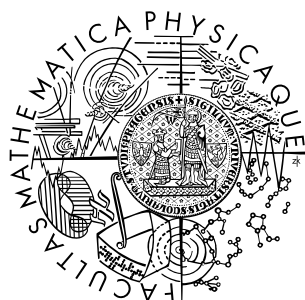


Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Pavel Parížek

Formal Verification of Components in Java

Department of Software Engineering
Advisor: Prof. František Plášil

Abstract

Title: Formal Verification of Components in Java
Author: Pavel Parížek
email: parizek@dsrg.mff.cuni.cz
phone: +420 2 2191 4235
Department: Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic
Advisor: Prof. František Plášil
email: plasil@dsrg.mff.cuni.cz
phone: +420 2 2191 4266
Mailing address (both Author and Advisor):
Dept. of SW Engineering, Charles University in Prague
Malostranské nám. 25
118 00 Prague, Czech Republic
WWW: <http://dsrg.mff.cuni.cz>
This thesis: <http://dsrg.mff.cuni.cz/~parizek/phd-thesis>

Abstract:

Formal verification of a hierarchical component application involves (i) checking of behavior compliance among sub-components of each composite component, and (ii) checking of implementation of each primitive component against its behavior specification and other properties like absence of concurrency errors. In this thesis, we focus on verification of primitive components implemented in Java against the properties of obeying a behavior specification defined in behavior protocols (frame protocol) and absence of concurrency errors. We use the Java PathFinder model checker as a core verification tool.

We propose a set of techniques that address the key issues of formal verification of real-life components in Java via model checking: support for high-level property of obeying a behavior specification, environment modeling and construction, and state explosion. The techniques include (1) an extension to Java PathFinder that allows checking of Java code against a frame protocol, (2) automated generation of component environment from a model in the form of a behavior protocol, (3) efficient construction of the model of environment's behavior, and (4) addressing state explosion in discovery of concurrency errors via reduction of the level of parallelism in a component environment on the basis of static analysis of Java bytecode and various heuristics. We have implemented all the techniques in the COMBAT toolset and evaluated them on two realistic component applications. Results of the experiments show that the techniques are viable.

Keywords

Software components, Java language, behavior protocols, concurrency errors, model checking, static analysis, state explosion, environment modeling

Acknowledgments

I would like to thank all those who supported me in my doctoral study and the work on my thesis. I very appreciate the help and counseling received from my advisor Prof. František Plášil. For the various help they provided me, I also thank my colleagues; in particular to (in alphabetical order): Jiří Adámek, Petr Hnětynka, Pavel Ježek, Tomáš Kalibera, Jan Kofroň, Tomáš Poch, and Ondřej Šerý.

My thanks also go to the institutions that provided financial support for my research work. Through my doctoral study, my work was partially supported by the Grant Agency of the Czech Republic projects 201/03/0911 and 201/06/0770, by the Czech Academy of Sciences project 1ET400300504, and the ITEA project OSIRIS.

Last but not least, I am in debt to my parents and Alena, whose support and patience made this work possible.

Contents

1	Introduction	5
1.1	Formal verification	5
1.2	Software components	8
1.3	Formal verification of software components	9
1.4	Problem statement	11
1.5	Goals of the thesis	11
1.6	Structure of the thesis	12
2	Background	13
2.1	Program verification frameworks	13
2.2	Compositional verification	15
2.3	Behavior Protocols	16
3	Goals revisited	19
4	Contribution: verification of Java code of primitive components	20
4.1	The whole picture — application of the assume-guarantee paradigm	21
4.2	Modeling component environment via behavior protocols	26
4.3	Obeying a frame protocol	27
4.4	Addressing state explosion in discovery of concurrency errors	29
4.5	Contribution reflected in publications	29
5	Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker	32
6	Specification and Generation of Environment for Model Checking of Software Components	42
7	Modeling Environment for Component Model Checking from Hierarchical Architecture	55
8	Partial Verification of Software Components: Heuristics for Environment Construction	70
9	Modeling of Component Environment in Presence of Callbacks and Autonomous Activities	79

10 Evaluation and related work	99
10.1 Method	99
10.2 Tools	103
10.3 Experiments	103
11 Conclusion	106
References	108

Chapter 1

Introduction

As increasingly complex and autonomous software systems are nowadays used in almost all domains, including mission- and safety-critical systems, reliability has become a key issue — there is growing need for reliable software systems.

Modern approaches to construction of reliable software systems include (i) usage of formal methods for analysis and verification of systems' behavior and (ii) decomposition of large and complex systems into well-defined units — software components. It is recommended to apply formal methods and decomposition at all stages of the development process to discover as many errors as possible and to ensure correct functionality with respect to the requirements. More specifically, use of formal verification at the design stage helps cut down the development expenses, since the cost of fixing an error is in general lower in case of design models than in the source code, while use of formal verification at the implementation stage helps discover low-level errors in the source code that may have been introduced into the system during implementation in a programming language.

Nevertheless, in this work we focus only on systems built from software components, and, in particular, on formal verification (and discovery of errors) during the implementation stage.

1.1 Formal verification

In software industry, the prevailing approach to discovery of errors in software systems (programs) is testing [11], which is based on monitoring and examining program's output during its execution; specifically, the output of a tested program is evaluated with respect to required and expected properties of the program's behavior. Although testing is a successful approach, it can never find all violations of a specific property (i.e. errors) in the implementation of a software system or prove that the system satisfies the property, since it checks only selected *executions paths* (paths in the state space) of the system [11] — e.g. only selected interleavings of parallel threads. For that reason, testing is not good at detection of subtle and difficult to reproduce errors (e.g. timing-dependent) that occur only in specific execution paths or depend on specific inputs. Well-known examples of such errors are *concurrency errors* (deadlocks and race conditions) that occur only for specific thread interleavings — in this case, use of formal methods can help.

Formal methods [14] are rigorous techniques for analysis and verification (hence formal verification) of design and implementation of software systems — in particular, they employ mathematical models of behavior of software systems and algorithms that process the models. The principal advantage of formal verification methods over software testing is the ability to (i) show whether a software system satisfies all the given properties (i.e. whether it is correct with respect to the properties) and (ii) find all violations of the properties (all errors) — formal methods aim at exhaustive checking of all execution paths in a software system, while testing can only find some of the errors. On the other hand, a drawback of formal methods is that they typically have very high time and memory requirements (i.e. they do not scale well) and give no answer for some inputs due to undecidability or intractability, while testing can be successfully applied to software systems of any size (it scales very well).

In between formal verification and software testing is the approach of runtime analysis (also called as dynamic analysis). Like testing, runtime analysis techniques check only selected execution paths of a software system and therefore cannot be used to verify that the system satisfies a specific property. However, the advantage of runtime analysis over testing is that the former can give some information about the execution paths that were not directly analyzed via limited use of formal techniques [35] — e.g. it can find a potential error on a different execution path than the one that was analyzed.

Techniques of formal verification are either automated or interactive. The group of automated techniques includes model checking and static analysis, while a typical example of interactive techniques is logical inference (theorem proving). Here in this work we focus on automated techniques of formal verification of software systems, i.e. on software model checking and static program analysis. An introduction to these two techniques follows.

Model checking

Model checking [21] was originally introduced as a technique for automated verification of finite state models of software and hardware systems against properties expressed via temporal logic, however, now it is used for all kinds of systems and properties, including potentially infinite systems like Java programs and high-level properties like obeying of a behavior specification defined via a state machine. Nevertheless, most model checkers support a proprietary and/or domain-specific low-level modeling language that allows to encode models in a way suitable for model checking — for example, SMV [45] uses guarded command language and SPIN [38] accepts models in Promela, which is a C-like programming language suitable for modeling communication protocols. There are only few model checkers that accept programs in mainstream programming languages like Java or C — these include Java PathFinder [60], Bogor [27], Blast [12] and MAGIC [19].

A typical model checker accepts a description (model or source code) of a software system and a set of properties as an input, and systematically traverses the state space of the system — i.e. it examines all possible execution paths, and, in particular, all possible interleavings of parallel threads — with the goal of detecting violations of the properties. If it finds a violation of a property (an error), it provides

a counter-example (an error trace) that identifies the execution path violating the property.

The main limitations of model checking with respect to verification of real-life software systems are *state explosion* [21] and *environment modeling* [51] (in this thesis also referred to as *problem of missing environment*).

The state explosion problem occurs especially in model checking of large and complex systems, as the size of the state space is roughly exponential with respect to the number of threads and components, and to the size of the input domains; it is typically manifested by the model checker running out of memory or time. Main approaches to addressing state explosion include abstraction [21], heuristics [32], symbolic model checking [45], and compositional reasoning [22][53]. Abstraction techniques help reduce the size of the whole state space (e.g. via restriction of data domains or slicing [58]) and thus make verification of system’s correctness more feasible, while heuristics help find specific kinds of errors in limited time and memory (i.e. before a model checker runs out of memory) via guiding the model checker to potential error states during the state space traversal. The basic idea of symbolic model checking is to represent the state space via formulas in a propositional logic, i.e. not as a state-transition graph. Finally, compositional techniques address state explosion via application of the divide-and-conquer approach — each component is verified in isolation — and therefore work well for systems that can be decomposed in a natural way (e.g. systems built from components). Since the idea of compositional reasoning is very related to the topic of this thesis, we provide a more detailed overview in Sect. 2.2.

The problem of missing environment is caused by the fact that model checking works only for closed systems, whose models have a single initial state (entry point) and reflect all possible values of inputs. However, most software systems are open, i.e. they do not feature a single entry point and their behavior depends on a particular environment — e.g. on data received over network or loaded from a file, or on actions performed by the user via GUI. Common solution to this problem is based on modeling and construction of an *artificial environment*, which closes the given open system [51]. Such an environment typically consists of (i) a test harness (driver) that exercises the given open system in various ways (simulating behavior e.g. of a user or a particular *real environment*) and provides an entry point (e.g. `main` method), and (ii) specification of a subset of possible values of inputs like method parameters and files with test data. Nevertheless, it is in general hard to construct an artificial environment that exercises a given open system in all reasonable ways that correspond to expected and valid usage of the system.

Static analysis

There exist many techniques of static analysis of programs — slicing [58], abstract interpretation [26], type inference, and also detection of specific control-flow (or syntactical) patterns in source code (e.g. [39]). The common characteristic of all these techniques is that they directly analyze the program’s source code, i.e. without the need of executing the program or systematically traversing its state space. In particular, static analysis allows to reason about the run-time behavior and output of a program at compile-time.

In general, the purpose of any static analysis technique is either to provide a specific information about the source code of a given program (e.g. specific abstraction of its behavior), or to check whether a specific property holds for the program's code (e.g. absence of deadlocks in all control-flow paths) [46]. However, the inherent limitation of static analysis is the ability to provide only imprecise or approximative answers [55], since (i) it does not have the knowledge of run-time values of all program's variables and (ii) works in general only with an abstraction of the source code (i.e. with an abstraction of the program's behavior) — computation of precise answers may be undecidable. There is also a trade-off between safety and precision of the analysis — e.g. for checking the code against a specific property, a safe (conservative) analysis reports also some spurious (false) errors in addition to real errors, while precise analysis reports only real errors (i.e. no spurious errors) but may not find all of them.

The majority of static analysis-based techniques do not suffer from the problem of missing environment, since they work for both closed and open systems. In case of an open system, it is only necessary to use (worst-case) assumptions about behavior of an environment of the system (fragment of a program) — e.g., it may be assumed that any value of a method parameter is possible. An obvious consequence is a lower precision of the analysis and reporting of spurious errors.

Combination of formal verification techniques

As indicated above, both the model checking and static analysis techniques have certain advantages and drawbacks with respect to each other — specifically, model checking is precise but suffers from state explosion and the problem of missing environment, while static analysis is typically efficient but produces approximate answers and spurious errors. A popular approach is to combine both techniques, so that one compensates for the drawbacks of the other and vice versa — typically, model checking is used as the main verification technique, and static analysis is used for construction of an abstract model of a software system subject to verification and for reduction of the state space size (e.g., in Bandera [25]). Note that some verification toolsets and frameworks also use theorem proving and runtime analysis as complementary approaches — the former as decision procedures that help construct abstract models of software systems' behavior (see e.g. [9]), and the latter to tell the model checker on which parts of the state space it should focus (see e.g. [35]).

1.2 Software components

A software component is, in the broadest sense, a unit of code that can be reused in different contexts without knowledge of its internals [57]. However, we use a more concrete definition: a *software component* is a reusable unit of code with explicit interfaces and well-defined behavior (functionality), which can be composed with other components to form a complex software system.

Each component has two kinds of interfaces — *provided interfaces*, which specify the services that the component provides to its clients, and *required interfaces* that specify the services it requires from the environment (e.g. from other components).

Components are then interconnected via *bindings* among interfaces; specifically, a provided interface of one component is bound to a required interface of another component. As for component’s behavior, it is necessary to specify its valid (expected) use by the clients and the reactions to such a valid usage — the component’s contract — in a formal way, i.e. via a formal behavior specification of some sort. The behavior specification can be defined in an event trace-based formalism (e.g. a process algebra [10]) or via pre/post-condition pairs.

A component-based software system is designed, implemented and deployed in a way that corresponds to a specific component model. In general, a *component model* is a conceptual framework that specifies a set of rules and concepts for all aspects of the lifecycle of both individual components and complete component-based systems — ranging from the particular definition of a component and the notion of a *component type* (defined, e.g., as a set of component interfaces) to an architecture definition language (ADL) that allows to express a structure of a component-based application. A *component platform* consists of a component model and a *runtime environment* for components, which implements the model.

There are two main kinds of component models — *flat* and *hierarchical*. The obvious difference between a hierarchical and flat component model is that the former supports nested components, while the latter does not. In case of a hierarchical component model, it is necessary to distinguish between two kinds of components (as a consequence of component nesting) — *primitive components* and *composite components*. Primitive components are leafs of a hierarchy, i.e. they are black-box entities that are implemented in a programming language like Java. Composite components are gray-box entities with externally visible structure — they are composed of nested sub-components interconnected via bindings among interfaces.

Flat component models and the corresponding platforms are typically developed by industry; the models are simple (i.e. provide less features), while the platforms provide stable and mature runtime environments. Well-known examples of such models are EJB [28] and CCM [47].

Component platforms based on hierarchical models — e.g. Darwin [43], Wright [4], SOFA [18] and Fractal [15] — are typically developed by academia. They support advanced features like multiple communication styles and behavior modeling and verification. However, on the other hand, the majority of them provide a very limited runtime environment (or none at all) [43][4]. Moreover, many hierarchical component models aim only at design of component applications, i.e. they support modeling of architecture and behavior, and completely neglect the component’s implementation and its verification.

Nevertheless, in this work we focus on those hierarchical component models and platforms that explicitly support both behavior modeling and component implementation in a programming language, in particular on SOFA [18] and Fractal [15].

1.3 Formal verification of software components

With respect to formal verification, there are two main differences between general software systems like Java programs and systems built from explicit components with well-defined interfaces and hierarchical structure (e.g. SOFA applications):

- 1) Individual components are typically equipped with a formal behavior specification of some kind, so that behavior compatibility among components at each level of nesting in a hierarchy (*behavior compliance*) can be formally verified;
- 2) A single component is an open system, i.e. the problem of missing environment is inherent to verification of implementation of individual components.

Ad (1) Most approaches to component behavior specification are based on formalisms like finite state machines (e.g. Labeled Transition Systems [10] - LTSs) and process algebras (e.g. behavior protocols [52] and CSP [37]) that allow to specify valid sequences of events on component interfaces, where an event corresponds to a method invocation or return, or to sending and receiving of a message. For example, behavior protocols [52] are used in the SOFA and Fractal component platforms, and CSP [37] is used in Wright. Then, two or more components are behaviorally compliant if they communicate without errors — e.g., in case of two components, one component should not emit an event that is not expected by the other. Nevertheless, checking of behavior compliance between components in a particular hierarchy makes sense only if implementation of each primitive component (e.g. Java code) in the hierarchy *obeys its behavior specification* (i.e. behaves in accordance with it). In case of design-oriented component models (e.g. Darwin), it is necessary to assume that the primitive component obeys its behavior specification, while in case of models like SOFA and Fractal, which explicitly support implementation of primitive components in real programming languages like Java and C, it is possible (and desirable) to formally verify that the implementation of each primitive component obeys its behavior specification.

Ad (2) A typical solution is to use a model of an artificial environment of the component subject to verification, so that a closed system can be constructed. In case of model checkers that accept programs in real programming languages, code of such an artificial environment has to be provided in order to create a complete program composed of the code of the component and environment. It is possible to use the most general environment (*universal environment*) that may call each method of the component at any time, for an arbitrary number of times, and in parallel with any of the other methods [29]. Nevertheless, a component is typically expected to work correctly only in some environments [3] — in that case, the obvious option is to use a restricted environment that behaves in the same way as a particular real environment of the component (e.g. the rest of a particular component-based system).

In general, the main advantages of building reliable software systems using well-defined components and their formal verification are:

(i) The possibility to increase performance of formal verification and make it feasible for real-life software systems; this is due to the opportunity to apply techniques of compositional reasoning, which exploit the natural modular (hierarchical) structure of such systems. For example, the problem of state explosion can be mitigated this way, since a single component typically has a smaller state space than the whole system.

(ii) Reuse of "correct" components in different contexts, e.g. in different applications. In particular, an isolated component can be formally verified together with a

specific model of its environment, using e.g. the assume-guarantee paradigm [53][30], and then employed in all software systems that interact with the component according to the model.

However, both advantages can be exploited only if the component platform supports formal specification of component behavior, behavior composition and compliance checking, and verification of implementation of primitive components.

1.4 Problem statement

As indicated above, many of state-of-the-art hierarchical component models support (i) specification of component behavior via finite state machines (e.g. LTS) and process algebras (e.g. behavior protocols), and (ii) formal verification of behavior compliance among components at the same level of nesting in a hierarchy via model checking. However, according to our knowledge, none of them supports checking whether an implementation of a primitive component obeys its behavior specification via automated methods of formal verification — i.e. via model checking and static analysis. The technique of model checking is in particular suitable for checking of implementation against such a property, since event trace-based behavior specifications describe the temporal behavior of a software system.

Considering formal verification of implementation of primitive components in general, i.e. against various properties (including obeying of behavior specification and absence of concurrency errors), an obvious idea is to apply the compositional approach and model check one isolated primitive component at a time in order to address the state explosion problem. Nevertheless, model checking of complex (real-life) and highly parallel components is still prone to state explosion, and, moreover, the problem of missing environment has to be addressed too in such a case.

To summarize, the challenge of automated formal verification of implementation of real-life primitive components against various properties has not been fully addressed yet.

1.5 Goals of the thesis

The general goal of the thesis is to address the challenge and issues mentioned above — i.e. automated formal verification of implementation of real-life primitive components against various properties — in the context of Java as the implementation language and behavior protocols as the formalism for component behavior specification. This includes:

- developing a model checking-based technique for verification of component Java code against the property of obeying a behavior specification defined via behavior protocols,
- solving the problem of missing environment for model checking of isolated primitive components implemented in Java, and

- addressing the problem of state explosion for verification of component Java code against the properties of obeying a behavior specification and absence of concurrency errors.

1.6 Structure of the thesis

The thesis is structured as a collection of already published papers with a unifying text. Chapter 2 provides necessary background and Chapter 3 describes revisited goals with respect to the background. Chapter 4 gives overview of our contribution with references to the included papers. The papers are included in Chapters 5-9. Then follows evaluation and related work (Chapter 10), and a conclusion.

Chapter 2

Background

In this chapter we present an overview of several verification frameworks for programs in Java — in particular, we discuss their advantages and drawbacks with respect to the goals of this thesis. Moreover, we provide an introduction to the compositional verification and assume-guarantee paradigm, and an overview of the formalism of behavior protocols.

2.1 Program verification frameworks

There exist several tools and frameworks that employ model checking for the purpose of verification of programs in mainstream languages like Java and C against various properties (e.g. absence of deadlocks and satisfaction of a temporal logic formula). For example, the Java PathFinder model checker [60] and Bandera toolset [25] aim at verification of Java programs, and MAGIC [19] is a framework for reasoning about programs written in the C language. In this section we focus on the verification tools for Java programs (i.e. Java PathFinder and Bandera), which are the most relevant with respect to our goals.

Java PathFinder

Java PathFinder (JPF) [60] is a highly extensible and configurable explicit-state model checker for Java programs. It accepts a complete Java bytecode program (with `main`) as an input, and works directly with the bytecode, i.e. on the level of bytecode instructions. In fact, JPF is implemented as a special Java virtual machine (JPF VM) that supports backtracking, state matching and non-determinism — it examines all execution paths of the given Java program. Each transition in the JPF state space corresponds to a set of bytecode instructions and each JPF state contains by default the full state of the JPF VM, including complete heap and stacks of all threads. Similar to other state-of-the-art software model checkers, JPF supports standard optimization techniques like partial order reduction (performed on-the-fly) and thread/heap symmetry reduction that help reduce the state space size.

The key feature of JPF is a high degree of extensibility and customizability; specifically, it provides an API for:

- plugins (listeners) that allow to monitor the state space traversal and execution of the checked program by JPF VM to a great detail, and to intercept it;
- custom search strategies (DFS, BFS, heuristic search) and various heuristics for state space traversal [32];
- customizable state management and representation — in particular, JPF can be configured to take into account only a subset of the full JPF VM state during the state space traversal (e.g. for state matching).

Besides that, JPF also provides an API — the `Verify` class — for non-deterministic data choice that can be used in test drivers to check a unit of code for different inputs; e.g. a call of `Verify.getInt(5)` means that the subsequent code is checked for each integer value in the range $0 \dots 5$.

As for properties a Java bytecode program can be checked against, JPF supports by default only low-level properties like absence of concurrency errors, uncaught exceptions and assertion violations. However, it can be extended via listeners and/or domain-specific extensions, which replace part of the JPF core, in order to check more complex and higher-level properties. The currently available domain-specific extensions include model checking of UML 2.0 state charts, symbolic execution (suitable for inputs with unbounded domains), and compositional verification.

Bandera toolset

Bandera [25] is a toolset for model checking of general Java programs, which accepts source code of a complete Java program and a set of properties expressed in temporal logic as an input. The toolset includes the core model checker Bogor [27] and several auxiliary tools that prepare input for Bogor and present its output in a user-friendly way. More specifically, since the input language of Bogor is BIR (Bandera Intermediate Representation), the auxiliary tools perform the following tasks: (i) translation of Java source code (input of Bandera) into a model in BIR, (ii) various abstractions and transformations of the model (e.g. slicing [33]), and (iii) translation of counter-examples in BIR back to Java.

Bogor is a general purpose explicit-state model checker, which verifies models defined in the BIR language against properties defined via temporal logic — primitive propositions in temporal logic expressions can refer to heap structure, data values and source code locations. The current version of BIR supports all features common to modern object-oriented programming languages (e.g. Java) like dynamic creation of objects and threads, virtual methods and garbage collection; in other words, it is possible to express all features of Java in BIR (there is no significant semantic gap between Java and BIR). Like JPF, Bogor too provides an API for non-deterministic data choice.

A key benefit of Bogor is its extensibility, since it allows to construct domain-specific model checkers on top of it. Specifically, Bogor supports custom strategies and heuristics for state space traversal, custom state storage and representation, and several property specification languages like regular expressions and CTL. The BIR modeling language is also extensible by constructs and primitives specific to

particular domains. Besides that, Bogor supports common state space reduction techniques, like partial order reduction and heap/thread symmetry. An example of a domain-specific extension to Bogor is the Cadena design and verification environment [34], which aims at component systems based on the CORBA component model (CCM) [47].

Comparison of Java PathFinder with Bandera

From the point of view of our goals, JPF and Bandera have basically the same capabilities, advantages and drawbacks. Both support all the important features of Java (JPF directly via special JVM, Bandera via translation to BIR), are highly extensible and configurable, and provide APIs that allow easy integration into larger development and verification frameworks (like Cadena [34] in case of Bandera). On the other hand, none of them

- supports checking of Java code against high-level properties based on event traces by default — an extension has to be used and/or created;
- is applicable to isolated software components directly, since both of them accept only complete Java programs with `main` (the problem of missing environment occurs); however, Bandera already provides a generator of an artificial environment for Java classes [59].

Regarding extensibility, an advantage of JPF over Bandera is the support for listeners that allow to monitor and intercept the state space traversal.

2.2 Compositional verification

The key idea behind compositional verification ([22], [13]) is application of the divide-and-conquer approach. The big task of verifying a software system at once is decomposed into smaller tasks of verifying the individual components one at a time, and the verification results for the whole system (i.e. related to global properties) are derived from the results for individual components (local properties) and interaction among the components. Such a derivation can be performed, e.g., via model checking of abstractions (models) of all the components and the interaction among them. Note also that the problem of missing environment is inherent to compositional model checking, since each component is checked in isolation.

Although a "naive" approach to compositional model checking, as described above, may help address state explosion at least partially, it does not help much in this respect if the level of interaction among components of a given system is high, and, moreover, the local properties of individual components do not have to be preserved at the global level [22]. Both these issues are addressed by the assume-guarantee paradigm, which is a popular approach to compositional model checking.

Assume-guarantee paradigm

Using the *assume-guarantee* paradigm (A-G), model checking of a single component C can be used to verify whether C satisfies a given property P when put into an

environment E that satisfies a specific assumption A (about E 's behavior). The property P can be associated with the behavior of the complete system (C and E together) or with C only — we focus on the case when P is a local property of C (of its implementation). The *assumption* A characterizes the expected behavior of all the environments for C , in which C is expected to work correctly (valid environments).

Actual checking of C 's behavior in a specific E is performed in two steps — first it is checked whether C satisfies P under the assumption A , and then it is checked whether A characterizes the E 's behavior correctly. The property P is *guaranteed* to hold for C in E only if both checks finish with a positive answer. The checking process can be formally expressed by the following inference rule (well-known as the *A-G rule*):

$$\frac{\begin{array}{c} \langle A \rangle C \langle P \rangle \\ \langle true \rangle E \langle A \rangle \end{array}}{\langle true \rangle C \parallel E \langle P \rangle}$$

The rationale behind use of an assumption A is that the behavior of C typically depends on the behavior of its environment E . Note, however, that application of the A-G paradigm makes sense only if the assumption A is simpler than the specific E it models, i.e. if A abstracts the real behavior of E , for example, via hiding internal communication between the parts of E .

