Making Components Fit: SPINing[*]

Jan Kofroň, Tomáš Poch, Ondřej Šerý
Charles University in Prague,
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic
{jan.kofron, tomas.poch, ondrej.sery}@dsrg.mff.cuni.cz
http://dsrg.mff.cuni.cz

Abstract

The more popular it is to build an application from reusable software components, the more desperate is the need for showing correctness of such a composition. This requires on one hand, being able to formally specify behavior of software components, while, on the other hand, providing appropriate tool support for verification of correctness of the composition. In this paper, we suggest use of the formalism of Extended Behavior Protocols and present a tool chain for verification of composition correctness of component applications. The advantage of the proposed approach is using a well-tested and supported model checker Spin as a backend. As a proof of the concept, we share our experience with application of the method.

1 Introduction

Building software application of reusable software components (only components in the following) has become a frequently used development technique, as it makes the development both cheaper and faster. Both academia and industry make a substantial effort to formalize this approach. As a result, a number of component models exist; either flat (EJB, CCM, COM) or hierarchical (Fractal [7], SOFA 2 [8], Darwin [18]), which allow nesting of components[1].

According the component paradigm, a new application is built of *components of the shelf* (COTS). They are typically black boxes, provided by several vendors, and designed independently of the target application. Thus, ensuring compatibility of such components becomes one of the main responsibilities of the designer. Generally, a syntactic-level compatibility (e.g., matching interface signatures) is insufficient and a deeper insight on component behavior is necessary.

The problem of ensuring component behavior compatibility is approached by many researchers in different ways [2, 3]. In this paper, we assume behavior specification of components using *Extended Behavior Protocols* (EBP) [17], a formalism specifying the behavior as a set of allowable interleaving of method calls on component's interfaces. The goal of the paper is to propose a method of checking behavior compatibility of components specified in EBP. The method is based on translation of EBP specification into Promela, the input language of the Spin model checker [15]. Then, the Promela model is used to check the compatibility of components and, possibly, further analysis provided by Spin.

The paper continues by an overview of the EBP formalism (Sect. 2). Description of the actual transformation of EBP to Promela and discussion of the related issues follows (Sect. 3). We applied the proposed method in the CoCoME contest and share our experience (Sect. 4). In Sect. 5, the related work is summarized.

2 Extended Behavior Protocols (EBP)

Extended Behavior Protocols is a formalism for behavior specification of components. It can be used in hierarchical component models like SOFA 2 [8] and Fractal [7, 1]. In these component models, an important concept regarding (not only) behavior specification is *component frame*. Component frame is a black-box view of a component where only the exported, i.e., provided (server) and required (client) interfaces are considered. Components directly implemented in a programing language are denoted as *primitive components*, while components implemented by a composition of *subcomponents* are referred to as *composite components*. The related concept is *component architecture*, which describes the actual composition: subcomponents' frames and bindings among them.

[*] This work was partially supported by the Grant Agency of the Czech Republic projects 201/06/0770 and GD201/05/H014

[1] Since the flat component models may be viewed as a special case of the hierarchical ones, we only consider the latter type in this paper

To support formal reasoning about behavior compatibility of communicating components, each component frame is equipped with a behavior specification—EBP. EBP specifies traffic on the component boundary, capturing dependencies of method calls on the component's provided and required interfaces. Having the behavior described for each frame, we can check whether the connected components fit together or not.

In case of EBP, the concept of behavior compatibility is expressed as absence of certain types of communication errors—*bad activity*, *no activity*, and *unbound-requires* error. The *bad activity* error denotes a situation in which an event (either a method call or a return from a method call) is emitted by a component on a required interface, but the component bound to the interface is unable to accept the event in the current state. In other words, emitting of an event is a non-blocking operation, which may yield the *bad activity* error. The *no activity* communication error denotes a deadlock, i.e., a situation when no component can emit an event while there is a component waiting for one. The *unbound-requires error* is a special case of the bad activity error when a method is invoked on an unbound required interface. Unbound required interfaces can appear when a component is used in a context which does not employ all of its functionality [23].

This section continues by an example and brief overview of the EBP syntax and semantics. Formal definitions can be found in [17].

2.1 EBP Specification Example

To give an intuitive insight on EBP, we present an EBP specification of a *LoginManager* component (Fig. 1) and an architecture, where it might be employed (Fig. 2). The *LoginManager* component is used by the *SessionManager* component as an arbitrator controlling access to a system. The *Login* interface is used to invoke the *login* method of *LoginManager*; the result is passed through the *Authorize* interface. The *LoginManager* component uses the *Logger* component for logging information about granting and denying access to the system.

Within the *behavior* part of the *LoginManager* specification in Fig. 1, there is an expression specifying the component behavior. The expression references the types and state variables defined within the preceding *types* and *vars* sections.

