Extending Behavior Protocols With Data and Multisynchronization*
Technical report

Jan Kofron

October 20, 2006

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

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

Abstract

Using behavior protocol [1] for behavior specification of components in hierarchical components model (SOFA [2], Fractal [3]) turned out to be very beneficial if one is interested in communication errors among the application components. Recently, during specification of a Fractal component application aimed at controlling the access to the Internet at airports allowing for several types of payments for the access, several issues regarding the behavior protocols as a specification platform have arisen. The two most important are (i) insufficient expensiveness of behavior protocol language when specifying some typical behavior patterns, and (ii) insufficient performance of the behavior protocol checker — a tool used for searching for composition errors among communicating components. This paper focuses on solution of the first issue by proposing several extensions to behavior protocols.

1 Introduction

The use of software components as building blocks of (distributed) applications has become widely spread technique of software construction. Therefore, assuring the compatibility of components from various vendors is a legitimate requirement. Restricting the compatibility relation to the type-compatibility of bounded components' interfaces is according to our experiences simply not sufficient. Additional kind of semantic (behavior) specification is therefore needed in order to check for the components' behavior compatibility.

As model checking techniques applicable on the kind of testing stated above are usually based on exhaustive state space traversing and the size of a state space of a software piece (a software component) is usually too large to be traversed using today's system, a suitable method of software behavior abstraction is to be used.

1.1 Behavior Protocols

Behavior protocols are a method of software component behavior specification [1]. A behavior protocol describes an associated component behavior as a set of sequences of events appearing on component interfaces. The component interfaces are divided into two groups: provided (server) interfaces — *provides* and required (client) interfaces — *requires*. On provides, method calls are accepted and on requires, method calls are emitted.

*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.

Having the components' behavior described by behavior protocols, automated checking for components' behavior compatibility can be performed yielding the information about components' composition correctness.

1.2 Goals and Structure of the Paper

Specification of a real-life component application aimed at controlling the access to the Internet [5] has revealed several issues regarding behavior protocols as a component specification platform. One of the most important ones has been insufficient expensiveness of behavior protocol language when specifying some typical behavior patterns. Therefore, in this paper, we propose several extensions to behavior protocols making them easier to use and enabling precise description of behavior in cases where an approximate specification has have to be used.

The rest of the paper is structured as follows: In Sect. 2 we describe the original behavior protocols. In Sect. 3 particular motivations for the proposed extensions to the formalism of behavior protocols are informally described, while Sect. 4 describes all the extensions formally. Sect. 5 presents several examples illustrating the meaning of the proposed extensions. Sect. 6 concludes the paper and proposes possible direction of future work.

2 Behavior Protocols and Component Behavior

2.1 Behavior Protocols

A behavior protocol is an expression describing the behavior of a software component as a set of sequences (traces) of events. An event may be a method call emitting (`!interface.method^`), method call accepting (`?interface.method^`), emitting of a return from a method call (`!interface.method$`) and accepting such a return (`?interface.method$`). These events can be combined together using various operators (regular and special ones) thus forming regular-like expressions describing the set of possible/allowed traces a component may perform.

As an example consider a component representing a file. It provides one interface that contains five methods to manipulate the file: `open`, `read`, `write`, `close`, and `status`. The supported behavior (i) starts with calling `open`, then an arbitrary interleaving of `read` and `write` follows and finally `close` has to be called; and (ii) allows `status` to be called at anytime (in parallel with (i)). The corresponding behavior protocol takes the form (for simplicity we use an abbreviation '`methodName`' for '`methodName^; methodName$`'):

```
(open; (read + write)*; close) | status*
```

2.2 Specifying Behavior of a Software Component

Software components are basic building blocks of today's application. Dividing the application logic into several smaller pieces with both semantics and interface defined more clearly enables for automatized reasoning about software component properties. A way to check for interacting components compatibility is to use of behavior protocols for their behavior specification.

Behavior of each software component [2, 3] is described by its *frame protocol* and expresses the black-box-view behavior of the component. That means that only the events on the component provided and required interfaces are taken into account. Furthermore, in the case of composite component, i.e. a component created by composing several other components, we obtain a grey-box-view behavior where the events of the first-level subcomponents are visible. The behavior at this level is described by the *architecture protocol* of a composite component, which is created by a *consent* [6] composition of the first-level subcomponents' frame protocols. The consent composition is parametrized by a set S of events corresponding to the methods of interfaces bounded between the two components being composed (if there are more than two components they are composed in a stepwise manner). The result of this operator application is basically a parallel composition

of the particular behavior protocols but the resulting traces synchronize on the complementary (in the sense of accepting vs. emitting) events from S.

The development of a component application is based on hierarchical composition of primitive components (i.e. components that do not contain other components) creating thus composite components. Errors on a particular level, i.e. among first-level subcomponents of a component, are denoted as composition errors, while errors between adjacent composition levels, i.e. between parent and child components, are captured by the behavior compliance.

We distinguish four kinds of composition errors: *bad activity*, *no activity*, *infinite activity*, and *unbound requires error*. Bad activity denotes a situation, when a component tries to emit a call on one of its required interfaces and the component bound to this required interface by its provided interface is not able (according to its frame protocol) to accept such a call. No activity denotes a deadlock and infinite activity a livelock (i.e. some components are working, but no progress to a final state can be made). The unbound requires error denotes the situation when a component tries to emit a request on an unbound required interface.

