it into `CACHE` (`put`) and finally returns it to the caller of `query`.

Now, let us focus on reentrance of the subcomponents. As `SWITCH` just "forwards" the calls and makes a simple decisions based on the answers of `CACHE`, we will assume that it is reentrant, i.e. the methods on `SWITCH:F1` and `SWITCH:F2` can be called by two threads simultaneously (one thread on each of these interfaces) without any negative impact on its functionality.

Since `ENGINE` does a time-consuming numerical computation, calling the `compute` method by several threads in parallel is feasible. Whether this is really possible depends on the way `ENGINE` manipulates its internal data structures. However, the reentrancy of `ENGINE` should not be understood as an implementation detail, as it influences reentrancy of `FORECAST`:

1) If `ENGINE` is reentrant (two threads can call the methods on `ENGINE:E` concurrently), `FORECAST` is reentrant as well (because `SWITCH` and `CACHE` are also reentrant).

2) If `ENGINE` is not reentrant, concurrent usage of `ENGINE:E` results in a corruption of its internal data structures. Note that calling `query` on `FORECAST:F1` and `FORECAST:F1` simultaneously can eventually result in a concurrent call of `compute` on `ENGINE:E` (in case none of the two requested forecasts have been previously computed). Therefore `FORECAST` is also not reentrant.

1.3. Software Architectures and Frameworks

The *setup* of cooperating components from Fig.1 is ready to communicate with an environment on its external connections (via ties to `FORECAST:F1`, `FORECAST:F2`). As an aside, since we have considered alternatives of the component `ENGINE` - a final setup can be considered as an instance of a component-based framework where `ENGINE` is a parameter. A particular choice of such a parameter can influence the way some other components in the setup need to communicate, which consequently can be OK for some environments but for others can mean an erroneous behavior in the setup. Obviously, a behavior specification could help analyze the problem.

To identify at which level and how behavior is to be specified it is advantageous to recall the key two approaches to component employment:

(a) In top-down design/refinement from scratch, the required functionality (like weather forecast) is first viewed by the environment as a black box component, *frame* in our terminology (Fig.2a). Later, by refinement, the component's *architecture* is elaborated (Fig.2b), and, eventually, setup is finalized. In [7] we showed that behavior protocols can be specified for both frame and architecture and their compliance verified by a tool.

(b) When reuse is considered, a potentially useful framework is identified and its parameters determined. The resulting architecture is than placed into the required frame, which finally yields a setup. An intuition-based bottom line is that (i) an architecture can serve in different frames (be part of different setups), (ii) the choice of actual parameters in a framework can cause communication errors in the resulting architecture. In [8] we showed that composition errors in an architecture can be statically detected assuming the behavior of its subcomponents is specified by means of behavior protocols.

1.4. Goal and Structure of the Paper

The message of Sect. 1.2 and 1.3 is twofold:
- it is beneficial to specify behavior,
- even though there are communication errors in an architecture, they can be avoided in a particular environment (abstracted as a frame).

From this perspective, the problem this paper focuses on reads: An architecture can contain composition errors (e.g., as the result of a choice of a framework's parameters). Whether a composition error occurs in a run of this architecture depends on the way the architecture is used in its environment, i.e. erroneous architecture is a relative concept. The key issue is how to statically verify that, for a given setup containing the architecture, no composition errors can occur in any run. The goal is to show that this is possible by employing behavior protocols and their consent operator in particular. The rest of the paper is structured as follows: Section 2 overviews the key concepts of behavior protocols, while Sect.3 presents the key contribution - shows how to check whether an architecture contains composition errors in a model environment. Related work is discussed in Sect.4; Sect.5 provides a conclusion and draws a picture of future work.

2. Behavior Protocols

2.1. Basics

To analyze the behavior of components and frameworks, a formal specification language has to be employed. In this paper, we use *behavior protocols* [7, 8, 9], which were developed as a part of our SOFA component model [5]. As discussed in [7], behavior protocols reminding regular expressions are more readable than a process algebra's notation, and their expressive power is strong enough to reasonably approximate behavior of components. In contrast to CCS [10], they always lead to finite state spaces and the compliance/equivalence relations are decidable.

