

Carmen : Software Component Model Checker

Aleš Plšek¹ and Jiří Adámek^{2,3}

¹ INRIA-Lille, Nord Europe, Project ADAM
USTL-LIFL CNRS UMR 8022, France

`ales.plsek@inria.fr`

² Distributed Systems Research Group
Charles University in Prague
Czech Republic

`adamek@dsrg.mff.cuni.cz`

³ Institute of Computer Science,
Academy of Sciences of the Czech Republic

Abstract. The challenge of model checking of isolated software components becomes more and more relevant with the boom of component-oriented technologies [20]. An important issue here is how to verify an open model representing an isolated software component (also referred as *the missing environment problem* in [17]).

In this paper, we propose on-the-fly simulation of the component environment to address the issue. We employ behavior protocols [18] and a system coordinating two model checkers: Java PathFinder [4] and BPChecker [15]. This approach allows us to enclose the model representing the behavior of a given component and consequently to exhaustively verify the model. Our solution was implemented as the Carmen tool [1]. We demonstrate scalability of our approach on real-life examples and show that, in comparison with the COMBAT model checker [17], we bring better performance, and also exhaustive and correct verification.

1 Introduction

Model checking [9], as one of the most popular approaches to formal verification of software systems, has already proven to be useful. However, the need for extracting a finite model from a target system (the "classical" model checking) forces researchers to seek approaches on model checking at the source-code level. Despite the complexities of these approaches, particularly *the state explosion problem*, there exist such model checkers (e.g. Java PathFinder [4] or Bandera [10]). One of the methods of coping with the issue of state explosion is decomposition of a system into small parts which can be verified separately.

Independently on this branch of research, widely popular Component-Oriented Programming [20] introduces software components – small compact units providing a certain functionality through strictly defined points. It is therefore natural to tackle the problem of software component verification, since components themselves bring the most straightforward way of decomposition – a property so intensively sought when fighting the state explosion problem.

In the scope of formal verification, we distinguish between closed and open systems. A closed system is autonomous, i.e. it does not communicate with another system. In the context of component programming, it is e.g. whole component application — there are no interfaces for the communication of the whole application with another component. On the other hand, an open system communicates with other entities; again, in the context of component programming, it is e.g. a single component, that communicates with other components (its environment) via interfaces. A behavior model of a closed system is called a closed model, while a behavior model of an open system is called an open model.

From the verification point of view, a behavior model specified by the code of a single component (an open model) is incomplete, as the behavior of the component depends not only on the decisions made by the component itself, but also on its environment. In the context of different environments, the behavior of the component can differ. However, the source-code level model checkers typically need a closed model as the input. Therefore, an important question arises here: How to enclose the model of a component and thus to allow formal verification? The challenge is also referred as *the missing environment problem* [17].

The goal of our research is to propose an answer to the question above. In this paper we design a method of on-the-fly simulation of software component's environment to achieve a closed model. In our solution, component's implementation and its behavior specification, given in a form of a behavior protocol [18], are processed by two cooperating model checkers - Java PathFinder [4] and BPChecker [15]. These cooperating tools then formally verify component's implementation against a behavior protocol and specified properties.

Our solution was implemented as the Carmen tool [1]. We compared Carmen with COMBAT — the tool presented in [17], addressing the same issue. We concluded that Carmen performs exhaustive verification, while COMBAT does not. Also, the state space traversed by Carmen is smaller, which is important for performance.

To reflect the goal, the structure of the paper is as follows. Section 2 introduces basic insights into Java PathFinder, Component-Oriented Programming, and Behavior Protocols. At the end of the section, we elaborate on the goal of our research. While Section 3 presents possible approaches to *the missing environment problem*, Section 4 describes in detail the concept we have chosen to implement. Section 5 demonstrates our contributions and scalability of the solution on real-life examples. In Section 6 we discuss related work. Section 7 concludes the paper.

2 Background

2.1 Java PathFinder

Java PathFinder (JPF) [4,21] is an explicit state software model checker. It verifies given program by traversing its state space and searching for implementation errors (e.g. deadlocks, unhandled exceptions,...) and property violations.