As to the behavior compliance, each component (and also a composite one) is supposed to have a frame protocol associated that is in the case of a composite component compared with the architecture protocol. The compliance is defined as the absence of errors in the composition of the architecture protocol and *inverted frame protocol* that is created by inverting (swapping provided and required interfaces) the component frame protocol. As this reduces the problem of detecting compliance errors to the problem of detecting composition errors, only composition is taken into account from now on.

3 Extensions to behavior protocols

The following sections describe various extensions of behavior protocols reflecting the most burning and important problems. The extensions include macros for reuse of protocol parts, data for passing parameters and storing information about actual state of stateful components, multisynchronization for synchronizing more than two components, and the "*until loop*" construct for easy specification of "*service components*" behavior.

3.1 Macros

Macros are a construct for reusing source code fragments and thus both enhancing readability and making error fixes faster and at one place. They are present in a variety of languages — C, C++, Latex, Promela, etc.

In a behavior protocol describing complex component behavior, a situation when a fragment of a behavior protocol or its modification is used several times also occurs. As an example, consider the behavior protocol of the IpAddressManager component from a complex component application [5] listed on Fig. 9. A rough structure of this protocol can be described by the expression on Fig. 1.

```
initialization;
(
  use_db1
  +
  (
    (use_db1 | ?start_using_db2^); !start_using_db2$;
    (use_db2 | ?stop_using_db2^); !stop_using_db2$
  )
)
```

Figure 1: Rough structure of the protocol of the IpAddressManager component.

By using macros for "implementing" *initialization*, *use_db*1, and *use_db*2 protocol parts, both main advantages of macros are demonstrated — first, the reuse of *use_db*1 fragment simplifies a potential error fix within this fragment, and second, the readability of this expression is undoubtedly much better in comparison with the original behavior protocol. Furthermore, as the *use_db*1 and *use_db*2 fragments have very similar structure, they can be both "implemented" by a single parameterized macro.

It is clear from the example above that incorporation of macros into behavior protocols greatly enhances the readability by reusing protocol fragments and thus shortening the expressions. We decided to use the CPP [4] syntax as it is simple yet powerful enough and also used in the Promela language. Furthermore, for preprocessing of behavior protocol files with CPP macros, an already existing implementation of the preprocessor can be used, e.g. [4].

3.2 Data

When modeling component behavior, data can be used for two main purposes — (1) to pass method-specific information to the implementation of a method, and (2) to store information across multiple method calls. Introducing of data to a modeling language must be done in a careful way, because the data may be a major cause of the state space explosion. To illustrate this claim, consider the behavior protocol $?i.m1 \mid ?i.m2$. A component that behaves according to this behavior protocol is able to accept the $m1$ and $m2$ methods on the i interface in parallel. The state space generated by this protocol consists of nine states. Adding a parameter of the integer type to each method that is used in the potential method bodies blows up the state space to $9 \times sizeof(integer) \times sizeof(integer)$[1]. Moreover, in almost all cases a data type of a much smaller domain, e.g. *byte*, would be sufficient.

The control flow of a program piece, in particular a component, often depends on data values. Therefore, to model the behavior of a component precisely, data has to be incorporated into the modeling language. As an example consider a part of the behavior protocol of the *FlyTicketClassifier* component from [5] on Fig. 2.

$$
\begin{array}{l}
?IFlyTicketAuth.CreateToken\ \{ \\
\quad (\\
\quad\quad !IAfFlyTicketDb.GetFlyTicketValidity; \\
\quad\quad (!IAfFlyTicketDb.IsEconomyFlyTicket + NULL) \\
\quad) \\
\quad + \\
\quad (\\
\quad\quad !ICsaFlyTicketDb.GetFlyTicketValidity; \\
\quad\quad (!ICsaFlyTicketDb.IsEconomyFlyTicket + NULL) \\
\quad) \\
\quad + \\
\quad NULL \\
\}
\end{array}
$$

Figure 2: A Part of the FlyTicketClasiffier behavior protocol.

In the component implementation, after accepting the *CreateToken* method request, the component calls the *GetFlyTicketValidity* method either on the *IAfFlyTicketDb* or *ICsaFlyTicketDb* interface depending on the parameter passed to the *CreateToken* method. In the case of a malformed or wrong parameter, no method is called (represented by the last *NULL* token). Similarly, the validity of the parameter is checked by the *GetFlyTicketValidity* methods and depending on the result the *IsEconomyFlyTicket* method is potentially called.

[1] Assuming the integer size to be 4 bytes, the resulting state space consists of more than 10^{20} states.

Because of absence of the parameters, the behavior protocol specifies a superset of the implemented component behavior — in particular, it allows the *GetFlyTicketValidity* method on the *ICsaFlyTicketDb* interface to be called even in cases when the same method on the *IAfFlyTicketDb* interface should be called instead. Even if such imprecisions usually don't extend the state space size, a behavior incompatibility among communicating components may be missed.

To illustrate the problem with specification of a stateful component, consider the aforementioned IpAddressManager component whose behavior protocol is listed on Fig. 9. The rough structure listed on Fig. 1 can be further simplified when using a variable for remembering which of the two available databases is currently used. The resulting protocol takes the form listed on Fig. 3.

$$
\begin{array}{l}
initialization; \\
(\\
\quad (start\_using\_db2;\ stop\_using\_db2)* \\
\quad | \\
\quad (use\_db1\_or\_db2)* \\
)
\end{array}
$$

Figure 3: Rough structure of the reduced IpAddressManager protocol.