A key concept behind the behavior protocols is an *event* – an abstraction of issuing a method call, a response to a call, or a general event not associated with a particular method. For instance, a call of `compute` on `ENGINE:E` is captured as `ENGINE:E.compute↑`, a response to the call as `ENGINE:E.compute↓`. Every event is *emitted* by a component and *accepted* by another component. Calling `compute` via `SWITCH:E` is seen as emitting `SWITCH:E.compute↑` by `SWITCH` (denoted as `!SWITCH:E.compute↑` from the perspective of `SWITCH`); at the same time `ENGINE:E.compute↑` is accepted by `ENGINE` (denoted as `?ENGINE:E.compute↑`). Above `!SWITCH:E.compute↑` and `?ENGINE:E.compute↑` are examples of *event tokens*.

Event tokens are primitive operands of a *behavior protocol* - an expression which specifies complex behavior as the set of desired sequences of events (traces). The operators employed in behavior protocols are: ";" which means *sequencing* of the traces described by the operands; "+" (*alternative*) stands for alternative choice of the traces, "*" denotes finite *repetition*, and "|" means *parallel interleaving* of the traces.

Let us illustrate the semantics of behavior protocols on the behavior specification of the `SWITCH`. It repeatedly accepts calls to the `query` method on both of its provides interfaces (`SWITCH:F1`, `SWITCH:F2`) in parallel. On every such call `SWITCH` reacts by calling `get` on `SWITCH:C` to find out whether the requested forecast is available in `CACHE`; depending on the finding, `SWITCH` either responds immediately to `query`, or calls `compute` on `SWITCH:E`. After `SWITCH` accepts a response from `compute` on `SWITCH:E`, it calls `put` on `SWITCH:C` to store the result into `CACHE`, and finally responds on `SWITCH:F1` or `SWITCH:F2`.

The behavior of `SWITCH` is specified by the following protocol:

```
Prot_SWITCH =
(
  ?SWITCH:F1.query {
    !SWITCH:C.get ;
    ( NULL +
      (!SWITCH:E.compute ; !SWITCH:C.put)
    )
  }
)* |
(
  ?SWITCH:F2.query {
    !SWITCH:C.get ;
    ( NULL +
      (!SWITCH:E.compute ; !SWITCH:C.put)
    )
  }
)*
```

Here, `!SWITCH:C.get` is an example of (a few) predefined useful abbreviations; it stands for `!SWITCH:C.get↑ ; ?SWITCH:C.get↓`. The abbreviation `?SWITCH:F1.query { !SWITCH:C.get ; ( ... ) }` stands for `?SWITCH:F1.query↑; !SWITCH:C.get ; ( ... ) ; !SWITCH:F1.query↓` – it describes acceptance of a call to the `SWITCH:F1.query` method, including the reactions specified by the protocol in braces, i.e. `!SWITCH:C.get ; ( ... )`. Furthermore, `NULL` stands for "empty" behavior (no event).

As $Prot_{SWITCH}$ specifies the events on the interfaces forming only the frame of `SWITCH`, we call $Prot_{SWITCH}$ the *frame protocol* of `SWITCH`. The frame protocol of `CACHE` is much more simple:

```
Prot_CACHE =
  (?CACHE:C.get + ?CACHE:C.put)* |
  (?CACHE:C.get + ?CACHE:C.put)*
```

In $Prot_{CACHE}$, we used the abbreviation `?CACHE:C.get`, standing for `?CACHE:C.get↑ ; !CACHE:C.get↓`. Finally, we introduce the frame protocols for reentrant and non-reentrant variants of `ENGINE`:

```
Prot_ENGINE-RE =
  ?ENGINE:E.compute* |
  ?ENGINE:E.compute*

Prot_ENGINE-NR =
  ?ENGINE:E.compute*
```

2.2. Group Protocol and Consent Operator

By a behavior protocol, we can describe not only behavior of a single component, but also behavior of a group of components – we talk about a *group protocol*.