Moreover, user's own properties can be defined. JPF operates at the program byte-code level which means that a real-life application written in Java is used as a model of a system. A custom Java virtual machine (JPF VM) is used to execute a given program in every possible execution path. The state space of a target program is a directed acyclic graph in principle with branches determined by Java bytecode instructions, thread interleavings, and possible values of input data. JPF fights the state-space explosion problem by implementing POR algorithm [9] and state matching heuristics [12].

2.2 Component-Oriented Programming

We employ the basic idea of Component-Oriented Programming [20] that is further extended in the hierarchical component models, e.g. [3,5]. Here, components are either *composite* (created as a composition of lower-level components) or *primitive* (implemented directly in a common programming language, e.g. Java). Components are viewed as black-box entities. Interfaces of components can be either *required* or *provided*. Through provided interfaces, services of the component are accessible, the required interfaces are connected to other components to intermediate delegation of tasks. By the term *environment* we denote all the components connected to the interfaces of a given component.

We implemented the Carmen tool, introduced in this paper, for the Fractal component model [3]. As a future work, we also plan to adapt Carmen for the SOFA component model [5]. We chose Fractal and SOFA because we had been experienced with the formal verification of the applications written in those component models and because checkers of behavior protocols had been already implemented for both of them [6,15].

2.3 Behavior Protocols

Behavior protocols [18] are a language for component behavior specification. They have been successfully applied to the SOFA [18] and Fractal [6,15] component models. To analyze behavior of components specified via behavior protocols, Behavior Protocol Checker (BPChecker) [15] was developed.

A behavior protocol describes communication of a component with its environment.⁴ On the semantic level, such a communication is defined as the set of all admissible sequences of *events* on the component's interfaces. There are two kinds of events: *requests* for method calls and *responses* to those requests.

Syntactically, behavior protocols are similar to process algebra [7]. The basic building blocks of a behavior protocol are *event tokens*, denoting the events. An

⁴ To be precise, a behavior protocol can describe not only the communication of a component with its environment, but also the interplay of events inside a composite component. However, this alternative usage of behavior protocols is out of scope of this paper, as our goal is to check consistency of a primitive component code with the protocol of the component; specification of a composite component behavior via behavior protocols is not needed here. For more details, see [18].

event token has the following syntax: `<prefix><interface>.<method><suffix>`. The prefix `?` denotes acceptance of an event, the prefix `!` denotes emission of an event. The suffix `↑` denotes a request (i.e. a method call), and the suffix `↓` denotes a response (i.e. return from a method). Therefore, for `i` being an interface name and `m` being a method name on `i`, `?i.m↑` stands for accepting the request for a call of `i.m`, while `!i.m↓` denotes the emission of the response for a call of `i.m`.

Behavior protocols are syntactically constructed from the event tokens using *operators*. There are operators for sequencing (`;`), alternative behavior (`+`), repetition (`*`), and arbitrary interleaving (`()`), that is useful for behavior specification of parallel processes.

Also, abbreviations are defined for behavior protocols; they serve as syntactic sugar, standing for complex but often used constructs. The abbreviation `?i.m` stands for `?i.m↑ ; !i.m↓`, i.e. acceptance of a request followed by the emission of the associated response (i.e. the typical part of the behavior of a component providing `i.m` to the outside world). Similarly, `!i.m` stands for `!i.m↑ ; ?i.m↓`. Finally, if `P` is an arbitrary protocol, `?i.m{P}` stands for `?i.m↑ ; P ; !i.m↓`, i.e. it describes a part of the behavior of a component providing `i.m`, where the protocol `P` describes what the component does inside of the implementation of `i.m`.

NULL stands for an empty protocol (specifying no behavior).

We demonstrate the usage of behavior protocols on a simple example shown in Fig. 1. Here, the functionality of the *Database* component is expressed by its behavior protocol. First, *Database* accepts the initialization call — `db.start`; this leads to calling `lg.log` and then the result of the `db.start` call is returned. After that, *Database* is able to absorb an arbitrary number of `db.get` or `db.put` calls, each resulting in an `lg.log` call. To finish the execution, the component is stopped by calling `db.stop`.