The topmost operator of the EBP behavior expression is the *parallel operator* '|'. Arbitrary interleaving of traces specified by the operands is allowed. The left operand (lines 10-12) captures the fact that the component is able to accept the *setMode* method on the *Config* interface. Execution of a method means accepting the method request, processing the method body and, finally, emitting a response. The method request and response are modeled as two different events. Thus, the left operand of '|' is equivalent to:

```
 1: component LoginManager {
 2:   types {
 3:     USER = {ROOT, OTHER}
 4:     MODE = {OPERATIONAL, MAINTENANCE}
 5:   }
 6:   vars {
 7:     MODE mode = MAINTENANCE
 8:   }
 9:   behavior {
10:     ?Config.setMode(MODE m) {
11:             mode <- m;
12:     }*
13:     |
14:     ?Login.login(USER user) {
15:       (!Logger.log(UNAUTH); !Authorize.accessDenied) +
16:       switch (mode) {
17:         OPERATIONAL: {!Logger.log(OK); !Authorize.allow }
18:         MAINTENANCE: {
19:           switch(user) {
20:             ROOT: { !Logger.log(OK); !Authorize.allow }
21:             default: {
22:               !Logger.log(BLOCKED); !Authorize.deny
23:             }
24:           }
25:         }
26:       }
27:     }*
28:   }
29: }
```

Figure 1. EBP specification of the LoginManager component

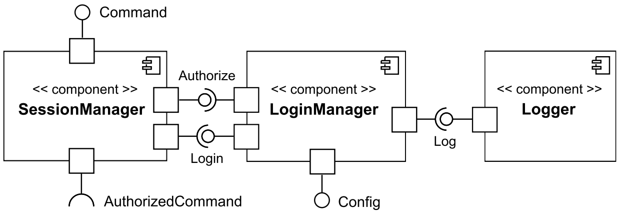

Figure 2. Environment of the LoginManager

?*Config.setMode(MODE m)*↑;
mode <– *m*;
!*Config.setMode*↓

which can be repeated any arbitrary (finite) number of times ('*'). This fragment means that first a method request ('↑') of the *setMode* method is accepted ('?') with a parameter *m* of the type `MODE`. The value of the parameter is consequently assigned to the *mode* variable and finally a method response ('↓') is emitted ('!'). Regarding the right operand (lines 16-29), first a *login* method call request is accepted with a parameter denoting whether a superuser (`ROOT`) or an ordinary user (`OTHER`) attempts to log in (line 16). *LoginManager* nondeterministically decides whether access should be granted or not. In the real implementation, the decision would be based on the login information and user database, which are abstracted away in this case.

If access is denied, *LoginManager* calls the *log* method of the *Logger* interface to log the unauthorized login attempt and notifies the user about the result via calling the *deny* method on the *Authorize* interface (line 17).

If the *LoginManager* is in the OPERATIONAL mode and the login information is correct, it logs the user in, records the access via the *Logger* interface and notifies the calling component about success via the *allow* method of the *Authorize* interface (line 19). In the MAINTENANCE mode, however, the system is set up to allow only the ROOT user to be logged in (line 22), while the OTHER users are BLOCKED (line 24). Again, all attempts to log in are logged using the *log* method of the *Logger* interface (lines 22 and 24).

2.2 EBP Syntax

An EBP specification starts with the *component* keyword followed by three main sections: (i) type definitions, (ii) local variable definition, and (iii) behavior definition. Within the *types* section, enumeration types of state variables and method parameters are defined. The definition of a type takes the form:

$$\textit{type\_name} = \{\ \textit{value}_1, \textit{value}_2, \ldots, \textit{value}_n\ \}$$

This defines a type *type_name*; variables of this type can hold only the enumerated values.

The *vars* section contains definition of state variables. These variables are local to the component and shared by all methods within the *behavior* section of the EBP. The variable definition takes the following form:

$$\textit{type\_name var\_name} = \textit{initial\_value}$$

This way, a state variable *var_name* of the type *type_name* is defined; the initial value of this variable is set to *initial_value*.

The basic building blocks of the expression in the *behavior* section are *events*. We distinguish six basic types of events with the following meaning:

1. $!\textit{if.method}(\textit{value}_1, \textit{value}_2, \ldots, \textit{value}_n)\uparrow$ – emitting a method call request with parameters $\textit{value}_1$, $\textit{value}_2$, $\ldots$, $\textit{value}_n$

2. $!\textit{if.method}\downarrow$ – emitting a method call response

3. $?\textit{if.method}(\textit{type}_1\ \textit{par}_1, \textit{type}_2\ \textit{par}_2, \ldots, \textit{type}_n\ \textit{par}_n, \textit{value}_{n+1}, \ldots, \textit{value}_m)\uparrow$ – accepting a method call request if the caller used actual values $\textit{value}_{n+1}$, $\textit{value}_{n+2}, \ldots, \textit{value}_m$ as parameters $n+1$ to m. The actual values of remaining parameters are restricted by the used (enumerated) type. These two kinds of parameters can be arbitrarily mixed.