The assume-guarantee paradigm was originally introduced for temporal logic model checking (e.g. in [53]) and the assumptions were defined manually in most cases. However, there are recent techniques that support checking against properties expressed via transitions systems (e.g. via LTS [30] and STS [56]) and automated construction of assumptions via learning [23]. In general, the A-G paradigm can be used for compositional verification of any software system that consists of well-defined parts (e.g. components or processes); specifically, an assumption about environment's behavior (*environment assumption*) can be defined in any suitable formalism (including behavior protocols), and satisfaction of any property supported by the model checker can be verified.

2.3 Behavior Protocols

The formalism of behavior protocols (BP) [52], developed in our research group, is a specific process algebra that we use for modeling of component behavior. Currently, it is supported by the SOFA and Fractal component platforms.

In general, a behavior protocol $prot$ is an expression that specifies a set $L(prot)$ of finite traces of atomic events on components' provided and required interfaces. The events in a protocol directly correspond to implementation-level events like invocation of a specific method on a specific interface — specifically, each atomic event has the syntactical form $\langle prefix \rangle \langle interface\ name \rangle . \langle method\ name \rangle \langle suffix \rangle$, where the prefix can be either $?$ (acceptance) or $!$ (emitting), and the suffix can be either \uparrow (method invocation) or \downarrow (return from a method). Therefore, four types of atomic events are supported:

<i>acceptance of a method invocation:</i>	<code>?interface.method↑</code> ,
<i>emit of a method invocation:</i>	<code>!interface.method↑</code> ,
<i>acceptance of a return from a method:</i>	<code>?interface.method↓</code> ,
<i>emit of a return from method:</i>	<code>!interface.method↓</code> .

More complex behavior protocols can be constructed from the atomic events via the following binary operators: `;` (sequence), `+` (choice), `*` (repetition), and `|` (parallel composition). The parallel composition operator generates all interleavings of event traces defined by its operands such that no synchronization is assumed. The empty protocol is denoted by `NULL`.

The formalism of behavior protocols also supports several useful shortcuts that enhance readability:

- `!i.m{prot}` stands for `!i.m↑ ; prot ; ?i.m↓`, and
- `?i.m{prot}` stands for `?i.m↑ ; prot ; !i.m↓`.

The protocol *prot* models a method body, which can be empty.

For the purpose of modeling the behavior of SOFA and Fractal components with behavior protocols, we distinguish between component’s frame and architecture. The *frame* of a component is formed by all the component’s external provided and required interfaces. The *architecture* of a given component is represented either by the component’s internal structure (a set of sub-components and bindings among their interfaces at the first level of nesting) in case of composite components, or by the implementation in a programming language in case of primitive components. We say that an architecture implements a frame.

Then, the component’s *frame protocol* specifies the valid sequences and interleavings of atomic events (method invocations and returns) on the component’s frame — in other words, it specifies how the component can be used by its environment and how it reacts to requests from the environment. The *architecture protocol* of a specific composite component models the composed behavior of its sub-components.

The main advantage of the formalism of behavior protocols is the built-in support for checking of behavior compliance among components equipped with frame protocols. For that purpose behavior protocols provide the *consent operator* (∇) [2], which is a special kind of parallel composition; specifically, application of the consent operator on two protocols

- generates all interleavings of event traces defined by its operands (like `|`),
- forces complementary events (e.g. `?i.m↑` and `!i.m↑`) to synchronize, i.e. to form an internal action (e.g. `τi.m↑`), and, in particular,
- identifies specific communication errors: deadlock (“no activity”) and no response to a call (“bad activity”).

Two behavior protocols are compliant if their composition via consent does not yield any communication errors. We distinguish between two kinds of behavior compliance: (1) *horizontal compliance* among components at the same level of nesting (e.g.

among all sub-components of a specific composite component), and (2) *vertical compliance* between a frame protocol and an architecture protocol of a specific composite component. Satisfaction of vertical compliance for a particular component means that the component's architecture implements its frame correctly.

Both kinds of behavior compliance are supported by the Behavior Protocol Checker (BPChecker) [42], which implements the consent operator. Technically, the search for communication errors in composition of two protocols is performed via exhaustive state space traversal, where each transition corresponds to an atomic event from one of the protocols. Checking of compliance between three or more protocols is implemented via subsequent application of the consent to pairs of protocols (the ∇ operator is associative).

Nevertheless, as indicated in Sect. 1.3, checking of behavior compliance among components in a particular hierarchy makes sense only if each primitive component in the hierarchy obeys its behavior specification. In the context of behavior protocols, we say that a primitive component *obeys its frame protocol*. Formal definition of this property (obeying of a frame protocol) for primitive components implemented in Java is in Sect. 4.3.

Chapter 3

Goals revisited

In the light of the facts mentioned in the previous chapter, we decided to address the general goal of the thesis (Sect. 1.5) on the basis of the assume-guarantee paradigm and Java PathFinder. The assume-guarantee paradigm provides a coherent theoretical framework for verification and analysis of behavior of isolated components. Java PathFinder (JPF) is a state-of-the-art verification tool (model checker) for Java bytecode programs.

We have chosen JPF over the Bandera toolset, since at the time we made the decision only the alpha release (not fully stable) of Bandera was available and JPF was more extensible and customizable — both JPF and Bandera did not support properties based on sequences of events (e.g. obeying of a frame protocol) at that time. The other JPF-related issues, which have to be solved, are that (i) it works only for complete Java programs with `main` (there is the problem of missing environment) and (ii) it is prone to state explosion when checking complex Java programs.

The detailed goals of the thesis are:

- G1) To sufficiently solve modeling and construction of an artificial environment for verification of component's Java code with JPF.
- G2) To extend JPF with support for checking component's Java code against the high-level property of obeying a frame protocol.
- G3) To address state explosion in model checking of component's Java code with JPF against the properties of obeying a frame protocol and absence of concurrency errors.

Chapter 4

Contribution: verification of Java code of primitive components

In this chapter we provide an overview of our contribution and, in particular, show how we addressed the goals G1-G3 from Chapter 3. Technical details and additional information on any part of our contribution can be found in the already published papers (forming also the Chapters 5-9 of this thesis).

Throughout the whole chapter, we illustrate the key ideas and concepts of our contribution on a part of the component application developed in the CRE project with France Telecom [1]. The whole application works as a provider of WiFi Internet access at airports, supporting e.g. payment via a credit card and assignment of IP addresses via DHCP. However, we focus only on the `DhcpServer` component (Fig. 4.1), which is responsible for management of IP addresses — in particular, it assigns IP addresses to newly connected clients via DHCP, tracks expiration of DHCP leases, and stores assigned IP addresses in a database.

More specifically, `DhcpServer` consists of four components: `IpAddressManager`, `Timer`, `DhcpListener` and `TransientIpDb`. The `DhcpListener` component interacts with clients via the DHCP protocol, `TransientIpDb` is a database of temporally assigned IP addresses, and `Timer` notifies `IpAddressManager` on expiration of DHCP leases. The `IpAddressManager` component is responsible for assigning proper IP addresses to clients — it does that on the basis of clients' MAC addresses provided via DHCP and information about mapping between MAC and IP addresses that is stored in the database (`TransientIpDb`).

The frame protocols of `IpAddressManager` and `TransientIpDb`, modified and simplified with respect to [1] for the purpose of illustrating our contribution, are depicted on Fig. 4.2 and Fig. 4.3, respectively. The simplified frame protocol of `IpAddressManager` captures only calls on provided methods of the component and interaction with `TransientIpDb`. The frame protocol of `TransientIpDb` states that (i) it is possible to call any method M of the component in parallel with any other method or, in particular, with another instance of the same method, and (ii) each method can be called for a finite number of times.

We use the `IpAddressManager` component in Sect. 4.3 for illustration of checking whether component's Java implementation obeys the frame protocol with JPF, and the `TransientIpDb` component for illustration of addressing state explosion

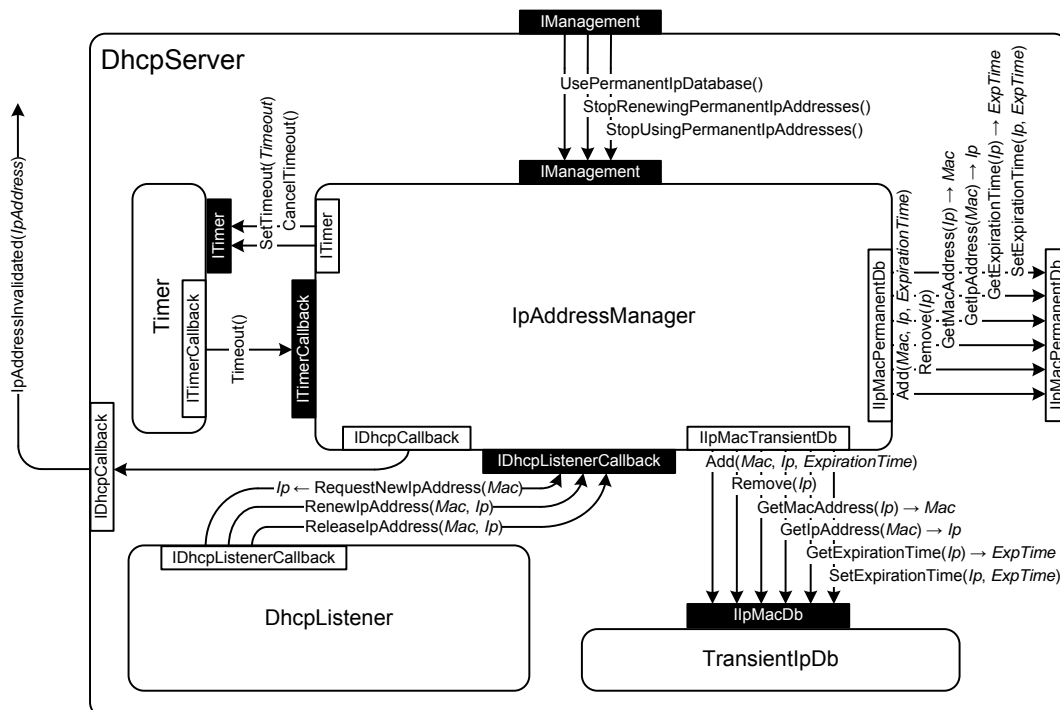


Figure 4.1: The DhcpServer component

in discovery of concurrency errors in Java code with JPF (in Sect. 4.4). A fragment of the Java code of `IpAddressManager` is depicted on Fig. 4.4 (only the `RequestNewIpAddress` method is included) and a fragment of the Java code of `TransientIpDb` is depicted on Fig. 4.5 (only business methods are included).

4.1 The whole picture — application of the assume-guarantee paradigm

Our approach to formal verification of Java implementation of isolated primitive components against various properties is based on the assume-guarantee paradigm (A-G) and Java PathFinder.

As an assumption we use the behavior model of a particular environment of a component subject to verification. The environment can be general (universal), specific to a particular context (Sect. 4.2), or even specific to a particular property (Sect. 4.4). The model of environment’s behavior — the environment assumption — is defined in the formalism of behavior protocols and reflected in the artificial environment (see below).

A ”guarantee” is a specific property of the Java code of the component (and its environment, in some cases), which is satisfied under the given assumption, i.e. in the environment modeled by the assumption. Although it is, in general, possible to use any property supported by JPF, we explicitly focus on the following two

```

(
  (
    (
      ?IDhcpListenerCallback.RequestNewIpAddress {
        !IIpMacTransientDb.GetIpAddress ;
        (!IIpMacTransientDb.Add + NULL)
      }
      +
      ?IDhcpListenerCallback.RenewIpAddress {
        !IIpMacTransientDb.GetIpAddress ;
        (!IIpMacTransientDb.SetExpirationTime + NULL)
      }
      +
      ?IDhcpListenerCallback.ReleaseIpAddress {
        !IIpMacTransientDb.GetIpAddress ;
        (!IIpMacTransientDb.Remove + NULL)
      }
    )*
    |
    ?ITimerCallback.Timeout {
      (
        !IIpMacTransientDb.GetExpirationTime ;
        (!IIpMacTransientDb.Remove + NULL)
      )*
    }*
  )
  ; ?IManagement.UsePermanentIpDatabase ;
  (
    # same as above
  )
  ; ?IManagement.StopUsingPermanentIpDatabase
)*

```

Figure 4.2: Frame protocol of the IpAddressManager component

```
?IIpMacTransientDb.Add*
|
?IIpMacTransientDb.Add*
|
?IIpMacTransientDb.Remove*
|
?IIpMacTransientDb.Remove*
|
?IIpMacTransientDb.GetMacAddress*
|
?IIpMacTransientDb.GetMacAddress*
|
?IIpMacTransientDb.GetIpAddress*
|
?IIpMacTransientDb.GetIpAddress*
|
?IIpMacTransientDb.GetExpirationTime*
|
?IIpMacTransientDb.GetExpirationTime*
|
?IIpMacTransientDb.SetExpirationTime*
|
?IIpMacTransientDb.SetExpirationTime*
```

Figure 4.3: Frame protocol of the TransientIpDb component


```

public class IpAddressManagerImpl
    implements IDhcpListenerCallback {

    protected IIpMacDb iIpMacTransientDb;

    public String RequestNewIpAddress(byte[] MacAddress) {
        String ipAddr = iIpMacTransientDb.GetIpAddress(MacAddress);

        Date expTime = new Date(System.currentTimeMillis()+3600000);

        iIpMacTransientDb.Add(MacAddress, ipAddr, expTime);

        return ipAddr;
    }

    ...
}

```

Figure 4.4: Fragment of Java implementation of the `IpAddressManager` component

properties of component’s Java code: (i) *obeying of a frame protocol* and (ii) *absence of concurrency errors*.

The process of checking of a primitive component against a specific property with JPF consists of the following four steps:

- 1) automated construction of the behavior model of a particular environment for the component subject to verification,
- 2) manual specification of possible values of method parameters,
- 3) automated generation of the artificial environment (Java code) from the behavior model and specification of values, and
- 4) verification of Java code of the complete program composed of the component and artificial environment against the given property with JPF.

The artificial environment is generated from the behavior model and specification of method parameter values in an automated way by the tool — Environment Generator for Java Pathfinder [49] — that we developed. The generated environment for a specific component (Java code) contains a driver class with the `main` method, which calls methods of the component’s provided interfaces according to the model of environment’s behavior, and stub implementations of all required interfaces of the component. For example, the environment for the `Timer` component contains `main`, which calls methods on the `ITimer` provided interface, and a stub implementation of the `ITimerCallback` required interface. Specification of possible values of method parameters is provided by the user in the form of a Java class that works as a container for the values.

```

public class TransientIpDbImpl implements IIpMacDb {

    protected Map ipMac = new HashMap();
    protected Map macIp = new HashMap();
    protected Map expTimes = new HashMap();

    public void Add(byte[] MacAddr, String IpAddr, Date ExpTime) {
        ipMac.put(IpAddr, MacAddr);
        macIp.put(MacAddr, IpAddr);
        expTimes.put(IpAddr, ExpTime);
    }

    public Date GetExpirationTime(String IpAddr) {
        Date expTime = (Date) expTimes.get(IpAddr);
        return expTime;
    }

    public String GetIpAddress(byte[] MacAddr) {
        String ip = (String) macIp.get(MacAddr);
        return ip;
    }

    public byte[] GetMacAddress(String IpAddr) {
        byte[] mac = (byte[]) ipMac.get(IpAddr);
        return mac;
    }

    public void Remove(String IpAddr) {
        byte[] mac = (byte[]) ipMac.get(IpAddr);

        if (mac != null) {
            ipMac.remove(IpAddr);
            macIp.remove(mac);
            expTimes.remove(IpAddr);
        }
    }

    public void SetExpirationTime(String IpAddr, Date ExpTime) {
        if (expTimes.containsKey(IpAddr)) {
            expTimes.put(IpAddr, ExpTime);
        }
    }
}

```

Figure 4.5: Java implementation of the TransientIpDb component

4.2 Modeling component environment via behavior protocols

As indicated in Sect. 1.1, it is in general hard to model and construct a reasonable artificial environment for a given component for the purpose of feasible model checking. Our solution to this problem is based on modeling the environment's behavior via behavior protocols — behavior model of an environment (artificial or real) for a specific isolated component C is denoted as an *environment protocol* of C .

Since an environment of C can be seen as another component E bound to C , the environment protocol of C is actually a frame protocol of E . A specific behavior protocol can be used as an environment protocol EP_C of C , if it is compliant with the frame protocol FP_C of C — i.e. the formula $EP_C \nabla FP_C$ has to hold.

We propose three different particular environment protocols, each having its own specific advantages and drawbacks with respect to the others — an inverted frame protocol, a context protocol, and an environment protocol based on a calling & trigger protocol.

An *inverted frame protocol* EP_C^{inv} of a component C models an environment that exercises C in all ways it was designed for — the environment performs all sequences and interleavings of method call-related events that are allowed by the C 's frame protocol. The inverted frame protocol of C is derived directly from C 's frame protocol via syntactical replacement of all $!$ with $?$ (and vice versa) in the protocol, however checking with JPF is prone to state explosion if the environment constructed from the inverted frame protocol is used.

A *context protocol* EP_C^{ctx} of C models the actual use of C in a particular component application, i.e. it is specific to a particular context. Technically, EP_C^{ctx} corresponds to composition of frame protocols of all the other components in the particular application. Since only a subset of component's functionality is often used by the application, the context protocol is often simpler than the inverted frame protocol and therefore checking with JPF is less prone to state explosion if the environment constructed from the context protocol is used. For example, the context protocol of **TransientIpDb** (Fig. 4.6) involves less parallelism than its inverted frame protocol. On the other hand, construction of the context protocol is prone to state explosion, since it involves state space traversal — BPChecker is used to create the composition of frame protocols.

The issues of the inverted frame protocol and context protocol (liability to state explosion) are addressed by use of an environment protocol that is based on the *calling & trigger protocol* of C (EP_C^{trig}). It models the actual use of C in a particular application (like the context protocol) and is derived syntactically from the frame protocols of all the other components in the application — thus, both the construction and checking with JPF are less (or not at all) prone to the state explosion. However, EP_C^{trig} is a valid model of environment's behavior only if the frame protocols of all the other components in the application (i.e. their behavior) satisfy certain constraints (see Chapter 9 for details). If the constraints are not satisfied, then EP_C^{ctx} or EP_C^{inv} has to be used.

Note that for any of the three specific environment protocols introduced above, it is not necessary to verify whether the clause $\langle true \rangle E \langle A \rangle$ in the A-G

```

(
  !IIpMacTransientDb.GetIpAddress ;
  (
    !IIpMacTransientDb.Add
    +
    !IIpMacTransientDb.SetExpirationTime
    +
    !IIpMacTransientDb.Remove
    +
    NULL
  )
)*
|
(
  !IIpMacTransientDb.GetExpirationTime ;
  (!IIpMacTransientDb.Remove + NULL)
)*

```

Figure 4.6: Context protocol of the `TransientIpDb` component

rule (Sect. 2.2) holds, since an artificial environment E for C is directly generated from the environment protocol (assumption A). In other words, the clause $\langle true \rangle E \langle A \rangle$ holds implicitly for all the three cases.

4.3 Obeying a frame protocol

We say that a primitive component implemented in Java obeys its frame protocol, if (i) it is able to accept all sequences and interleavings of method calls on its provided interfaces that are specified by the frame protocol (e.g. without throwing of an exception), and (ii) it reacts correctly to each call on a provided interface, i.e. performs valid sequences and interleavings of calls of methods on its required interfaces. Each execution path in the Java program composed of a component and its environment has to be compliant with the component's frame protocol — the set of execution paths is determined by the environment and thus the environment should use the component correctly with respect to its frame protocol.

Put formally, a component C obeys its frame protocol FP_C iff the formula $L(\text{Java}_C/\text{calls}) \subseteq L(FP_C)$ holds for any valid environment E for C (such that $FP_E \nabla FP_C$), where $\text{Java}_C/\text{calls}$ denotes all sequences of invocations and returns of/from methods of C 's provided and required interfaces that occur in the Java program composed of C and E . The property of obeying a frame protocol is, in general, a specific relation between Java code and a behavior protocol.

For illustration, the fragment of Java code of `IpAddressManager` depicted on Fig. 4.4 obeys the fragment of the component's frame protocol (Fig. 4.7), while the

fragment of the Java code depicted on Fig. 4.8 violates the frame protocol — the incorrect code contains also the call of the `SetExpirationTime` method.

```
...
?IDhcpListenerCallback.RequestNewIpAddress {
    !IIpMacTransientDb.GetIpAddress ;
    (!IIpMacTransientDb.Add + NULL)
}
...
```

Figure 4.7: Fragment of the frame protocol of the `IpAddressManager` component

```
...
public String RequestNewIpAddress(byte[] MacAddress) {
    String ipAddr = iIpMacTransientDb.GetIpAddress(MacAddress);

    Date expTime = new Date(System.currentTimeMillis()+3600000);

    iIpMacTransientDb.Add(MacAddress, ipAddr, expTime);

    Date expTime2 = new Date(System.currentTimeMillis()+2400000);
    iIpMacTransientDb.SetExpirationTime(ipAddr, expTime2);

    return ipAddr;
}
...
```

Figure 4.8: Fragment of Java implementation of the `IpAddressManager` component that *violates* the frame protocol

In order to allow checking of the (high-level) property of obeying a frame protocol with JPF, it is necessary to extend and/or customize it, since it supports only low-level properties like deadlocks and assertion violations by default. Our approach is based on combination of JPF and `BPChecker` in such a way that both checkers cooperate during traversal of their own state spaces. More specifically, we have created a listener for JPF that monitors the traversal of the state space of the given Java program (component + environment) in JPF and notifies the `BPChecker` of all method call-related events on the provided and required interfaces of the given component. When the `BPChecker` receives an event from the JPF listener, it checks whether the event is allowed in the current state of its state space (determined by the frame protocol) — in case of a negative answer, it tells JPF to stop the state space traversal and report an error trace.

4.4 Addressing state explosion in discovery of concurrency errors

Well-established approaches to addressing the state explosion in discovery of specific kinds of errors via model checking include use of heuristics for state space traversal [32] and guiding of a model checker by results of static analysis or runtime analysis [35]. For concurrency errors it means, for example, use of a heuristic that prefers state space paths involving aggressive thread interleaving and guiding of the model checker by a counter-example (reported by static or runtime analysis) that identifies a potential deadlock. A common characteristic of these approaches is that they can be applied only during actual model checking (of a complete program) — however, an isolated component is not such a program.

We propose an alternative and complementary approach — construction of a *reasonable environment* for an isolated component on the basis of static analysis of component’s code and various heuristics. The key idea behind the approach is that the reasonable environment for a specific component will be simpler than the component’s *full environment* (corresponding to a particular environment protocol) e.g. in terms of the maximal number of threads running in parallel (*level of parallelism*), and therefore its use will mitigate the state explosion and also help discover the specific errors in limited time and memory.

Since here we aim at the discovery of concurrency errors in component Java code with JPF, we construct the reasonable environment in such a way that only those component’s methods that feature potential concurrency errors are executed in parallel by the environment. To be more specific, we reduce the level of parallelism in an environment protocol, yielding a *reduced environment protocol*, which is then used a model of behavior of a *reduced environment*. The methods to be executed in parallel by the reduced environment (i.e. methods that feature potential concurrency errors) are identified via static analysis that searches for suspicious bytecode patterns (e.g. unsynchronized access to a shared variable) in pairs of Java methods and a heuristic that assigns weights to the patterns (Chapter 8). When the reduced environment is constructed, JPF is applied to the complete Java program composed of a component and reduced environment in order to check parallel execution of the identified methods for presence or absence of real concurrency errors.

For illustration, consider the `TransientIpDb` component (Fig. 4.5), whose methods involve unsynchronized read and write accesses to shared variables. Reduction of the particular environment protocol of `TransientIpDb` (Fig. 4.9) on the basis of search for suspicious bytecode patterns would produce the reduced environment protocol depicted on Fig. 4.10, which specifies parallel execution of selected pairs of methods. E.g. parallel execution of the methods `Add` and `GetExpirationTime` is specified, since both of them access the `expTimes` variable.

4.5 Contribution reflected in publications

Our contribution is reflected in the included papers in the following way:

```

!IIpMacTransientDb.Add*
|
!IIpMacTransientDb.Remove*
|
!IIpMacTransientDb.GetMacAddress*
|
!IIpMacTransientDb.GetIpAddress*
|
!IIpMacTransientDb.GetExpirationTime*
|
!IIpMacTransientDb.SetExpirationTime*

```

Figure 4.9: Environment protocol of the `TransientIpDb` component

```

(
  !IIpMacTransientDb.Add*
  |
  !IIpMacTransientDb.GetExpirationTime*
)
+
(
  !IIpMacTransientDb.Add*
  |
  !IIpMacTransientDb.Remove*
)
+
...

```

Figure 4.10: Reduced environment protocol of the `TransientIpDb` component — search for suspicious patterns

Chapter 5 [PPK07] presents our approach to checking Java code of an isolated primitive component against the property of obeying a frame protocol with JPF; in particular, technical details on cooperation between JPF and BPChecker and mapping between their state spaces are provided there.

Chapter 6 [PP07a] describes our approach to the generation of an artificial environment (Java code) for a given primitive component and introduces the use of an inverted frame protocol as a model of the environment's behavior.

Chapter 7 [PP07b] introduces the concept of a context protocol and also provides a syntactical algorithm for its derivation.

Chapter 8 [PP07c] presents our approach to addressing state explosion in discovery of concurrency errors via reduction of the level of parallelism in a component environment on the basis of search for suspicious Java bytecode patterns (corresponding to potential concurrency errors). In particular, it provides details on the supported patterns and an evaluation of the approach on several real-life components implemented in Java.

Chapter 9 [PP08] introduces the calling & trigger protocol and syntactical constraints on components' frame protocols that have to be satisfied in order to make the calling & trigger protocol a valid model of environment's behavior. Moreover, it also provides the formal definition of an environment protocol on the basis of the consent (∇) operator and an evaluation of all particular environment protocols that we use — inverted frame protocol, context protocol, and calling & trigger protocol.

[**PPK07**] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker, In Proceedings of 30th IEEE/NASA Software Engineering Workshop, published by IEEE Computer Society, ISBN 0-7695-2624-1, ISSN 1550-6215, pp. 133-141, January 2007.