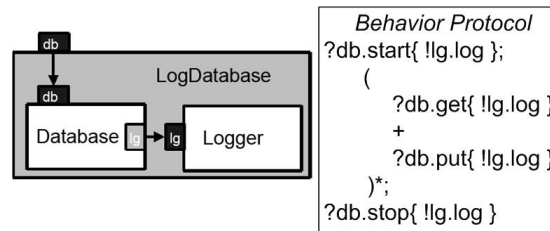


Fig. 1. Motivation Example: the *LogDatabase* composite component consisting of the *Database* and *Logger* subcomponents. The small black and gray boxes denote provided and required interfaces. E.g., `db` is a provided interface of *Database*, while `lg` is a required interface of *Database*. The behavior protocol of *Database* is shown.

2.4 Goal Revisited

Behavior protocols give to a component application developer the option to check consistency of his or her design from the point of view of component behavior. For example, if behavior protocols of all three components in Fig. 1 are provided by the developer, it is possible to check correctness of communication between *Database* and *Logger*, as well as compliance of the *LogDatabase* internals behavior (determined by the protocols of *Database* and *Logger*) with the *LogDatabase* protocol itself [18]. However, once we take also primitive components into consideration (i.e. the components that are not composed of subcomponents, but directly implemented in some programming language instead — and there must be such components in each application), things get more complicated.

Let us assume that *Database* is primitive (and is implemented in Java). Now, we cannot assure the correctness of the *Database* implementation by pure behavior protocol analysis, as there are no behavior protocols describing the behavior of *Database* internals.

Moreover, we can look at the problem from another point of view: we want to use verification tools for Java code. One of the options to specify the properties to verify is to use *assertions* — conditions that must be true when the control reaches given places in the code⁵. However, as the code of the component is an open code (it has no predefined entry point and the behavior of the code depends on how the environment will use it), it is not possible to use the code verification tool to check the properties expressed as assertions. As mentioned in the introduction, this issue is called missing environment problem [17].

Therefore, the goal of our work is the following: to design and implement a tool that (1) checks the compliance of Java implementation of a primitive component with the behavior protocol of the component (i.e. verifies that the code does what the protocol specifies), and (2) at the same time it checks validity of the assertions in the Java code; only those runs that correspond to the behavior specified via the protocol are taken into consideration.

We chose Java as both the SOFA and Fractal component models (where the behavior protocols were already applied) use Java as the implementation language.

3 Cooperation of Model Checkers

The problem we tackle in this paper is a verification problem. To solve it, we decided rather than developing a brand new tool to adapt an existing model checker. From our study, the Java Path-Finder tool (JPF) emerged as the best option. It provides wide functionality and can be easily modified and extended.

However, JPF itself does not allow to cope with all the issues of a single component verification. Since JPF allows to verify only closed models, we introduce

⁵ Contrary to the classic assertions used for software testing, assertions in formal verification are much more powerful tool, as the verification tool checks the validity of assertions for *all* possible runs.

behavior protocols to substitute the environment of the component and thus to enclose the model. During the verification it is then necessary to observe the communication of the component with the environment represented by behavior protocols. To do this, we employ an additional model checking tool – BPChecker. The specific details of such a checker cooperation form our main contribution.

The task of the cooperation is to synchronize the verification of the component implementation, performed by JPF, and the verification of component external behavior, performed by BPChecker, whenever a communication between the component and its environment occurs. Such a synchronization can be achieved using two different concepts discussed further: Virtual Environment or Environment Simulation.

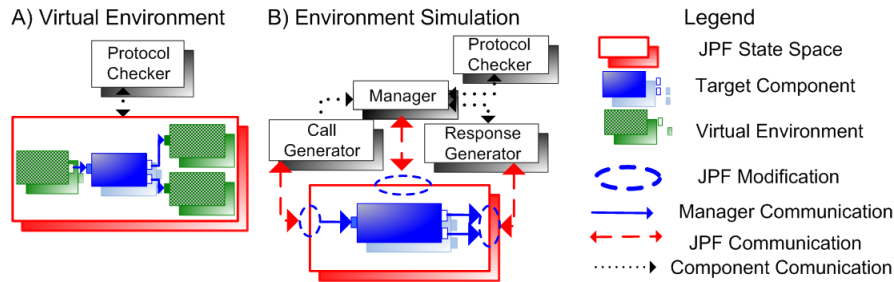


Fig. 2. JPF and BPChecker Cooperation, Proposed Concepts

3.1 Virtual Environment Concept

The key idea of this concept, presented in Fig. 2 A), is to automatically generate virtual environment of a component (i.e. a Java code), creating a closed system that can be verified by JPF. Such a code has to provide an entry point (the `main` method). Moreover, the virtual environment has to be able to perform every sequence of events described in the component's behavior protocol. This guarantees that JPF will be able to analyze all the behavior alternatives that are relevant.