Using (state) variables, a single implementation of the *use_dbx* protocol part with conditional calls to a particular database is sufficient, as the *use_db*1 and *use_db*2 protocol parts are very similar (see Fig. 9). Therefore, also, the fix of a potential error can be made in one place.

The last concerned issue regarding the variables inside behavior protocols are method return values. As it is an one-directional data transmission as well as in the previous case of parameter passing, we do not extend behavior protocols by such a construct; the data-dependent control-flow can be modeled using an artificial method call passing necessary data.

3.3 Multisynchronization

Behavior protocols provide a natural mechanism for synchronizing two communicating components. The synchronization can be achieved by a simple emitting of a shared event, i.e., performing a method call. However, there are two issues regarding such synchronization:

1. The synchronization is done in an one-directional way so if the component on the active (emitting) side reaches the synchronization point before the other one does, a bad-activity error appears.

2. No more than two components may be synchronized in a simple way[2].

Both the aforementioned issues become a problem in cases when general synchronization of more than two components is needed. As an example, consider the following situation occurring also in [5]: There are three components A, B, and C, whose interfaces are bound as shown on Fig. 4. The (simplified) corresponding behavior protocols for components B and C are listed on Fig. 5.

The cooperation scenario is following: First, the A component creates the data necessary for initialization of the B and C components and pass it to them. After the B and C components complete their initialization, they are ready to start their work. A problem lies in construction of the protocol for the A component. The first option is to initialize the components in the order of B, C. In this case, however, after initialization of the B component through its $BPI1$ interface, the component may emit a request to the A component (using $BRI1 \rightarrow API1$ binding) before

[2]Of course, as the synchronization of a fixed number of components can be modeled by a finite automaton, it can be also described by a behavior protocol. However, the protocol is quite complex and its length grows with the number of components involved.

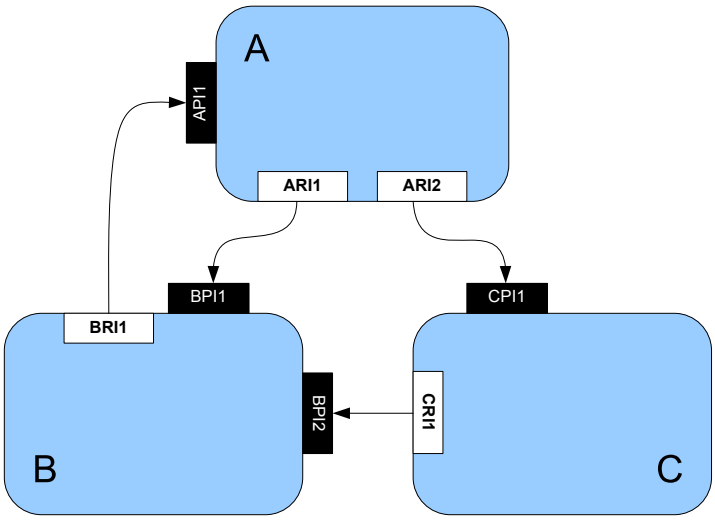

Figure 4: Synchronization of three components.

the C component has finished its initialization. This would result in a bad-activity error — the A component is not ready to accept the request at this point. Similarly, using the initialization order of C, B may result in a call to the non-initialized component B using the $CRI1 \rightarrow BPI1$ binding. The only way to cope with this problem in this particular situation is to use the parallel operator, which results in a behavior protocol listed on Fig. 6. This protocol models more than the desired behavior and thus may cause problems with encapsulation of the components into a supercomponent[3].

$B : ?BPI1.init;\ (!BRI1.m1 \mid ?BPI2.m2)$
$C : ?CPI1.init;\ !CRI1.m$

Figure 5: Behavior protocols corresponding to the B and C components on Fig. 4.

$!ARI1.init;\ (?API1.m1 \mid !ARI2.init)$

Figure 6: Behavior protocol corresponding to the A component on Fig.4.

As sketched above, there is no possible correct sequential order of initialization of the components B and C — without introducing an artificial parallelism, both orders result in a behavior protocol not composable (i.e., yielding composition errors) with the other protocols.

Therefore, we would like to have an easy-to-use synchronization mechanism that would allow for synchronization of more than two components. In [5], this problem was solved via extending the semantics of behavior protocols by introducing *atomic actions*. An atomic action is a special event consisting of other ordinary events. All the events inside of an atomic action are executed atomically in a single step. Although this enabled for behavior specification of all components involved in the demo application, several issues arose. Violation of associativity of the consent operator and much more complex formal system were the most important ones. In each proposed solution of these issues, at least one of the aforementioned problems persisted. Therefore, we don't consider

[3]In cases the A component reacts to requests from B and C by interacting with other components.

the use of atomic actions for synchronization of more components as a suitable approach. In the following paragraphs, we discuss several other options for achieving the multisynchronization.

First, let us consider the use of a shared variable for synchronization of several components. The synchronized components access this variable and, at the synchronization point, they block or continue their execution according to the variable value. There are at least two issues to be mentioned regarding this approach: (1) the initial value of the synchronization variable has to be set by an arbitrary process, and (2) the synchronization variable couldn't be reused in a simple way — all processes have to pass the synchronization point before the variable reuse, which must be assured by, e.g., using an additional variable or other notification construct. Furthermore, at least one process has to be aware of the total number of processes synchronized at a point for initial variable assignment.