[**PP07a**] P. Parizek and F. Plasil. Specification and Generation of Environment for Model Checking of Software Components, In Proceedings of Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2006), ENTCS, volume 176, issue 2, published by Elsevier B.V., ISSN 1571-0661, pp. 143-154, May 2007.

[**PP07b**] P. Parizek and F. Plasil. Modeling Environment for Component Model Checking from Hierarchical Architecture, In Proceedings of Formal Aspects of Component Software (FACS'06), ENTCS, volume 182, published by Elsevier B.V., ISSN 1571-0661, pp. 139-153, June 2007.

[**PP07c**] P. Parizek and F. Plasil. Partial Verification of Software Components: Heuristics for Environment Construction, In Proceedings of 33rd EUROMICRO SEAA conference, published by IEEE Computer Society, ISBN 0-7695-2977-1, ISSN 1089-6503, pp. 75-82, August 2007.

[**PP08**] P. Parizek and F. Plasil. Modeling of Component Environment in Presence of Callbacks and Autonomous Activities, Accepted for publication in proceedings of TOOLS EUROPE 2008, LNBIP, to be published by Springer-Verlag, June 2008.

Chapter 5

Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker

Pavel Parízek¹,
František Plášil,
Jan Kofroň¹

Contributed paper at **30th IEEE/NASA Software Engineering
Workshop (SEW-30)**.

In conference proceedings,
published by IEEE CS,
pages 133–141,
ISBN 0-7695-2624-1,
ISSN 1550-6215,
January 2007.

The original version is available electronically from the publisher's site
at <http://doi.ieeecomputersociety.org/10.1109/SEW.2006.23>.

¹Regarding the relative contribution of me and my colleague Jan Kofroň, I have designed and implemented the plugin for JPF that manages the cooperation between JPF and BPChecker, while Jan Kofroň performed the experiments and implemented a minor extension of BPChecker that was necessary to make the cooperation work.

Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker*

Pavel Parizek, Frantisek Plasil, Jan Kofron

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

*Academy of Sciences of the Czech Republic
Institute of Computer Science
{plasil,kofron}@cs.cas.cz
<http://www.cs.cas.cz>*

Abstract

Although there exist several software model checkers that check the code against properties specified e.g. via a temporal logic and assertions, or just verifying low-level properties (like unhandled exceptions), none of them supports checking of software components against a high-level behavior specification. We present our approach to model checking of software components implemented in Java against a high-level specification of their behavior defined via behavior protocols [1], which employs the Java PathFinder model checker and the protocol checker. The property checked by the Java PathFinder (JPF) tool (correctness of particular method call sequences) is validated via its cooperation with the protocol checker. We show that just the publisher/listener pattern claimed to be the key flexibility support of JPF (even though proved very useful for our purpose) was not enough to achieve this kind of checking.

Keywords: software components, behavior protocols, model checking, cooperation of model checkers

1. Introduction

Model checking is one of the approaches to formal verification of finite state hardware and software systems. A model checker usually accepts a finite model of a target system and a property expressed in some property specification language, and checks whether the model satisfies the property via traversal of the state space that is generated from the model. Especially model checking of

software is a popular research topic nowadays, mainly because there are several issues that have to be solved before the technique can be used for real-life applications.

A general problem of model checking is the necessity to create a model of the system to be checked. Manual construction of the model is an error-prone process, and even if the model is automatically extracted from a specification of the system or from the source code, it is typically an abstraction of the system - therefore, a model checker may find errors in the model that are not present in the original program and vice versa.

In case of properties to be checked, the most common way to express them is via a temporal logic (LTL, CTL) and in the form of assertions. However, it is also possible to check for a predefined set of properties - deadlocks or properties specific to a certain class of systems such as device drivers.

As to software model checking at the program source code level, a crucial problem is the size of state space triggered by the model of a program (i.e. the problem of state explosion). Despite that, there exist such model checkers. For Java programs, these are most notably the Java PathFinder (JPF) [5] and Bandera [7] tools. (An advantage of JPF over Bandera is that the most recent release of the latter is an alpha version, not being fully stable yet, and that JPF is also more extensible). The properties checked are either predefined (e.g. absence of a deadlock) or to be specified in LTL (Bandera) and via assertions related to the code (JPF). A typical feature of both Bandera and JPF is the combination of static program analysis and model checking.

*The work was partially supported by the Grant Agency of the Czech Republic (project number 201/06/0770) and France Telecom under the external research contract number 46127110.

The former is used to create a program model; to lower the state space size, abstraction techniques are applied - these include partial order reduction [13] and data abstraction [13].

State explosion can be also mitigated by the decomposition of a software system into small and well-defined units, components. Typically, a software component generates a smaller state space than the whole system and therefore can be checked with fewer requirements on space and time. Nevertheless, model checking of code of software components usually brings along the problem of missing environment, which means that it is not possible to model check an isolated component, because it does not form a complete program with an explicit starting point (e.g. the `main` method). In order to overcome this obstacle, it is necessary to create a model of the environment of the component subject to model checking, including the specification of possible values of method parameters, and then check the whole program, composed of the environment and component.

A specific feature of software components is the existence of ADLs (Architecture Description Languages) used to specify component interfaces, and, first of all, composition of components via bindings of their interfaces (i.e. to specify the architecture of a component-based application at a higher level of abstraction than code). Some ADLs even include the option to specify behavior of the components, typically in a LTS-based formalism [15, 18, 16, 17].

An obvious challenge, not addressed yet to our knowledge, is to check the code of software components against a high-level behavior specification provided at the ADL component specification level.

1.1. Goal and structure of the paper

The goal of the paper is to show how the challenge mentioned above can be addressed for software components implemented in the Java language and a high-level specification of their behavior defined via behavior protocols [1] employed in ADL. We present our approach that integrates the Java PathFinder model checker with the behavior protocol checker [4].

The remainder of the paper is organized as follows. Sect. 2 introduces an example of a component ADL specification, Sect. 3 provides an overview of behavior protocols and Sect. 4 introduces the Java PathFinder model checker. Sect. 5 presents the key contribution - the description of our solution for model checking of primitive (non-composed) software components' code against behavior protocols that makes JPF cooperate with the protocol checker. Sect. 6 provides results of evaluation and the rest of the paper contains related work and conclusion.

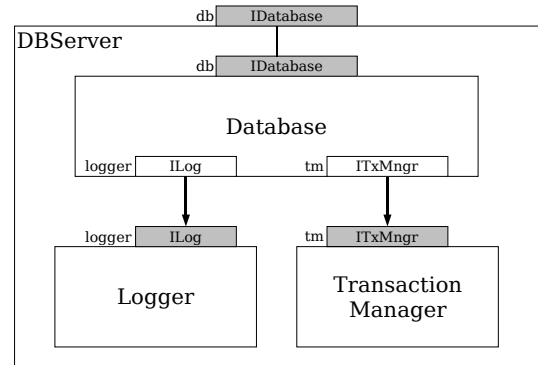


Figure 1: Architecture of the DBServer component

2. Example

In this section we provide an example, which will be used to illustrate the ideas presented throughout the rest of the paper. Consider the component architecture in Fig.1. Here the component `DBServer` provides the `IDatabase` interface and contains three primitive subcomponents - `Database`, `Logger` and `Transaction Manager`. The `IDatabase` interface is implemented by the delegation to the `Database` subcomponent. The other two subcomponents of the `DBServer` component are bound to the required interfaces of the `Database` subcomponent.

Fragments of an ADL specification for the `DBServer` and `Database` components may take the following form:

```
frame DBServer {
  provides:
    IDatabase db;
  protocol:
    ?db.start ; (?db.add || ?db.get ||
?db.remove)* ; ?db.stop
};

frame Database {
  provides:
    IDatabase db;
  requires:
    ILog logger;
    ITxMngr tm;
  protocol:
    // presented in Sect. 3
};
```

These fragments specify the *frame* (boundary, a collection of interface instances) of the components `DBServer` and `Database`. For instance, the specification states that `Database` has two required interfaces (`logger` of the type `ILog` and `tm` of the type `ITxMngr`); in a similar vein, `db` of the type `IDatabase` is its provided interface. The protocol section of each of the frames contains the behavior specification (in the form of behavior protocols explained in Sect. 3) of the respective component.

Fragments of Java source code of all interfaces and implementation of the Database component follow:

```
public interface IDatabase
{
    public void start();
    public void stop();
    public void add(String key, Object data);
    public Object get(String key);
    public void remove(String key);
}

public interface ILog
{
    public void log(String message);
}

public interface ITxMngr
{
    public void init();
    public void destroy();
    public void begin();
    public void commit();
    public void rollback();
}

public class DatabaseImpl implements IDatabase
{
    private ILog logger;
    private ITxMngr tm;

    public void start()
    {
        logger.log("start");
        tm.init();
    }

    public void stop()
    {
        logger.log("stop");
        tm.destroy();
    }

    public void add(String key, Object data)
    {
        tm.begin();
        ... // adding data
        if (ok) tm.commit();
        else tm.rollback();
    }

    public Object get(String key)
    {
        // similar to the add method
    }

    public void remove(String key)
    {
        // similar to the add method
    }
}
```

3. Behavior Protocols

3.1. Basics

A behavior protocol is an expression that describes the behavior of a software component in terms of atomic events on the provided and required interfaces of a component, i.e. in terms of accepted and emitted method call requests and responses on those interfaces. The semantics of a behavior protocol is defined in terms of Labeled Transition System (LTS), where transitions are labeled by atomic events.

Each atomic event in a behavior protocol has the following syntax: <prefix> <interface>.<method> <suffix>. The prefix ? denotes an accept event and the prefix ! denotes an emit event. The suffix † stands for a request (i.e. a method call) and the suffix ‡ stands for a response (i.e. return from a method).

Several useful shortcuts are defined: an expression of the form !i.m is a shortcut for the protocol !i.m† ; ?i.m‡, an expression of the form ?i.m is a shortcut for the protocol ?i.m† ; !i.m‡ and an expression of the form ?i.m{prot} is a shortcut for the protocol ?i.m† ; prot ; !i.m‡. The NULL keyword denotes an empty protocol.

The protocol section of the ADL example in Sect. 2 illustrates how most of the operators of behavior protocols are applied. It includes the sequence operator ;, the repetition operator *, the alternative operator +, and the or-parallel operator ||. There is also an and-parallel operator |, yielding all the possible interleavings of the event traces defined by its operands. The or-parallel operator is a shortcut (p || q stands for p + q + (p | q), where p and q are behavior protocols).

A behavior protocol defines a possibly infinite set of traces, where each trace is a finite sequence of atomic events.

The following protocol specifies a part of the Database component's behavior.

```
?db.start† ; !logger.start† ; ?logger.start‡
; !tm.init† ; ?tm.init‡ ; !db.start†
```

It starts with accepting request for start call on db, then, as a reaction, issues the request for start call on logger and accepts the response, does the same for the init call on tm, and, finally, issues a response to start call on db.

For every component, we assume its *frame protocol* [1] is specified in ADL. The frame protocol describes the external behavior of a component, which means the protocol contains only the events on the external interfaces determined by the component's frame. For every composite component, its *architecture protocol* can be generated as a parallel composition of the frame protocols of the subcomponents at the first level of nesting [1].

The frame protocol of the Database component, with the syntactical shortcuts mentioned above applied, might be:

```

?db.start(!logger.start ; !tm.init) ;
(
  ?db.add(!tm.begin ; (!tm.commit +
!tm.rollback))
  ||
  ?db.get(!tm.begin ; (!tm.commit +
!tm.rollback))
  ||
  ?db.remove(!tm.begin ; (!tm.commit +
!tm.rollback))
)* ;
?db.stop(!logger.stop ; !tm.destroy)

```

The behavior specified by this protocol reflects the expected usage pattern of the component and also its reaction to each call accepted on its `db` interface. For example, it states that when the component accepts a request for `add` call on `db`, it should (in the following order)

- 1) call the `begin` method on `tm`,
- 2) call one of the `commit` and `rollback` methods on `tm`, and, finally,
- 3) issue a response to the `add` call on `db`.

In addition, the protocol states that calls of `add`, `get`, `remove` on `db` can be accepted in parallel and this can be repeated a finite number of times.

Important feature of behavior protocols is the notion of behavior compliance which allows to say whether two components, equipped with frame protocols, can communicate without errors or not. *Horizontal compliance* of components that are at the same level of nesting is evaluated via a mechanism similar to parallel composition of their frame protocols the results of which are not only the traces produced by the `|` operator, but also all erroneous traces reflecting communication errors (such as no activity and bad activity [2]). *Vertical compliance* between a frame and an underlying architecture is evaluated by being treated as horizontal compliance between the architecture's protocol and inverted frame protocol (constructed from the frame's protocol by replacing all accept events with emit events and vice versa)[3].

Obviously, the whole component-based system, in which the horizontal and vertical compliance is verified at all levels of component nesting, works fine under the assumption that the code of each primitive component really implements what was specified by its frame protocol. More precisely, on its frame interfaces the component has to accept/issue such method call-related event sequences that correspond to the traces specified by the frame protocol - it has to *obey* its frame protocol [1].

3.2. Protocol Checker

For the purpose of static checking of compliance between two protocols, we use the static protocol checker [4] developed in our research group. Taking two protocols as arguments, it creates a parse tree for each of these protocols and then produces a composite parse tree that determines the state space reflecting the parallel

composition of the two protocols. A transition in the state space represents execution of an atomic event. In each step of state space traversal, the checker acquires the list of possible transitions from the current state. In search for communication errors, it systematically, in the DFS manner, explores all branches in the state space that correspond to those transitions.

In addition, in our research group, we have also developed a runtime protocol checker to check whether a component obeys its frame protocol in a particular run. The tested component is equipped by interceptors at its frame's interfaces which notify the runtime checker on the method call related events. Not needing to traverse the whole state space (and employ backtracking), the run time checker just selects the transition that corresponds to an actually observed event; if there is no such available in the state space, it reports a violation of the frame protocol's obeying.

4. Java PathFinder

Java PathFinder (JPF) [5] is a modern software model checker for Java byte code. More specifically, it is a specialized Java Virtual Machine (JPF VM), which runs on top of the underlying host JVM, and, in contrast to the standard JVM, executes the program in all possible ways. The state space of a target program is a tree in principle, with branches determined by the threads' instructions interleaving and possible values of input data.

Like other model checkers for concurrent programs, JPF supports partial order reduction (POR) [13]. The purpose of this technique is to lower the state space size via including in the state space only one interleaving of instructions that are both independent and executed in different threads. The consequence is that JPF actually traverses a reduced state space where each state is associated with one of the following events ("points") in the byte code execution:

(a) *Scheduling point*. The current instruction is thread scheduling relevant (e.g. it accesses a shared variable, starts/stops a thread, blocks a thread, etc.)

(b) *Value point*. A value selection takes place (see below).

In order to enable checking of a code unit (e.g. a method) for different values of input data (e.g. method parameters), JPF contains the static class `Verify` that provides methods for a systematic selection of values of virtually any type. The methods of `Verify` are to be called in the checked code. For example, if the checked code unit executes `Verify.random(3)`, an integer value from the range 0..3 is selected. However, after reaching an end state, JPF backtracks (recursively) up to the `Verify.random(3)` call and selects another value from 0..3; this is repeated until all the values from this interval have been used for execution. Obviously, employing methods of `Verify` increases the state space size since each selected value triggers a different branch in the state space.

By default, JPF searches the state space of the checked program for "low-level" properties like deadlocks,

unhandled exceptions and failed assertions, however since it is extensible via the publisher/listener pattern, it allows to observe the course of the state space traversal. This way, listeners can check for specific (and more complex) properties in each visited state.

Each state of a checked program, as stored by JPF, consists of the heap, static area and stacks of all threads, thus representing the current state of the checked program at a particular scheduling or value point. When traversing the state space, JPF checks whether the current state has been already visited. In a positive case, it backtracks to the nearest scheduling or value point, for which there exist an unexplored branch and continues along that. This backtracking is based on keeping a stack representing the currently explored path in the state space (an item in the stack determines the list of not yet visited branches).

5. Model Checking Against Behavior Protocols

5.1. Motivation - Analysis of Options

Our key desire is to check whether a primitive component, implemented in Java, obeys its frame protocol. Since JPF is, without any extension, able to check only low-level properties (Sect. 4), and obeying a frame protocol is a quite high-level property, checking for this property in JPF is not directly possible. We identified the following options to address this problem:

(i) *Protocol assertions*: To enhance the component's code with assertions reflecting the frame protocol, and then let JPF check for violation of the assertions.

(ii) *State spaces integration*: To modify JPF in such a way that (a) any method call on an external (frame) interface of the component will be respected in POR, i.e. there will be a state associated with the call, and (b) the state space representing the frame protocol will be an integral part of the state space searched by JPF; the later can be achieved by some kind of parallel composition of the protocol related and code related state spaces.

(iii) *Checkers' cooperation*: To modify POR as described above (ii(a)) and keep the program code and protocol related state spaces separated and let model checker for each of them cooperate, i.e. to let JPF and protocol checker cooperate.

Since (i) inherently involves the kind of program analysis not easily reusable from JPF, and (ii) means a major modification of both JPF and model checker (moreover triggering the need to cope with portability issues with respect to future JPF versions), we have decided to go for (iii) whereas a key modification (not a major one) seemed to be necessary mainly at the protocol checker side.

5.2. Cooperation of Java PathFinder and Protocol Checker

Since JPF and the protocol checker work on different levels of abstraction - JPF at the level of byte code instructions and the protocol checker at the level of behavior protocols - and their states represent different information, it is necessary to define a mapping from the JPF state space, which is the lower-level one, into the state space of the protocol checker. Fortunately, this is possible since both state spaces can reflect all executions of the checked program in terms of frame methods' calls (even though at a different level of abstraction). The mapping is implemented as a JPF listener. The listener traces all executions of the invoke and return byte code instructions that are corresponding to methods of the provided and required interfaces of a target component, and notifies the protocol checker of such instructions in the form of atomic events, thus telling the protocol checker which transition from the list of all possible transitions it should take. The notification is done during traversal of the JPF state space in both the onward and backward directions.

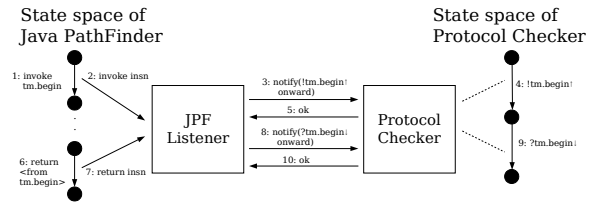


Figure 2: Communication between the JPF and Protocol Checker - traversal of state spaces in the onward direction

When the protocol checker is notified about an event that does not correspond to any element of the list of available transitions in the current state, it reports a violation of the frame protocol to JPF. In a similar vein, JPF notifies the protocol checker when it reaches an end state (i.e. and end of a branch of its state space, corresponding to the end of the main method), and if, in that case, the protocol checker is not in an end state of its own state space (e.g. it expects some more events to occur), an error is reported as well.

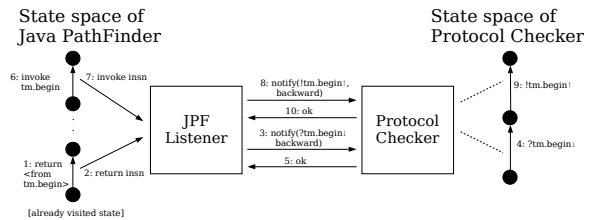


Figure 3: Communication between the JPF and Protocol Checker - traversal of state spaces in the backward direction

Communication between the Java PathFinder and the protocol checker during checking of the beginning of the add method, provided by the Database component, is depicted on Fig. 2 and Fig. 3. In both figures, the left part

shows the JPF state space and the right part shows the state space of the protocol checker; the numbers determine order of the related activities. Fig. 2 illustrates traversal of both state spaces in the onward direction and Fig. 3 the process of backtracking from an already visited state.

5.3. Modifications of JPF

In the process of implementing cooperation of JPF with the protocol checker, we had to enhance the functionality of JPF (i.e. to make several modifications of its source code) in order to support the mapping from the JPF state space into the state space of the protocol checker. The modifications include:

(i) *POR modification*. The code responsible for partial order reduction was modified by adding a new *frame call* point reflecting execution of an *invoke* or *return* instruction that corresponds to an event in the frame protocol. Even though this addition increases the state space size for most programs, it was inherently necessary.

(ii) *State representation extension*. Unfortunately, the relation between a frame call point and a state of the protocol checker may not be unique (so that no mapping can be found for this JPF state). In particular this happens in a specific case of correspondence between an *if-else* statement and an alternative in a frame protocol; below, the source code fragment and the corresponding part of the frame protocol (in two variants) illustrate such case:

```
// Java code
...
boolean b = Verify.randomBool();
if (b) {
    mA(); mB();
}
else {
    mC(); mD();
    b = true;
}
mE(); mF();
...

// fragments of frame protocol
// variant 1
(mA ; mB ; mE ; mF)
+
(mC ; mD ; mE ; mF)

//variant 2
(mA ; mB ; mE ; mF)
+
(mC ; mD ; mX ; mY)
```

Looking at the source code, it is clear that `mE()`; `mF()` will be always executed with `b` set to `true`. Consequently, when JPF backtracks at some point after executing `mF()` for the first time, to check the other *if-else* statement branch, it reaches an already visited state at the end of the *if-else* statement (since `b == true` is kept) and backtracks again,

not executing the `mE` and `mF` methods for the second time. At that point, the protocol checker will report a protocol violation though, since it expects `mE` and `mF` to be called. This happens even though the code obeys the protocol in variant 1. However, considering the variant 2, the code does not obey the protocol, but the protocol checker will again report a protocol violation, however not because the code does not obey the protocol, but again since it expects `mX` and `mY` to be called.

A solution to this problem was to assign a unique counterpart to a JPF state by the following JPF extension: Each state representation contains also the frame call trace for each thread (in addition to heap, static area and thread stack frames). Therefore the states with the same heap, static area and thread stacks, but with different frame call traces for a certain thread, are differentiated and their mapping to protocol checker state space is easy to determine. In the example above, when the state representation extension is applied, JPF is forced to execute the `mE` and `mF` methods for the second time because the two branches of the *if-else* statement produce different frame call traces.

5.4. Modifications of Protocol Checker

We have extended the static protocol checker with a new functionality in order to let it accept notifications from a JPF listener and drive the traversal of the protocol state space according to the received atomic events. In this respect, the added functionality is similar to the runtime protocol checker; put differently, the extended protocol checker can be viewed upon as the runtime protocol checker with support of backtracking. When the extended protocol checker receives an event, it checks whether it is possible to perform a corresponding transition in its state space in the desired direction (onward/backward); in a negative case, it reports a violation of the protocol to the JPF listener.

5.5. The Whole Picture - Making the Pieces Work Together

The tool for model checking of primitive components against behavior protocols, created via cooperation of JPF and the protocol checker, accepts as input implementation of a primitive component (i.e. its byte code), its environment (see below) and the specification of the component's architecture and frame protocol in the form of ADL.

When executed, the tool runs JPF with the protocol checker on the program composed of the component and its environment. The output is a success message, if the implementation obeys the frame protocol; otherwise the stack of the protocol checker and stacks of all threads are printed as a counterexample.

The environment of a target component is generated by another tool (environment generator) from its frame protocol [20]. Possible values of method parameters have to be provided in the form of a special Java class that serves as a container for the sets of values.

6. Evaluation

6.1. Discussion

Even though the proposed solution works “reasonably well” as documented by the experimental results provided in Sect. 6.2, a key drawback of this solution is that it increases the state space unnecessarily by considering the continuation after each `if-else` statement twice (by putting it into separate branches) in specific cases similar to the one described in Sect 5.3. To illustrate this, consider again the Java code from the example in Sect. 5.3 and the following fragment of the corresponding frame protocol:

```
((mA ; mB) + (mC ; mD)) ; mE ; mF
```

Here, the protocol asks the methods `mE` and `mF` to be executed only once. However, JPF with the state representation extension executes `mE()`; `mF()`; for the second time after backtracking to process the `else` branch (`mC()`; `mD()`). This way of handling the `if-else` statement continuations is the main cause of deterioration in performance (Sect.6.2).

We envision two solutions to this problem: (a) *Coordination of backtracking*. The idea, instead of extending the state representation with frame call traces, is to allow JPF to backtrack only if the protocol checker is also currently in an already visited state. Technically, if JPF

Table 1: State space size / time required of the two JPF modification alternatives for a component when checked against three versions of its frame protocol

JPF modification	Protocol (1)	Protocol (2)	Protocol (3)
POR	17 states / 3.3 sec	74 states / 2.5 sec	5085 states / 11.8 sec
POR + states representation	17 states / 2.7 sec	309 states / 2.7 sec	59011 states / 227 sec

is in a state when backtracking is desirable it asks the protocol checker for a permission to do so (which can be denied). However, a downside of this technique is the necessity to additionally modify the JPF core, with all related drawbacks (portability to new JPF versions, ...). (b) *State space integration*. This option was already mentioned in Sect. 5.1. The basic idea is to create JPF state space with compound states, each covering both the program code and behavior protocol substates. Here, backtracking coordination would be addressed implicitly by requiring it to be desirable in both substates of the state in question.

Both solutions are equivalent with respect to backtracking since both of them allow JPF to backtrack only if both the current state in the program code state space and the current state in the protocol state space allow to backtrack. However, an advantage of the first solution (coordination of backtracking) is that it is much easier to implement and can be made distributed without much effort, i.e. each checker can run in a separate address space/node, obviously helping fight state explosion.

Nevertheless, the bottom line is that just the publisher/listener pattern claimed to be the key extensibility

support of JPF (even though it proved very useful for our purpose) was not enough to achieve JPF cooperation with our protocol checker. In particular, out of this pattern, we had to extend the JPF internal state representation (internal state model in [6]) and furthermore we faced the problem of backtracking coordination. If these two issues were directly supported via JPF API, the JPF extensibility would be substantially enhanced, since we believe at least the former issue would be a prevailing problem of checking the validity of particular method call sequences (traces) via JPF, regardless the underlying state machine variant.

6.2. Experimental Results

As mentioned above, we have implemented several extensions and modifications to the original JPF code in order to make it possible to check whether a Java implementation of a primitive component obeys its frame protocol.