While verification of such an enclosed model (code of the component + virtual environment) is simple (this can be done with just a minor modification of JPF [17]), generating a virtual environment from the protocol has many issues. The reason is that some forms of behavior protocols (e.g. those specified using the alternative and repetition operators) can not be equivalently expressed by a Java code. Therefore, no virtual environment can correspond to such protocols, and consequently the verification process cannot be correct. Despite its disadvantages, this concept was implemented in the COMBAT model checking tool [17].

3.2 Environment Simulation Concept

The idea of the Environment Simulation concept is to use JPF to analyze only the code of the verified component itself and to handle the events on the external interfaces of the component via a modification of JPF — see Fig. 2 B). Every time the *Manager* detects communication initiated by the verified component, it interrupts the verification process and let the *Response Generator* simulate appropriate environment responses according to the behavior protocol of the verified component. The information about the appropriate environment responses is taken from the *Protocol Checker*, that is run in a special mode. At the same time, *Call Generator* is used to simulate the calls initiated by the environment. Finally, the *Protocol Checker* is used not only to obtain the information about the environment responses, but also to check that the events emitted by the verified component respect the protocol.

The Environment Simulation concept allows to simulate any form of behavior protocols, including the alternative and repetition operators, providing correct and exhaustive form of verification. Moreover, as *Manager* can interrupt the verification process at any time and force JPF to explore another execution path, it is possible to control verification and to smoothly integrate additional heuristics.

In the light of the outlined options, the Environment Simulation concept was chosen to implement. Based on this decision, the Carmen project [1] was founded. More extensive description of the project can be found also in [19].

4 Environment Simulation

Based on the discussion above, we propose to develop a Software Component Model Checker, which implements the Environment Simulation concept. To facilitate the verification, we have to simulate a component environment by generating events that will be absorbed by the component. The component is then forced by JPF to respond to these artificially created events, its behavior is evaluated and thus the component is being verified.

4.1 Cooperation

Since each tool operates at a different level of abstraction - JPF with byte-code instructions and BPChecker with events, we need to define a proper mapping between their state spaces to achieve cooperation. This would be possible if states that represent absorbed events could be identified. Whereas this is inherently satisfied inside the BPChecker state space, the JPF state space represents only the component itself. To tackle this problem, we have extended the JPF state space with states that represent communication between the component and its environment. Therefore we are able to find a mapping between state spaces of the checkers. See Fig. 3 for an illustration example.

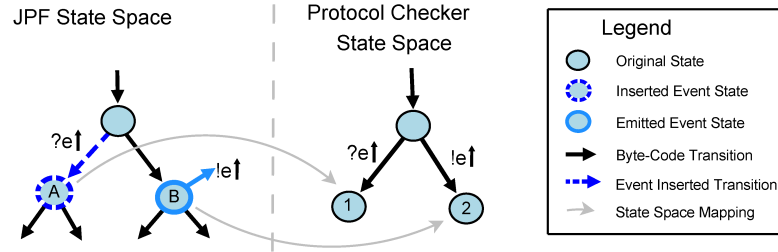


Fig. 3. State Space Mapping

4.2 Environment Simulation

The environment simulation process generates events that occur on interfaces of the component. These events have to be then inserted into the JPF, afterwards the verification can continue. We are able to determine which event has to be inserted in cooperation with BPChecker. Consequently, state space extensions allow to simulate an absorbed event by creating a new state and by inserting it into the JPF. Verification then continues from a newly inserted state and thus, the component is forced to react to the new event.