Another option is to use the existing communication mechanism of emitting and accepting events and extend it by broadcast/multicast options. As a broadcast usually means delivery of a message to all nodes, which is definitely not suitable for all situations in behavior protocols, a sort of multicast seems to be a better solution. Nonetheless, at least three issues arise in this case: (1) Not only the exact number, but also the identities of all participating components has to be known to at least one component, (2) there must be the "sending" component which is the central part of this construct, and (3) the emit synchro-event must be blocked until all recipients are able to accept this event that does not correspond with the semantics of behavior protocols.

The final option discussed here is based on a special (neither emit nor accept) event that is shared by the components participating on a synchronization. This event is blocking, as expected. The set of synchronization events is passed to the composition operator when composing components together. A component is allowed to execute a synchronization event in its current state if and only if the other component being composed with this one is also able to execute this event in its current state. The components then execute the event in a single atomic step. Unlike the common method events, the synchronization event is not transformed (does not result in a τ-event) by the composition. Therefore, it can be reused in future compositions thus providing synchronization of more than two components.

3.4 Until loops

During the work on the project [5], a lack of expensiveness was identified when specifying *service component* behavior. By the term *service component* we mean a component providing some functionality through its provided interfaces. The behavior protocol of such a component takes the form "(?*provide_service*)$*$". In some cases, however, it is desirable to make the component stop providing the service; in these situations the component first should accomplish the current serving, if there is any, followed by switching to a stop state.

As an example consider the behavior protocol on Fig. 7. In this case, the component provides *service1* and *service2* as long as other components ask for the services. However, the component cannot be stopped explicitly.

$$((?service1\{!process\_service1\}) + (?service2\{!process\_service2\}))*$$

Figure 7: Example of behavior protocol of a service component without a stop state.

A behavior protocol specifying behavior of a service component that can be explicitly stopped is listed on Fig. 8. Although precisely describing the desired behavior, this protocol is not well readable and hard to understand; nonetheless, it shows that behavior of a service component that may be stopped still can be described by a behavior protocol. Therefore, introducing a syntactic abbreviation for comfortable expressing of such behavior will be sufficient.

```
((?service1{!process_service1}) + (?service2{!process_service2}))*;
(?stop +
((
  (?service1{!process_service1^;?stop^;?process_service1$}) +
  (?service2{!process_service2^;?stop^;?process_service2$}) +
  (?service1{?stop^;!process_service1}) +
  (?service2{?stop^;!process_service2}) +
);  !stop$)
)
```

Figure 8: Behavior protocol of a service component with stop.

4 Formal definition

In the following sections, we provide a formal description of all the behavior protocols extensions informally described above and discuss related issues.

4.1 Macros

The syntactical and semantical rules are adopted from well-established macro system originally used for the C language. Macros in behavior protocols are allowed to use parameters, but for the sake of simplicity, recursion of macros is not permitted. The macro definition has the following syntax:

```
#define macro_name(p1, ...,pn) macro_body
```

By this line, one declares a macro called *macro_name* with formal parameters p_1,...,p_n. In a protocol below this macro definition, all occurrences of the string '*macro_name(*a_1,..., a_n*)*' are replaced with '*macro_body*', where all occurrences of 'p_i' are replaced by 'a_i', for each $i \in 1 \ldots n$.

For macro processing, we use the C preprocessor [4] supporting a lot of preprocessing features that can be used in future, e.g. conditional preprocessing of code fragments.

Furthermore, as a technicality, we also use a macro statement for including files:

```
#include file.pr
```

4.2 Method parameters

The method parameters can be only of symbolic values — the values can be tested for equality, but other kind of comparing, e.g. '<' and '<=', is not provided. The domain of parameter values is defined at the beginning of a protocol source file via the *parvalues* statement:

```
parvalues = {value1, value2, ..., valuen}
```

By this line, n different values of method parameters are defined. There can be multiple *parvalue* statements in a protocol source file — all the values then share a single "namespace" and can be used independently of the group they are defined. In other words, the parameter values are not of any type.

To allow for passing method parameters at the caller side, the original event token denoting emitting a request $!interface.method\hat{}$ is extended in the following way:

$!interface.method(a_1, \ldots, a_n)\hat{}$

The $a_1, \ldots a_n$ are values of method parameters. The parameter values have to be defined within a *parvalues* statement before they can be used in a method call. To grant the backward compatibility, a parameterless method call can be expressed both as $!interface.method()\hat{}$ and $!interface.method\hat{}$. To enhance the readability in some cases, a syntactic abbreviation regarding this event is defined:

$$!interface.method(a_1, \ldots, a_n)$$

corresponds to:

$$!interface.method(a_1, \ldots, a_n)\hat{}; ?interface.method\$$$

At the callee side, the expression denoting processing a method request $?interface.method\{\ldots\}$ is again extended:

$$?interface.method(p_1, \ldots, p_n)\{method\_body\}$$

Here, the formal parameters $p_1, \ldots, p_n$ are set to values corresponding to the ones the caller has chosen. In the *method_body*, *switch* statements testing the values of the parameters are to be used. The syntax of the expression testing a parameter p_i takes the following form:

$$\begin{array}{l} switch(p_i)\ \{ \\ \quad value_1 : \{\ protocol_1\ \} \\ \quad value_2 : \{\ protocol_2\ \} \\ \quad \vdots \\ \quad value_n : \{\ protocol_n\ \} \\ \quad default : \{\ protocol_d\ \} \\ \} \end{array}$$

The *default* branch is optional — it can be omitted. The *switch* statements may be nested; that means that the $protocol_i$ expressions may also contain *switch* statements. The parameters (and their values) are visible only while processing a particular method call, i.e., in the *method_body* in the expression above.