We have run several tests¹ to get a performance comparison between the versions of JPF with the modification of state representation turned off and on, and to show the impact of the complexity of environment and size of data domains on the time and space requirements for checking. All tests were done on a non-trivial, yet simple, primitive component (roughly 100 lines of Java code). The

code of the component is such that its state space mapping to the protocol state space is unique. This component has a provided interface `i1` and three required interfaces `i2`, `i3`, and `i4`, each of them featuring some of the methods `m1, m2, ..., m5`. The component was checked against the three versions of its frame protocol stated below, with the component’s environment also generated from the frame protocol. The simple protocol (1) contains just an alternative and nested call operators, while the protocol (2) employs also the repetition operator. The most complex protocol (3) contains in addition the and-parallel operator.

```
(1) ?i1.m1{!i2.m1 ; !i3.m1 ; !i4.m1} ;
?i1.m2{!i4.m2 ; !i3.m2 ; !i2.m2}
```

```
(2) ?i1.m1{!i2.m1 ; !i3.m1 ; !i4.m1} ; (
?i1.m3{!i4.m3 ; !i2.m3 ; (!i2.m4 + NULL);
```

¹All tests were performed on Intel Pentium 4 HT, 3.0 GHz, 2.0 GB RAM, running Windows 2003 Server Enterprise Edition SP1, and Sun Java SDK build 1.4.2_04-b05


```
!i2.m6 ; (!i4.m4 + !i4.m5))* ;
?i1.m2{!i4.m2 ; !i3.m2 ; !i2.m2}
```

```
(3) ?i1.m1{!i2.m1 ; !i3.m1 ; !i4.m1} ; (
(?i1.m3{!i4.m3 ; !i2.m3 ; (!i2.m4 + NULL) ;
!i2.m6 ; (!i4.m4 + !i4.m5)) |
(?i1.m4{!i4.m3 ; !i3.m5 ; !i3.m6; (!i4.m4 +
!i4.m5)) })* ; ?i1.m2{!i4.m2 ; !i3.m2 ;
!i2.m2}
```

Table 1 illustrates the effects of the two JPF modifications (POR only and both POR and state representation extension) in terms of the state space and time requirements growth.

Table 2 shows the performance of the modified JPF (POR+states representation) for data domains of increasing complexity (one-, two-, and four-value data domains are considered). The abstract data sets were used for eleven variables in the source code. Generally, doubling the size of a data domain of a single globally accessible variable results in twice as large state space (exponential growth)

Table 2: State space / time required for different data domains

	Protocol (1)	Protocol (2)	Protocol (3)
One-value domain	17 states / 2.7 sec	309 states / 2.7 sec	59011 states / 228 sec
Two-value domain	17 states / 2.4 sec	749 states / 3.2 sec	163968 states / 389 sec
Four-value domain	17 states / 2.4 sec	2499 states / 4.5 sec	1099386 states / 1548 sec

allowing usually only small data domains to be taken into account. Nonetheless, such verification still provides valuable information, more thorough than simple testing.

From these experimental results, it is clear that a drawback of modifications to JPF we made is the growth of the state space, which results in increase of time requirements of the checking process. Despite that, the state space of a typical primitive component can be still traversed in a reasonable time: In addition to these performance tests, we have also successfully applied this JPF and protocol checker cooperation to a non-trivial component-based application consisting of 20 components with the architecture and behavior specified via ADL and behavior protocols (over 300 lines); the verification of a component took from few minutes to 24 hours in the worst case.

7. Related work

Besides the Java PathFinder model checker, there exist other tools for model checking of finite-state software systems [7, 9, 10, 12]. As far as we know, these model checkers require the checked property to be specified in a particular firmly determined way (e.g. custom property specification language, assertions, etc.). Specifically, none of them targets software components, let alone checking the components' code against behavior properties specified at

the ADL level, as apparent from their short characteristics provided below.

The Bandera tool set [7] is designed for model checking of Java programs against temporal logic expressions. It supported the Spin and JPF model checkers originally, but the next generation of the tool set employs Bogor [8] as the core model checker.

Similar to Bandera, the Zing model checker [9] targets concurrent object-oriented programs. It accepts a model of a target program, defined in a custom specification language, as input and verifies it against user-defined assertions.

The SLAM model checker [10] is a part of the SDV tool for formal verification of device drivers for the Windows operating systems. It is specific in that it creates a Boolean abstraction of a target program and uses the principle of refinement to discard errors that are present in the abstraction but not in the original program. Properties to be checked are to be specified in a low-level language called SLIC [11].

The MAGIC tool [12] aims at formal verification of C programs against finite state machine specifications. It uses

compositional approach, which means that it decomposes a software system into several components (i.e. procedures written in the C language), and then verifies each component separately. More specifically, it is able to verify that a finite state machine (LTS) is a safe abstraction of a C procedure by employing the abstract-verify-refine paradigm [21].

Charmy [19] is an extensible tool for architectural analysis. It allows graphical UML-like specification of a system architecture including topology editor, sequence and state charts. The specification can be checked in an automatized way for absence of static specification errors, e.g. for each send message operation in a component there has to be a receive message operation in another component, messages with the same name must have the same number of parameters, etc. The architecture specification can be translated (again in an automatized way) into Promela (Spin specification language), and it can be checked for an arbitrary property expressible in LTL.

So the bottom line is that none of these checkers employs the idea of checking a given model against a specific property via cooperating with another model checker. However, the technique of integrating a model checker with another tool for automated verification has been applied several times in the following form: A model checker is used in a theorem prover as a decision procedure for temporal properties [14, 22]. Typically, this approach is applied to

software systems with a large (or even infinite) state space. For example, an integration of the Isabelle/IOA theorem prover with the μ cke model checker is presented in [14]. The Isabelle tool employs the μ cke tool as an oracle for μ -calculus formulas related to I/O automata.

8. Conclusion and future work

In this paper, we presented our approach to model checking of software components implemented in the Java language against their behavior specification (behavior protocols [1]), which makes the Java PathFinder model checker [5] cooperate with the protocol checker [4]. The key benefits include a quick realization, decent performance, and relatively easy maintainability when facing a new version of JPF. We showed, however, that just the publisher/listener pattern claimed to be the key flexibility support of JPF (even though proved very useful for our purpose) was not enough to achieve JPF cooperation with the protocol model checker.

As to future work, our current research goals include (a) extending JPF with a direct support for behavior protocols (the options “protocol assertions” and “state spaces integration” in Sect. 5.1). (b) Coordination of backtracking mentioned in Sect. 6.1 - this is of our highest priority.

Acknowledgments

We would like to record a special credit to Jiri Adamek for valuable hints and comments regarding the cooperation of JPF and the protocol checker.

References

- [1] F. Plasil and S. Visnovsky: Behavior Protocols for Software Components, *IEEE Transactions on Software Engineering*, vol. 28, no. 11, Nov 2002
- [2] J. Adamek and F. Plasil: Component Composition Errors and Update Atomicity: Static Analysis, *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 4, John Wiley, 2005
- [3] J. Adamek and F. Plasil: Erroneous Architecture is a Relative Concept, *Proceedings of Software Engineering and Applications (SEA)*, published by ACTA Press, ISBN 0-88986-425-X, pp. 715-720, Nov 2004
- [4] M. Mach, F. Plasil, and J. Kofron: Behavior Protocol Verification: Fighting State Explosion, *IJCIS Vol.6, Number 1, ACIS, ISSN 1525-9293*, pp. 22-30, 2005
- [5] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda: Model Checking Programs, *Automated Software Engineering Journal*, Vol. 10, No. 2, Apr 2003
- [6] P. C. Mehltz, W. Visser, and J. Penix: The JPF Runtime Verification System, NASA Ames Research Center, <http://javapathfinder.sourceforge.net>
- [7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zhueng: *Bandera: Extracting Finite-state Models from Java Source Code*, ICSE 2000, pages 439-448
- [8] Robby, M. Dwyer, and J. Hatcliff: *Bogor: An extensible and highly-modular model checking framework*, In *FSE 03: Foundations of Software Engineering*, pages 267-276, ACM, 2003
- [9] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie: *Zing: a model checker for concurrent software*, Technical report, Microsoft Research, 2004
- [10] T. Ball and S. K. Rajamani: *The SLAM Project: Debugging System Software via Static Analysis*, *POPL 2002*, ACM, Jan 2002
- [11] T. Ball and S. K. Rajamani: *SLIC: A Specification Language for Interface Checking (of C)*, MSR-TR-2001-21, Microsoft Research, 2002
- [12] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith: *Modular Verification of Software Components in C*, *IEEE Transactions on Software Engineering*, vol. 30, no. 6, June 2004
- [13] E. Clarke, O. Grumberg, and D. Peled: *Model Checking*, MIT Press, Jan 2000
- [14] T. Hamberger: *Integrating Theorem Proving and Model Checking in Isabelle/IOA*, Technical report, T.U. Munich, August 1999
- [15] R. Allen and D. Garlan: *A Formal Basis for Architectural Connection*, In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, issue 3, pp. 213-249, July 1997
- [16] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer: *Specifying Distributed Software Architectures*, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, vol. 989, pp. 137-153, 1995
- [17] D. Giannakopoulou, J. Kramer, and S. C. Cheung: *Analysing the Behaviour of Distributed Systems using Tracta*, *Journal of Automated Software Engineering*, vol. 6(1), Jan 1999
- [18] F. Plasil, D. Balek, and R. Janecek: *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*, *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, IEEE CS Press, May 1998
- [19] P. Inverardi, H. Muccini, and P. Pelliccione: *CHARMY: An Extensible Tool for Architectural Analysis*, *ESEC-FSE'05, The fifth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering. Research Tool Demos*. Lisbon, Portugal, 2005
- [20] P. Parizek and F. Plasil: *Specification and Generation of Environment for Model Checking of Software Components*, Technical Report No. 2005/5, Dep. of SW Engineering, Charles University, Nov 2005
- [21] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith: *Counterexample-guided abstraction refinement*, In *Computer Aided Verification*, pages 154-169, 2000
- [22] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas: *PVS: Combining Specification, Proof Checking, and Model Checking*, *Proceedings of CAV'96*, 1996

Chapter 6

Specification and Generation of Environment for Model Checking of Software Components

Pavel Parízek,
František Plášil

Contributed paper at **Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2006)**.

In *Electronic Notes in Theoretical Computer Science*,
published by Elsevier B.V.,
Volume 176, Issue 2,
pages 143–154,
ISSN 1571-0661,
May 2007.

The original version is available electronically from the publisher's site at <http://dx.doi.org/10.1016/j.entcs.2006.02.036>.

Specification and Generation of Environment for Model Checking of Software Components

Pavel Parizek^{a,1}, Frantisek Plasil^{a,b,1}

^a *Department of Software Engineering
Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
{parizek, plasil} @ nanya.ms.mff.cuni.cz*

^b *Institute of Computer Science
Academy of Sciences of the Czech Republic
Prague, Czech Republic
plasil @ cs.cas.cz*

Abstract

Model checking of isolated software components is inherently not possible because a component does not form a complete program with an explicit starting point. To overcome this obstacle, it is typically necessary to create an environment of the component which is the intended subject to model checking. We present our approach to automated environment generation that is based on behavior protocols [9]; to our knowledge, this is the only environment generator designed for model checking of software components. We compare it with the approach taken in the Bandera Environment Generator tool [12], designed for model checking of sets of Java classes.

Keywords: Software components, behavior protocols, model checking, automated generation of environment

1 Introduction

Model checking is one of the approaches to formal verification of software systems that gets a lot of attention at present. Still, there are some obstacles that have to be addressed, at least partially, before model checking of software can be widely used in practice. Probably the biggest problem is the size of state space typical for software systems. One solution to this problem (state explosion) is the decomposition of a software system into small and well-defined units, components.

Nevertheless, a component usually cannot be checked in isolation, because it does not form a complete program inherently needed to apply model checking. It is, therefore, necessary to create a model of the environment of the component subject to model checking, and then check the whole program, composed of the

¹ This work was partially supported by the Czech Academy of Sciences (project 1ET400300504) and France Telecom under the external research contract number 46127110.

*This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

environment and component. The environment should be created in a way that minimizes the increase of the state space size caused by the composition.

1.1 Goals and Structure of the Paper

The paper aims at addressing the problem of automated generation of environment for model checking of software components implemented in the Java language. The main goal is to present our approach that is based on behavior protocols [9] and to compare it with the approach taken in the Bandera Environment Generator tool [12], which is the only other Java focused approach we are aware of.

The remainder of the paper is organized as follows. Sect. 2 provides an example to illustrate the problem of environment generation and Sect. 3 introduces the Bandera Environment Generator (BEG) [12]. Sect. 4 starts with an overview of behavior protocols [9] and then presents the key contribution - the description of our approach to specification and generation of environment based on behavior protocols. Sect. 5 provides comparison of the two approaches and briefly mentions our proof of concept implementation. The rest of the paper contains related work and a conclusion.

2 Motivation

In order to illustrate how an environment can be created, we present a simple example - a Java class `DatabaseImpl` and a handwritten environment for this class, assuming `DatabaseImpl` is the intended subject to model checking. The class implements one interface and requires one internal reference of an interface type to be set. Therefore, it can be also looked upon as a `Database` component with one provided and one required interface.

Key fragments of source code of the `DatabaseImpl` class look as follows:

```
public interface IDatabase {
    public void start();
    public void stop();
    public void insert(int key, String value);
    public String get(int key);
}

public class DatabaseImpl implements IDatabase {
    private ILogger log;

    public void start() {
        log.start();
    }

    public void stop() {
        log.stop();
    }
}
```

```

public void insert(int key, String value) {
    ...
}

public String get(int key) {
    ...
}
}

```

In general, an environment should allow the model checker (i) to search for concurrency errors (typically reflected by introducing several threads that are executed in parallel), and (ii) to check all the control flow paths (usually addressed by a random choice of parameter values for all methods).

Captured by “the important” fragments of its source code, such environment could take the following form:

```

public class EnvThread extends Thread {
    IDatabase db;
    ...

    public void run() {
        db.insert(getRandomInt(), getRandomString());
        String val = db.get(getRandomInt());
        ...
    }
}

public static void main(String[] args) {
    IDatabase db = new DatabaseImpl();
    db.setLogger(new LoggerImpl());

    db.start();

    new EnvThread(db).start();
    new EnvThread(db).start();
    ...

    db.stop();
}

```

In the example, two threads of control, which enable the model checker to search for concurrency errors, are created. A random choice of parameter values for the purpose of checking all the control flow paths is employed as well (`getRandom...` calls).

Obviously, creating an environment by hand is hard and tedious work even in simple cases. A straightforward solution to this problem is to automatically generate the environment from a higher-level abstraction than the code provides. In Sect. 3 and 4, we present two solutions based on this idea.

3 Environment Generator in Bandera

3.1 Bandera

Bandera [6] is a tool set designed for model checking of complete Java programs, i.e. those featuring a main method. It is composed of several modules - model extractor, model translator, environment generator, and model checker, to name the key of them. The model extractor extracts a (finite) internal model from Java source code and the model translator translates the internal model into the input language of a target model checker. Here, the Bandera tool set supported the Spin and Java PathFinder model checkers originally, but currently it is intended mainly for a Bandera specific model checker (Bogor [11]).

3.2 Bandera Environment Generator

The Bandera Environment Generator (BEG) [12] is a tool for automated generation of environment for Java classes. Given a complete Java program, the user of the BEG tool has to decompose the program into two parts - the tested *unit*, i.e. the classes to be tested, and its *environment*. Since the environment part is usually too complex for the purpose of model checking, it is necessary to create an abstract environment. This abstract environment can be generated from a model created

- either from assumptions the user provided, or
- from a result of code analysis of environment classes (if available).

The model can specify, for example, that a certain method should be called five times in a row, or that it should be executed in parallel with another specific method.

Since, usually, there exist no environment classes in case of software components, we will further consider only the first option - i.e. that the abstract environment is generated from user-specified assumptions. For this purpose, the BEG tool provides two formal notations - LTL and regular expressions. The actual specification (“environment specification” in the rest of this section) takes the form of program action patterns (method calls, assignments, etc), illustrated below.

An environment specification for the `DatabaseImpl` class presented in Sect. 2, written in the input language of the BEG tool, could be as follows:

```
environment
{
  instantiations
  {
    1 LoggerImpl log;
    IDatabase db = new DatabaseImpl();
    db.setLogger(log);
    int x = 5;
  }

  -- high level specification of the environment behavior
  regular assumptions
  {
```

```

    T0: (db.get() | db.insert())*
    T1: (db.get(x) | db.insert(5, "abcd"))*
  }
}

```

The `instantiations` section allows the user to specify how many instances of a certain type should be created and under which names they can be referenced. In this example, two objects are instantiated - the `log` instance of the `LoggerImpl` class and the `db` instance of the `DatabaseImpl` class.

The `regular assumptions` section contains regular expressions describing the behavior of the environment with respect to the tested classes. Each regular expression defines a sequence of actions that should be performed by a single thread of control. In our example, two threads of control are defined, both modeling a sequence of calls to the `insert` and `get` methods on the `IDatabase` interface.

Notice that the whole execution is characterized by the specified threads (`T0`, `T1`) - there is no “main” thread. Consequently, calls to the `start` and `stop` methods on the `IDatabase` interface cannot be reasonably modeled in such an environment specification.

The BEG tool also allows to specify parameter values of method calls on the tested classes. If the value of a parameter is not specified, as in the thread `T0` above, then it is non-deterministically selected from all the available values of a given type (e.g. from all allocated instances of a given class in the case of a reference type) during model checking. As a parameter to a method call, it is even possible to use a variable defined in the `instantiations` section (such as `x` above).

As the BEG tool is not intended specifically for software components, but rather for plain Java classes, it is necessary to manually specify the environment for the classes that implement a target component; an alternative would be to develop a tool for automatic translation of an ADL specification of the component’s architecture and behavior into the input language of the BEG tool.

However, since the most recent Bandera release is an alpha version only [6], not being fully stable yet, we have decided to use the Java PathFinder model checker (JPF) [13]. Consequently, we faced the problem to create an environment generator, since none was available (BEG is not intended for components and, moreover, the latest Bandera version does not allow to use the Java PathFinder as a target model checker any more).

4 Environment Generator for Java PathFinder

We have built our own environment generator for model checking of components implemented in the Java language. Our approach stems from the assumption that components are during design specified in an ADL (Architecture Description Language), which, in particular, includes specification of their provided and required interfaces and also specification of their behavior. The latter is done via behavior protocols [9]. In this section we show how this behavior specification can be advantageously employed for generating an environment necessary for component model checking.

4.1 Behavior protocols

A behavior protocol is an expression that describes the behavior of a software component in terms of atomic events on the provided and required interfaces of the component, i.e. in terms of accepted and emitted method call requests and responses on those interfaces.

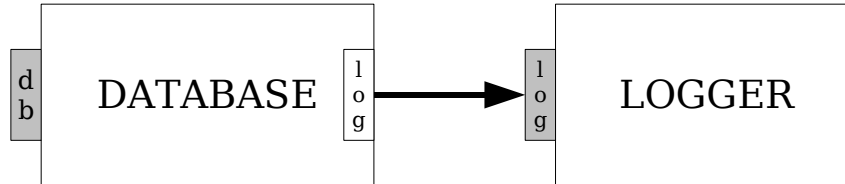


Fig. 1. The DATABASE and LOGGER components, defined in Sect. 2

A protocol example for the Database component from Fig. 1 is below:

```
?db.start↑ ; !log.start ; !db.start↓ ; (?db.insert || ?db.get)* ;
?db.stop{!log.stop}
```

Since this protocol specifies the interplay on the external interfaces of Database, it is its *frame protocol* [9]. Informally speaking, it specifies the Database functionality that starts with accepting request for **start** call on **db**. As a reaction it calls **start** at **log** and issues response to the **start** call on **db**. This is followed by accepting **insert** on **db** in parallel with **get** on **db** finitely many times. At the end, it accepts a request for a **stop** call on **db** and, as a reaction, it calls **stop** at **log** and issues response to the **stop** call on **db**.

Each event has the following syntax: `<prefix><interface>.<method>` `<suffix>` (where the suffix is optional; the events having no suffix are syntactical shortcuts explained below). The prefix `?` denotes an *accept* event and the prefix `!` denotes an *emit* event. The suffix `↑` stands for a request (i.e. a method call) and the suffix `↓` stands for a response (i.e. return from a method). An expression of the form `!i.m` is a shortcut for `!i.m↑;?i.m↓`, an expression of the form `?i.m` is a shortcut for `?i.m↑;!i.m↓` and an expression of the form `?i.m{prot}` is a shortcut for `?i.m↑;prot;!i.m↓`, where `prot` is a protocol. The NULL keyword denotes an empty protocol.

The example above presents also several operators. The `;` character is the sequence operator, `*` is the repetition operator and `||` is the or-parallel operator. Behavior protocols support also an alternative operator `+` and an and-parallel operator `|`. In fact, the or-parallel operator is only a shortcut; e.g. `a || b` stands for `a + b + (a | b)`. The `|` operator denotes all the possible interleavings of traces that correspond to its operands.

A behavior protocol defines a possibly infinite set of event traces, each of them being finite.

Each component has a frame protocol associated with it, and a composite component can have also an architecture protocol [9]. The frame protocol of a component describes its external behavior, what means that it can contain only the events on external interfaces of the component. On the other hand, the architecture protocol describes the behavior of a component in terms of composition of its subcomponents at the first level of nesting.

4.2 Cooperation of Java PathFinder with the Protocol Checker

When checking a component application specified via ADL with behavior protocols, it is necessary (i) for each composite component in the hierarchy to check compositional compliance of subcomponents at the first level of nesting and also compliance of a frame protocol with an architecture protocol (ii) and for each primitive component to verify that an implementation of the component obeys its frame protocol. For the purpose of checking compliance of protocols, we use the protocol checker [7] developed in our research group, and for checking that a primitive component obeys its frame protocol, we use a tool created via cooperation of JPF with our protocol checker [8]. The tool has to be applied to a program composed of a target component and its environment.

Communication between JPF and the protocol checker during checking of the **Database** component is depicted on Fig. 2. The left part of the schema shows the JPF traversing the code (state space) of the component and the right part shows the state space of the protocol checker, which is determined by the frame protocol of the component. A plugin for JPF, which we have developed, traces execution of the invoke and return instructions that are related to methods of the provided and required interfaces of a target component, and notifies the protocol checker of those instructions in the form of atomic request and response events. The protocol checker verifies that the trace constructed from the received events is compliant with the frame protocol of the component. When the protocol checker encounters an unexpected event or a missing event, it tells JPF to stop the state space traversal and to report an error (counter example) to the user.

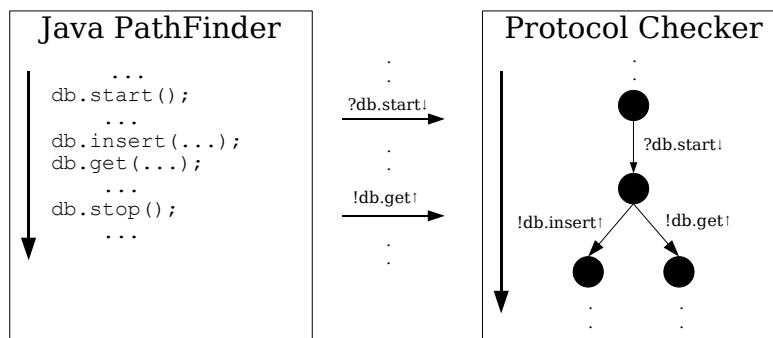


Fig. 2. Communication between the Java PathFinder and Protocol Checker

4.3 Modeling the Environment with Inverted Frame Protocol

The environment of a component can be advantageously modeled by its inverted frame protocol [1], constructed from the components frame protocol by replacing all the accept events with emit events and vice versa. The inverted frame protocol constructed this way forces the environment

- to call a certain method of a particular provided interface of the component at the moment the component expects it, and
- to accept a certain method call issued on a particular required interface of the component at the moment the component “wishes” to do so.

The inverted frame protocol of the Database component introduced above is:

```
!db.start↑ ; ?log.start ; ?db.start↓ ; (!db.insert || !db.get)* ;  
!db.stop{?log.stop}
```

Our environment generator accepts all syntactically valid frame protocols with the exception of protocols of the form $?a + !b$ and $!a^* ; ?b$. The reason for not supporting frame protocols of the form $?a + !b$ is that the environment driven by inversion of such a protocol cannot determine how long it should wait for the $!b$ event to occur before it emits a call that corresponds to the $?a$ event and therefore disables the other alternative (i.e. $!b$). Protocols of the form $!a^* ; ?b$ are not supported for a similar reason - the environment is not able to determine when the repetition $!a^*$ is going to finish. It is recommended to use protocols of the form $!a^* ; !b$ instead (wherever possible) because in such case the $!b$ event tells the environment that the repetition has finished.

In order to minimize the size of the state space that JPF has to traverse, our environment generator performs several transformations of the frame protocol of the target component before creating the inverted frame protocol and generating the code of the environment. The key goal of the transformations is to

- get as many instances of the alternative operator $+$ as possible at the outermost level of protocol nesting. The advantage of this approach is that all these alternatives can be checked in parallel by multiple instances of JPF, thus lowering the time requirements for model checking of the target component.
- reduce the number of repetitions, and also event interleavings caused by the $|$ operator, even at the cost of accuracy.

For example, our generator transforms

- an iteration over some subprotocol to an alternative between an empty protocol and a sequence of two copies of the subprotocol (e.g. the protocol $!a^*$ is transformed to the protocol $NULL + (!a ; !a)$),
- a sequence that contains some alternatives to an alternative between all possible sequences (e.g. the protocol $!a ; (!b1 + !b2)$ is transformed to the protocol $(!a ; !b1) + (!a ; !b2)$),
- an and-parallel operator connecting two subprotocols, both of them being alternatives, to an alternative between selected pairs of subprotocols connected by the $|$ operator - the pairs are selected in a way ensuring that each element of the two

- alternatives is present at least in one of the pairs (e.g. the protocol $(!a1 + !a2) | (!b1 + !b2)$ is transformed to the protocol $(!a1 | !b1) + (!a2 | !b2)$), and
- an and-parallel operator with three or more subprotocols to an alternative between selected pairs of subprotocols, where each pair is connected by the `|` operator and followed by a sequence of subprotocols that do not belong into the selected pair; the pairs are selected in such a way that the first subprotocol is paired with the second, the second with the third, and so on (e.g the protocol $a | b | c | d$ is transformed to the protocol $((a | b) ; c ; d) + ((b | c) ; a ; d) + ((c | d) ; a ; b)$).