4. $?\textit{if.method}\downarrow$ – accepting a method call response

5. $\textit{var} <- \textit{value}$ – assigning the value *value* to the state variable *var*

A single event represents the simplest form of EBP. The following operators are available to derive more complex protocols: '$+$' (alternative), ';' (sequence), '$|$' (parallel composition), '$*$' (repetition), NULL (nulary operator), *while*, and *switch*. The NULL operator specifies a set containing only the empty trace. The *while* operator takes the following form:

while (*var* == *value*) { *protocol* }

This expression denotes repetition of the *protocol* as long as the value of the *var* state variable is equal to *value*. The value of the variable *var* is tested each time before the *protocol* takes place.

The *switch* operator takes the following syntax:

switch (*var*) {
  $\textit{value}_1$: { $\textit{protocol}_1$ }
  $\textit{value}_2$: { $\textit{protocol}_2$ }
  ...
  default: { $\textit{protocol}_d$ }
}

According to the value of *var* (either a state variable or a method parameter), the $\textit{protocol}_i$ takes place. The *default* branch is optional. It takes place if the value does not match any other case. If the *default* branch is not present, a default branch with the *NULL* protocol is automatically generated. Further, the following syntactic abbreviation, referred to as *method call processing*, is defined. The abbreviations are sufficient in most cases. Therefore, since the abbreviations are also easier to read, their use is preferred.

$?\textit{if.method}(\textit{type}_1\ \textit{par}_1, \ldots, \textit{type}_n\ \textit{par}_n)\ \{\textit{protocol}\}$

stands for:

$?\textit{if.method}(\textit{type}_1\ \textit{par}_1, \ldots, \textit{type}_n\ \textit{par}_n)\uparrow;$
$\quad\textit{protocol};\ !\textit{if.method}\downarrow$

2.3 Semantics of EBP

Semantics of the EBP formalism can be split into two, relatively independent, parts. First, we describe the semantics of a single EBP specification. Second, we discuss composition of EBPs and detection of the communication errors.

Single EBP specifies the behavior as a set of finite traces consisting of events of the six types presented in the previous section. The set of traces is given by means of a Deterministic Finite Automaton with Assignment (DFAA) introduced in [17]. DFAA is an ordinary Deterministic Finite Automaton augmented with variables and guards. The guards, being conditions over the variables, are associated with transitions. A transition is enabled (may be taken) if the associated guard holds[2].

[2] To ensure deterministic nature of DFAA, if there are two (or more) transitions coming out from a state and labeled by the same event, their guards are required to be mutually exclusive.

When capturing an EBP specification, DFAA states and (guarded) transitions capture the control flow, while DFAA variables directly correspond to the EBP state variables. The transformation of EBP into DFAA is quite straightforward. The variables in DFAA correspond one-to-one to the variables in the particular EBP. The transformation of the behavior section follows the idea of the well-known transformation of a regular expression into a deterministic finite automaton. A single event behavior is represented by an automaton containing two states connected by an transition labeled by the event and a true guard. An automaton for a more complex specification is constructed using the bottom-up approach following the parse tree of the specification[3].

A model of the entire system is obtained by composition of DFAAs of participating components. Since DFAAs have the same expressive power as finite state automata, the result of composition can be again viewed as an automaton—*composite automaton*. A state of the composite automaton, *composite state*, represents a combination of states of particular DFAAs and their variable valuation. The initial composite state captures the situation where all DFAAs are in their initial states and all variables have their initial values. A transition represents either an internal change of single DFAA (assignment to a variable) or communication between two DFAAs (acceptance of complementary events—'?e' on one side and '!e' on the other). In other words, the composite state space represents synchronous product of DFAAs; the coresponding events are paired, while the others are arbitrarily interleaved.

If there is a reachable composite state in which a DFAA accepts a '!e' event (the component may emit a method call or a return from a method call) and there is no DFAA accepting the complementary '?e' event (no component can accept the method call or return), *bad activity* arises. *No activity* is reported when a composite state is reached such that at least one DFAA is not in its final state but no DFAA can accept a '!e' event. As clear from above, the information needed to identify an error is hidden in the list of output transitions of a single composite state. Thus, the behavior compatibility we are checking is a reachability property of the composite state space.

3 Deriving Promela From EBP

Having components and their EBP specification, an important task is to verify correctness of components' composition. For this purpose, we use the Spin model checker. Basically, an EBP specification is translated into Promela and the actual state space traversal is left up to Spin. The whole process is depicted in Fig. 3.

