

Software Component Verification: On Translating Behavior Protocols to Promela* Technical Report

Jan Kofron

October 23, 2006

*Department of Software Engineering
Charles University in Prague
Czech Republic
kofron @ neny.ms.mff.cuni.cz*

*Institute of Computer Science
Academy of Sciences of the Czech Republic
Czech Republic
kofron @ cs.cas.cz*

Abstract

Using software components is a modern approach for building extensible and reliable applications. To ensure high dependability, a component application should undergo verification, e.g. model checking, to prove it has certain properties. The implementation of an application is usually too complex to be verified at a formal level; therefore, a model being an abstraction of the implementation is to be used. Behavior protocols [10] are a platform for modeling of software component behavior. In this paper, we propose a method for translation behavior protocols to Promela [7], which is consequently used as the input for the Spin model checker [7]. Having the Promela code describing the component behavior, one can efficiently check for the behavior compatibility and LTL (Linear Temporal Logic) properties of cooperating software components.

1 Introduction

Using software components for building reliable (distributed) applications belongs to modern and promising trends of the future of software development. Each software component is potentially a subject for future reuse; clearly defined interface and semantics are thus a necessity. To combine components from various vendors, the developer needs a common way for component specification. Unlike description of component interfaces (ADL's, headers, ...), a standard for the specification of component semantics (behavior) has not been established yet.

In this paper, we focus on checking behavior compatibility in hierarchical component models like Fractal [11] and SOFA [6]. In these component models, there are two kinds of components: *primitive* components that have no internal structure (from the architectural point of view) and are directly implemented in a programming language, e.g. Java, and *composite* components consisting of other (either primitive or composite) components. The architecture of an application, i.e., the way the components are connected and nested, is described in an *Architecture Description Language* (ADL) file. Here, the specified component behavior or a link to an external file containing the specification may be present.

*This work was partially supported by the Grant Agency of the Czech Republic project GD201/05/H014 and the Czech Academy of Sciences project 1ET400300504.

1.1 Goals and Structure of the Paper

The goal of our research is to devise a method for verification of behavior compatibility of co-operating components of real-life applications specified using behavior protocols [10] as well as verifying LTL (Linear Temporal Logic) properties of particular components.

Behavior protocols [10] are a method for specification of software component behavior in the SOFA [6] and Fractal [1] component models. Having an entire component application specified by behavior protocols, one can check for behavior compatibility of the application's components.

In this paper, we present how component behavior compatibility written in behavior protocols can be checked in the Spin model checker [7]; since Spin uses Promela[7] as its input language, we also propose an algorithm for translating behavior protocols into Promela and discuss the benefits of this approach.

The structure of the paper is as follows: In Sect. 2 we describe behavior protocols. In Sect. 3 we describe details of the behavior compatibility check, while in Sect. 4 we show how specified behavior can be converted into Promela. Sect. 5 evaluates the contribution. Sect. 6 discusses related work, while Sect. 7 concludes the paper and proposes possible directions of our future work.

2 Behavior Protocols

A *behavior protocol* [10] is an expression describing the behavior of a component; the *behavior* means the activity on component interfaces viewed as finite sequences (traces) of accepted and emitted method call events. A behavior protocol is syntactically composed of event denotations (tokens), operators and parentheses. For a method m on an interface i , there are four event token variants:

Emitting an invocation:	$!i.m^\wedge$
Accepting an invocation:	$?i.m^\wedge$
Emitting a response:	$!i.m\$$
Accepting a response:	$?i.m\$$

Furthermore, three syntactic abbreviations of method calls are defined:

- Issuing a method call: $!i.m$ is an abbreviation for $!i.m^\wedge;?i.m\$$
- Accepting a method call: $?i.m$ is an abbreviation for $?i.m^\wedge;!i.m\$$
- Processing of a method: $?i.m\{expr\}$ stands for $?i.m^\wedge;expr;!i.m\$$ meaning that the behavior protocol ' $expr$ ' defines the m 's reaction to the call in terms of emitting and accepting other events.