To define the semantics of protocols with parameters informally described above, we describe a way to construct a behavior protocol without parameters modeling the same behavior as a given behavior protocol with parameters.

At the caller (client) side, the emit-request event $!interface.method(a_1, \ldots, a_n)\hat{}$ is transformed into $!interface.method\_a_1\_\ldots\_a_n\hat{}$.

At the callee (server) side, the situation is more interesting. In the case of accepting a request $?interface.method(p_1, \ldots, p_n)\{method\_body\}$, in the *method_body*, for each combination of values of parameters $p_1, \ldots, p_n$, there is a protocol (possibly NULL) defined that is valid for a particular parameter values combination. Thus, the accepting of a request $?interface.method(p_1, \ldots, p_n)$ $\{method\_body\}$ is transformed into alternative of branches starting with accept tokens for all combinations of possible parameter values:

$$\begin{array}{l} ?interface.method\_a_1\_a_1\_\ldots\_a_1\{\ protocol\_1\_1\_\ldots\_1\ \}\ + \\ ?interface.method\_a_1\_a_1\_\ldots\_a_2\{\ protocol\_1\_1\_\ldots\_2\ \}\ + \\ \vdots \\ ?interface.method\_a_1\_a_1\_\ldots\_a_n\{\ protocol\_1\_1\_\ldots\_n\ \}\ + \\ ?interface.method\_a_2\_a_1\_\ldots\_a_1\{\ protocol\_2\_1\_\ldots\_1\ \}\ + \\ \vdots \\ ?interface.method\_a_n\_a_n\_\ldots\_a_n\{\ protocol\_n\_\ldots\_n\ \}\ + \end{array}$$

The protocols *protocol_1_1_..._1*, ..., *protocol_n_..._n* are except for the *switch* statements copies of the original protocol *method_body* in the parameterized version. The *switch* statements are (according to the value of a tested formal parameter) replaced with the branch of the *switch* statement corresponding to the actual branch of the alternative operator. This transformation is recursively applied to the branches. The resulting behavior protocol does not contain parameterized method requests and models the same behavior as the original one.

4.3 State variables

A state variable is a component local variable used for storing information across multiple method calls. As to the data domain, the same rules as for parameters are applied, i.e., the state variables can hold only the values previously defined by a *parvalues* statement. Each state variable is defined after the *parvalues* statements and before the behavior protocol. The syntax is as follows:

var local_var <– initial_value

By this statement, a state variable *var_name* being initialized to *initial_value* is declared. The values of state variables can be changed in method bodies via assignment either to a constant or to a value of a parameter passed to this method. The assignment statement is executed in a single non-interruptible step. The assignment of a *value* to the variable *local_var* takes the form:

local_var <– value

For testing a state variable value the same mechanism as in the case of parameters, i.e., the *switch* statement, is used.

To define the semantics, we describe construction of a finite automaton corresponding to the behavior and modeling the desired behavior. However, the situation in this case is more complex as the values of state variables may be changed "in parallel" by multiple parallel branches of a behavior protocol; this results in a situation when behavior of a parallel branch affects the behavior of another one. The algorithm for construction of a finite automaton A_{main} corresponding to a behavior protocol containing state variables follows:

1. Eliminate the method parameters as described in previous section.
2. Create a finite automaton A corresponding to the behavior protocol:
   - Each *switch* statement model as an alternative of all the branches; mark the initial state of this alternative with "SWITCH".
   - Do not include a transition for the assignments of state variables; mark the states, where a transition for assignment should start with "ASSIGN".
   - All the other operators model as original protocol operators.
3. Let n be the number of all possible combination of state variables' values. Create n copies of the A automaton and denote them by A_i for $i \in 1 \ldots n$; each automaton represents a combination of state variables' values.
4. For each state marked with "ASSIGN", add a transition to the "same" state of an automaton A_i representing the same combination of variables' values except for the variable being assigned. Label this transition with the assignment statement.
5. For each state marked with "SWITCH", keep only the transition leading from this state that corresponds to the state variable value represented by this automaton; delete only the first transition of each branch but one, but keep the following transitions and states.
6. Delete unreachable states and transitions in all automatons.

7. The resulting automaton A_{main} is the automaton created from A_i, $i \in 1 \ldots n$; the only initial state of A_{main} is the initial state of the A_i that A_i represents the combination of initial state variables' values. The set of accepting states is the union of the sets of accepting states of particular A_i.

4.4 Multisynchronization

The synchronization of multiple behavior protocols is based on usage of special protocol events being neither requests nor responses. Let us denote these events *joining events*. A *joining event* is a blocking event executed simultaneously and atomically by all participating components as well as the emit-accept pair of ordinary events by two communicating components. However, the joining events are not "internalized" (i.e., converted to τ-events), but they keep their form and meaning even after composition. Thus, they can be used in further compositions to synchronize additional components on the same joining events. On the other hand, a protocol may contain a joining event not used in a particular composition; then, such a joining event does not block and "is executed" in the same way as an ordinary event on an interface that is not (yet) bound.