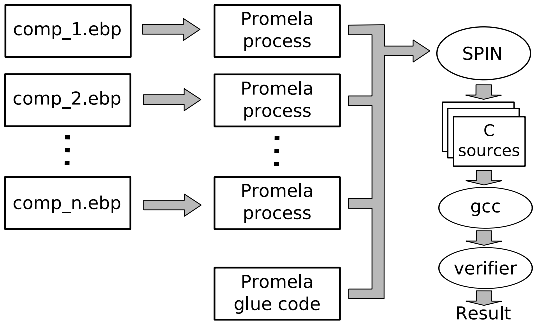

Figure 3. Overview of the transformation process

First, DFAAs are created from the EBP specification of individual components. Second, Promela processes representing the components are created from DFAAs. Third, this Promela representation of components along with a bit of necessary glue code is fed into Spin to get a specialized instance of a state-space traverser in C, which is then compiled and executed to perform the actual verification.

3.1 Basic mapping

In this section, we describe the basics of the EBP to Promela mapping. In some cases, there are more than just a single mapping option. For the sake of simplicity, we present only the final mapping here. The choices made are justified in the following section.

3.1.1 Representation of single component specification

An EBP component specification is represented as a Promela process. The process is a straightforward implementation of DFAA in Promela. The current state of DFAA is stored in the local variable `state`. A Promela variable exists for every EBP variable. The transition relation is reflected by two tables—`transitions` and `guards`. Indexed by a state and an event, the `transitions` table yields the index of the target state of the transition or `NONE`, if there is no transition from the given state labeled by the given event. The `guards` table contains guard for each transition. It is also indexed by a state and an event. The actual guard expression is represented as a C function to which the elements of `guards` point. Based on the current values of the local variables, the C function of the guard evaluates to either `true`, if the guard is satisfied, or `false` otherwise. There is also a table `final` indexed by a state, which identifies the final states of DFAA.

[3] The resulting automaton may contain non-deterministic choices, i.e., a state featuring multiple transitions labeled by the same event but non-exclusive guards. This case is handled by, modifying the involved transitions to contain only mutually exclusive and equivalent guards, and applying the standard determinization algorithm.

```
 1:proctype component_1()
 2:{
 3:  int state = 4; // initial state
 4:
 5:  do
 6:  ::emit(state, transitions[state][0],
 7:      guards[state][0], RESP_OPEN)
 8:  ::accept(state, transitions[state][1],
 9:      guards[state][1], REQ_OPEN)
10:  ::emit(state, transitions[state][2],
11:      guards[state][2], RESP_READ)
12:  ::accept(state, transitions[state][3],
13:      guards[state][3], REQ_READ)
14:  ::emit(state, transitions[state][4],
15:      guards[state][4], RESP_WRITE)
16:  ::accept(state, transitions[state][5],
17:      guards[state][5], REQ_WRITE)
18:  ::emit(state, transitions[state][6],
19:      guards[state][6], RESP_CLOSE)
20:  ::accept(state, transitions[state][7],
21:      guards[state][7], REQ_CLOSE)
22:  ::final(final[state])
23:  od;
24:
25:DONE:
26:  skip;
27:}
```

Figure 4. Promela representation of EBP

```
 1:#include "component_0.pr"
 2:#include "component_1.pr"
 3:init {
 4:  atomic { // Initialization
 5:    process_count = 2;
 6:    run component_1();
 7:    run component_2();
 8:  }
 9:  timeout; // Verification stopped
10:  d_step {
11:    if
12:    :: event != NONE ->
13:         printf("BAD ACTIVITY");
14:         assert(0);
15:    :: else -> end_of_run = 1;
16:    fi;
17:  }
18:  timeout;
19:  d_step {
20:    if
21:    :: process_count > 0 ->
22:         printf("NO ACTIVITY");
23:         assert(0);
24:    :: else -> skip
25:    fi;
26:  }
27:}
```

Figure 5. Init of the EBP verification task

The process consists of a single do cycle. In every iteration of the cycle and based on the current DFAA state, the process can either emit, or accept an event, make an assignment to a state variable, or finish the execution (recall that an EBP generates finite traces).

For example, an EBP protocol specifying behavior of a simple file component:

?open; (?read + ?write)*; ?close

gets translated into the Promela process listed in Fig. 4. Recall that each method call is an abbreviation for sequence of two events, e.g., ?open↑; !open↓. Therefore, four methods yield eight atomic events. The nine cases in the body of the do cycle correspond to these eight atomic events (lines 6-21) and a single check for a final state (line 22). The constants RESP_OPEN, REQ_OPEN, etc. are shared among all processes and identify the events. The transitions and guards tables contain columns only for events used by the actual protocol (minimizing the number of cases of the do cycle).