Moreover, by employing the backtracking strategy we are able to simulate every possible sequence of events. The component is therefore verified against every behavior of its environment that is in conformance with a behavior protocol of this component.

An absorbed event is however representing also a data which are being transformed from an environment to the component and these data have to be generated as well. We refer to this in Section 5.2.

4.3 Verification

The central unit of the verification process is *Manager*. It communicates with both the checkers, arbitrates the cooperation between JPF and BPChecker and determines future steps of the verification. Manager evaluates states of the checkers and decides which events will be simulated on interfaces of the component. Figuratively speaking, JPF represents the component, BPChecker represents its environment and Manager provides a connecting layer between them.

To better illustrate the role of Manager, we introduce code snippets of methods which are used by Manager to control the progress of the verification. The method `stateAdvanced()` listed in Fig. 4 handles a situation when JPF advanced a new state. First, Manager verifies if there was any emitted event and whether it was in compliance with a given behavior protocol (line 2-3). Consequently, Manager tries to simulate a next event, if BPChecker proposes any event, it is simulated, both tools are notified and we proceed to a new state (lines

```

void stateAdvanced() {
2  if eventEmitted()
    BPChecker.verifyEvent(emittedEvent);
4  newEvent = BPChecker.getEvent();
    if newEvent != null
6    JPF.simulateEvent(newEvent);
    BPChecker.eventSimulated(newEvent);
8    stateAdvanced();
    else
10   if JPF.isEndofExecutionPath() && BPChecker.isNotAccepting()
        reportErrorBehavior();
12 }

14 void stateBacktracked() {
    if isEventToBacktrack()
16   BPChecker.backtrackEvent(event);
    newEvent = BPChecker.getEvent();
18   if newEvent != null
        JPF.simulateEvent(newEvent);
20   BPChecker.eventSimulated(newEvent);
        stateAdvanced();
22 }

```

Fig. 4. Manager Arbitrating an Advanced/Backtracked State

5-8). If there is no event to simulate, Manager only verifies that both tools are in accepting states in case the end of an execution path was reached.

The method `stateBacktracked()`, listed in Fig. 4 (line 15), handles situations when JPF backtracked from an already explored state. The task of Manager is to backtrack also a simulated event and then to simulate a new one (lines 20-22). If there is no event to simulate, nothing is to be done since all paths starting by events were already explored.

Thanks to these notification methods Manager is able to coordinate cooperation of both checkers and thus to achieve an exhaustive verification of all execution paths of the component implementation.

4.4 Motivation Example Revisited

In this section we revisit the motivation example from Section 2.3 to demonstrate the verification process. The Fig. 5 shows the implementation of the `Database` component together with its behavior protocol. The arrows are showing the correspondences between events of the behavior protocol and method calls inside the component implementation code.

From JPF point of view, every event absorbed by a component is represented inside the JPF VM as a thread which invokes a given method on a particular

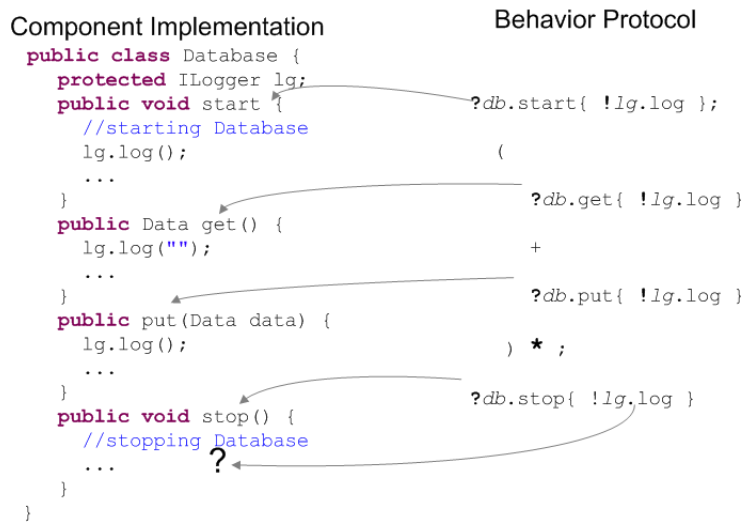


Fig. 5. Example of the Verification

interface. On the other hand, an emitted event is represented as a thread invoking a method on an interface of another component.