The operators in behavior protocols include those used in regular-expressions ('+', ';', and '*') and a special one (the parallel operator '|' generating an alternative of all possible interleavings of operands' event tokens).

As an example, consider the following protocol:

```
!Callback.AddrInvalidated *
|
(
  ?Mgmt.UsePermanentDb^;
  (
    !IpMacPermanentDb.GetAddr *
    |
    (!Mgmt.UsePermanentDb$;
     ?Mgmt.StopUsingPermanentDb^)
  );
);
```

!Mgmt.StopUsingPermanentDb\$
)*

The protocol describes behavior of a DHCP server, which can behave in two different modes: (1) IP addresses for new clients are automatically generated by the DHCP server (by a preconfigured pattern e.g. valid IP address range) or (2) the IP/MAC address mappings are permanently stored in an external database component and the DHCP server assigns IP addresses to new clients according to their MAC address and the mapping stored in the database.

This behavior is reflected in the behavior protocol as follows: parallel composition of the !Callback.AddrInvalidated token with the rest of the protocol means that the component may call this method any time during its execution. Additionally, the component is able to accept a Mgmt.UsePermanentDb method request. Before accepting a Mgmt.StopUsingPermanentDb method request, it must emit exactly one Mgmt.UsePermanentDb response and it may emit any number of IpMacPermanentDb.GetAddr calls. Before responding to the Mgmt.StopUsingPermanentDb, it accepts a potential response to the IpMacPermanentDb.GetAddr to accomplish this method call, if a request event of this method has been emitted and no corresponding response accepted yet. In the first mode, only !Callback.AddrInvalidated may be emitted, while in the second mode, which is entered by accepting the Mgmt.UsePermanentDb event, also methods of the IpMacPermanentDb interface are called. The second mode of DHCP server is exited by emitting the Mgmt.StopUsingPermanentDb\$ event.

3 Component Behavior Compatibility

In hierarchical component models, there are two behavior compatibility relations to be taken into account. The first compatibility relation describes the correctness of the communication among components on a particular level of nesting, while the subject of the other relation is to capture the compatibility between each component and its subcomponents. In behavior protocols, the correctness of communication of the former relation is called *horizontal compliance* or absence of composition errors. The latter one is denoted as *vertical compliance*.

For evaluation of both vertical and horizontal compliances a special composition operator *consent* [2] is used. This operator is basically a parallel composition operator synchronizing the protocols on a set of events — let us denote this set \mathcal{S} . When synchronizing, two complementary events (differing in their prefixes — ‘!event’ and ‘?event’) from \mathcal{S} form a τ -event. The emitting and accepting events are thus executed in a single atomic transition. Unlike other composition operators, application of the consent operator yields not only the traces corresponding to correct communication among components, but also *error traces* describing the erroneous behavior. The consent operator is able to capture three types of composition errors — *bad activity*, *no activity*, and *divergence*.

The bad-activity error denotes a state when a component (according to its behavior protocol) may emit an event, but there is no other component able to accept such an event¹. Technically, as behavior protocols do not support any type of multiple bindings (one-to-many, many-to-one, many-to-many), there is only one possible and known recipient of each request. In cases where a multiple binding is needed, behavior protocols are transformed via a syntactical work-around [1].

The no-activity error denotes a deadlock, i.e., a state where no component is able to perform any action (neither emit nor accept an event) and at least one component is not in an end state. As the components of an application under consideration are not required to form a closed system, some interfaces may not be bound; in this case, the emit events of such unbound required interfaces are considered to be accepted by “the environment” any time, while “the environment” is considered as being able to emit an event of an unbound provided interface only in the states a component is able to accept it. If this default behavior concerning unbound interfaces is not desirable, a component representing the environment may be constructed adjusting the expected behavior.

¹Note that in the semantics of behavior protocols requests do not block, as opposed to e.g. Java method calls; therefore, the case when the recipient is not ready to accept a request is considered as an error.