In the example, technical details are hidden in a set of macros emit, accept, assign, and final. The definitions can be seen in Fig. 6. Note that they are simplified for the sake of brevity.

3.1.2 Communication

The macros use a single shared variable event to achieve synchronous communication. A special value NONE indicates that any component willing to emit an event is free to do so. Emitting an event consists of checking that event == NONE and setting event to (the identifier of) an event to be emitted. Any component ready to accept the emitted token can do so by checking that event contains (the identifier of) the correct event and resetting it back to NONE. To prevent race conditions and to reduce size of the generated state space, the two actions together with the necessary state transition are enclosed into a d_step block. An assignment macro checks that no communication is under way (i.e., that event == NONE) and then performs the actual assignment.

3.1.3 Composition error detection

Put all together, the EBP components represented as processes can be simulated in Spin. However, the main motivation was not only to simulate the components but also to check their behavior compatibility, i.e., to detect potential communication errors. This is done in the entry process init, which can be seen in Fig. 5. First, it instantiates processes for all EBP components and stores the count in process_count variable. Then it waits (using Promela timeout statement—line 14) until all communication ceases. This can be caused by one of the following three reasons: (i) all components successfully executed their protocols and are now in their final states, (ii) bad activity, i.e., a component emitted an event, but no component was ready to accept it, or (iii) no activity, i.e., no component can communicate any further, but some components are not in their final states. The case (ii) is easy to identify via a check of the event shared variable (line 22). If it differs from NONE, the last action was emit of an event which was not accepted (recall that the timeout statement

```
#define emit(state,tr_dst,guard,eId)                        \
  d_step {                                                  \
    event == NONE && tr_dst != NONE && SATISFIED(guard); \
    event = eId;                                            \
    state = tr_dst;                                         \
  }
#define accept(state,tr_dst,guard,eId)                      \
  d_step {                                                  \
    event == eId && tr_dst != NONE && SATISFIED(guard);  \
    event = NONE;                                           \
    state = tr_dst;                                         \
  }
#define assign(state,tr_dst,guard,var,val)                  \
  d_step {                                                  \
    event == NONE && tr_dst != NONE && SATISFIED(guard); \
    var = val;                                              \
    state = tr_dst;                                         \
  }
#define final(is_final_state)                               \
  atomic {                                                  \
    is_final_state == 1 && end_of_run == 1;                 \
    process_count--;                                        \
    goto DONE;                                              \
  }
```

Figure 6. Definition of helper macros

got executable). If `event == NONE`, then both (i) and (iii) could possibly occur. In order to distinguish between these two cases, the `end_of_run` shared variable is set to signalize the end of execution. Then using the `final` macro, any component that is in its final state decreases the `process_count` variable. After another `timeout` (line 30) either `process_count == 0`, being the (i) case (everything is alright), or it is equal to the number of deadlocked components, being the (iii) case (no activity), as those components have not decreased `process_count`.

3.2 Issues of transformation

While designing the EBP to Promela mapping, we had to overcome a number of technical details and make choices between alternative approaches. The most important ones of the details and choices made are explained and justified in this section.

3.2.1 Representation of DFAA

The first question was whether to (i) encode the DFAA representation of an EBP using an explicit state variable (`state`), and a transition table (`transitions`), etc. (as described below), or (ii) use a program counter of a Promela process as the representation of the current DFAA state. In the (ii) case, labels would correspond to states of the DFAA while transitions would be encoded as `goto`s among the labels. The (i) option was eventually chosen, as it allows hiding the transition table from Spin state space (see the following paragraph). Taking the (ii) option resulted in a huge Promela source code. In some cases, the result was even behind limits of the Spin model checker, which failed to process it.

3.2.2 State space size

As usual, a problem was also the size of the state space of the Promela model. The first (naïve) implementation of (i) generated large state spaces with huge state identifiers. In order to fight this problem, the static (large) tables `transitions`, `guards`, `final` were encoded using the `c_code` block and thus effectively hidden from Spin, so that they did not inflate the state identifiers. A single state, as seen by Spin, then contains only the global variables `event`, `process_count`, `end_of_run`, EBP state variables of all components, one `state` variable for each component, and a program counter for each process (`init`, `component_1`, `component_2`, ..., `component_n`).[4]

Moreover, use of the constructs `atomic` and `d_step` in the set of macros helped us to squeeze the ratio between number of Promela model transitions and the number of transitions in DFAA representation close to 2:1, which is the optimal case for one-to-one communication (as each time two processes are involved). We consider this to be a very satisfiable result.