4.4 Specification of Values of Method Parameters

Our solution to specification of the possible values of method parameters is based on the idea that the user defines the set of values which are to be considered as parameters. From the implementation point of view, these sets are to be put into a special Java class serving as a container for all the sets of values. The value of a method parameter of certain type is later non-deterministically selected from the set of values considered for that type and method. In addition to the sets of values common for the whole component, it is also possible to define sets that are specific to a particular method or interface.

Below is a fragment of the specification of values for the `Database` component:

```
putIntSet("IDatabase", "insert", new int[]{1, 2, 5, 10});
putIntSet("", "", new int[]{1, 3, 5, 12});
putStringSet("", "", new String[]{"abcd", "EFGH1234"});
```

The first statement defines a set of integer values that is specific to the `insert` method of the `IDatabase` interface. The other two statements define the sets of integers and strings that are to be applied to all methods of the `Database` component interfaces.

The main drawback of this approach is that the user has to define on his/her own the sets of values in such a way that will force the model checker to check all the control flow paths in the component.

5 Evaluation

In this section we compare the two approaches to modeling the environment described above, i.e. the approach of the BEG tool and our approach based on behavior protocols.

The main differences between them are:

- The BEG tool allows to specify parallelism only at the outermost level of regular expressions that specify behavior of the environment (there is no such limit in case of behavior protocols).
- Behavior protocols have no support for method parameters, therefore the possible values of method parameters must be specified separately in a special Java class, while the BEG tool allows to specify the values of method parameters directly in the expressions that specify behavior of the environment.

It is worth to mention that there is also a difference in that the BEG tool targets plain Java classes with informally specified provided and required interfaces, while our approach targets the software components having provided and required interfaces defined in an explicit way.

As a speciality, another advantage of support for specification of parameter values directly in expressions that specify behavior is that it enables the environment generator to select a proper version of an overloaded method - or to generate a code that will non-deterministically invoke all versions of the method that conform to the specification.

We have created an implementation of the environment generator that uses the inverted frame protocol of a component as a model of the environments behavior. It aims at components that use the Fractal Component Model [5] and expects that the Fractal ADL is used to define components. We have successfully applied our environment generator to a component-based application composed of 20 components. Transformations of the frame protocol, described in Sect. 4.3, reduce the size of the state space determined by the protocol approximately thirty times in case of more complex components, therefore lowering also the time required for model checking of the components, all that at the cost of accuracy, though. Nevertheless, model checking of more complex components with environment generated from their frame protocols with no transformations applied is not feasible. Despite the abstractions of the environment introduced by transformations of the frame protocol, the technique is still much more systematic than simple testing. Let us again emphasize that model checking of a component without an environment is not possible at all, because JPF is applicable only to complete Java programs, not isolated software components.

6 Related work

Except for the Bandera Environment Generator [12], we are not aware of any other approach to specification and generation of environment for model checking of software components or parts of object-oriented programs. Nevertheless, there exist model checkers for object-oriented programs that do not need to generate an environment because these tools usually extract a finite model from a complete program (featuring the main method) and then check the model - an example of such a model checker is Zing [2].

There are also tools that solve the problem of automatic generation of environment for fragments of procedural programs (e.g. drivers, libraries, etc). An example of such a tool is the SLAM [4] model checker, which is a part of the SDV tool for verification of device drivers for the Windows operating system. Given a program, the checker creates a Boolean abstraction of the program (all value types approximated by Boolean) and then checks whether some desired temporal properties hold for the abstraction. It uses the principle of refinement to discard errors that are present in the abstraction but not in the original program (false negatives). The environment for device drivers is defined by the interfaces provided by the Windows kernel. The SLAM tool models the environment via training [3]. Here, the basic principle is that, for a certain procedure P that is to be modeled, it first takes several drivers

that use the procedure P, then it runs the SDV tool on those drivers and therefore gets several Boolean abstractions of the procedure P, and finally merges all those abstractions and puts the resulting Boolean abstraction of the kernel procedure P into a library for future reuse.

Our tool for environment generation is partially based on [10]. The tool that is described in the thesis, designed for the Bandera tool set, also uses the inverted frame protocol idea; it is also focused on components compliant to the Fractal Component Model [5]. We decided not to use this tool mainly because it generates an environment that increases the state space size quite significantly, since it does not employ any of transformations described in Sect. 4.3 and also does not provide any means for specification of method parameter values - all that makes it almost unusable in practice.

7 Conclusion

Direct model checking of isolated software components is usually not possible because model checkers can handle only complete programs. Therefore, it is necessary to create an environment for each component subject to model checking.

In this paper, we have compared two approaches to generating environment of components, resp. classes - namely the Bandera Environment Generator (BEG) tool [12] in Sect. 3, and our approach that is based on behavior protocols [9] in Sect. 4. Main differences between the two approaches lie in the level of support for parallelism, in support for specification of parameter values, and in the fact that the BEG tool is focused on plain Java classes while our approach targets software components with explicitly defined provided and required interfaces.

As to future work, an automated derivation of sets of values used for nondeterministic choice of method parameters is our current goal. It is motivated by the fact that manual definition of such sets requires the user to carefully capture a way that will let the model checker to check all the control flow paths in a target component. A viable approach to the derivation of possible parameter values could be to use static analysis of Java source code (or byte code).

Acknowledgments

We would like to record a special credit to Jiri Adamek and Nicolas Rivierre for valuable comments and Jan Kofron also for many hints regarding the integration of the protocol checker with JPF.

References

- [1] Adamek, J., and F. Plasil, *Component Composition Errors and Update Atomicity: Static Analysis*, Journal of Software Maintenance and Evolution: Research and Practice, **17**(2005), pp. 363-377
- [2] Andrews, T., S. Qadeer, S. K. Rajamani, J. Rehof and Y. Xie, *Zing: a model checker for concurrent software*, Technical report, Microsoft Research, 2004
- [3] Ball, T., V. Levin and F. Xie, *Automatic Creation of Environment Models via Training*, TACAS 2004, 93-107

- [4] Ball, T., S. K. Rajamani, *The SLAM Project: Debugging System Software via Static Analysis*, POPL 2002, ACM, 1-3
- [5] Bruneton, E., T. Coupaye, M. Leclercq, V. Quma, and J. B. Stefani, *An Open Component Model and its Support in Java*, In Proceedings of the International Symposium on Component-based Software Engineering (ICSE 2004 - CBSE7), LNCS, **3054**(2004), May 2004
- [6] Corbett, J. C., M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby and H. Zhueng, *Bandera: Extracting Finite-state Models from Java Source Code*, ICSE 2000, ACM, 439-448
- [7] Mach, M., F. Plasil and J. Kofron, *Behavior Protocol Verification: Fighting State Explosion*, International Journal of Computer and Information Science, **6**(2005), 22-30
- [8] Parizek, P., F. Plasil and J. Kofron, *Model Checking of Software Components: Making Java PathFinder Cooperate with Behavior Protocol Checker*, Tech. Report No. 2006/2, Dep. of SW Engineering, Charles University, Jan 2006
- [9] Plasil, F., and S. Visnovsky, *Behavior Protocols for Software Components*, IEEE Transactions on Software Engineering, **28**(2002)
- [10] Potrusil, T., "Specifying Missing Component Environment in Bandera", Master Thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2005
- [11] Robby, M. Dwyer and J. Hatcliff, *Bogor: An extensible and highly-modular model checking framework*, In FSE 03: Foundations of Software Engineering, pp. 267-276, ACM, 2003
- [12] Tkachuk, O., M. B. Dwyer and C. S. Pasareanu, *Automated Environment Generation for Software Model Checking*, 18th IEEE International Conference on Automated Software Engineering (ASE03), p. 116, 2003
- [13] Visser, W., K. Havelund, G. Brat, S. Park and F. Lerda, *Model Checking Programs*, Automated Software Engineering Journal, **10**(2003)

Chapter 7

Modeling Environment for Component Model Checking from Hierarchical Architecture

Pavel Parízek,
František Plášil

Contributed paper at **Third International Workshop on Formal Aspects of Component Software (FACS'06)**.

In *Electronic Notes in Theoretical Computer Science*,
published by Elsevier B.V.,
Volume 182,
pages 139–153,
ISSN 1571-0661,
June 2007.

The original version is available electronically from the publisher's site
at <http://dx.doi.org/10.1016/j.entcs.2006.09.036>.

Modeling Environment for Component Model Checking from Hierarchical Architecture

Pavel Parizek^{a,1}, Frantisek Plasil^{a,b,1}

^a *Department of Software Engineering
Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
{parizek, plasil} @ neny.ms.mff.cuni.cz*

^b *Institute of Computer Science
Academy of Sciences of the Czech Republic
Prague, Czech Republic
plasil @ cs.cas.cz*

Abstract

Application of model checking to isolated software components is not directly possible because a component does not form a complete program - the problem of missing environment occurs. A solution is to create an environment of some form for the component subject to model checking. As the most general environment can cause model checking of the component to be infeasible, we model the environment on the basis of a particular context the component is to be used in. More specifically, our approach exploits hierarchical component architecture and component behavior specification defined via behavior protocols, all that provided in ADL. This way, the environment represents the behavior of the rest of the particular application with respect to the target component. We present an algorithm for computing the model of environment's behavior that is based on syntactical expansion and substitution of behavior protocols.

Keywords: Software components, behavior protocols, environment for model checking, hierarchical component architecture

1 Introduction

Various methods of formal verification have already proven to be useful for finding errors in large and complex software systems, and particularly in critical systems, thus helping increase reliability of such systems. At present, one of the most popular approaches to verification of software systems is model checking [4], which is an algorithmic technique for checking whether a finite model of a target system satisfies a certain property. Typically, the model has the form of a finite labeled transition system and the property can be expressed as a temporal logic expression (LTL, CTL). Checking whether a property is satisfied in the model is based on an exhaustive traversal of the state space determined by the model. This way, model

¹ This work was partially supported by the Grant Agency of the Czech Republic (project number 201/06/0770).

*This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

checking can help to find concurrency errors like deadlocks, which are very subtle and quite hard to discover with traditional approaches such as testing. However, the main advantage of model checking - traversal of the complete state space (i.e. checking of the property in each state) - is also its main weakness. Especially in case of more complex software systems, the state space may be large enough to make model checking of a system not feasible; this is well-known as the *state explosion problem*.

A completely different approach to building more reliable software systems is to decompose large and complex systems into smaller and well-defined units - software components. Typically, components are considered to be entities with well-defined provided (server) and required (client) interfaces, and in some cases also with formally specified behavior. A component-based application is a collection of individual components, which are interconnected via well-defined bindings between their interfaces.

Components that have no externally observable internal structure, while having real implementation in a certain programming language, are called *primitive components*. Components containing nested subcomponents, i.e. components with observable internal structure, are called *composite components*. The structure of a composite component, commonly referred to as *component architecture*, is typically defined in an Architecture Description Language (ADL) [2][8][9]. Usually, definition of a composite component in an ADL specifies also the external provided and required interfaces of all components, bindings between the component and its subcomponents, and optionally component behavior (e.g. in an LTS-based formalism).

An example of a composite component, which will be used to illustrate the ideas presented throughout the rest of the paper, is depicted on Fig. 1. The component `DBServer` provides the `db` interface of type `IDatabase` and contains four primitive subcomponents - `Database`, `Logger`, `Transaction Manager`, and `Backup Scheduler`. The `db` interface of the `DBServer` component is implemented by delegation to the `Database` subcomponent. The `Logger` and `Transaction Manager` subcomponents are bound to the required interfaces of the `Database` component, and the `Transaction Manager` component is bound also to the required interface of the `Backup Scheduler` component. In the rest of the paper, we will be interested especially in the `Transaction Manager` component, which provides the `start`, `stop`, `begin`, `commit` and `abort` methods on its provided interface `tm` of type `ITxMngr`, and the `backup` method on its provided interface `bk` of type `IBackup`.

1.1 Problem of Missing Environment

A viable approach is the application of model checking to individual software components, for example, in order to verify that the component's implementation satisfies a formal specification of the component's behavior. As an individual component obviously generates a smaller state space than the whole application, the problem of state explosion is also mitigated this way, at least partially.

However, considering only primitive components, the problem with model checking of these components is that they are not complete programs (e.g. with `main` method) - and model checkers typically analyze only complete programs. This

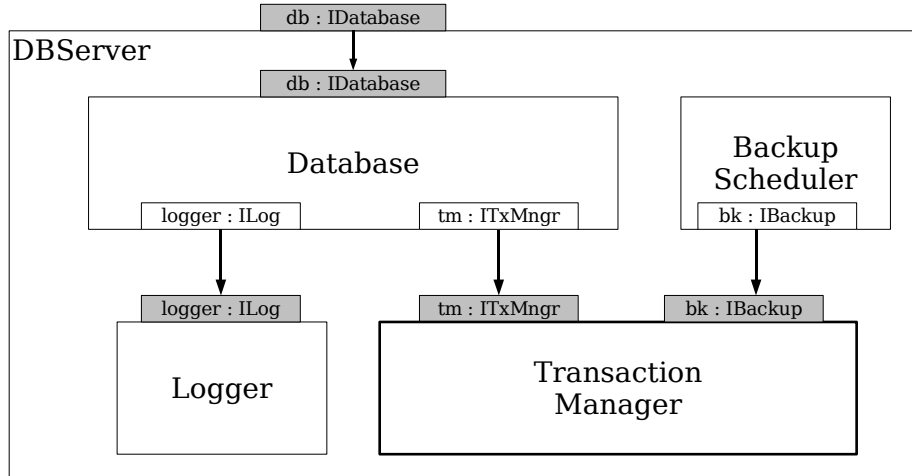


Fig. 1. Architecture of the `DBServer` component

triggers the *problem of missing environment*. An obvious solution is to create an environment of some form for each primitive component subject to model checking, and then separately check the complete programs, each composed of a primitive component and its environment.

Such an environment has to fulfill the following key requirements:

- It should be created in a way that minimizes the state space size of the program composed from the component and its environment, while at the same time, it should be complex enough to exercise the target component under all reasonable behaviors (sequences and parallel interleavings of method calls) and all combinations of input values.
- It should allow the model checker to search for concurrency errors; this is typically reflected by calling methods of the component by more threads of control.
- It should force the model checker to check all the control flow paths in the component's code; this is usually addressed by calling each method of the component several times with different combinations of parameter values - particular combination for a method invocation being selected non-deterministically via means provided by the model checker.

An environment for `Transaction Manager` (from Fig. 1) that fulfills all three requirements could take the form depicted in Fig. 2 (only fragments of its Java code are presented). From that it is clear that manual construction of the environment is tedious and error-prone process. Therefore, we aim at creating a tool that would generate the environment in an automated way, i.e. an environment generator. As an input, the environment generator will get the specification of a component's environment, which has to determine all behaviors and combinations of input values. Having the proper environment specification, the tool is able to produce a reasonable environment for component model checking, which fulfills all the requirements stated above.

```

public class EnvDbThread extends Thread
{
    ITxMngr tm;

    public void run()
    {
        String id = tm.begin(getRandomString());
        if (getRandomBool()) tm.commit(id);
        else tm.abort(id);
        ...
    }
}

public class EnvBkThread extends Thread
{
    IBackup bk;

    public void run()
    {
        bk.backup(); ...
    }
}

public static void main(String[] args)
{
    TransactionMngrImpl tm = new TransactionMngrImpl();

    tm.start();

    new EnvDbThread(tm).start();
    new EnvDbThread(tm).start();
    new EnvBkThread(tm).start();

    tm.stop();
}

```

Fig. 2. Fragments of Java code of environment for the **Transaction Manager** component

The specification itself may be divided into (i) a model of the environment's behavior and (ii) a definition of possible combinations of input values (i.e. method parameters). All of this can be provided manually by the user or retrieved, for example, from the ADL specification of the whole component-based application. In this text, we focus on modeling of the environment's behavior (our current approach to definition of possible combinations of input values is described in [10]).

1.2 Goals and Structure of the Paper

The paper aims at addressing the problem of modeling the environment for model checking of primitive software components that have their behavior specified in the formalism of behavior protocols [13]. The main goal is to present our approach to modeling of the environment's behavior, which exploits the definition of a component's architecture and specification of the component behavior via behavior protocols provided in ADL.

The remainder of the paper is organized as follows. Sect. 2 provides an overview of behavior protocols. Sect. 3 presents the key contribution - our solution to computing the model of environment's behavior from (i) the graph of bindings between components in the architecture and (ii) the behavior specifications of all the components (defined via behavior protocols) in the architecture. The rest of the paper contains evaluation, related work and a conclusion.

2 Behavior Protocols

For specification and modeling of behavior of software components, we use the formalism of behavior protocols.

A behavior protocol is an expression that specifies the behavior of a software component in terms of specific atomic events on the component's provided and required interfaces, those events being accepted and emitted method call requests and responses. Each behavior protocol defines a possibly infinite set of traces, where each trace is a finite sequence of atomic events - we use $L(prot)$ to denote the set of traces specified by a protocol $prot$. The semantics of a behavior protocol is defined in terms of labeled transition system (LTS), with transitions labeled by atomic events.

Syntactically, a behavior protocol reminds a regular expression, with a set of atomic actions working as the underlying alphabet. Each atomic event has the following syntax: $\langle prefix \rangle \langle interface \rangle . \langle method \rangle \langle suffix \rangle$. The prefix $?$ denotes an accept event, the prefix $!$ denotes an emit event, the suffix \uparrow denotes a request (i.e. a method call), and the suffix \downarrow denotes a response (i.e. return from a method). Several shortcuts, which make the protocols more readable, are also defined. For example, an expression of the form $?i.m\{prot\}$ is a shortcut for the protocol $?i.m\uparrow; prot; !i.m\downarrow$, and an expression of the form $?i.m$ is a shortcut for the protocol $?i.m\uparrow; !i.m\downarrow$. The `NULL` keyword denotes an empty protocol.

In addition to standard operators $;$ (sequence), $+$ (alternative), and $*$ (repetition), behavior protocols provide the and-parallel operator $|$, which generates all the possible interleavings of event traces defined by its operands, and the or-parallel operator $||$ ($p || q$ stands for $p + q + (p | q)$).

The component's *frame protocol* [13] describes the external behavior of the component by defining all the valid sequences of events (i.e. traces) on the component's external interfaces. For composite components, the *architecture protocol* describes the composed behavior of all subcomponents at the first level of nesting; it is generated as a parallel composition of frame protocols of the subcomponents.

The frame protocol of **Transaction Manager** (Sect. 1) is

```
( ?tm.start ; ?tm.begin* ; (?tm.begin* | ?tm.commit* |
?tm.abort*) ; ?tm.stop ) | ?bk.backup*
```

It is a parallel composition of two subprotocols. The first of them specifies that the component should accept finite number of calls of `backup` on the `bk` interface. The second subprotocol states that the component has to accept call of `start` on its `tm` interface and then a finite number of calls of `begin` on `tm`, then it should accept calls of `begin`, `commit` and `abort` on `tm` in parallel, and finally it should accept the call of `stop` on `tm`.

The frame protocol of Database might be

```
?db.start{!logger.start ; !tm.start} ;
(
  ?db.add{!tm.begin ; (!tm.commit + !tm.abort)}
  +
  ?db.get{!tm.begin ; (!tm.commit + !tm.abort)}
  +
  ?db.remove{!tm.begin ; (!tm.commit + !tm.abort)}
)* ;
?db.stop{!logger.stop ; !tm.stop},
```

the frame protocol of DBServer might be

```
?db.start ; (?db.add + ?db.get + ?db.remove)* ; ?db.stop,
```

and the frame protocol of Backup Scheduler might be `!bk.backup*`.

An advantage of using behavior protocols for specification of component's behavior is the possibility to check whether the components equipped with frame protocols are behaviorally compliant, i.e. whether the components communicate without errors. We distinguish between (i) the *horizontal compliance* of components at the same level of nesting and (ii) the *vertical compliance* of a frame of a composite component with the underlying architecture (expressed by the architecture protocol). Nevertheless, checking of behavior compliance makes sense only under the assumption that the implementation of each primitive component satisfies its frame protocol (we say that the component *obeys* its frame protocol). This holds only if the component accepts/issues only such method-call related event sequences on its external provided and required interfaces that are specified by the component's frame protocol. An obvious approach to checking whether a component obeys its frame protocol is to use code model checking; for that purpose we have a tool [11] that accepts only complete programs as input, and therefore we need to create an environment that, together with the component, makes a complete program accepted by our tool.

3 Modeling the Environment with Behavior Protocols

As indicated in Sect. 1.1, a model of the environment's behavior has to be supplied as a part of the environment specification that is provided to an environment generator. The model of the environment's behavior should reflect the fact that the resulting environment has to represent the behavior of all other components that

can possibly be bound to the target component. As an example, consider the component architecture on Fig. 1; the environment for `Transaction Manager` should represent at least the behavior of `Database` and `Backup Scheduler` with respect to `Transaction Manager`.

Our first solution to modeling of the environment’s behavior, presented in [10], uses the *inverted frame protocol* [1] of the target component, which is constructed from the component’s frame protocol by replacing all the accept events by emit events and vice versa. Such a model is the most general one, as the component’s frame protocol specifies all the sequences of events the component can accept/issue on its external (provided and required) interfaces. A drawback of this solution is that the environment generated this way can be very complex, frequently making model checking of the program composed of the component and its environment suffer from state explosion. In [10] we presented an attempt to mitigate this drawback by designing heuristic transformations and approximations of the frame protocol that simplify the resulting environment to an extent that makes checking feasible. However, a problem with this approach is that the resulting environment exercises the target component only by a subset of the behaviors defined by the component’s frame protocol; therefore, checking whether the component obeys its frame protocol is not exhaustive in such a case.

In order to solve this problem, we propose a new approach to modeling the environment’s behavior on the basis of a particular context - in our case, an architecture the component is expected to be used in. More specifically, our approach exploits (i) the definition of the architecture the target component is a part of, and (ii) the behavior specification (defined as behavior protocols) of all the components that form the architecture. Here, the basic idea is to use *context protocol* of the target component, which specifies the actual use of the target component by the other components of the architecture (and vice versa), as the model of the environment’s behavior - it is an idea similar to that of using context constraints for compositional reachability analysis of LTSs, which was presented in [3].

Using the context protocol instead of the inverted frame protocol as the model of the environment’s behavior is useful especially in the case, where a particular component-based application exploits only a subset of the functionality provided by the target component - the context protocol then specifies only a subset of behaviors determined by the inverted frame protocol, thus helping mitigate the problem of state explosion.

To illustrate the advantage of using the context protocol instead of the inverted frame protocol, consider again the architecture on Fig. 1, having the frame protocols of the `Transaction Manager`, `Database`, and `Backup Scheduler` components as presented in Sect. 2. Since the frame protocol of `Database` effectively specifies call of `begin` on its required interface `tm` followed by an alternative between calls to `commit` and `abort` on `tm`, all that repeated for a finite number of times, then, despite the fact that the frame protocol of `Transaction Manager` specifies parallel calls of those methods, the context protocol for `Transaction Manager` is

```
( !tm.start ; (!tm.begin ; (!tm.commit + !tm.abort))* ; !tm.stop )
| !bk.backup*
```

Such a context protocol for **Transaction Manager** obviously specifies a subset of behaviors determined by the component’s (inverted) frame protocol, and, therefore, model checking of **Transaction Manager** with the environment modeled by this context protocol will have lower time and space requirements, than if the inverted frame protocol was used for this purpose.

Notice also, that the behavior determined by the component’s context protocol has to be a subset of behavior specified by the component’s inverted frame protocol for the checking of a component against its frame protocol to work correctly; otherwise the model checker could report some “false errors” in addition to violations of the component’s frame protocol by its implementation, since also the traces not allowed by the frame protocol will be defined in the context protocol (with corresponding behavior being encoded in the generated environment) in this case. In other words, for the inverted frame protocol IF_C of the component C and its context protocol CTX_C , the formula $L(CTX_C) \subseteq L(IF_C)$ must hold.

3.1 Computing the Model of Environment’s Behavior

Technically, our approach is to compute a behavior protocol that models behavior of target component’s environment from (i) frame protocols of the other components at the same level of nesting, (ii) the inverted frame protocol of the parent component and (iii) the bindings between component’s interfaces; we denote the output, i.e. the model of the environment’s behavior, to be the *environment protocol* of the target component. The ideal algorithm for this purpose is the one that fulfills the following two requirements:

- It should take only a fraction of time required by actual model checking of the target component, as the task of environment construction is only a prerequisite to the process of model checking, which has big time and space requirements on its own.
- The algorithm should be precise; i.e. the resulting environment protocol should represent exactly those behaviors that can be exercised on the target component by other components taking part in the particular architecture, i.e. it should specify the same behavior like the target component’s context protocol. Representing a subset of those behaviors would prevent exhaustive model checking and representing a superset of those behaviors could possibly reduce efficiency of the checking (by increasing the state space size).

However, for the algorithm to have low time requirements (which is our top priority), it is necessary to make a compromise on the second requirement, as computing the environment protocol that specifies exactly the same behavior as the context protocol could be a very time- and space-consuming task for some inputs. In such cases, the algorithm should produce an environment protocol that is a superset of the context protocol (in terms of behavior specified by it) in an efficient way. Nevertheless, considering the inverted frame protocol IF_C of the component C , its context protocol CTX_C , and its environment protocol E_C , then the formula $L(CTX_C) \subseteq L(E_C) \subseteq L(IF_C)$ must hold. Consequently, the component’s environment protocol will specify the same behavior as its inverted frame protocol in the worst case.

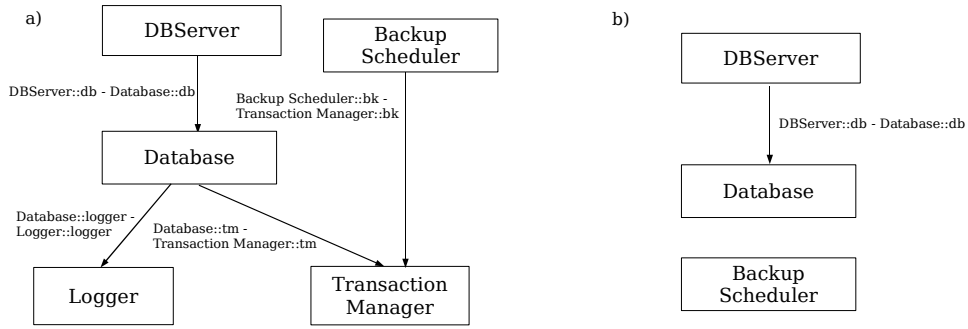


Fig. 3. a) graph of bindings between components; b) binding trees for the **Transaction Manager** component

3.1.1 Syntactical Approach

We designed an algorithm, which is based on syntactical expansion and substitution of (parts of) behavior protocols. Its input consists of the frame protocols of all the components in the architecture (except the target one) and the graph of the bindings between the components, and its output is the environment protocol of the target component. The algorithm is divided into three steps, described below.