The divergence error² denotes presence of a cycle within the component behavior when there is no way to reach an accepting state. As mentioned in Sect. 2, behavior protocols describe only finite traces; therefore, each cycle from which no accepting state is reachable is considered as an error.

According to our experience, we believe that undesired infinite program execution appears very rarely when using behavior protocols as the specification platform; therefore we omit the detection of this type of errors in Promela models obtained from behavior protocols.

As an example of how the consent operator works, consider the following behavior protocols P_A and P_B and their consent composition $P_A \nabla_S P_B$ ³ where $S = \{lg.start^\wedge, lg.start^\$, lg.log^\wedge, lg.log^\$, lg.stop^\wedge, lg.stop^\$\}$:

$$\begin{aligned} P_A &: ?db.start\{!lg.start\}; \\ &\quad (?db.get\{!lg.log\}+?db.put\{!lg.log\})^*; \\ &\quad ?db.stop\{!lg.stop\} \\ P_B &: ?lg.start; ?lg.log^*; ?lg.stop \\ P_A \nabla_S P_B &: ?db.start\{\tau lg.start\}; \\ &\quad (?db.get\{\tau lg.log\}+?db.put\{\tau lg.log\})^*; \\ &\quad ?db.stop\{\tau lg.stop\} \end{aligned}$$

4 Translating Behavior Protocols to Promela

Behavior Protocol Checker [8] is a tool for evaluation of horizontal and vertical compliance on models described by behavior protocols. Evaluation of these relations employs exhaustive traversal of the model state space and thus is quite time-consuming. The current version of the Behavior Protocol Checker is limited to state spaces of the size in the order of magnitude of 10^8 states, which is not sufficient in some cases.

The Spin model checker [7] is a state-of-the-art explicit model checker featuring LTL checking abilities, bit-state hashing, and quite user friendly interface. It is able to traverse state spaces of several orders of magnitude higher sizes than Behavior Protocol Checker is. Additionally, there are a number of extensions of Spin extending the Promela language, e.g. dSpin [5], that can be used in the future for translating possible behavior protocols extensions.

These facts motivated us for formulating the “Protocols-to-Promela” translation rules allowing for checking for the composition errors and LTL properties of models described by behavior protocols in Spin.

4.1 Modeling of Communication

The main problem to solve when translating behavior protocols to Promela [7] is modeling of the component communication.

The first idea one probably gets is to use processes to model components and message channels to model communication among them. There are two modes regarding message treatment in Spin — in the first mode, the send-message command gets blocked in the case of the full message buffer, while in the second mode, the message gets lost in such a case. Unfortunately, in behavior protocols, we want each emit event that cannot be accepted immediately to cause a bad activity error. Thus, modeling such behavior using message channels would require incorporation of an algorithmic mechanism (e.g. message counting) to detect bad activity errors; still, bad activity would be probably detected at the end of the verification hardening an error trace construction and finding the error cause.

Another approach for modeling component communication is based on variables. Each component is modeled as a process; moreover, each method of an exported (provided / required)

²Sometimes referred to as *infinite activity*.

³Note that this composition yields no composition errors.

interface is associated with two boolean variables reflecting a wait for a call request (*wrq*) and response (*wrs*). Each time a component starts to wait for a method call, it sets the *wrq* variable to *true*. Later on, when another component decides to emit a call of this method, it first checks the value of *wrq*, and according to the value it either performs the call (it reassigns *false* to *wrq*) or stops the checking process by printing information about the error discovered (similarly with the *wrs* variable). As to the other types of composition errors, the no-activity is detected in a natural way as a Promela deadlock, and, as explained in Sect. 3, the divergence composition error is not detected at all.