3.2.3 Communication

There was also an option to use Promela channels for communication among components instead of a single synchronization variable `event`. However, besides expanding the state identification vector, the asynchronous semantics of Promela channels would not allow us to identify bad activity errors[5]. In the case of an independent subset of communicating components, the Promela `timeout` statement would not help to identify the error situation.

3.2.4 Partial order reduction

One drawback of the proposed solution is that it is not well-suited for the Spin partial order reduction algorithm (POR). Mainly due to use of the global communication variable `event`, application of POR does not result in any substantial reduction of the model size. Another reason is the fact that the components perform almost exclusively globally visible communication and little internal computation. Such a scenario is inherently problematic for partial order reduction.

3.2.5 Error trace back-projection

Since the aim of the verification is to find communication errors of components specified in EBP, the error trace

[4] Note that during the state space traversal, the program counters stay constant as the components are in their `do` cycle (`d_step` hides the internal steps) and the main process waits in its first `timeout` statement.

[5] Note that a Promela channel either blocks or looses the message if the recipient is not ready (in case of the full message buffer), while in EBP, this situation leads to a bad activity error.

Table 1. Number of states and transitions in the CoCoME model

Protocol	*Spin States*	*Spin Transitions*	*EBP Transitions*	*Spin/EBP Trans*
EnterpriseServer	3014	5557	2348	2.37
CashDeskLine	15518	37421	12038	3.11
StoreApplication	382702	643624	321495	2.00
TradingSystem	586481	1599684	736580	2.17
Data	1891216	4204144	1794818	2.34
StoreServer	2998646	6366667	2828501	2.25

found by Spin should be back-projected into the EBP language. This feature is implemented transparently in the `emit`, `accept`, and `assign` macros, using `printf` to log each event. This way, the error trace found by Spin can be simulated (via the simulation mode of Spin) in order to reconstruct the EBP error trace.

4 Experience: CoCoME

We applied the proposed approach for checking EBP using Spin in the CoCoME (Common Component Modeling Example) contest [26] with satisfactory results. The main goal of CoCoME was to provide a common platform for comparison of different approaches to modeling component applications. So far, comparing different techniques was very difficult as their authors typically applied them to distinct case studies (and often only toy ones).

In the CoCoME contest, all participating groups shared a common assignment and applied their modeling techniques to it. The core of the assignment was an information system for sale and stock management of multiple stores. The assignment was specified by simple UML model clarified by an example implementation. The goal of each participating group was to create a model of the application to show strengths and weaknesses of their formalism.

In our case, we applied the SOFA 2 component model and EBP for behavior specification of components. The result model of the CoCoME assignment contains 18 primitive and 9 composite components in our case. The prototype implementation in Java (provided as a part of the assignment) takes 9,296 LOC.

Using the proposed transformation of EBP into Promela and the Spin model checker, we succeed to verify correctness of communication of the components. Hierarchical structure of the CoCoME application was exploited to check the whole application part by part. Basically, it suffices to check each composite component separately (i.e., to detect composition errors yielded by composition of all the subcomponents' frames together with the component's own frame).

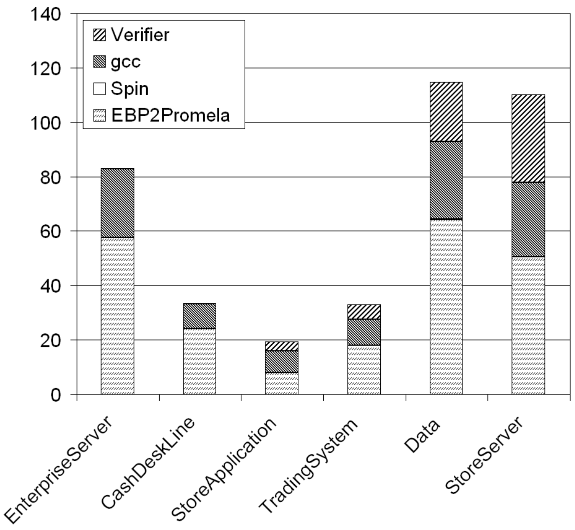

Figure 7. Time (in seconds) spent in different stages of the tool chain

Table 1 presents results of verification for each composite component in CoCoME. The first column denotes names of composite components. The other columns contain the number of states visited by Spin, number of the Spin transitions, the number of transitions of the automaton simulated by the Promela model (EBP Transitions), and the ratio between Spin transitions and EBP transitions. The ratio is reasonably small, which suggest good scaling of the method.

Fig. 7 visualizes proportion of the time spent in different stages of the tool chain[6,7]. A significant amount of time is spent by the transformation itself. This is most obvious in the case of the EnterpriseServer component, which deserves a comment. Its transformation takes nontrivial time, while the verification itself is done almost instantly. The transformation is done for each component frame separately—no information about the other components is used. Thus, if a relatively complex component (e.g., with high degree of parallelism) is used in an environment which utilizes only a small portion of the component's functionality, the huge

[6] Tests were executed on Intel(R) Xeon(R) CPU E5345 @ 2.33GHz processor.