The first step is the reduction of the graph of bindings to a subgraph that contains only the paths that start at a parent component or at a component with no provided interfaces (or with all its provided interfaces unbound) and end at a component that is bound to the target component. In fact, the subgraph is a set of acyclic graphs, which we call *binding trees* (despite the fact that some of them may actually be DAGs) - there is one binding tree for the parent component, if defined in the architecture, and one binding tree for each component with no provided interfaces (or with all its provided interfaces unbound). Note that binding trees defined in this way do not reflect bindings of the target component's required interfaces to provided interfaces of other components in the architecture. However, despite that, our algorithm supports cyclic dependencies between components, since each cycle of method calls must be initiated by a call that does not belong to the cycle - and that call will be reflected in the resulting environment protocol (although possibly indirectly in some cases).

Considering the architecture on Fig. 1, the graph of bindings on Fig. 3a, and the **Transaction Manager** component as the target one, this step of the algorithm will produce a subgraph that is depicted at the Fig. 3b. The first binding tree of the subgraph consists of two nodes - the root node corresponding to the **DBServer** component, and its child node corresponding to the **Database** component - and one edge that represents the binding between the two components, and the second binding tree consists of one node corresponding to the **Backup Scheduler** component.

In the second step, a part of the environment protocol is constructed for each binding tree via syntactical expansion of protocols during traversal of a tree in the DFS manner. The frame protocol of the component (or inverted frame protocol in case of a parent component) corresponding to the root node of a tree represents the initial version of the part of the environment protocol for the particular binding

tree. Then, when backtracking over an edge from a node A to a node B (which is the parent of A) during DFS, all bindings between the required interfaces of the component C_B (represented by the node B) and provided interfaces of the component C_A (represented by the node A) are taken, and for each of these bindings all the calls on the corresponding required interface of C_B (as defined in its frame protocol) are replaced with reactions to those calls (as defined in the frame protocol of C_A) in the current version of the part of the environment protocol. If the frame protocol of C_A specifies two or more reactions to some specific method call that are connected via the and-parallel operator, it is necessary to use all these reactions together with the connecting and-parallel operators preserved for the purpose of replacing the corresponding call.

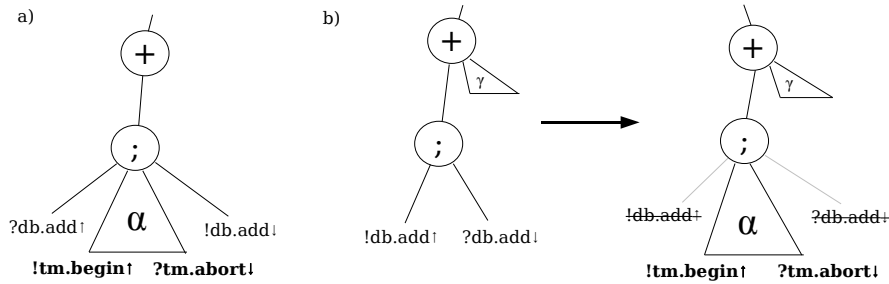


Fig. 4. a) parse tree for the frame protocol of Database; b) illustration of one step in construction of the environment protocol for Transaction Manager - replacement of call to db.add with a reaction to the call (specified in the frame protocol of Database)

For illustration of the syntactical expansion of protocols, consider the first binding tree on Fig. 3b and the Transaction Manager component as the target one. Then, when backtracking over the edge that represents the binding between DBServer and Database, the call !db.add will be expanded to a reaction to this call that is specified in the frame protocol of Database, i.e. to a subprotocol !tm.begin ; (!tm.commit + !tm.abort), as depicted on Fig. 4 (the figure showing only fragments of parse trees of the protocols).

In the third step of the algorithm, parts of the environment protocol for all binding trees are connected via the and-parallel operator, thus forming the resulting environment protocol for the target component. Using the and-parallel operator is necessary because calls delegated from the parent component (if it exists) can be performed in parallel with calls performed by components that have no provided interfaces (or have all of them unbound). Considering our example, there are two binding trees, one with the DBServer component as its root node and the second with the Backup Scheduler component as its root node; therefore, the two parts of the environment protocol that correspond to these binding trees will be connected with the and-parallel operator.

Finally, the environment protocol is simplified to contain only events that represent calls on the provided interfaces of the target component, as all other events are not relevant for modeling the environment of the target component and can be therefore safely ignored.

The output of our algorithm for the **Transaction Manager** component is depicted in Fig. 5. It is an environment protocol that specifies the same behavior as the component’s context protocol presented in Sect. 3, i.e. both protocols specify the same set of event sequences. The presence of an alternative between subprotocols of the form `(!tm.begin ; (!tm.commit + !tm.abort))` is only a syntactical difference, which could be handled by preprocessing of some form before the environment is actually generated from the environment protocol. The reason for the environment protocol to have this form, not allowing parallel invocation of methods on the `tm` interface, is that the frame protocol of **DBServer** specifies no parallelism; calls on `db` specified in the inverted frame protocol of **DBServer** are replaced with reactions to those method calls that are specified in the frame protocol of **Database**, when the environment protocol is constructed.

```
(
  !tm.start ;
  (
    (!tm.begin ; (!tm.commit + !tm.abort))
    +
    (!tm.begin ; (!tm.commit + !tm.abort))
    +
    (!tm.begin ; (!tm.commit + !tm.abort))
  )* ;
  !tm.stop
)
|
!bk.backup*
```

Fig. 5. Environment protocol for the **Transaction Manager** component

Consequence of the environment protocol for **Transaction Manager** specifying only repetition of alternative calls on the `tm` interface (i.e. real usage of the component in the given architecture) is that the environment modeled by this environment protocol will determine smaller state space (of the program composed of the component and environment) than if the component’s inverted frame protocol, which allows for parallel calls of these methods, is used for modeling of the environment’s behavior.

4 Evaluation

As already mentioned in Sect. 3, an advantage of modeling the environment’s behavior via environment protocol is that the environment protocol reflects the real usage of the target component in the specific architecture the component is used in, and, therefore, it will typically specify a subset of behaviors determined by the inverted frame protocol of the target component. On the other hand, a drawback of using the environment protocol is that checking of the component has to be performed again for each architecture the component is used in, since a different subset of behaviors defined by the component’s frame protocol may be exploited in each component architecture. In any case, checking whether the target component

obeys its frame protocol is exhaustive (with respect to the specific architecture), if the environment protocol is used - not like when the heuristic transformations of the inverted frame protocol are applied, which make checking of the component not exhaustive (although more feasible in most cases).

The algorithm for computing the environment protocol, which is described in Sect. 3.1.1, works well and produces expected results; its time and space requirements being fractional with respect to actual checking of the component. For example, the algorithm is able to detect that methods of the target component are called sequentially or alternatively in the given architecture, even though the component's frame protocol allows for parallel calls of the methods of the component (see the example in Sect. 3.1.1 for illustration). Nevertheless, it is hard to estimate the state space reduction achieved by our approach in general, as the level of reduction depends on each specific case (i.e. how the target component is used in the particular architecture); a systematic analysis of this issue is subject of our current research. We also have a proof-of-concept implementation that was tested on several examples, including the one presented in this paper.

However, our solution has also some drawbacks. One of them is that the syntax-based algorithm does not produce a correct environment protocol if the component is able to perform some calls on its required interfaces autonomously. Specifically, such calls will not be included in the resulting environment protocol, which is then incorrect because it will not force the environment to wait for these calls to happen (i.e. wait till the component issues the calls). The reason for not including autonomous calls on component's required interfaces in the environment protocol is that our algorithm does not reflect bindings of the target component's required interfaces to the provided interfaces of other components in the architecture.

Second problem of our approach is that syntactical expansion of protocols may not produce a correct result in cases, when the frame protocol of a certain component specifies more reactions to some method call that are connected via the sequence operator. In that case, it is generally not possible to decide, in an efficient way, which reaction is the appropriate one for the particular method call; on the other hand, subprotocols that represent both reactions should be typically equal in terms of behavior, since they specify the body of the same method.

Third drawback of our solution, less significant than the first two, is that our algorithm may produce an environment protocol that specifies a superset of behaviors determined by the context protocol for some inputs. Nevertheless, one of our design requirements was to develop an efficient algorithm with respect to both time and space, and this has been achieved with designing the algorithm to produce the environment protocol that specifies more behaviors than the context protocol in some cases.

5 Related work

The problem of model checking of isolated software components, which form a component-based application, can be seen as a variation of compositional model checking [5], whose basic idea is to (i) decompose a target system into several components, (ii) verify local properties of the components via model checking, and (iii)

deduce global properties of the whole system from the local properties of the components. The key point of this approach is checking properties of a composition of a selected component with a model of its environment, instead of checking properties of the isolated component; by using an environment, it is guaranteed that the checked local properties are preserved also at the global level. The difference between compositional model checking and our approach is that the former aims at checking global properties of the whole program (or a set of processes) via checking local properties of individual components (or processes), while our approach aims at checking the properties specific to individual components (e.g. obeying of a frame protocol).

For verification of properties of software components, the assume-guarantee approach [7][12] is often used. The idea is to check a component only in such environments that satisfy certain assumptions typically provided by the user; we can say that the assumptions model the valid environments of a component subject to model checking. This way, the need to check the component in all possible environments (or in a universal environment), what is usually an infeasible task, is avoided. Application of model checking to a component with an environment satisfying a certain assumption then verifies whether the component satisfies the given property under the given assumption. If the model checker returns a positive answer, it is guaranteed that the component, when used in an environment that satisfies the specific assumption, must satisfy the given property in this environment. In order for the checking of a component to be of practical use, the assumptions should together model the real environment of the component (e.g. an architecture the component is to be used in). The most popular means for expressing the assumptions is the temporal logic (LTL), commonly used also for specification of the properties.

However, in our specific case, we use the environment protocol as the assumption about the environment for model checking of components. This way, we do not have to check whether such an assumption holds for the environment actually used, as the environment is generated on the basis of the environment protocol, and therefore the assumption represented by the environment protocol holds trivially. Typical application of the assume-guarantee paradigm also requires the assumptions to be manually created by the user; in our case, we have to manually define the frame protocols of all the components in the architecture - the environment protocol is then automatically computed from these frame protocols and bindings between components. We are aware of only one automatic approach to generating the assumptions for compositional verification, which is based on incremental learning [6].

6 Summary

Direct model checking of isolated software components is typically not possible because a component does not form a complete program which is accepted as an input by a typical program model checker (the problem of missing environment). Therefore, a solution is to create an environment of some form for the component that is subject to model checking.

We proposed to model the environment on the basis of a particular component

architecture the target component is expected to be used in; the architecture being a context for the component. Specifically, since we aim at hierarchical component architectures with component behavior modeled via behavior protocols, we model the component's environment behavior with an environment protocol computed from frame protocols of other components taking part in the given architecture.

We have presented an algorithm for computing the environment protocol, which is based on syntactical expansion and substitution of frame protocols. Finally, we showed that the solution for modeling the environment's behavior on the basis of the environment protocol is more efficient than our previous approach [10] based on an inverted frame protocol; the reason for this is that the environment protocol reflects the real usage of the target component in the given architecture, while the inverted frame protocol specifies the most general environment for the component.

As for future work, we plan to extend our current algorithm for computing the environment protocol with support for components that call methods on their required interfaces autonomously.

Acknowledgments

We would like to record a special credit to Jan Kofron for valuable comments regarding the design and implementation of the algorithm for computation of the environment protocol.

References

- [1] Adamek, J., and F. Plasil, *Erroneous Architecture is a Relative Concept*, Proceedings of Software Engineering and Applications (SEA), ACTA Press, pp. 715-720, Nov 2004
- [2] Allen, R., "A Formal Approach to Software Architecture", PhD Thesis, School of Computer Science, Carnegie Mellon University, 1997
- [3] Cheung, S. C., J. Kramer, *Context Constraints for Compositional Reachability Analysis*, ACM Transactions on Software Engineering and Methodology, **5**(1996), pages 334-377, October 1996
- [4] Clarke, E. M., O. Grumberg, and D. Peled, "Model Checking", MIT Press, 2000
- [5] Clarke, E. M., D. E. Long, and K. L. McMillan, *Compositional Model Checking*, In Proceedings of the 4th Symposium on Logic in Computer Science, June 1989
- [6] Cobleigh, J. M., D. Giannakopoulou, and C. S. Pasareanu, *Learning Assumptions for Compositional Verification*, In Proceedings of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), April 2003
- [7] Giannakopoulou, D., C. S. Pasareanu, and H. Barringer, *Assumption Generation for Software Component Verification*, In Proceedings of the 17th IEEE Conference on Automated Software Engineering (ASE), IEEE CS, 2002
- [8] Magee, J., and J. Kramer, *Dynamic Structure in Software Architectures*, Proceedings of FSE'4, 1996
- [9] Medvidovic, N., *ADLs and dynamic architecture changes*, Joint Proceedings SIGSOFT'1996 Workshops, ACM Press, Oct 1996
- [10] Parizek, P., and F. Plasil, *Specification and Generation of Environment for Model Checking of Software Components*, Accepted for publication in Proceedings of FESCA 2006, ENTCS, Mar 2006
- [11] Parizek, P., F. Plasil, and J. Kofron, *Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker*, Accepted for publication in Proceedings of 30th IEEE/NASA Software Engineering Workshop (SEW-30), IEEE CS, Apr 2006
- [12] Pasareanu, C. S., M. B. Dwyer, and M. Huth, *Assume-Guarantee Model Checking of Software: A Comparative Case Study*, In Proceedings of the 6th SPIN Workshop, LNCS, **1680**(1999), 1999
- [13] Plasil, F., and S. Visnovsky, *Behavior Protocols for Software Components*, IEEE Transactions on Software Engineering, **28**(2002)

Chapter 8

Partial Verification of Software Components: Heuristics for Environment Construction

Pavel Parízek,
František Plášil

Contributed paper at **33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)**.

In conference proceedings,
published by IEEE CS,
pages 75–82,
ISBN 0-7695-2977-1,
ISSN 1089-6503,
August 2007.

The original version is available electronically from the publisher's site at <http://doi.ieeecomputersociety.org/10.1109/EUROMICRO.2007.46>.

Partial Verification of Software Components: Heuristics for Environment Construction

Pavel Parizek, Frantisek Plasil

*Charles University, Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{parizek,plasil}@dsrg.mff.cuni.cz
<http://dsrg.mff.cuni.cz>*

*Academy of Sciences of the Czech Republic
Institute of Computer Science
plasil@cs.cas.cz
<http://www.cs.cas.cz>*

Abstract

Code model checking of software components suffers from the well-known problem of state explosion when applied to highly parallel components, despite the fact that a single component typically comprises a smaller state space than the whole system. We present a technique that mitigates the problem of state explosion in code checking of primitive components with the Java PathFinder in case the checked property is absence of concurrency errors. The key idea is to reduce parallelism in the calling protocol on the basis of the information provided by static analysis searching for concurrency-related patterns in the component code; by a heuristic, some of the pattern instances are denoted as “suspicious”. Then, the environment (needed to be available since Java PathFinder checks only complete programs) is generated from a reduced calling protocol so that it exercises in parallel only those parts of the component’s code that likely contain concurrency errors.

Keywords: software components, model checking, concurrency errors, Java PathFinder, static analysis

1. Introduction

For object-oriented programs, several verification and reasoning frameworks are built around code model checkers to check whether a finite model of the code of a target program violates a desired property (reported by providing a counterexample). Such a property can be predefined in the model checker (e.g. absence of deadlocks), expressed as an external temporal logic formula, and specified as an assertion directly in the code of a program. Well-known examples of such frameworks are the SLAM model checker [3] and Java PathFinder [22], the latter being both a highly

customizable code model checker and a verification framework, which works as a special JVM upon byte code.

Model checking of complex software systems that involve high degree of parallelism is prone to the well-known state explosion problem. All viable approaches to address it are based on abstraction [5] (e.g. partial order reduction and predicate abstraction), compositional reasoning and heuristics. In particular, heuristics are used to direct the state space traversal (*directed model checking* [6]) and to identify the parts of the state space that are likely irrelevant with respect to given properties. The key goal of heuristics is to help (i) discover errors in limited time and space and (ii) report short and easy-to-read counterexamples. Even though this way *partial verification* is done in general (since some parts of the state space are omitted), heuristics perform well for verification against specific types of errors [6].

For hierarchical component-based systems with formal behavior specification, various properties specific to components can be checked, such as correctness of composition (assembly) [11], [18], and whether the code of a primitive component obeys the behavior specification. In [17], we presented a technique of code model checking of primitive software components against their behavior specification (defined via behavior protocols [18]) that is based on cooperation of the Java PathFinder (JPF) [22] with the behavior protocol checker (BPChecker) [10]. Although the approach presented in [17] typically works well, for a heavily parallel component state explosion can still occur.

1.1. Behavior Protocols

For modeling and specification of behavior of hierarchical software components, in our group, we use the formalism of behavior protocols [18] (a specific process algebra). As behavior, the set of finite traces of atomic events

corresponding to accepted and emitted method calls on component interfaces is considered. A behavior protocol $prot$ specifies a set of traces denoted as $L(prot)$: in particular, the behavior of a component on its external interfaces is defined by its *frame protocol*.

A behavior protocol reminds a regular expression upon an alphabet of atomic events, syntactically written as $\langle prefix \rangle \langle interface \rangle . \langle method \rangle \langle suffix \rangle$. The prefix $?$ means accepting, $!$ emitting, the suffix \uparrow means a request (of a method call) and \downarrow a response (return from a call). Several shortcuts are defined: $?i.m$ is a shortcut for $?i.m\uparrow$; $!i.m\downarrow$ and $!i.m$ stands for $!i.m\uparrow$; $?i.m\downarrow$. In addition to the standard regular operators ($;$, $+$, $*$), there is also $|$ (and-parallel), which generates all interleavings of the event traces defined by its operands.

Concepts presented in this paper will be illustrated on a part of the component application developed in CRE project [1] for Fractal [4] (Fig. 1). Here we are interested especially in the `TransientIpDb` and `IpAddressManager` primitive components that form a part of the `DhcpServer` composite component. The frame protocol of `TransientIpDb` (featuring the interface `IIpMacDb`) might be:

```
?IIpMacDb.Add* | ?IIpMacDb.Remove* |
?IIpMacDb.GetMacAddress* |
?IIpMacDb.GetIpAddress* |
?IIpMacDb.GetExpirationTime* |
?IIpMacDb.SetExpirationTime*
```

It states that each method can be executed repeatedly in parallel with other methods on the interface.

An advantage of frame protocols is the possibility to

check whether the components are behaviorally compliant (i.e. they communicate without errors). For that purpose behavior protocols introduce the *consent operator* ∇ , a special case of parallel composition; it supports synchronization via merging accepting and emitting events of a method call into internal events, and also identifies communication errors (deadlock and no response to a call). We have implemented the consent operator in the behavior protocol checker (BPChecker) [10].

1.2. Model Checking of Software Components and Behavior Protocols

At the first sight, code model checking of software components mitigates the state explosion problem, since a single component obviously comprises a smaller state space than the whole system. Unfortunately, this is not directly possible, since typical code model checkers, including the Java PathFinder, check only a complete program (featuring the `main`), which is not typical for a component - *problem of missing environment* [16]. A solution to it is to construct a software environment that, together with the component, makes a complete program. For this purpose, we developed the environment generator for the Java PathFinder [15]; as input, it accepts behavior specification of an environment as a behavior protocol (the component's *environment protocol*) and its output is a set of Java classes forming the environment, which communicates with the component interfaces according to the environment protocol.

An environment protocol of a primitive component can be constructed in two ways: (i) by forming the inverted frame protocol (derived from the frame protocol by replacing emit

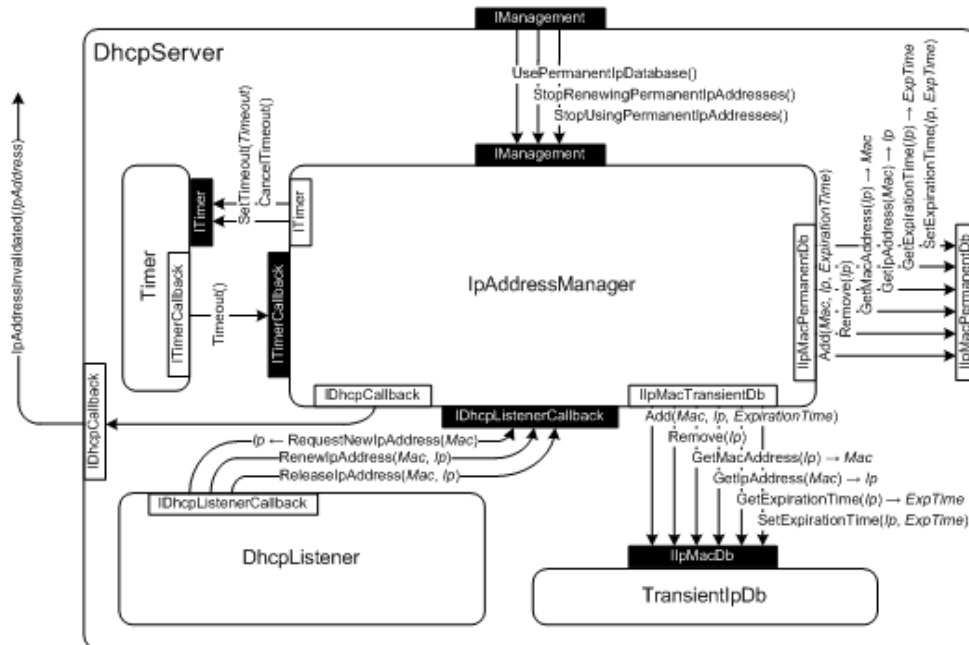


Figure 1: Architecture of the `DhcpServer` component

events with accept events and vice versa) [16], and (ii) by composition of frame protocols of other components in the particular architecture via the consent operator [14]. For illustration, the inverted frame protocol of `TransientIpDb` (and also its environment protocol) is:

```
!!IpMacDb.Add* | !!IpMacDb.Remove* |
!!IpMacDb.GetMacAddress* |
!!IpMacDb.GetIpAddress* |
!!IpMacDb.GetExpirationTime* |
!!IpMacDb.SetExpirationTime*
```

In general, an environment protocol specifies both invocations of the component’s methods by the environment (events of the form `!m`) and acceptances of component’s calls to the environment (events of the form `?n`). However, it is hard to generate environment which accepts calls according to such a protocol, since in Java there is no explicit construct for acceptance of a method call depending upon history of other calls. Fortunately, for checking the component we use JPF cooperating with BPChecker, which verifies whether both incoming and outgoing calls are done according to the frame protocol. Therefore, it is enough to generate an environment which accepts the calls in any order and just its outgoing calls respect the environment protocol. Consequently, an environment protocol can be restricted to method invocations (*calling protocol*). For example, the environment protocol `(!a;?b) | !c;(?!d+!e) | (!b+?d;!e)` is restricted to the calling protocol `!a | (!c;!e) | (!b+!e)`.

1.3. Goals and Structure of the Paper

The goal of this paper is to address the state explosion problem for code model checking of primitive components with JPF in case the checked property is absence of concurrency errors (deadlocks, race conditions). For this purpose, the paper proposes a technique to keep the state space size in “reasonable” limits by heuristically reducing the parallelism in the environment so that it exercises in parallel only those parts of the primitive component’s code which likely contain concurrency errors; these parts are identified via a static code analysis (searching for “suspicious” patterns in the component code).

An additional goal is to illustrate the feasibility of the proposed technique and its benefits (support for discovery of concurrency errors in limited time and space and provision of short and easy-to-read counterexamples) on the results of experiments performed on several primitive components.

To reflect these goals, the remainder of the paper is organized as follows. Sect. 2 presents details of the proposed technique - heuristic reductions of parallelism in the environment on the basis of information provided by a static analysis of code. Further, Sect. 3 shows experimental

results of applying the proposed technique to several primitive components and Sect. 4 provides an evaluation of the technique. The rest of the paper contains related work and a conclusion.

2. Heuristics for Environment Construction

As indicated in Sect. 1.3, the basic idea of the technique is to reduce the parallelism in the environment of a primitive component on the basis of static code analysis that identifies those parts of the component code that likely contain concurrency errors. In general, this is done in the following 4-step process which involves several heuristics:

- (1) Acquiring a calling protocol of the component subject to checking;
- (2) by static code analysis, identifying those methods of the component whose parallel executions would likely cause concurrency errors;
- (3) reducing the level of parallelism in the calling protocol so that parallel composition is preserved only between method calls identified in (2) - creating a *reduced calling protocol*;
- (4) constructing an environment corresponding to the reduced calling protocol and applying JPF to the complete program composed of the component and environment codes.

Here we focus only on (2) and (3), since the other steps are described in [14] and [16].

2.1. Identification of Methods Likely Causing Concurrency Errors

The purpose of the step (2) above is only to identify those methods of a component subject to checking, whose parallel executions likely cause concurrency errors. The algorithm for methods’ identification has to fulfill the following requirements; it has to

- (i) have low time complexity,
- (ii) support detection of deadlocks and race conditions,
- (iii) accept isolated primitive components as input,
- (iv) provide a Java API so that it can be integrated with the existing environment generator [15].

Even though there exist solutions for detection of potential concurrency errors in Java (e.g. Jlint [9] and FindBugs [8]), none of them we are aware of fulfills all these requirements. In particular, the existing solutions either accept only complete programs [20], detect only a single type of concurrency errors (typically race conditions) [20], or do not provide a Java API [9]. The proposed solution is based on searching for four concurrency-related patterns (in our experience frequently occurring in Java applications) in the byte code of pairs of methods and assigning weights (likeliness of an error) to pattern instances. The patterns are illustrated below (*synch* means *synchronized*).