4.2 Protocols-to-Promela Translation Rules

Now, we describe the proposed “Protocols-to-Promela” translation algorithm. As aforementioned, for each method, two boolean variables are declared⁴; we denote them *method variables*:

```
bool interface1_method1[2];
bool interface1_method2[2];
...
```

At the beginning, for each behavior protocol a Promela process is created and for all methods the protocol allows to accept in its initial state, the corresponding variable is set to true:

```
interface1_method1[0] = true;
interface1_method3[0] = true;
...
```

Consequently, each process notifies a special process **Main** about finishing its initialization. The **Main** process is used for running the other processes and synchronizing their execution at the beginning and at the end of the verification.

Now, the input protocols are parsed and corresponding Promela code is generated according to each protocol structure. The following table describes the mapping of the behavior protocol operators to Promela.

BP operator	Promela mapping
sequence ‘;’	‘;’
alternative ‘+’	if...fi with each guard representing the beginning of an alternative branch
repetition ‘*’	do...od with break
and-parallel ‘ ’	new process types each representing a parallel branch

Additionally to these basic mapping principles, there are several issues to be addressed concerning particular operators.

If the ‘+’ operator is used to combine various branches starting with an accept event (*‘?interface.method’*), after accepting a particular request all the other requests must not be accepted. Therefore, the corresponding method variables have to be reset to false. Mixing of emit and accept events in various branches of an alternative operator is considered as a bad practice and therefore not supported.

If the protocol encapsulated by a repetition operator ‘*’ starts with a request event, the **do...od** statement has to include additional conditional branch **‘:skip -> break;’** for nondeterministic termination of the cycle to model any arbitrary number of protocol repetition. Note here that the semantics of this transformation includes also an infinite number of repetitions of the protocol inside the **‘do...od’** statement.

If the protocol encapsulated by the ‘*’ operator starts with an accept event, it is up to the other components to decide how many times they would call the component associated with this behavior protocol. To model the appropriate behavior and to avoid the deadlock at the end of the

⁴Because of the Promela syntax, the method identifiers used in protocols are modified in the Promela output from *‘interface.method’* to *‘interface_method’*.

cycle, again, the ‘do...od’ statement includes an additional conditional branch. It either accept the call following after the repetition part or, if there are no further events, the ‘::endofrun -> break;’ statement terminating the cycle at the end of the verification is used. The ‘endofrun’ variable is set to true by the Main process when no other process is able to perform any action anymore.

As stated in the table above, the ‘|’ (‘and-parallel’) behavior protocol operator is modeled using a new process for each parallel branch. The processes are created by the process representing the protocol and started simultaneously (ensured by the ‘atomic’ statement). The parent process waits for termination of the child processes before it continues emitting and accepting further events. The notification is performed via a shared variable.

To enhance the readability of the the resulting Promela models, on Fig. 1, we present macros for modeling events⁵.

```
#define waitforrequest(method) method[0] = true;
#define waitforresponse(method) method[1] = true;
#define acceptrequest(method) method[0] == false; lock = false
#define acceptresponse(method) method[1] == false; lock = false
#define emitrequest(method) \
    lock == false; \
    lock = true; \
    if \
        ::method[0] -> method[0] = false; \
        ::else -> assert(false);\
    fi
#define emitresponse(method) \
    lock == false; \
    lock = true; \
    if \
        ::method[1] -> method[1] = false; \
        ::else -> assert(false);\
    fi
#define cancelwaitrequest(method) method[0] = false;
#define cancelwaitresponse(method) method[1] = false;
```

Figure 1: Basic macros for improving the readability of Promela code.