A joining event may occur in a behavior protocol as an ordinary event. The token denoting a joining event starts with '@' followed by the name of the joining event; for instance:

@*synchro*

The information what joining events are shared by particular components is expressed as a parameter of the composition operator; thus, the syntax and semantics of the *consent* composition operator are extended accordingly.

To formally describe the semantics of the joining events, we use the same approach as in previous case — we describe the semantics using the theory of finite automata. For the sake of simplicity, at this point, we only describe the part regarding the joining events.

Let us denote the language generated by a behavior protocol P by $L(P)$ and the set of traces accepted by a finite automaton A by $L(A)$. Because for each behavior protocol P there exists a finite automaton A_P such that $L(P) = L(A_P)$ (and vice versa), we define the result of the consent composition of two behavior protocols in terms of a finite automaton. Let there be two behavior protocols BP_A and BP_B and the automata $A = (Act, Q_A, q_A, F_A, N_A)$ and $B = (Act, Q_B, q_B, F_B, N_B)$ such that $L(BP_A) = L(A)$ and $L(BP_B) = L(B)$. Let Act be the set of all event tokens forming the alphabet, Q_A be the set of states, $q_A \in Q_A$ be the initial state, $F_A \subseteq Q_A$ be the set of accepting states and $N_A \subseteq Q_A \times Act \times Q_A$ be the transition relation (and similarly for B). Let $J = \{@j_1, \ldots, @j_n\}$ be the set of joining events, S the set of synchronization events and U the set of unbound events; let J, S, and U are pairwise disjoint. Then, the automaton C accepting the language $L(C) = L(BP_A \nabla_{S,U,J} BP_B)$ contains a transition labeled by $@j_i$ from states corresponding to the following rule:

$$((q_1, q_2), @j_i, (q_1', q_2')) \in N_C \textit{ iff } (q_1, @j_i, q_1') \in N_A \textit{ and } (q_2, @j_i, q_2') \in N_B \textit{ and } @j_i \in J$$

4.5 Until loops

The syntax of a protocol describing a *stoppable service component* follows:

do *service_part* **until** *?stop_method*

Service_part is an arbitrary behavior protocol, while *stop_method* is a method of a service component provided interface used to make the component stop providing the service. The *!stop_method*\$ response may be emitted after accomplishing (i.e., reaching of a final state of) the current execution specified by *service_part*.

To formally define the semantics, we again present a procedure for construction of a behavior protocol having the original syntax and modeling the same behavior.

First recall that each behavior protocol corresponds to a finite automaton accepting the same language as the behavior protocol generates. Let the A_{SP} be a finite automaton so that $L(A_{SP}) = L(service\_part)$. Let us denote the states of A_{SP} by $s_0 \dots s_n$. Let A'_{SP} be a copy of the automaton A_{SP} and let us denote its states by $s'_0 \dots s'_n$. We construct an automaton A_{SSC} created from A_{SP} and A'_{SP} in the following way:

1. Add all states $s_1 \dots s_n$ and $s'_1 \dots s'_n$ and all transitions to A_{SSC}.
2. Add a new state s_{final} to A_{SSC}.
3. Redirect all loop transitions leading to the initial state of A'_{SP} from states s'_i of A'_{SP} corresponding to the top-most repetition operator to the state s_{final}. Label this transition with the !*stop_method*$ token.
4. Add a new state s_{tmp} to A_{SSC}; add a transition from the initial state of A_{SP} to s_{tmp} and label it with the ?*stop_method*^ token; add a transition from s_{tmp} to s_{final} and label it with the !*stop_method*$ token.
5. Except for the initial state of A_{SP}, add transitions from s_i to s'_i for all $i \in \{1 \dots n\}$ and label them with the ?*stop_method*^ token.
6. Let the initial state of A_{SSC} be only the initial state of A_{SP} and the accepting state of A_{SCC} only the state s_{final}.
7. Delete unreachable states and corresponding transitions.

For the finite automaton A_{SSC}, let us create a regular expression (being also a behavior protocol) SSC such that $L(A_{SSC}) = L(SSC)$ [7]. The behavior protocol SCC has the original syntax and specifies the same behavior as the original protocol stated above.

5 Examples

In this part, we present several examples illustrating the aforementioned procedures and algorithms.

5.1 Parametrized method calls

First, consider the behavior protocol on Fig. 10 containing parameterized method calls. As in the case of parameterized method calls a potential interleaving of a protocol part with another parallel branch does not interfere, for the sake of simplicity, this example contains no parallel operator. A protocol yielding the same set of traces (up to the method name modifications, of course) and containing no parametrized method calls is stated on Fig. 11. The $CreateToken\_OTHER1, \dots,$ $CreateToken\_OTHERn$ method accepts represents all the other possible values of the *airlines* parameter.

5.2 Multisynchronization

To demonstrate the necessity of a multisynchronization mechanism in behavior protocols, consider the following fragments of original behavior protocols corresponding to *Arbitrator* (Fig. 12), *Token* (Fig. 13), and *DhcpServer* (Fig. 14) components. Here, the synchronization is performed via *atomic actions*.