Besides event tokens seen in Sect. 2.1 , we use τe for *internal events* – events occurring on a binding in a group G (here, the qualification e starts with the binding name <I_R-I_P>, where I_R, I_P are the requires and the provides interface, which are bound) – and *error tokens* to denote *composition errors*. As they are described in [8] and [11] in detail, we provide a brief overview only: A *bad activity* (denotation εe) occurs when a component C emits an event on its requires interface I_R, which is bound to a provides interface I_P of another component D, and D is not able to absorb the event on I_P (demanded by its frame protocol). *No activity* (denotation $\varepsilon\varnothing$) means that the components run into a deadlock - no component in the group can emit an event, but there is a component which has not reached its final state. *Divergence* (denotation $\varepsilon\infty$) denotes a situation when the communication of the components in the group never stops. Finally, *unbound requires error* (denotation $\varepsilon\not\rightarrow$e) occurs when a component C emits an event e on its requires interface, which is unbound.

As the behavior of every component in a group is specified separately by its frame protocol, we will advantageously construct the group protocol from these frame protocols via a repeated application of the *consent* operator [8]. Technically, the consent operator (denoted as ∇) composes the group protocols $Prot_{G1}$, $Prot_{G2}$ of two (disjoint) component subgroups G1, G2 into the group protocol Prot of the group G composed of G1 and G2. In addition to $Prot_{G1}$, $Prot_{G2}$, consent also takes the set S of all the events on the ties between the groups G1, G2 (*set of synchronizing events*) as a parameter: Prot = $Prot_{G1}$ ∇_S $Prot_{G2}$. As a frame protocol can be viewed as a group protocol of a group consisting of a single component, the group protocol of a non-trivial group G can be constructed from frame protocols of the components in G step by step by a repeated application of ∇.

Informally, the semantics of the consent operator can be described by the following three rules: (1) The behavior of G1 and G2 is synchronized on the events from S; (2) the sequences of events, which are not in S, are arbitrarily interleaved; (3) Composition errors are identified. The formal definition of ∇ can be found in the [8].

To show how ∇ works, we provide the behavior of the CACHE-SWITCH group:

```
Prot_CACHE-SWITCH = Prot_CACHE ∇_S1 Prot_SWITCH =
  (
    ?SWITCH:F1.query {
      τ<SWITCH:C-CACHE:C>.get ;
      ( NULL +
        (!SWITCH:E.compute ;
         τ<SWITCH:C-CACHE:C>.put)
      )
    }
  )* |
  (
    ?SWITCH:F2.query {
      τ<SWITCH:C-CACHE:C>.get ;
      ( NULL +
        (!SWITCH:E.compute ;
         τ<SWITCH:C-CACHE:C>.put)
      )
    }
  )*
```

Here, the abbreviation τ<SWITCH:C-CACHE:C>.put stands for τ<SWITCH:C-CACHE:C>.put↑ ; τ<SWITCH:C-CACHE:C>.put↓ and the set of synchronizing events is

```
S1 = {
  <SWITCH:C-CACHE:C>.get↑
  <SWITCH:C-CACHE:C>.get↓
  <SWITCH:C-CACHE:C>.put↑
  <SWITCH:C-CACHE:C>.put↓
}.
```

To demonstrate how consent identifies composition errors, consider the composition

$Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ = $Prot_{CACHE\text{-}SWITCH}$ ∇_{S2} $Prot_{ENGINE\text{-}NR}$

Here, $Prot_{ENGINE\text{-}NR}$ specifies the behavior of the non-reentrant ENGINE and S2 is the set of synchronizing events on the <SWITCH:E-ENGINE:E> binding.