Considering our example, both checkers are in their initial states at the beginning, there are no threads in JPF VM. Manager therefore asks BPChecker for a list of events which can be simulated. According to the protocol, an event `?db.start` is proposed. This event is then simulated, a new thread which invokes the method `start` on the interface `db` is created inside JPF. From now JPF starts with the verification of the component's code. This process is monitored by Manager and interrupted whenever the component tries to communicate with its environment. Here, such situation occurs when the component invokes `lg.log`. Manager immediately stops the verification and verifies that the emitted event conforms to a given behavior protocol. Then the thread is interrupted until the moment when BPChecker proposes a simulation of an event which represents a response to the invoked call. In between, the verification of parallel threads inside the JPF state space can continue.

Looking at the behavior protocol in Fig. 5, we can see that an event `lg.log` does not have any corresponding method call in the component implementation, in Fig. 5 indicated by a question mark. During the verification, JPF executes the method `stop`, reaches its end and notifies BPChecker. However, the protocol specifies that during the `stop` method execution, an event `!lg.log` will be emitted. Since no such event occurred, it is an obvious behavior protocol violation and an extensive report (including stack traces of both checkers) will be send to the developer.

Except the component's behavior, which is verified whenever the component emits an event, the JPF checker itself verifies additional properties, e.g. the presence of deadlocks, unhandled exceptions or any other user defined properties. This finally leads to an exhaustive verification of every property along all the execution paths of the component's implementation.

5 Evaluation

As the biggest contribution of our work we consider the Environment Simulation concept that straightforwardly solves *the missing environment problem*. Contrary to the COMBAT checker [17] described in Section 6, we do not require any reductions of behavior protocols; therefore, our approach provides an exhaustive simulation of the environment and correct verification of components.

To show the quality of our method, we developed a prototype implementation – Carmen Project [1]. The tool verifies Fractal software components [3] implemented in Java against their behavior protocols and the sets of user-defined properties. In this section we present the performance evaluation and discuss the limitations of our tool.

5.1 Case Studies and Performance Evaluation

For performance evaluation, we used real-life case studies from the Component Reliability Extensions for Fractal component model (CRE) [6] and CoCoMe [2] projects.

CRE is an application that manages the airport services for wireless internet connection. It consists of more than twenty components. We have selected three non-trivial components for verification⁶: the *FlyTicketClassifier* component classifies air tickets and provides connections to the appropriate database, the *ValidityChecker* component verifies the airtickets, and the *Arbitrator* component controls the whole system.

For the second part of the performance evaluation, we used the Fractal implementation of the *Store* and *CashDeskExample* components from the CoCoMe case study [8], addressing the simulation of cash desk system in a supermarket.

For the performance evaluation, we did several comparison tests between Carmen and COMBAT, using the code of the components mentioned above. The following parameters have been monitored: *Unique States* (number of unique states that were reached), *Visited States* (total number of reached states), *Time* (total time of the verification), and *States/Second* (the number of states visited per second).

The results of the performance evaluation⁷ are presented in Table 1.

⁶ More detail information regarding these components, the whole case study, and the Carmen documentation can be found at the project web page [1].

⁷ All the tests were run on Pentium 4 3.0 GHz with 2.0 GB RAM, Windows Server 2003 OS.

Table 1. Performance Comparisons

Case Study	Component Name	Checker	# States		Time	States/Second
			Unique	Visited		
CRE	FlyTicketClassifier	Carmen	922	1 920	3s	640
		COMBAT	6 519	10 254	4s	2563
CRE	ValidityChecker	Carmen	435	592	2s	296
		COMBAT	4 033	9 324	4s	2331
CRE	Arbitrator	Carmen	6 074	14 898	34s	438
		COMBAT	166 977	378 437	9m:30s	663
CoCoMe	CashDeskApp	Carmen	3 480 851	6 644 606	1h:32m:17s	1200
		COMBAT	4 839 108	10 541 046	33m:26s	5 254
CoCoMe	Store	Carmen	574 538	1 717 282	2h:29m:09s	192
		COMBAT	11 669 994	28 728 733	1h:49m:08s	4 387

While COMBAT verifies a closed system (including the generated environment), Carmen simulates the environment during the verification and therefore the progress of the verification is slower. This can be observed when verifying the components from the CRE case study — *FlyTicketClassifier*, *ValidityChecker*, and *Arbitrator*. However, the state space of COMBAT is larger, which is caused by the necessity to include the generated environment. Thus, the total verification time is better for Carmen in all the three cases and the difference between the total verification times (Carmen vs. COMBAT) is the bigger the larger the state space is.