The patterns (P1) and (P2) are deadlock-related. Specifically, (P1) captures nesting of synchronized blocks in reverse order, while (P2) identifies the calls to the

Object.wait and Object.notify methods that are nested inside two synchronized blocks (i.e. call of LB.notify is never reached after LB.wait was executed).

```

    m1                m2
(P1) synch (L1) {    synch (L2) {
      synch (L2) {    synch (L1) {
        ..           ..
      }              }
    }                }

(P2) synch (LA) {    synch (LA) {
      synch (LB) {    synch (LB) {
        LB.wait();    LB.notify();
      }                }
    }                  }

```

The patterns (P3) and (P4) are race conditions-related. In particular, (P3) captures the situation when reading and writing to the same attribute is possible simultaneously due to synchronized blocks guarded by locks of different objects, and (P4) identifies unsynchronized accesses to a shared attribute, for instance via unsynchronized calls to methods of Java collection classes (e.g. HashMap, LinkedList, or TreeSet).

```

    m1                m2
(P3) X x;           Y y;
      synch (x) {    synch (y) {
        this.attr = .. .. = this.attr;
      }              }

(P4) List ll = ..   List ll = ..
      ll.add("abc"); ll.remove(1);

```

The weight of each pattern instance reflects the likeliness of the corresponding concurrency error occurrence (e.g. if in P1 the types t1 of L1 and t2 of L2 differ, then an error is more likely than when they are the same, since different types imply different objects - a consequence of this is the nesting of synchronized blocks in reverse order). The total weight of a pair of methods <m1, m2> is determined as the sum of weights of all the pattern instances identified in the method pair. The actual values of the weights are determined by a *weight function* upon classes of instances of P1-P4 providing values from the range <0,1> (the lower the value the smaller likeliness of an error; zero means no likeliness). The function is to be provided by the user. Based on a series of experiments, we have “tuned up” the function specified in Table 1, where the classes are determined by the relation of types t1 and t2.

The algorithm, which locates a specific pattern (one of P1-P4) in the code and assigns weights to its instances, is further denoted as a *heuristic detector*. Implementation of a detector is based on the ASM library [2].

Table 1: Weights of concurrency-related patterns

pattern	P1 (t1 = t2)	P1 (t1 != t2)	P2	P3 (t1 = t2)	P3 (t1 != t2)	P4 (t1 = t2)	P4 (t1 != t2)
weight	0.3	1	0.5	0.25	0.8	0.25	0.9

2.2. Creating a Reduced Calling Protocol

The basic idea of the step (3) (beginning of Sect. 2) is to reduce the number of occurrences of parallel compositions in the calling protocol by replacing a parallel operator with an explicit specification of method calls interleaving via simplified sequencing. However, the reduced calling protocol has to preserve the parallel compositions involving methods identified in the step (2) as likely containing concurrency-related errors (Sect. 2.1). More precisely, the proposed technique reduces a calling protocol of the form

$$\text{InitP} ; (p_1 | p_2 | \dots | p_N) ; \text{FinishP} \quad (\text{I})$$

where InitP, FinishP and all p_i are calling protocols.

Three types of reduction are proposed: *sensitive composition*, *recursive reduction of parallelism*, and *parallel prefixes*. All these reductions accept as input a calling protocol, e.g.

```
!init; (!a | (!c; !e) | (!b; !e)); !finish (i)
```

The output of each reduction of a calling protocol CP is a reduced calling protocol CP_{red} , which may be syntactically very different. However, each trace in $L(CP_{red})$ has to be a prefix of a trace from $L(CP)$ (i.e. $\forall t_{red} \in L(CP_{red}) \exists t \in L(CP) \exists t_{suf} : t = t_{red} t_{suf}$) so that behavior not allowed by CP is not present in CP_{red} . For sensitive composition and recursive reduction of parallelism, the prefixes correspond to complete traces (i.e. t_{suf} is the empty string so that $L(CP_{red}) \subseteq L(CP)$), while for parallel prefixes, t_{suf} is not empty and $L(CP_{red})$ contains proper prefixes.

The key idea of the *sensitive composition* is as follows: $(p_1 | p_2 | \dots | p_N)$ in (I) is replaced by

$$(\dots; p_{k-1}; p_k; p_{k+1}; \dots; (p_i | p_j)) + (\dots; p_{k-1}; p_k; p_{k+1}; \dots; (p_i | p_j)) + \dots + (p_i; \dots; p_N) \quad (\text{II})$$

where an alternative with the parallel operator is introduced for any protocol tuple $\langle p_i, p_j \rangle$ such that its cumulative weight (explained below) is non-zero; basically, p_i and p_j contain methods involving instances of patterns P1-P4. The sequence $\dots; p_{k-1}; p_k; p_{k+1}; \dots$ contains all of the protocols p_1, \dots, p_N except for p_i and p_j . The last alternative, purely “sequential”, is introduced only if there is a tuple with zero cumulative weight. Notice that replacement of parallel composition by sequencing is very simplified: each alternative specifies a set of traces with a common prefix followed by interleavings of events described by $(p_i | p_j)$. This reflects the fact that the only “sensitive” (likely producing concurrency errors) protocols in the alternative are p_i and p_j . The sequence $\dots; p_{k-1}; p_k; p_{k+1}; \dots$ is intentionally chosen as a prefix (not a postfix) of $(p_i | p_j)$ to exclude this sequence from JPF backtracking triggered by execution of all interleavings of $(p_i | p_j)$. Sensitive

composition is illustrated on the following example. Given the protocol (i), all the tuples are:

- <!a, (!c;!e)> (ii) cum. weight 1.3
- <!a, (!b+!e)> (iii) cum. weight 0.25
- <(!c;!e), (!b+!e)> (iv) cum. weight 0

For each protocol tuple, all pairs of methods, whose calls are specified in the tuple, are identified; for the tuple (ii) those are <!a, !c> and <!a, !e>. By applying the heuristic detectors to the code of these pairs, the weight of each pair is acquired (0.5 for <!a, !c> and 0.8 for <!a, !e>). The cumulative weight of a protocol tuple is determined as the sum of weights of all its method pairs, i.e. the weight of (ii) is 1.3. The alternatives from (II) are determined by the cumulative weights of the tuples as follows:

- (!b+!e) ; (!a | (!c;!e)) (ii')
- (!c;!e) ; (!a | (!b+!e)) (iii')
- !a ; ((!c;!e) ; (!b+!e)) (iv')

Thus, the reduced calling protocol takes the form

$$\begin{aligned} & !init; ((!b+!e) ; (!a | (!c;!e))) + \quad (v) \\ & (!c;!e) ; (!a | (!b+!e)) + \\ & !a ; ((!c;!e) ; (!b+!e)); !finish \end{aligned}$$

The basic idea of the *recursive reduction of parallelism* is that $(p_1 | p_2 | \dots | p_N)$ in (I) is replaced by

$$p_{k+1}; \dots; p_N; (p_1 | \dots | p_k) \quad (III)$$

where each p_m from $p_{k+1}; \dots; p_N$ is removed from the parallel composition in one step of the reduction; i.e. after the first step of reduction (where $m = N$), the protocol takes the form $p_N; (p_1 | \dots | p_{N-1})$. Reduction is performed as long as the proportional weight of p_m is lower than a user-defined threshold; this assumes that ordering of $p_1 | \dots | p_N$ is determined by their proportional weights (p_1 having the highest and p_N the lowest one). The proportional weight of p_m is determined as the sum of cumulative weights of the tuples $\langle p_i, p_m \rangle$ and $\langle p_m, p_j \rangle$, where $1 \leq i, j < m$, divided by the sum of cumulative weights of all the tuples $\langle p_i, p_j \rangle$ over $\{p_1, \dots, p_m\}$, where $i \neq j$. Recursive reduction of parallelism is illustrated on the following example. Given the protocol (i), the proportional weights of the protocols !a, (!c;!e), (!b+!e) have to be determined. Since (i) contains the tuples (ii), (iii) and (iv), having cumulative weights 1.3, 0.25 and 0, the proportional weights of the protocols !a, (!c;!e), (!b+!e) are 1, 0.84 and 0.16. Thus, (i) can be reduced to (!b+!e) ; (!a | (!c;!e)) in one step, as the proportional weight of !b+!e is lower than the threshold set to 0.2 (on the basis of a number of experiments).

Both the sensitive composition and recursive reduction of parallelism preserve InitP and FinishP in CP_{red} , since $L(CP_{red}) \subseteq L(CP)$ holds for these reductions. However, InitP may be typically empty if the component has no explicit initialization phase. The *parallel prefixes* reduction takes advantage of this by considering only prefixes of traces in $L(CP)$ which start with interleavings of protocol tuples that have non-zero cumulative weight. Assuming that InitP in (I) is empty, the basic idea is to replace $(p_1 | \dots | p_N)$ by

$$(p_i | p_j) + (p_i | p_j) + \dots \quad (IV)$$

where an alternative is introduced for any tuple $\langle p_i, p_j \rangle$ such that its cumulative weight is non-zero (weights evaluated as in case of sensitive composition); naturally, the “rest” of traces in $(p_1 | p_2 | \dots | p_N)$; FinishP is not considered. The inherent assumption is that concurrency errors will be discovered by considering only the prefixes of traces in $L(CP)$ (supposing InitP is empty). For illustration, consider protocol (i). Assuming it is modified by eliminating !init, the following protocol tuples are acquired:

- <!a, !c> (vi) cum. weight 0.5
- <!a, !e> (vii) cum. weight 0.8
- <!a, !b>, <!b, !c>, <!b, !e>, <!c, !e> (viii), cum. weight 0

Since only the tuples with non-zero cumulative weight are considered in (IV), the result is (!a | !c) + (!a | !e).

3. Tools & Experiments

This section describes the experiments that we performed to show the impact of the proposed reductions on time and space complexity of component checking with JPF. For that purpose, we have created a prototype tool that supports all the proposed reductions of parallelism and provides heuristic detectors for all the patterns P1-P4.

In search for real-life examples of concurrency errors in the code, we have manually examined a number of components, ranging from those of the demo application developed in [1] to those from the Perseus project [19]. Typically, “interesting” components contained pattern instances in the combinations {P1, P2} and {P3, P4}. The components are listed in Tab. 2 and Tab. 3. Since *Pessimistic Concurrency Manager* was the only strong deadlock-prone candidate, we created a testing component (*OrderProcessor*) where we injected several deadlocks.

The tables show for each of these pattern combinations and the analyzed component characteristics of several JPF runs, each of them for the environment generated by a different reduction (including none) of the component’s calling protocol. For each environment, two variants of JPF runs were measured - first, for the standard DFS algorithm for state space traversal and, second, for the heuristic search (HS, [6]) that maximizes thread interleavings.

The run characteristics are: the total number of states traversed by JPF, length of the provided counterexample, elapsed time to find the first error and size of memory. Detailed description of the discovered errors is in [13].

The reason for not performing experiments with parallel prefixes for *IpAddressManager* is that its calling protocol has the InitP part non-empty.

4. Evaluation

Results of the experiments (in Tab. 2 and 3) show that the proposed reductions make discovery of concurrency errors in the code of primitive components with JPF more feasible by lowering the time and space complexity. Moreover, shorter

and easy-to-read counterexamples are provided, since less parallelism (i.e. parallel interleavings of fewer threads) has to be modeled by JPF and therefore the path to an error state is typically shorter than if no reduction is applied. Surprisingly, when heuristic search was applied, JPF reported only the last transition of the counterexample (1) in the tables) - likely a bug.

An obvious question is (a) which of the reductions should be applied in the checking process and (b) in which order. Since there is no simple relation among the languages $L(CP_{red_pp})$, $L(CP_{red_sc})$ and $L(CP_{red_rrp})$ for a particular CP , an obvious answer to (a) is all of them, while as to (b) the speed assessment indicated by the experiments from Tab. 2 and Tab. 3 might be the driving factor (CP_{red_pp} means the result of parallel prefixes reduction of CP , etc.). Therefore, we recommend to apply the reductions in the following order: (i) parallel prefixes; after no error was discovered by a run of JPF with the environment generated from CP_{red_pp} , similar steps are to be taken for (ii) sensitive composition and (iii) recursive reduction of parallelism (no particular order of preference of these two). If an error is discovered, after it is fixed the same reduction is to be repeated in the checking process. In general, since traces from $L(CP_{red})$ are only prefixes of (not all) traces from $L(CP)$, a JPF run upon a component with the environment generated from CP_{red} may not find all the errors that would be identified with the environment generated from CP ; this was the case of

checking `IpAddressManager` for race conditions (Tab. 2b). Therefore, (iv) “no reduction” is also to be applied, however it might not be feasible for components with heavily parallel behavior (Tab. 2c).

As to patterns, another question is (a) in which order and (b) combinations they are to be applied. The answer to (a) is easy: they do not directly depend on each other so that there is no recommended order. As for (b), there is a trade-off: the more patterns are applied, the higher the cumulative weights of tuples (Sect. 2) and therefore the resulting CP_{red} contains more parallelism. The other side of the coin is the more parallelism the higher the complexity of JPF checking. As a compromise, the combinations {P1, P2} and {P3, P4} are feasible since instances of both the patterns in a combination are not likely to be detected at the same time.

It may seem that heuristic detectors are sufficient for discovery of concurrency errors of specific types in the code (i.e. there is no need to run JPF to find such errors). However, heuristic detectors can issue both false positives and negatives, since the pattern detection is undecidable in general (e.g. consider that the types t1 and t2 in P1 are available statically, while the actual instances L1 and L2 only at runtime). Thus, JPF has to be used to decide whether there are “real” concurrency errors in the code.

It should be emphasized that p_i in (I) (Sect. 2.2) are general calling protocols, so that if p_i takes again the form `InitP ; (p1 | p2 | ... | pN) ; FinishP`, the reduction can be applied

Table 2: Detection of race conditions (patterns P3 and P4)

	No reduction	No reduction (HS)	Parallel prefixes	Parallel prefixes (HS)	Sensitive composition	Sensitive compos. (HS)	Recursive red. of parallelism	Recursive red. of parallelism (HS)
a) in <code>TransientIpDb</code> (project: <i>CRE</i> , size: 65 lines of code (loc) in Java)								
No. of states	1189	-	865	261355	16849	-	1189	-
Length of CE	61	-	25	no error	41	-	61	-
Time in seconds	2	-	2	165	15	-	2	-
Memory in MB	7	out of memory	7	167	8	out of memory	7	out of memory
b) in <code>IpAddressManager</code> (project: <i>CRE</i> , size: 240 loc in Java)								
No. of states	105652	-	-	-	172245	171537	155644	156067
Length of CE	44	-	-	-	no error	no error	no error	no error
Time in seconds	199	-	-	-	332	327	264	265
Memory in MB	13	out of memory	-	-	19	308	14	259
c) in <code>Pessimistic Concurrency Manager</code> (project: <i>Perseus</i> , size: 400 loc in Java)								
No. of states	-	-	877233	1129069	172	-	-	-
Length of CE	JPF failed	-	no error	no error	50	-	JPF failed	-
Time in seconds	-	-	505	550	1	-	-	-
Memory in MB	-	out of memory	25	500	11	out of memory	-	out of memory

recursively. This recursive application of the reduction technique was tested only on “toy” components, since we found it hard to obtain any real-life component with behavior featuring nested parallelism.

A drawback of the proposed technique is that all the patterns P1-P4 involve just two methods, i.e. concurrency errors that span more methods are not considered. Also, the selected four patterns naturally do not cover all possible concurrency problems in Java; therefore, our prototype tool is extensible so that more patterns can be easily added.

5. Related work

In particular, we are not aware of any other technique that addresses the state explosion in code checking of software components via application of heuristics (for reduction of parallelism) when constructing a component environment (in typical code model checkers, heuristics are used to guide state space traversal). The Bandera Environment Generator (BEG) [21] can generate an environment for sets of Java classes; however, the environment’s behavior specification has to be provided by the user (i.e. it is not derived from the component’s behavior specification), who ad-hoc determines the level of parallelism in the environment.

While there are very few techniques for component environment generation, a lot of related research has been done in detection of concurrency errors in the code. This includes (i) static analysis upon an abstraction of the code (e.g. Chord [12]), (ii) dynamic detection of errors during a run of an instrumented program (e.g. Eraser [20]), (iii) search for predefined bug patterns in the code (e.g. Jlint [9] and FindBugs [8]), and (iv) model checking (e.g. SLAM [3] and Java PathFinder [22]). In general, each technique based on (i)-(iii) reports false positives and misses some of the concurrency errors that are discovered by other such

techniques. Specifically, static analysis suffers from over-abstraction of the code and reporting false positives (spurious errors). Even though (ii) reports no false positives, it checks only selected execution paths, consequently not discovering all errors. Despite that tools searching for predefined bug patterns (iii) typically report false positives and fail to identify all errors, they are used in practice because of their low time and space complexity. Short characteristics of the selected tools based on (i)-(iii) are below.

The Chord tool [12] is a static detector of race conditions that combines four different techniques of static analysis (e.g. call graph construction and lock analysis) in order to minimize the number of false positives it reports.

Popular dynamic detector of race conditions is Eraser [20], which uses the well-known lockset algorithm. This tool can be applied only to binary executables.

The generic FindBugs tool [8] locates predefined patterns via a combination of linear byte code scan and data- and control-flow analysis. It aims at detection of all kinds of errors in Java code (e.g. null pointer dereference), but it has a limited support for concurrency errors - it is focused rather on incorrect usage of Java concurrency-related API (e.g. `Thread.run()` is used instead of `Thread.start()`). In a similar vein, the generic Jlint tool [9] searches for instances of predefined bug patterns in byte code; unlike FindBugs, it can detect potential deadlocks and race conditions within the inherent limits of static pattern analysis.

A technique similar to what we proposed in Sect. 2 and 3 is the combination of runtime analysis with model checking [7], where the purpose of runtime analysis is to detect potential race conditions and deadlocks. The model checker (JPF) is used to check whether the potential errors detected by runtime analysis are real or not. While our technique is based on a specific generation of environment (needed to make a component complete program anyway) focused on

Table 3: Detection of deadlocks (patterns P1 and P2)

	No reduction	No reduction (HS)	Parallel prefixes	Parallel prefixes (HS)	Sensitive composition	Sensitive compos. (HS)	Recursive red. of parallelism	Recursive red. of parallelism (HS)
a) in <code>OrderProcessor</code> (testing component, size: 100 loc in Java)								
No. of states	77133	29713	359	788	1526	8428	1527	1361
Length of CE	52	(1)	19	(1)	36	(1)	37	(1)
Time in seconds	28	14	1	3	2	7	2	2
Memory in MB	6	86	5	5	5	8	5	7
b) in <code>Pessimistic Concurrency Manager</code> (project: <i>Perseus</i> , size: 400 loc in Java)								
No. of states	142	-	54	4990	90	25449	93	7372
Length of CE	121	-	33	(1)	69	(1)	72	(1)
Time in seconds	1	-	1	5	1	44	1	11
Memory in MB	8	out of memory	5	13	11	47	10	26

concurrency errors identified by static analysis, the technique [7] directs JPF checking of a complete program to focus on particular concurrency errors identified in a specific preceding run.

6. Conclusion and future work

In this paper, we addressed the state explosion problem encountered in JPF code model checking of primitive software components in case the checked property is absence of concurrency errors. Since JPF checks only complete programs, an environment has to be provided for a component to make it a complete program. In [16, 14], we described how such an environment can be generated from the behavior specification of the component and of its deployment context (specifically, from its calling protocol). The key idea is to reduce parallelism in the calling protocol on the basis of the information provided by static analysis of the component code, searching for concurrency-related patterns; by a heuristic, some of these patterns are denoted as “suspicious”. Then, the environment is generated in such a way that it exercises in parallel only those parts of the component’s code that likely contain concurrency errors.

By results of several experiments, we have shown that the main benefit of the proposed three reductions of calling protocol is the possibility to generate an environment allowing discovery of concurrency errors via JPF with reasonably low time and space complexity. Even though use of these reductions may prevent discovery of some of the errors (which would be detected when no reduction was employed), there is a trade-off: checking with no reduction likely provides no result, since state explosion occurs.

As a future work, we plan to generalize the proposed technique with support for byte code patterns involving an arbitrary number of parallel methods; this way, the static code analysis should be able to detect more potential concurrency errors. In addition, we will focus on more elaborated definition of the weight function - with respect to (i) specific code features (like the number of attributes shared by methods), and (ii) probability of parallel execution of component’s methods.

Acknowledgments

This work was partially supported by the Grant Agency of the Czech Republic (project number 201/06/0770). Special credit also goes to Pavel Jezek for his key role in designing the demo application in [1].

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
- [2] ASM: Java bytecode manipulation framework, <http://asm.objectweb.org>
- [3] T. Ball, S. K. Rajamani: The SLAM Project: Debugging System Software via Static Analysis, POPL 2002, ACM
- [4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J. B. Stefani: The FRACTAL component model and its support in Java. *Softw., Pract. Exper.* 36(11-12), 2006
- [5] E. Clarke, O. Grumberg, and D. Peled: Model Checking, MIT Press, Jan 2000
- [6] A. Groce, W. Visser: Heuristics for Model Checking Java Programs, Proceedings of the 9th International SPIN Workshop on Model Checking of Software, 2002
- [7] K. Havelund: Using Runtime Analysis to Guide Model Checking of Java Programs, In SPIN Model Checking and Software Verification, LNCS 1885, 2000
- [8] D. Hovemeyer, W. Pugh: Finding Bugs is Easy, ACM SIGPLAN Notices, vol. 39, pages 92-106, Dec 2004
- [9] Jlint, <http://artho.com/jlint/>
- [10] M. Mach, F. Plasil, and J. Kofron: Behavior Protocol Verification: Fighting State Explosion, IJIS, Vol.6, Number 1, ACIS, ISSN 1525-9293, pp. 22-30, 2005
- [11] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer: Specifying Distributed Software Architectures. Proc. 5th European Software Engineering Conference
- [12] M. Naik, A. Aiken, and J. Whaley: Effective Static Race Detection for Java, In Proceedings of PLDI’06, ACM
- [13] P. Parizek, F. Plasil: Heuristic Reduction of Parallelism in Component Environment, Tech. Report No. 2007/2, Dep. of SW Engineering, Charles University, Mar 2007
- [14] P. Parizek, F. Plasil: Modeling Environment for Component Model Checking from Hierarchical Architecture, Accepted for publication in Proceedings of FACS’06, ENTCS, Sep 2006
- [15] P. Parizek: Environment Generator for Java PathFinder, <http://dsrg.mff.cuni.cz/projects/envgen>
- [16] P. Parizek, F. Plasil: Specification and Generation of Environment for Model Checking of Software Components, Accepted for publication in Proceedings of FESCA 2006, ENTCS, 2006
- [17] P. Parizek, F. Plasil, and J. Kofron: Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker, SEW’06, IEEE CS
- [18] F. Plasil, S. Visnovsky: Behavior Protocols for Software Components, IEEE Trans. on Soft. Eng. 28(11), 2002
- [19] Perseus project, <http://perseus.objectweb.org>
- [20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson: Eraser: A Dynamic Data Race Detector for Multithreaded Programs, ACM Transactions on Computer Systems, 1997
- [21] O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu: Automated Environment Generation for Software Model Checking, Proceedings of ASE’03, 2003
- [22] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda: Model Checking Programs, Automated Software Engineering Journal, vol. 10, no. 2, Apr 2003

Chapter 9

Modeling of Component Environment in Presence of Callbacks and Autonomous Activities

Pavel Parízek,
František Plášil

Contributed paper at **46th TOOLS Conference on Objects, Models, Components, Patterns (TOOLS EUROPE 2008)**.

Accepted for publication in conference proceedings,
to be published by Springer-Verlag,
LNBIP,
June 2008.

Modeling of Component Environment in Presence of Callbacks and Autonomous Activities

Pavel Parizek¹, Frantisek Plasil^{1,2}

*¹Charles University in Prague, Faculty of Mathematics and Physics,
Department of Software Engineering, Distributed Systems Research Group
{parizek,plasil}@dsrg.mff.cuni.cz
<http://dsrg.mff.cuni.cz>*

*²Academy of Sciences of the Czech Republic
Institute of Computer Science
{plasil}@cs.cas.cz
<http://www.cs.cas.cz>*

Abstract. A popular approach to compositional verification of component-based applications is based on the assume-guarantee paradigm, where an assumption models behavior of an environment for each component. Real-life component applications often involve complex interaction patterns like callbacks and autonomous activities, which have to be considered by the model of environment's behavior. In general, such patterns can be properly modeled only by a formalism that (i) supports independent atomic events for method invocation and return from a method and (ii) allows to specify explicit interleaving of events on component's provided and required interfaces - the formalism of behavior protocols satisfies these requirements. This paper attempts to answer the question whether the model involving only events on provided interfaces (calling protocol) could be valid under certain constraints on component behavior. The key contribution are the constraints on interleaving of events related to callbacks and autonomous activities, which are expressed via syntactical patterns, and evaluation of the proposed constraints on real-life component applications.

Key words: assume-guarantee reasoning, behavior protocols, modeling of environment behavior, callbacks, autonomous activities

1 Introduction

Modern software systems are often developed via composition of independent components with well-defined interfaces and (formal) behavior specification of some sort. When reliability of a software system built from components is a critical issue, formal verification such as program model checking becomes a necessity. Since model checking of the whole complex ("real-life") system at a time is prone to state explosion, compositional methods have to be used. A basic idea of compositional model checking [6] is the checking of (local) properties of isolated components and inferring (global) properties of the whole system from the local

properties. This way, state explosion is partially addressed, since a single isolated component typically triggers a smaller state space compared to the whole system.

A popular approach to compositional model checking of component applications is based on the assume-guarantee paradigm [18]: For each component subject to checking, an assumption is stated on the behavior of the component’s environment (e.g. the rest of a particular component application); similarly, the “guarantee” are the properties to hold if the component works properly in the assumed environment (e.g. absence of concurrency errors and compliance with behavior specification). Thus, a successful model checking of the component against the properties under the specific assumption guarantees the component to work properly when put into an environment modeled by the assumption.

Specific to program model checkers such as Java PathFinder (JPF) [21] is that they check only complete programs (featuring `main()`). Thus checking of an isolated component (its implementation, i.e. for instance of its Java code) is not directly possible ([17], [10]), since also its environment has to be provided in the form of a program (code). Thus, program model checking of a primitive component is associated with the *problem of missing environment* [14]. A typical solution to it in case of JPF is to construct an “artificial” environment (Java code) from an assumption formed as a behavior model as in [14][20], where the behavior model is based on LTS defined either directly [10], or in the formalism of behavior protocols [19]. Then, JPF is applied to the complete program composed of the component and environment.