[7] Time spent in Spin is negligible in comparison to the other stages.

DFAA of the complex component is translated into Promela model, which is consequently compiled by Spin and gcc. Finally, the verifier detects that most states of the complex component are unreachable in this case and does not explore them. In the case of EnterpriseServer, this situation occurs and the verifier terminates almost instantly. The same component is used also in the Data and StoreServer composite components. But in these cases, the environment utilizes more of its functionality and the resulting state space traversed by Spin is larger.

To sum it up, our participation in the CoCoME contest allowed us not only to apply our approach to a real-life-sized component application but also to present this all on a common platform used also by the other researchers. The results suggest that the approach is feasible and scales well. The effort spent on the development was significantly lowered by using the supported and well-tested third party tool—Spin model checker. Moreover, the entire EBP specification[8] along with the transformation tool chain[9] is available for download.

5 Evaluation and Related Work

While we were choosing model checker to use, we kept several things in mind: (i) the chosen model checker should be stable, memory efficient, yet fast enough for practical cases, (ii) its input language should be easy to use for representation of finite automata, and (iii) it should be either usable as a library or provide a command line interface to be used from our frontend.

Besides satisfying the requirements, Spin is suitable for software model checking (conversely to, e.g., SMV [19], which was designed for hardware model checking). Moreover, a Promela model can be easily interfaced with C routines and data thus allowing useful optimizations.

Spin is a very mature tool and there is quite a large body of work using it as a model checking back-end. The work aiming at verification via translation into the Promela language ranges from hardware specification languages such as SystemC [25], through various communication protocols specification languages as LEP and SDL [4, 22], general formalisms as Petri nets [24, 14] and logics [5], to rather high-level specifications as UML [10, 9] and Web services [20, 13, 16].

As to our work, the most relevant works are those focusing on high-level specifications. In [10], the authors propose an approach to analyze designs of CORBA applications via translation of a subset of UML (class, deployment, and state chart diagrams) into Promela. Then, they are checking for LTL properties and absence of deadlocks. A similar method for analysis of BPEL modeled as UML activity diagrams is presented in [9]. It is based on transformation of a UML meta-model into the Promela language. Again, absence of deadlocks is verified, while the method stresses the correctness of mapping defined as a homomorphism.

Further, in the field of Web services, other specialized formalisms are being translated into Promela to allow easier analysis (including LTL checking) by Spin. In [20], the author describes translation from WSFL (Web Services Flow Language), a language for description of Web service aggregation. A WSAT tool [13] translates BPEL4WS into a set of (guarded) finite state automata followed by translation into Promela, while focusing on problems of translation of XML data and XPath expressions.

In contrast to the aforementioned work, EBP is a formalism for behavior specification of components. Although there is a lot of work focusing on reasoning about behavior compatibility/communication correctness of software component, to our knowledge, none of them takes into account interplay among interfaces of a particular component, which we consider as an important aspect when reasoning about component behavior. To mention at least some of the works focused on component behavior—in [11], the authors model the component behavior as a labeled Petri Net. In this case, the behavior compatibility relation is based on the subset relation of services requests. Since Petri nets are able to capture not only regular languages, but also a subset of context-free and even context-sensitive languages, verification of component compatibility defined as a subset relation is challenging. Due to complexity of the method, scaling to real applications would be problematic.

EBP stem from BP; BP are also a language for specification of component behavior, however, they only contain two basic events for method call request and response. That means that neither method parameters nor variables are supported and no switch and while statements can be used. In this sense, BP can be seen as a more abstract specification platform than EBP.

In comparison to the Promela language, which could also be used to model behavior of components, EBP benefits from direct support of abstractions commonly used in component based development (e.g., components, interfaces and method calls). Using Promela, these abstractions have to be modeled manually.

In [12], the authors define *Interface automata* for modeling behavior of components. They use an optimistic view that components are considered compatible if there exists an environment that can interact with them without errors. In contrast to Interface automata, which are based on message passing, EBP directly supports method invocation, which brings it closer to code[10]. We also believe that the textual

[8] `http://dsrg.mff.cuni.cz/cocome_sofa.phtml`

[9] `http://sofa.objectweb.org/fm/`

[10] The two formalisms also differ in their view on subtyping. More information on the EBP view can be found in [17], however, a proper discussion is beyond the scope of this paper.

form of EBP is more convenient for user and helps scalability.