As $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ is relatively complex (however the developer does not have to write it by hand, since it is automatically generated), we show just one of the *erroneous traces* described by $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$, (i.e. a trace which ends by a composition error):

```
?SWITCH:F1.query↑ ;
τ<SWITCH:C-CACHE:C>.get↑ ;
τ<SWITCH:C-CACHE:C>.get↓ ;
τ<SWITCH:E-ENGINE:E>.compute↑ ;
?SWITCH:F2.query↑ ;
τ<SWITCH:C-CACHE:C>.get↑ ;
τ<SWITCH:C-CACHE:C>.get↓ ;
ε<SWITCH:E-ENGINE:E>.compute↑
```

This trace corresponds to the following behavior: query on SWITCH:F1 is called, SWITCH reacts by a call to SWITCH:C.get (which is immediately returned) and by a call to SWITCH:E.compute. Then, a call to query on SWITCH:F2 is accepted, what results in another call of SWITCH:C.get and a trial to call SWITCH:E.compute. However, this results in a bad activity error (ε<SWITCH:E-ENGINE:E>.compute↑), as the non-reentrant ENGINE is not able to accept a request before it responded the previous one.

All the erroneous traces described by $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ are caused by parallel access to ENGINE:E, resulting from simultaneous access to SWITCH:F1 and SWITCH:F2.

3. Checking Composition Errors in a Given Environment

The architecture protocol $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ presented above demonstrates an important fact: although the architecture of FORECAST described by $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ contains composition errors, those errors occur only in

special cases - specifically if two threads call `query` on `FORECAST:F1` and `FORECAST:F2` in parallel. If we limited the way the methods on the frame of `FORECAST` are called accordingly - by a frame protocol - the composition errors would not occur. To allow `query` be called only sequentially, we define the frame protocol of `FORECAST` as follows:

$Prot_{FORECAST}$ =
`(?FORECAST:F1.query + ?FORECAST:F2.query)*`

To check whether the architecture contains composition errors when used in the frame with the frame protocol $Prot_{FORECAST}$, we use the following technique:

(1) We *invert* $Prot_{FORECAST}$, simply by replacing all "?" in the protocol by "!" and vice versa. This way, we obtain an *inverted frame protocol* $Prot_{FORECAST}^{-1}$, specifying behavior of a *model environment* of `FORECAST` – i.e. behavior of a hypothetical component, which, bound to all the interfaces of `FORECAST`, behaves exactly how it anticipates.

(2) We compose $Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ with $Prot_{FORECAST}^{-1}$ to see how the architecture behaves in the model environment, i.e. how it behaves when forming a setup with the `FORECAST` frame:

$Prot_{FORECAST}^{-1}$ =
`(!FORECAST:F1.query + !FORECAST:F2.query)*`

$Prot = Prot_{FORECAST}^{-1} \nabla_{S3} Prot_{CACHE\text{-}SWITCH\text{-}ENR}$ =

```
(
  τ<FORECAST:F1-SWITCH:F1>.query {
    τ<SWITCH:C-CACHE:C>.get ;
    ( NULL +
      (τ<SWITCH:E;ENGINE:E>.compute ;
       τ<SWITCH:C-CACHE:C>.put)
    )
  } +
  τ<FORECAST:F2-SWITCH:F2>.query {
    τ<SWITCH:C-CACHE:C>.get ;
    ( NULL +
      (τ<SWITCH:E-ENGINE:E>.compute ;
       τ<SWITCH:C-CACHE:C>.put)
     )
  }
)*
```

```
S3 = {
  <FORECAST:F1-SWITCH:F1>.query↑
  <FORECAST:F1-SWITCH:F1>.query↓
  <FORECAST:F2-SWITCH:F2>.query↑
  <FORECAST:F2-SWITCH:F2>.query↓
}.
```

We just remark that the abbreviation `τ<FORECAST:F1-SWITCH:F1>.query{τ<SWITCH:C-CACHE:C>.get;(...)}` stands for `τ<FORECAST:F1-SWITCH:F1>.query↑; τ<SWITCH:C-CACHE:C>.get ; (...) ; !τ<FORECAST:F1-SWITCH:F1>.query↓`.