Because the components are bound and communicate each with both others, there in no sequential order in which they can be initialized in a correct way (i.e., not rising up a bad activity in later phases of the protocols). The *atomic actions* (events between '[' and ']') provide the power we need in this case, however, they violate the associativity of the consent operator, which

we consider as an important property of composition operators. On Fig. 15 - 17, there are the corresponding fragments of behavior protocols stated using the proposed synchronization method — *joining events*.

6 Evaluation and conclusion

In this paper, we presented several extensions to behavior protocols enhancing the usability of this platform for specification of behavior of software components. Two data entities were introduced addressing problems with method parameters and stateful components. Further, a synchronization mechanism allowing for synchronization of more than two components via new kind of events called *joining events* extending the expressive power of behavior protocols was introduced. Last, but not least, the *do..until* statement being only a syntactic abbreviation greatly simplifies the readability of specification of *service components*.

Using the extended version of behavior protocols for the Airport Internet Access application [5] resulted in much shorter, more readable yet more precise specification.

The future work will focus on implementing a tool for checking behavior compatibility of components specified by this newly proposed version of behavior protocols and evaluating the contribution on more case studies.

References

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

[2] SOFA — `http://sofa.objectweb.org`

[3] E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, J-B. Stefani: An Open Component Model and Its Support in Java. 7th SIGSOFT International Symposimum on Component-Based Software Engineering (CBSE7), LNCS 3054, Edinburgh, Scotland, May 2004.

[4] GCC, the GNU Compiler Collection, `http://gcc.gnu.org/`.

[5] Adamek, J., Bures, T., Jezek, P., Kofron, J., Mencl, V., Parizek, P., Plasil, F.: Component Reliability Extensions for Fractal Component Model. `http://kraken.cs.cas.cz/ft/public/public_index.phtml`

[6] Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance and Evolution: Research and Practice 17(5), Sep 2005

[7] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction To Automata Theory, Languages, and Computation, Addison-Wesley, 2001, Second edition

[8] Holzmann, G. J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, September 2003.

[9] Demartini, C., Iosif, R., Sisto, R: dSPIN: A Dynamic Extension of SPIN, Lecture Notes In Computer Science; Vol. 1680, 1999, ISBN:3-540-66499-8

7 Appendices

```
(
  ?IDhcpServerLifetimeController.Start^;
  !IListenerLifetimeController.Start^;
  [?IListenerLifetimeController.Start$,!IDhcpServerLifetimeController.Start$]
)
;
(
  (
  (
    (
    ?IDhcpListenerCallback.RequestNewIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;
      (
      (
        !IIpMacTransientDb.Add_1;
        !ITimer.SetTimeout
      ) + NULL
      )
    }
    +
    ?IDhcpListenerCallback.RenewIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;(
      (
        !IIpMacTransientDb.SetExpirationTime_1;
        !ITimer.SetTimeout
      ) + NULL
      )
    }
    +
    ?IDhcpListenerCallback.ReleaseIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;(
      (
        !IIpMacTransientDb.Remove_1;
        !IDhcpCallback.IpAddressInvalidated_1;
        (!ITimer.CancelTimeouts_1 + NULL)
      ) + NULL
      )
    }
    ) *
    |
    (
    ?ITimerCallback.Timeout{
      (
      !IIpMacTransientDb.GetExpirationTime_2;(
        (
        !IIpMacTransientDb.Remove_2;
        !IDhcpCallback.IpAddressInvalidated_2
        ) + NULL
      )
      ) *
```

```
    }
    ) *
)

)
 +
(
(
   (
   (
      ?IDhcpListenerCallback.RequestNewIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;
      (
         (
         !IIpMacTransientDb.Add_1;
         !ITimer.SetTimeout
         ) + NULL
      )
      }
       +
      ?IDhcpListenerCallback.RenewIpAddress{
      !IIpMacTransientDb.GetIpAddress_1; (
         (
         !IIpMacTransientDb.SetExpirationTime_1;
         !ITimer.SetTimeout
         ) + NULL
      )
      }
       +
      ?IDhcpListenerCallback.ReleaseIpAddress{
      !IIpMacTransientDb.GetIpAddress_1; (
         (
         !IIpMacTransientDb.Remove_1;
         !IDhcpCallback.IpAddressInvalidated_1;
         (!ITimer.CancelTimeouts_1 + NULL)
         ) + NULL
      )
      }
   ) *
   |
   (
      ?ITimerCallback.Timeout{
      (
         !IIpMacTransientDb.GetExpirationTime_2; (
         (
            !IIpMacTransientDb.Remove_2;
            !IDhcpCallback.IpAddressInvalidated_2
         ) + NULL
         )
      ) *
      }
   ) *
```

```
    )
    |
    ?IManagement.UsePermanentIpDatabase^
  );!IManagement.UsePermanentIpDatabase$;(
    (
    (
      ?IDhcpListenerCallback.RequestNewIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;(
        (
        !IIpMacPermanentDb.GetIpAddress;(
          (
          !IIpMacTransientDb.Add_1;
          !ITimer.SetTimeout
          ) + NULL
        )
        ) + (
        !IIpMacTransientDb.Add_1;
        !ITimer.SetTimeout
        )
      )
      }
      +
      ?IDhcpListenerCallback.RenewIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;(
        (
        (
          !IIpMacPermanentDb.GetIpAddress;(
          (
            !IIpMacTransientDb.SetExpirationTime_1;
            !ITimer.SetTimeout
          ) + NULL
          )
        ) + (
          !IIpMacTransientDb.SetExpirationTime_1;
          !ITimer.SetTimeout
        )
        ) + NULL
      )
      }
      +
      ?IDhcpListenerCallback.ReleaseIpAddress{
      !IIpMacTransientDb.GetIpAddress_1;(
        (
        !IIpMacTransientDb.Remove_1;
        !IDhcpCallback.IpAddressInvalidated_1;
        (!ITimer.CancelTimeouts_1 + NULL)
        ) + NULL
      )
      }
    ) *
    |
    (
```

```
        ?ITimerCallback.Timeout{
        (
          !IIpMacTransientDb.GetExpirationTime_2;(
          (
            !IIpMacTransientDb.Remove_2;
            !IDhcpCallback.IpAddressInvalidated_2
          ) + NULL
          )
        ) *
        }
      ) *
      )
      |
      ?IManagement.StopUsingPermanentIpDatabase^
    );!IManagement.StopUsingPermanentIpDatabase$
    )
) *
```