Component-interaction Automata [6] provide a framework upon which a concrete verification system can be built. For example, the authors define composition of particular automata in a very general way thus allowing for different composition styles in various approaches. Since the component behavior is defined directly as an automaton, the formalism is not easy to use as a specification platform for developers, it can rather serve as a target platform for translation of a higher-level language. To compare this formalism with EBP, the main difference inheres in absence of data support; although this does not influence the expression power of the formalism, the usability in case some data are needed is hereby decreased.

Since model checking of code is an undecidable problem, the models corresponding to EBP specification have to be limited in several ways. These limits include inability to express recursive calls, absence of a global heap, and only regularly expressible sequences of events. Despite these limitations, EBP turned out to be an easily usable yet abstract enough to make the verification process feasible.

6 Conclusion and Future Work

We have presented a method for checking correctness of component composition using the EBP formalism for behavior specification and its automated translation into Promela. Employing Spin as a model checking back-end provides additional features for free (e.g., LTL checking, bit state hashing, etc.) while taking away the burden of development, testing and support of a proprietary model checker.

The strong point of our work is application of our approach to a real-life-sized case study in the CoCoME contest. CoCoME constitutes a common platform for a number of component models and related formalisms, making our approach easily comparable to other approaches.

The weak point and our future work is checking EBP against actual Java code of components. This is already implemented for the old version of Behavior protocols [21] (via combination of Java PathFinder and a proprietary Behavior protocol checker) and can be in principle used with substantial manual modification of the EBP specification. However, making the process suit EBP without necessity of manual inspection is an important piece of the whole puzzle.

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] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[4] C. Bazlio, E. H. Haeusler, and M. Endler. Language-oriented formal analysis: a case study on protocols and distributed systems. *ENTCS*, 184:189–207, 2007.

[5] D. Bianculli, A. Morzenti, M. Pradella, P. S. Pietro, and P. Spoletini. Trio2promela: A model checker for temporal metric specifications. In *ICSE COMPANION ’07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 61–62. IEEE Computer Society, 2007.

[6] L. Brim, I. Cerna, P. Varekova, and B. Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. *SIGSOFT Softw. Eng. Notes*, 31(2):4, 2006.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An Open Component Model and Its Support in Java. In *CBSE*, pages 7–22, 2004.

[8] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA*, pages 40–48, 2006.

[9] H. Cao, S. Ying, and D. Du. Towards Model-based Verification of BPEL with Model Checking. In *Proceedings of the 6th International Conference on Computer and Information Technology (CIT’06)*, page 190. IEEE Computer Society, 2006.

[10] J. Chen and H. Cui. Translation from Adapted UML to Promela for CORBA-Based Applications. In *SPIN*, volume 2989 of *LNCS*, pages 234–251. Springer, 2004.

[11] D. Craig and W. Zuberek. Verification of component behavioral compatibility. In *Proceedings of 2-nd Int. Conf. on Dependability of Computer Systems (DepCoS’07)*, pages 294–304. IEEE Computer Society, 2007.

[12] L. de Alfaro and T. A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[13] X. Fu, T. Bultan, and J. Su. Wsat: A tool for formal analysis of web services. In *CAV*, pages 510–514, 2004.

[14] G. C. Gannod and S. Gupta. An Automated Tool for Analyzing Petri Nets Using SPIN. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 404. IEEE Computer Society, 2001.

[15] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003.

[16] R. Kazhamiakin, M. Pistore, and M. Roveri. Formal Verification of Requirements using SPIN: A Case Study on Web Services. In *Proceedings of Software Engineering and Formal Methods (SEFM'04)*, pages 406–415. IEEE Computer Society, 2004.

[17] J. Kofron. *Behavior Protocols Extensions*. PhD thesis, Charles University in Prague, 2007.

[18] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[19] K. L. McMillan. *Symbolic model checking — an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[20] S. Nakajima. Model-checking verification for reliable web service. In *OOPSLA 2002 Workshop on Object-Oriented Web Services*, November 2002.

[21] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In *Proceedings of 30th Annual IEEE/NASA Software Engineering Workshop (SEW'06)*, pages 133–141. IEEE Computer Society, 2006.

[22] A. Prigent, F. Cassez, P. Dhaussy, and O. Roux. Extending the Translation from SDL to Promela. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 79–94. Springer-Verlag, 2002.

[23] R. Reussner, I. Poernomo, and H. W. Schmidt. Reasoning about software architectures with contractually specified components. In *Component-Based Software Quality*, pages 287–325, 2003.

[24] C. Stehno. System Specification and Verification Using High Level Concepts—A Tool Demonstration. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 246–249. Springer-Verlag, 2002.

[25] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM Semantics in Promela and Its Possible Applications. In *SPIN*, volume 4595 of *LNCS*, pages 204–222. Springer, 2007.

[26] CoCoME (Common Component Modeling Example), `http://www.cocome.org`.