As the semantics of behavior protocols declares that various events may interleave during the execution, but only one event is executed at a time, we use the `atomic` statement to ensure the same semantics in Promela. A protocol waiting for, consequently accepting a method request, and finally emitting a method response looks as follows:

?interface.method

The corresponding Promela code fragment takes the form:

```
waitforrequest(interface_method);
acceptrequest(interface_method);
emitresponse(interface_method);
```

The counterpart, i.e., the component emitting the request and consequently accepting a response is described by the following behavior protocol:

!interface.method

Here, the corresponding Promela code fragment is:

⁵For sake of simplicity, in case of the bad activity error, no error message is printed, but only `assert(false)` is used.

```

atomic {
    emitrequest(interface_method);
    waitforresponse(interface_method);
}
acceptresponse(interface_method);

```

4.3 Example

In this section, we present an example of a translation of a simple behavior protocol to Promela. As an input, we use the behavior protocol of the DHCP server described in Sect. 2. The Promela source corresponding to this behavior protocol contains one master process type — `Dhcpserver` — that initializes all necessary variables to bring the component process to the initial state; in this case, only the `Mgmt.UsePermanentDb` method call may be accepted. After the initialization, the `Dhcpserver` process decrements the global (shared) control variable `initializer` to notify the other processes about finishing its initialization; the initial value of the `initializer` variable is set by the `Main` process (not listed here). After all processes finish the initialization, they start to communicate using method variables (e.g. `Mgmt.UsePermanentDb`). From now on, the Promela source is generated according to the structure of the input behavior protocol. The resulting Promela model corresponding to the original behavior protocol is listed on Fig. 2.

5 Evaluation

We have successfully applied the proposed technique on a non-trivial component application [1] consisting of approximately 20 components. The verification time when using the proposed approach and the Spin model checker⁶ took less than ten minutes, which we definitely consider as acceptable time.

The Promela code is generated from a behavior protocol in the time linear in the length of the input. Due to the way the semantics of behavior protocols is modeled in Promela, the resulting state space of Promela code is usually about ten times larger than the state space generated by the model specified by behavior protocols. This is because each event has to be explicitly awaited and also emitting an event means several lines of code. Compared with time required for verification of such systems using the proprietary Behavior Protocol Checker[8], Spin is more than an order of magnitude faster. Additionally, an arbitrary LTL property of the model can be verified.

As to alternative approaches, the first idea the reader probably gets after reading the sections above is to specify the component behavior directly in Promela. It is likely that the resulting code would be shorter than the code generated from behavior protocols. On the other hand, it would be significantly longer than a behavior protocols model. Nevertheless, the main problem lies in variables. Since Promela features various data types (integers, booleans, structures, enums), the application designer is in temptation to use them. Our experience shows that description of a slightly more complex system in Promela then usually yields a too large model state space impossible to traverse (even the use of bit-state hashing can't provide a reasonable level of reliability). Therefore, even a bit tricky because of absence of variables, behavior protocols provide a suitable specification platform for testing behavior compatibility of communicating components.

6 Related Work

In [3], the authors use a finite-automata-based description of component behavior and they compose automata for arbitrary components to obtain an automaton modeling behavior of a composite component. They differentiate between control and functional behavior of components. The properties are modeled using the alternation-free μ -calculus, for which an efficient model checking algorithm exists. Using reconfiguration controllers, it is possible to express and check for properties

⁶We have used hash-compact state storing method.

```

byte join = 2;

proctype Dhcpserver() {
  waitforrequest(Mgmt_UsePermanentDb);
  initializer--; initializer == 0;
  atomic {run ds1(); run ds2(); }
}

proctype ds1() {
  do
    ::atomic {
      emitrequest(Callback_AddrInvalidated);
      waitforresponse(Callback_AddrInvalidated);
    }
    acceptresponse(Callback_AddrInvalidated);
    ::skip -> break;
  od;
}

proctype ds2() {
  do
    ::acceptrequest(Mgmt_UsePermanentDb);
    join = 2;
    atomic {run ds3(); run ds4() };
    join == 0;
    atomic {
      emitresponse(Mgmt_StopUsingPermanentDb);
      waitforrequest(Mgmt_UsePermanentDb);
    }
    ::endofrun -> break;
  od;
}

proctype ds3() {
  do
    ::atomic {
      emitrequest(IIPMacPermanentDb_GetAddress);
      waitforresponse(IIPMacPermanentDb_GetAddress);
    }
    acceptresponse(IIPMacPermanentDb_GetAddress);
    ::skip -> break;
  od; join--;
}

proctype ds4() {
  atomic {
    emitresponse(Mgmt_UsePermanentDb);
    waitforrequest(Mgmt_StopUsingPermanentDb);
  }
  acceptrequest(Mgmt_StopUsingPermanentDb); join--;
}

```

Figure 2: Promela code of the DHCP server protocol.

concerning the component application state after and even during an architectural reconfiguration. However, even if the checking process proposed in this paper is performed independently for each composition level, computational requirements of the entire process are substantial even in cases of simple components; complex components are thus beyond the abilities of today's computational systems.