Figure 9: Behavior protocol of the IpAddressManager component

```
?IFlyTicketAuth.CreateToken(airlines) {
  switch (airlines) {
    AIRFRANCE : {
      !IAfFlyTicketDb.GetFlyTicketValidity;
      (!IAfFlyTicketDb.IsEconomyFlyTicket + NULL)
    }
    CZECHAIRLINES : {
      !ICsaFlyTicketDb.GetFlyTicketValidity;
      (!ICsaFlyTicketDb.IsEconomyFlyTicket + NULL)
    }
    default : {NULL}
  }
}
```

Figure 10: A behavior protocol containing parametrized method calls.

$$
\begin{array}{l}
?IFlyTicketAuth.CreateToken\_AIRFRANCE\ \{ \\
\quad !IAfFlyTicketDb.GetFlyTicketValidity; \\
\quad (!IAfFlyTicketDb.IsEconomyFlyTicket + NULL) \\
\} \\
+ \\
?IFlyTicketAuth.CreateToken\_CZECHAIRLINES\ \{ \\
\quad !ICsaFlyTicketDb.GetFlyTicketValidity; \\
\quad (!ICsaFlyTicketDb.IsEconomyFlyTicket + NULL) \\
\} \\
+ \\
?IFlyTicketAuth.CreateToken\_OTHER1\ \{NULL\} \\
+ \\
?IFlyTicketAuth.CreateToken\_OTHER2\ \{NULL\} \\
+ \\
\vdots \\
+ \\
?IFlyTicketAuth.CreateToken\_OTHERn\ \{NULL\}
\end{array}
$$

Figure 11: A behavior protocol containing no parametrized method calls.

$$
\begin{array}{l}
(\\
\quad ?ITokenCallback.Notify\hat{}; \\
\quad !IArbitratorCallback.Notify\hat{}; \\
\quad [?IArbitratorCallback.Notify\$, !ITokenCallback.Notify\$]; \\
\\
\quad ?IArbitratorLifetimeController.Start\hat{}; \\
\quad [!ITokenLifetimeController.Start\hat{}, !IDhcpServerLifetimeController.Start\hat{}]; \\
\quad [?ITokenLifetimeController.Start\$, ?IDhcpServerLifetimeController.Start\$, \\
\qquad !IArbitratorLifetimeController.Start\$] \\
) \\
; \\
rest\_of\_protocol
\end{array}
$$

Figure 12: A fragment of a behavior protocol of the Arbitrator component.

$$
\begin{array}{l}
((\\
\quad ?IToken.SetEvidence \\
\quad | \\
\quad ?IToken.SetValidity \\
\quad | \\
\quad (?IToken.SetAccountCredentials + NULL) \\
) + NULL) \\
; \\
!ITokenCallback.Notify \\
; \\
?ITokenLifetimeController.Start \\
; \\
rest\_of\_protocol
\end{array}
$$

Figure 13: A fragment of a behavior protocol of the Token component.

```
?IDhcpServerLifetimeController.Start
;
rest_of_protocol
```

Figure 14: A fragment of a behavior protocol of the DhcpServer component.

```
(
  ?ITokenCallback.Notify^;
  !IArbitratorCallback.Notify^;
  ?IArbitratorCallback.Notify$; !ITokenCallback.Notify$;
  @syncEnvironment;
  ?IArbitratorLifetimeController.Start^;
  !ITokenLifetimeController.Start^; !IDhcpServerLifetimeController.Start^;
  ?ITokenLifetimeController.Start$ | ?IDhcpServerLifetimeController.Start$;
  !IArbitratorLifetimeController.Start$
);
@syncAll
;
rest_of_protocol
```

Figure 15: A fragment of a synchronized version of the Arbitrator component.

```
((
  ?IToken.SetEvidence
  |
  ?IToken.SetValidity
  |
  (?IToken.SetAccountCredentials + NULL)
) + NULL)
;
!ITokenCallback.Notify
;
?ITokenLifetimeController.Start
;
@syncAll
;
rest_of_protocol
```

Figure 16: A fragment of a synchronized version of the Token component.

```
?IDhcpServerLifetimeController.Start
;
@syncAll
;
rest_of_protocol
```

Figure 17: A fragment of a synchronized version of the DhcpServer component.