As to the verification results for the CoCoMe case study (*CashDeskApp*, *Store*), Carmen again generates considerably smaller state spaces and the verification times are reasonable. However, the COMBAT tool achieves better total verification time. We believe that this is caused by the recent progress of the COMBAT tool which was ported to a newer version of JPF (version 4), whereas Carmen uses an old one (version 3.3.1). We reflect this finding in our future work (Sect. 7).

The bottom line is that Carmen is able to verify complex components in a reasonable time without any reductions of behavior protocols. The confrontation with COMBAT, which requires additional reductions of behavior protocols, has revealed that Carmen reaches fully correct verification and comparable performance.

5.2 Tool Limitations

Even though our approach potentially achieves exhaustive verification, the real-life application brings several limitations. Specification of parameters that are passed to the methods when generating events is the most important burden to deal with. The range of possible values has to be manually specified and its extensiveness directly affects the state space size. Therefore, the values should

be chosen with respect to the component implementation, to allow the checker to explore maximum of execution paths. The details are out of scope of this paper, we briefly discuss some of them in Sect. 6.

More detailed evaluation of Carmen and a discussion of its limitations can be found in [19].

6 Related Work

COMBAT [17] uses, similarly to the approach applied in our work, JPF in cooperation with BPChecker. It generates a virtual environment that is verified together with the component (see Section 3.1). For more information about the environment generation see [16]. However, the significant disadvantage of this checker lies in the absence of any solution to repetition and alternative operator problems addressed by Carmen. Instead, behavior protocols are simplified in order to avoid unsupported forms of protocols. These constraints consequently lead to a non-exhaustive verification of components. Nevertheless, we demonstrate the performance comparisons between both the approaches in Section 5.

Also, our approach is related to the assume-guarantee principle in model checking [13]. The tools based on this principle report the description of all the environments in which a given model satisfies a given property. We also use the idea of environment, but in the opposite manner: the description of the environment behavior (the calls from the environment to the component described in the behavior protocol) is given by the developer and the tool checks whether the property is satisfied in the environment. Note that the property itself is also specified by the behavior protocol (the reaction of the component to the calls made by the environment).

When searching for an equivalent alternative to Java PathFinder [4], we have been considering an alternative — Bandera [10]. It is a set of tools and modules which are designed to verify Java programs. Bandera accepts a complete Java program as an input and translates it into a language that can be verified by a specified model checker. Although Bandera is not intended to verify software components, it decomposes a target program into a part which is verified and the rest that is represented by specially generated environment. This approach is very similar to the Environment Generation concept presented in Section 3.1. Bandera also allows to use value domains for specifications of method parameters of given classes. However, the recent release of Bandera is an alpha version which is not fully stable yet.

Finally, we chosen Java PathFinder [4] as the basis of our implementation since it allows modification of its core implementation and is designed to support extendability by additional plugins.

In our work, we mainly focus on the verification of the order in which the methods of the component are called; another big issue is to cope with the values of the parameters that are passed to the methods. We use very simple heuristic approach to solve this problem, more sophisticated methods can be found e.g. in [11] or [14]: under-constrained execution [11] is a special kind of symbolic

execution, where some of the symbolic values (e.g. those that origin from the parameter values) are marked as under-constrained. If an error involves an under-constrained operand, an error message is produced only if the error occurs for all possible values of the operand (according to its type). This approach reduces the number of spurious errors. In [14], symbolic execution with lazy initialization is used to adapt Java PathFinder for verification of open systems: the method parameters are initialized during the execution in a lazy way; the exact value domains are not required from the developer.