In the resulting protocol `Prot`, there are no composition errors; this complies with what we intuitively expected: limiting the usage of `FORECAST` by a frame protocol allowing only sequential calls of `query` would eliminate composition errors. Obviously, the steps (1) and (2) give us a general method to check whether a given architecture is erroneous in the context of a given frame.

4. Related work

Probably the closest to our work is [12] where the problem of identifying a component's behavior errors in all potential environments is addressed. The approach is based on an extension of classical model checking: Checking for a given property of a component yields one of the three following results: (i) the property is preserved in all environments; (ii) the property is violated in all environments; (iii) all the environments in which the property is satisfied are furnished.

In [13], the authors focus on testing interface compatibility. Via interface automata, they check whether there exists an environment in which a given interface (module) works correctly. In addition, they check the errors caused by such method call chains which commence in the component under consideration (e.g., recursive call of a non-reentrant method). Note that, in our approach, the origin (environment or component) is not important for error identification. Contrary to both [12] and [13], choosing a pragmatic view important in practice, we check whether a given component behaves properly in a specific environment; on the other hand it might be interesting to investigate the existence of a "reasonable" environment.

The observation that the behavior of a component depends on the way it interacts with its environment (and vice versa) has been targeted by a number of researchers at different levels of granularity, ranging from dynamically modifiable Usage Policy for a single CORBA object [14], over mandatory calls [15] and the Alloy framework [16] considering cooperation among multiple plugins. To our knowledge, none of them comes up with the idea to relativize the fact that a software architecture contains communication errors. The authors of "predictable assembly" [17] envision a framework for reasoning on assembling of components featuring with specific properties. For behavior specification they consider CSP [18] as an example.

5. Conclusion and future work

We have relativized the fact that a software architecture containing communication errors is erroneous, by showing that for a particular environment, the internal communication errors can be avoided. This observation is important for component architecture reuse. The key instruments allowing to articulate the idea precisely are: (i) The separation of the "frame" and architecture" abstractions allowing the trick with inverted frame to represent a model environment ; (ii) The concept of behavior protocols and, in particular, their composition errors (we introduced in [1]) which capture possible erroneous behavior of cooperating components. Unlike typical process algebras (e.g. CCS [10], CSP [18]), composition errors reflect the inherent asymmetry of a procedure call (caller takes the initiative, while callee is passive). Currently, we are about to finish a new version of protocol checker in our SOFA model and

also working on enhancing the Fractal ADL by behavior protocols and making the checker available in Fractal. In the near future we consider identifying "the largest" frame (or "the best environment"), such that all the communication errors in a given architecture would be avoided in it.

6. References

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

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

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

[4] OMG Corba Component Model Speci.cation, http://www.omg.org/technology/documents/formal/components.htm

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

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

[7] F. Plasil, S. Visnovsky, Behavior Protocols for Software Components, *IEEE Transactions on Software Engineering, 28*(11), 2002

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

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

[10] R. Milner, *A Calculus of Communicating Systems* (LNCS, Springer-Verlag., 1992)

[11] J. Adamek, F. Plasil, 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

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

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

[14] W. DePrince Jr., C. Hofmeister, Enforcing a lips Usage Policy for CORBA components, *Porceedings of EUROMICRO'03*, IEEE CS Press, 2003

[15] M. Barnett 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

[16] R. Chatley, S. Eisenbach, J. Magee, Modeling a Framework for Plugins, *Proceedings of the SAVCBS 2003 Workshop*, Helsinki, Finland (http://www.cs.iastate.edu/SAVCBS/)

[17] K. C. Wallnau, Volume III: A Technology for Predictable Assembly from Certifiable Components, CMU/SEI-2003-TR-009

[18] A. W. Roscoe, *The Theory and Practice of Concurrency* (Prentice-Hall, 1998)