In the Bandera toolset [4], the Java source code is translated to Promela (or another input language of a model checker) to check various properties using Spin. In cases of complex software units, the resulting Promela model yields extremely large state spaces impossible to traverse in a reasonable time with reasonable amount of memory. However, the flexibility and power of the Promela language is demonstrated. Also, if separate checking of particular components is desired, one has to provide a testing environment [9, 12], since the Bandera toolset is able to accept closed code only.

Java PathFinder (JPF) [13] does not translate the Java source code into a modeling language; Instead, it uses a custom Java virtual machine executing the bytecode and checking for three properties: absence of failed assertions, unhandled exceptions, and absence of deadlocks. Moreover, it can be extended to check for a large scale of other properties. Again, however, JPF accepts close code only, so the problem of a suitable environment also has to be solved. As to the scalability of this approach, as Java is a programming language, the state space of even a simple application is quite large.

7 Conclusion and Future Work

In this paper, we have shown how the compliance relation for communicating components may be evaluated using behavior protocols and the Spin model checker; in particular, we have proposed a procedure for translating behavior specification in behavior protocols to Promela. The mapping of behavior protocol operators and modeling of the components' communication can be done in several ways affecting the size of the resulting model. Using the methods described in this paper, absence of composition errors may be checked even for complex system consisting of tens of components (like in [1]). As the behavior compatibility is checked for each composite component separately, extending of a component-based application by additional components usually increases the checking time requirements in an additive way only. We have also discussed the advantage of using behavior protocols in comparison with Promela as the language for component behavior specification. Nonetheless, during the work on the project [1], we have identified several issues complicating the creating of behavior specification; in particular, it is the absence of procedures/macros and variables.

Our future work will focus on extending the behavior protocol by macros that would allow reusing of protocol fragments in complex protocols. Further, we will focus on stateful components — here variables for storing the component state would be beneficial; therefore, other subject of our future work is extending behavior protocols by variables of some limited domains.

References

- [1] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil. Component reliability extensions for Fractal component model, http://kraken.cs.cas.cz/ft/public/public_index.phtml, 2006.
- [2] J. Adamek and F. Plasil. Component composition errors and update atomicity: Static analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), 2004.
- [3] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005)*, August 2006.

- [4] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
- [5] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN*, pages 261–276, 1999.
- [6] P. Hnetynka and F. Plasil. Dynamic reconfiguration and access to services in hierarchical component models. In *Proceedings of CBSE 2006*, pages 352 – 359. Springer-Verlag, 2006.
- [7] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.
- [8] M. Mach, F. Plasil, and J. Kofron. Behavior protocol verification: Fighting state explosion. *International Journal of Computer and Information Science*, 6(1), 2005.
- [9] P. Parizek and F. Plasil. Specification and generation of environment for model checking of software components. In *Presented at Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2006)*, 2006.
- [10] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.
- [11] R. Rouvoy and P. Merle. Towards a model-driven approach to build component-based adaptable middleware. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 195–200, New York, NY, USA, 2004. ACM Press.
- [12] O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proc. of the Eighteenth IEEE Int. Conf. on Automated Software Engineering*, 2003.
- [13] W. Visser, P. Mehltz, J. Penix, D. Giannakopoulou, C. Pasareanu, and M. Mansouri-Samani. Java Pathfinder, <http://javapathfinder.sourceforge.net>, 2006.