7 Conclusion

In this paper, we present our approach to model checking of software components. Our solution verifies software components implemented in the Java language against their behavior specifications (behavior protocols [18]) and sets of user-defined properties. To achieve the goal we designed a system that coordinates two model checking tools: Java PathFinder [4] and BPChecker [15]. Our solution was implemented as the Carmen tool [1].

Carmen employs on-the-fly simulation of software component environment to enclose the model representing implementation of an isolated software component. We consider this feature as the biggest contribution of our work.

Scalability of our approach was tested on real-life examples and the results show that our solution provides reasonable performance and brings fully correct verification.

As a future work we plan to improve performance of our tool by porting it to the most recent version of Java PathFinder (and thus to fully use its state-of-the-art verification heuristics).

8 Acknowledgments

Special thanks go to the Distributed Systems Research Group, in particular to Jan Kofroň and Pavel Parizek, for helping with BPChecker integration and for assistance during performance testing.

This work was partially supported by the Grant Agency of the Czech Republic project 201/08/0266, by the ANR/RNTL project Flex-eWare and by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

References

1. Carmen Project. <http://www.lifl.fr/~plsek/projects/carmen/>. 2008.
2. CoCoMe Project. <http://agrausch.informatik.uni-kl.de/CoCoME>. 2008.
3. Fractal Project. <http://fractal.ow2.org/>. 2008.
4. Java PathFinder Model Checker. <http://javapathfinder.sourceforge.net/>. 2008.

5. SOFA Project. <http://sofa.objectweb.org/>. 2008.
6. 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. 2008.
7. Bergstra, J. A., Ponse, A., Smolka, S.A. *Handbook of Process Algebra*. Elsevier, 2001.
8. Bulej, L., Bures, T., Thierry Coupaye, Decky, M., Jezek, P., Parizek, P., Plasil, F., Poch, T., Nicolas Rivierre, Sery, O., Tuma, P. CoCoME in Fractal. *in Proceedings of the CoCoME project*, Jun 2007.
9. Clarke, E., Grumberg, O., Peled, D. Model Checking. *MIT Press*, Jan 2000.
10. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Laubach, R. S., Zheng, H. Bandera: Extracting Finite-state Models from Java Source Code. *in proc. of the 22nd International Conference on Software Engineering*, June, 2000.
11. Engler, D., Dunbar, D. Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable. *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.
12. Groce, A., Visser, W. Heuristics for Model Checking Java Programs. *Int. Journal on Software Tools for Technology Transfer (STTT)*, Vol. 6, No. 4.
13. Giannakopoulou, D., Pasareanu, C.S., Barringer, H. Component Verification with Automatically Generated Assumptions. *Journal of Automated Software Engineering, Kluwer Academic Publishers, Volume 12, Issue 3*, July 2005.
14. Khurshid, S., Pasareanu, C.S., Visser, W. Generalized Symbolic Execution for Model Checking and Testing. *Proc. of TACAS 2003. Warsaw, Poland*, April 2003.
15. Mach, M., Plasil, F., Kofron, J. Behavior Protocol Verification: Fighting State Explosion. *Published in the Int. Journal of Computer and Inf. Science, Vol.6, No.1, ACIS, pp. 22-30*, Mar 2005.
16. Parizek, P., Plasil, F. Specification and Generation of Environment for Model Checking of Software components. *In Proc. of Formal Foundations of Embedded Software and Component-Based Software Architectures*, 176, no. 2, May 2007.
17. Parizek, P., Plasil, F., and Kofron, J. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. *In Proceedings of 30th IEEE/ NASA Software Engineering Workshop (SEW-30)*, Jan 2007.
18. Plasil, F., Visnovsky, S. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28, no. 11, Nov 2002.
19. Plsek, A. Extending Java PathFinder with Behavior Protocols. Master Thesis, available at <http://www.lifl.fr/~plsek/projects/carmen/download/documents/masterThesis.pdf>. 2006.
20. Szyperski, C. *Component Software: Beyond Object-Oriented Programming, 2nd ed.* Addison-Wesley Professional, Boston, 2002.
21. Visser, W., Havelund, K., Brat G., Park, S., Lerda, F. Model Checking Programs. *Automated Software Engineering Journal. Vol. 10, No. 2*, April 2003.