In general, real-life component applications feature circular dependencies among components involving complex interaction schemes. Nevertheless, for the purpose of program model checking of an isolated component, these schemes have to be abstracted down to interaction patterns between the component and its environment pairs. Based on non-trivial case studies [1][8], we have identified the following four patterns of interaction between a component C and its environment E to be the most typical ones (*C-E patterns*):

a) *synchronous callback* (Fig. 1a), executed in the same thread as the call that triggered the callback;

b) *asynchronous callback* (Fig. 1b), executed in a different thread than the trigger call;

c) *autonomous activity* (Fig. 1c) on a required interface, which is performed by an inner thread of the component;

d) *synchronous reaction* (Fig. 1d) to a call on a component’s provided interface.

In Fig. 1, each of the sequence diagrams contains activation boxes representing threads (T_1 and T_2) running in the component and environment in a particular moment of time. More specifically, in Fig. 1a, m denotes a method called on the component by the environment, and τ denotes the trigger (invoked in m) of the callback b ; note that all calls are performed by the same thread (T_1). As to Fig. 1b, the only difference is that the callback b is asynchronous, i.e. it is performed by a different thread (T_2) than the trigger τ . In case of Fig. 1c, the method s called on the component by the environment (in thread T_1) starts an autonomous activity performed by an inner thread (T_2), which calls the method a of the environment.

The latter overlaps with the call of m issued by the environment. Finally, in Fig. 1d, r denotes a synchronous reaction to the call of the method m issued by the environment (both performed in the same thread).

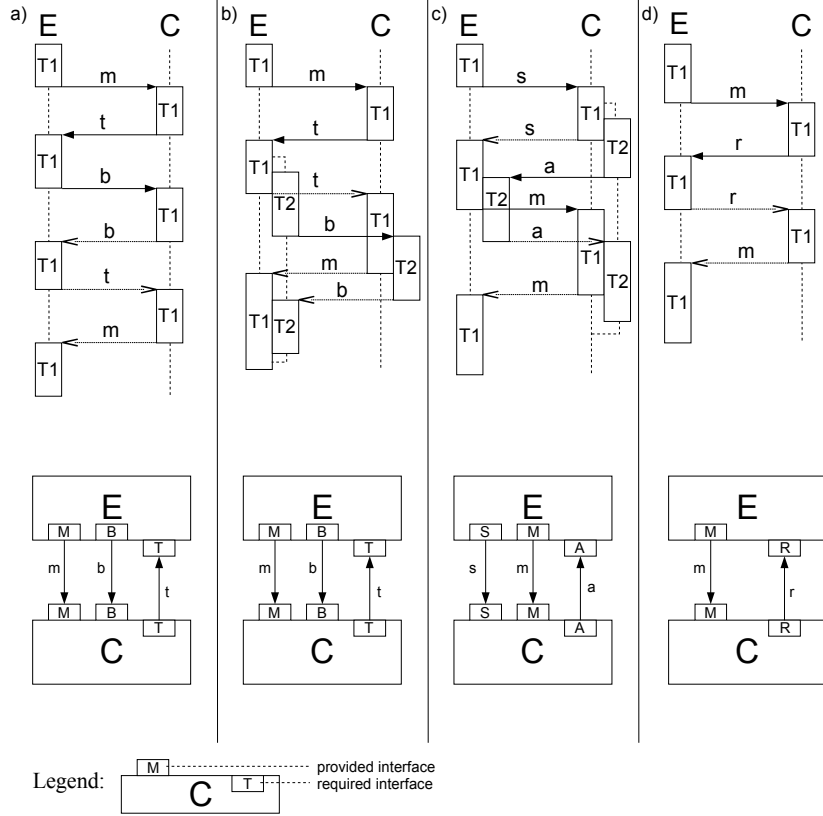


Figure 1: Interaction patterns (C-E patterns) between a component and its environment

These sequence diagrams clearly show that proper modeling of these C-E patterns via a specific formalism is possible only if the formalism allows to explicitly model the interleaving of method invocations and returns from methods on c 's provided and required interfaces. Specifically, a method call as whole cannot be modeled as an atomic event (like in [10]); instead, independent constructs for method invocation (*invocation event*), return from a method (*return event*) and method execution (*method body*) have to be supported by the formalism. We say that a model of environment's behavior is *valid* if it precisely describes all occurrences of the C-E patterns in the interaction between a component and its environment.

The formalism of behavior protocols [19], developed in our group, supports independent invocation and return events on c 's provided and required interfaces (details in Sect. 2) and therefore allows to model all the C-E patterns properly. In our former work, we introduced two specific approaches to modeling of environment's behavior: *inverted frame protocol* [14] and *context protocol* [15].

Both of them are generic, i.e. not limited to any particular communication pattern between \mathbb{E} and \mathbb{C} , and valid. The key difference between these two is that the inverted frame protocol models \mathbb{E} that exercises \mathbb{C} in all ways it was designed for (*maximal-calling environment*), while the context protocol models the actual use of \mathbb{C} in the given context of a component-based application (*context environment*). Specifically, the context protocol may be simpler than the inverted frame protocol, e.g. in terms of level of parallelism (i.e. the assumption on environment behavior is weaker), if the particular application uses only a subset of \mathbb{C} 's functionality.

Unfortunately, the actual JPF model checking of a component combined with an environment determined by any of these two modeling approaches is prone to state explosion, in particular for two reasons:

(1) Java code of \mathbb{E} is complex, since it has to ensure proper interleaving of the events on \mathbb{C} 's provided interfaces triggered by \mathbb{E} with the events on \mathbb{C} 's required interfaces triggered by \mathbb{C} itself. Technically, since there is no direct language support for expressing acceptance of a method call depending upon calling history in Java, the interleaving has to be enforced indirectly, e.g. via synchronization tools (`wait`, `notify`) and state variables.

(2) As to the context environment, its construction is also prone to state explosion, since the context protocol is derived from behavior specifications of the other components in a particular component application via an algorithm similar to the one employed in behavior compliance model checking [11]. In [15] we presented a syntactical algorithm for derivation of a context protocol, which has a low time and space complexity; however, it does not support autonomous activities and does not handle cycles in architecture (and thus callbacks) properly in general.

The issues (1) and (2) are particularly pressuring when \mathbb{C} is designed to handle a high level of parallel activities (threads). Then, this has to be reflected in \mathbb{E} to exercise \mathbb{C} accordingly. To alleviate the state explosion problem associated with these issues we proposed in [16] a simplified approach to modeling environment behavior: *calling protocol*. Roughly speaking, a calling protocol models precisely only the events on \mathbb{C} 's provided interfaces, i.e. it models only the calls issued by \mathbb{E} , under the assumption that the calls issued by \mathbb{C} are accepted by \mathbb{E} at any time (and in parallel) - this is an overapproximation of the desired behavior of \mathbb{E} . Thus the Java code of \mathbb{E} is simple, since it does not have to ensure proper interleaving of the events on \mathbb{C} 's provided and required interfaces. On the other hand, capturing this interleaving is necessary for an appropriate modeling of the C-E patterns in general, and thus for validity of a calling protocol-based model of environment's behavior. An open question is whether there are constraints on behavior of \mathbb{C} under which the calling protocol-based approach could provide a valid model of \mathbb{E} .

1.1 Goals and Structure of the Paper

The goal of this paper is to answer the question whether the calling protocol-based approach can provide a valid model of environment behavior in the context of the

C-E patterns, if, in the component’s behavior specification, certain constraints are imposed on the sequencing and interleaving of the C-E events with other events on the component interfaces.

The structure of the paper is as follows. Sect. 2 provides an overview of the formalism of behavior protocols and its use for modeling of environment behavior. Sect. 3 presents the key contribution of the paper - an answer to the question of validity of the calling protocol-based approach under certain constraints on component behavior and an algorithm for automated construction of a valid calling protocol-based model of environment’s behavior. Sect. 4 shows experimental results and the rest contains evaluation, related work and a conclusion.

2 Behavior Protocols

The formalism of behavior protocols - a simple process algebra - was introduced in [19] as a means of modeling behavior of software components in terms of traces of atomic events on the components’ external interfaces. Specifically, a *frame protocol* FP_C of a component C is an expression that defines C ’s behavior as a set $L(FP_C)$ of finite traces of atomic events on its provided and required interfaces.

Syntactically, a behavior protocol reminds a regular expression over an alphabet of atomic events of the form $\langle \text{prefix} \rangle \langle \text{interface} \rangle . \langle \text{method} \rangle \langle \text{suffix} \rangle$. Here, the prefix $?$ denotes acceptance, while $!$ denotes emit; likewise, the suffix \uparrow denotes a method invocation and \downarrow denotes a return from a method. Thus, four types of atomic events are supported: $!i.m\uparrow$ denotes emitting of a call to method m on interface i , $?i.m\uparrow$ acceptance of the call, $!i.m\downarrow$ emitting of return from the method, and, finally, $?i.m\downarrow$ denotes acceptance of the return. Several useful shortcuts are also defined: $!i.m\{P\}$ stands for $!i.m\uparrow ; P ; ?i.m\downarrow$ (*method call*), and $?i.m\{P\}$ stands for $?i.m\uparrow ; P ; !i.m\downarrow$ (*method acceptance*). Both in $!i.m\{P\}$ and $?i.m\{P\}$, a protocol P models a *method body* (possibly empty). As for operators, behavior protocols support the standard regular expression operators (sequence $(;)$, alternative $(+)$, and repetition $(*)$); moreover, there are two operators for parallel composition: (1) Operator $|$, which generates all the interleavings of the event traces defined by its operands; the events do not communicate, nor synchronize. It is used to express parallel activities in the frame protocol of C . (2) Operator ∇_S (“consent”), producing also all interleavings of the event traces defined by its operands, where, however, the neighboring events from S (with “opposite” prefix) are complementary - they synchronize and are forced to communicate (producing internal action τ similar to CCS and CSP). An example of such complementary events would be the pair $!I.m\uparrow$ and $?I.m\downarrow$. This operator

- a) synchronous callback: $FP_{Ca} = ?M.m \{!T.t\{?B.b\}\}$
- b) asynchronous callback: $FP_{Cb} = ?M.m\uparrow ; !T.t\uparrow ; ?T.t\downarrow ; ?B.b\downarrow ; !M.m\downarrow ; !B.b\downarrow$
- c) autonomous activity: $FP_{Cc} = ?S.s ; !A.a\uparrow ; ?M.m\uparrow ; ?A.a\downarrow ; !M.m\downarrow$
- d) synchronous reaction: $FP_{Cd} = ?M.m \{!R.r\}$

Table 1: Frame protocols of C in Fig. 1

is used to produce the composed behavior of cooperating components, while s comprises all the events on the component's bindings. Moreover, it also indicates communication errors (deadlock and "bad activity" - there is no complementary event to $!T.m!$ in a trace, i.e. a call cannot be answered).

Using behavior protocols, a quite complex behavior can be modeled - see, e.g., [1] for a behavior model of a real-life component application. Advantageously, it is possible to model the explicit interleaving of events on both the provided and required interfaces of a component in its frame protocol. Specifically, the frame protocol of C in Fig. 1 in the alternatives a) - d) would take the form as in Tab. 1.

2.1 Modeling Environment via Behavior Protocols

Consider again the missing environment problem and the setting on Fig. 1. Obviously, the environment of an isolated component C can be considered as another component E bound to C . Thus the model of E 's behavior can be a frame protocol of E . Since a required interface is always bound to a matching provided interface, the former issuing calls and the latter accepting calls, the corresponding events in both frame protocols ought to be complementary. For example, the frame protocols of E in Fig. 1 in alternatives a) - d) would have the form as in Tab. 2.

- a) $FP_{Ea} = !M.m\{?T.t\{!B.b\}\}$
- b) $FP_{Eb} = !M.m!;?T.t!;!T.t!;!B.b!;?M.m!;?B.b!$
- c) $FP_{Ec} = !S.s;?A.a!;!M.m!;!A.a!;!M.m!$
- d) $FP_{Ed} = !M.m\{?R.r\}$

Table 2: Frame protocols of E in Fig. 1

Obviously, an event issued by E (such as $!M.m!$) has to be accepted by C (such as $?M.m!$) at the right moment and vice versa. As an aside, this (behavior compliance [19]) can be formally verified by parallel composition via consent, $FP_E \nabla_s FP_C$, which should not indicate any communication error; for $FP_{Eb} \nabla_s FP_{Cb}$ this is obviously true, since FP_{Eb} was created by simply replacing all $?$ by $!$ and vice versa - FP_{Eb} is the *inverted frame protocol* (FP_{Cb}^{-1}) of C_b . Because of that and since here S comprises all events on the interfaces M , T and B , the consent operator produces traces composed of τ only.

In general, any protocol FP_E for which $FP_E \nabla_s FP_C$ does not yield any composition error is called *environment protocol* of C , further denoted as EP_C . In Sect. 1, we proposed three specific techniques to construct C 's environment protocol: (i) inverted frame protocol (EP_C^{inv}), (ii) context protocol (EP_C^{ctx}) and (iii) calling protocol (EP_C^{call}). Event though these techniques aim at "decent" exercising of C , an environment protocol may be very simple, designed to help check a specific property. Assume for instance that the interface M of C in Fig. 1 features also a method x and $FP'_{Ca} = ?M.m\{!T.t\{?B.b\}\} + ?M.x$. Then, $EP'_{Ca} = !M.x$ would be an environment protocol since $EP'_{Ca} \nabla_s FP'_{Ca}$ does not yield any composition error.

Nevertheless, the three techniques (i) - (iii) are much more of practical importance; below, they are illustrated on a part of the component architecture created in our group for the solution to the CoCoME assignment [8] (Fig. 2); the solution was based on the Fractal component model [3] and behavior protocols. Here we focus especially on the *Store* component, the functionality of which is not fully used by *StoreApplication* that accesses *Store* indirectly via *Data*. Specifically, the actual use of *Store* employs only a subset of the traces allowed by its frame protocol FP_{Store} (Fig. 3a); for example, FP_{Store} states that it is possible to call any method of the *StoreQueryIf* interface at most four times in parallel; however, assume that the *queryProductById* and *queryStockItem* methods on this interface are (indirectly) called three times in parallel by *StoreApplication* and the other methods are called only twice in parallel, or not at all in parallel. Therefore, the context protocol EP^{ctx}_{Store} (Fig. 3c) of *Store* is much simpler in terms of level of parallelism than its inverted frame protocol EP^{inv}_{Store} (Fig. 3b). Since the *Store* component has no required interface, its calling protocol EP^{call}_{Store} is equal to its context protocol, thus being obviously a valid model of the *Store*'s environment behavior in this special case.

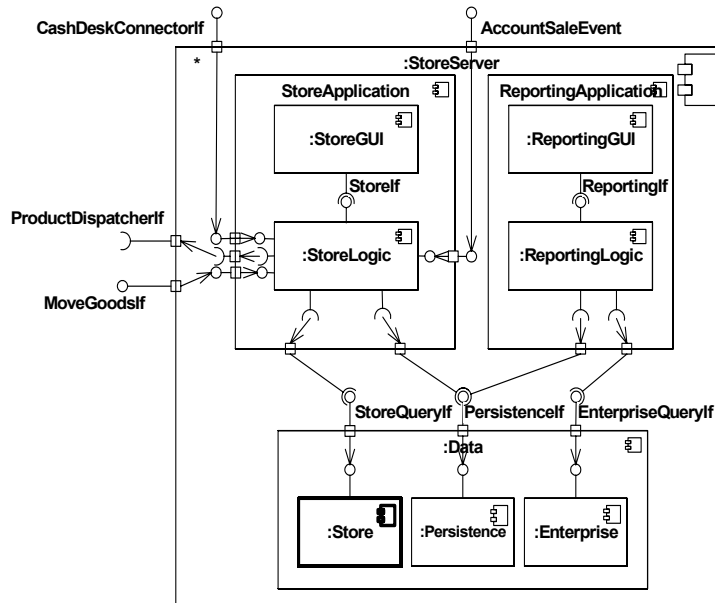


Figure 2: Architecture of the *StoreServer* component

In summary, the basic idea of the techniques (i)-(iii) for construction of a model of *C*'s environment (*E*) behavior is as follows:

Re (i) The inverted frame protocol EP^{inv}_C of a component *C* is constructed directly from the component's frame protocol FP_C by replacing all the prefixes ? by ! and vice versa.

Re (ii) The component's context protocol EP^{ctx}_c is derived via consent composition of the frame protocols of all the other components bound to C at the same level of nesting and the context protocol (or inverted frame protocol) of the C 's parent component (if there is one).

Re (iii) The calling protocol can be derived in two ways: either via syntactical omitting of events on required interfaces from the inverted frame protocol or context protocol, or directly from the frame protocols of all the components in an

```

a)  $FP_{Store} = ($ 
    ?StoreQueryIf.queryProductById +
    ?StoreQueryIf.queryStockItem +
    # calls of other methods on StoreQueryIf follow
 $)^*$ 
    |
    # the fragment above repeated three more times

b)  $EP^{inv}_{Store} = ($ 
    !StoreQueryIf.queryProductById +
    !StoreQueryIf.queryStockItem +
    # calls of other methods on StoreQueryIf follow
 $)^*$ 
    |
    # the fragment above repeated three more times

c)  $EP^{ctx}_{Store} = ( !StoreQueryIf.queryStockItem^* ; \dots )^*$ 
    |
    (
    !StoreQueryIf.queryProductById*
    +
    !StoreQueryIf.queryStockItem*
    )*
    |
    !StoreQueryIf.queryProductById*
    |
    (
    \dots ;
    ( !StoreQueryIf.queryProductById*; \dots )
    +
    ( \dots ; !StoreQueryIf.queryStockItem* )
    +
    \dots
    )
    )*

```

Figure 3: a) a fragment of the frame protocol of *Store*; b) a fragment of the inverted frame protocol of *Store*; c) a fragment of the context protocol of *Store*

architecture (including the one subject to checking) via a syntactical algorithm described in Sect. 3.2. In both cases, events on C 's required interfaces, i.e. calls of E from C , are modeled implicitly in such a way that they are allowed to happen at any time and in parallel with any other event on any C 's interface; technically, the environment protocol based on a calling protocol takes the form

$$EP^{call}_c = \langle \text{calling protocol} \rangle \mid ?m1^* \mid \dots \mid ?m1^* \mid \quad (E1)$$

$$\mid ?m2^* \mid \dots \mid ?mN^*$$

where m_1, \dots, m_N represent methods of the component's required interfaces (obviously several instances of the same method can be accepted in parallel; nevertheless, the number N and the number of appearances of each $?m_i^*$ have to be finite). Such EP_c^{call} is compliant with FP_c (assuming that compliance holds for frame protocols of all components in the application, which C belongs to), i.e. there are no communication errors, for the following reasons: (a) an environment modeled by EP_c^{call} calls C only in a way allowed by FP_c , since EP_c^{call} is derived from the frame protocols of the components cooperating with C at the same level of nesting (assuming their behavior is compliant); (b) an environment modeled by EP_c^{call} can accept any call from C at any time and in parallel with any other event on a C 's interface (both provided and required).

3 Calling Protocol vs. Callbacks and Autonomous Activities

As indicated at the end of Sect. 1, an environment protocol based on a calling protocol (EP_c^{call}) is an imprecise model (overapproximation) of E 's behavior in general, since it models in detail only the events on the C 's provided interfaces and assumes a generic acceptance of calls on the required interfaces. Therefore, specifically, it is not possible to model detailed interleaving of events on these interfaces, which is necessary for proper modeling of callbacks and autonomous activities.

In this section, we propose certain syntactical constraints on C 's frame protocol FP_c to ensure that no other events than those related to synchronous reactions, triggers of callbacks, and autonomous activities take place on the required interfaces of C ; also, we answer the question whether EP_c^{call} can be a valid model of E 's behavior if these constraints are satisfied.

The key idea is to express the constraints on FP_c via the following syntactical schemes (for simplicity, names of interfaces are omitted in event identifications):

(A) To express synchronous callbacks (and synchronous reactions) correctly, the constraint is that in FP_c the events corresponding to a particular callback b and a trigger t for b have to be nested according to the scheme

$$FP_c = \alpha_1 \text{ op}_1 ?m \{ \alpha_2 \text{ op}_2 !t \{ ?b \} \text{ op}_3 \alpha_3 \} \text{ op}_4 \alpha_4 \quad (\text{A1})$$

where α_i may involve only synchronous reactions (C-E pattern (d)) and arbitrary behavior protocol operators except consent (∇), and op_i is either the sequence operator ($;$) or the alternative operator ($+$). Specifically, the frame protocol $FP'_c = \alpha_1 \text{ op}_1 ?m ; \alpha_2 ; !t ; \alpha_3 ; ?b \text{ op}_2 \alpha_4$, which would be the only option when using the LTS-based approach of [10], violates the constraint. An example of a frame protocol that satisfies the constraint is

$$FP'_{ca} = ?m1 \{ !r1 \} ; ?m2 \{ !t \{ ?b \} + !r2 \} ; ?m3 \{ !r3 \} \quad (\text{EX-A1})$$

(B) To express asynchronous callbacks (and (A)) correctly, the constraint on FP_c is that it is necessary to use parallel composition of the events corresponding to a particular callback b with other events, including the trigger t of b , according to the scheme

$$FP_c = \beta_1 \text{ op}_1 ?m1 ; \beta_2 ; !t1 ; ((?t1 ; \beta_3 ; !m1 \text{ op}_2 \beta_4) | ?b) \quad (\text{B1})$$

where β_i is composed of behavior protocols satisfying the constraint A connected via arbitrary behavior protocol operators except the consent (∇), and op_i is again either $;$ or $+$. Specifically, a violation of the constraint would be to use explicit sequencing of events like in $?m1; !t1; ?t1; ?b1; !m1; !b1$ (Tab. 1b), since an asynchronous callback runs in a different thread than the trigger and therefore unpredictable thread scheduling has to be considered. An example of a frame protocol that satisfies the constraint is

$$FP'_{cb} = ?m1 ; ?m2! ; !t1 ; ((?t1 ; !m2! ; ?m3\{!r3\}) | ?b) \text{ (EX-B1)}$$

(C) To express autonomous activities on required interfaces (and (B)) correctly, the constraint is that it is also necessary to use parallel composition (as in (B)), since such activities are performed by C 's inner threads and thus non-deterministic scheduling of the threads has to be considered. Specifically, the events corresponding to a particular autonomous activity a have to be composed via the and-parallel operator with other events that can occur after the start of the inner thread (in method s). Thus, when involving autonomous activities, FP_C has to comply with the scheme

$$FP_C = \gamma_1 op_1 ?s! ; ((!s! ; \gamma_2) | !a) \text{ (C1)}$$

where γ_i is composed of behavior protocols satisfying the constraint B connected via arbitrary behavior protocol operators except the consent (∇). For example, the frame protocol $FP'_C = \gamma_1 op_1 ?s ; \gamma_2 ; !a! ; \gamma_3 ; ?a! ; \gamma_4$ is not valid, since the events for the autonomous activity a are not allowed to happen before the call to s returns (i.e. before $!s!$ occurs). An example of a frame protocol that satisfies the constraint is

$$FP'_{cc} = ?m1 ; ?s! ; ((!s! ; (?m2 + ?m3\{!r3\})) | !a) \text{ (EX-C1)}$$

In summary, to satisfy the constraints, a frame protocol has to be constructed in a hierarchical manner, with synchronous reactions and synchronous callbacks (compliant to the constraint A) lower than asynchronous callbacks (compliant to B), and with autonomous activities (compliant to C) at the top.

3.1 Calling & Trigger Protocol

An important question is whether from a FP_C (and frame protocols of other components at the same level of nesting as C) satisfying the constraints A, B, and C an EP^{call}_C can be derived such that it would be a valid model of behavior of C 's environment; i.e., whether it suffices to model precisely only the interleaving of events on C 's provided interfaces when callbacks and autonomous activities are considered. To answer this question, it is sufficient to consider the possible meanings of an event on a required interface in the frame protocol FP_C satisfying the constraints; such an event can be:

- (1) A synchronous reaction r to a call on a provided interface, when r is not a trigger of a callback.
- (2) An autonomous activity a on a required interface, when a is not a trigger of a callback.
- (3) A trigger t of a callback b (either synchronous or asynchronous).

In cases (1) and (2), it is appropriate to model r , resp. a , implicitly (as in E1), since it has no relationship with any event on C 's provided interfaces. On the other hand, a trigger t of a callback b (case 3) cannot be modeled implicitly, since b can be executed by E only after C invokes t - if t were modeled implicitly, then E could execute b even before t was invoked by the component.

Therefore, the answer to the question of sufficiency of the constraints is that the environment protocol based on a calling protocol (EP^{call}_C) is not a valid model of E 's behavior if the interaction between C and E involves callbacks, since triggers of callbacks are modeled implicitly in EP^{call}_C - precise interleaving of a callback and its trigger has to be preserved in a valid model of E 's behavior.

As a solution to this problem, we propose to define the environment protocol of a component C on the basis of a *calling & trigger protocol* that models a precise interleaving of the events on C 's provided interfaces (including callbacks) and triggers of callbacks. In principle, the environment protocol takes the form

$$EP^{trig}_C = \langle \text{calling \& trigger protocol} \rangle \mid ?m1^* \mid \dots \mid \quad (E2)$$

$$\mid ?m1^* \mid ?m2^* \mid \dots \mid ?mN^*$$

where $m1, \dots, mN$ are all the methods of the C 's required interfaces except triggers of callbacks. Compliance of EP^{trig}_C with FP_C holds for similar reasons like in case of EP^{call}_C (end of Sect. 2.1) - note that although an environment modeled by EP^{trig}_C can accept triggers of callbacks from a component C only at particular moments of time, C will not invoke any trigger at an inappropriate time, since frame protocols of C and components cooperating with C at the same level of nesting are assumed to be compliant.

An environment protocol based on a calling & trigger protocol for (A1) has to comply with the scheme

$$EP^{trig}_C = (\alpha_{1_prov}^{-1} \ o p_1 \ !m\{\alpha_{2_prov}^{-1} \ o p_2 \ ?t\{!b\} \ o p_3 \ \alpha_{3_prov}^{-1}\} \ o p_4 \quad (A2)$$

$$\ o p_4 \ \alpha_{4_prov}^{-1}) \mid \alpha_{1_req}^{-1} \mid \dots \mid \alpha_{4_req}^{-1}$$

where $\alpha_{i_prov}^{-1}$ denotes the events on provided interfaces from α_i^{-1} and $\alpha_{i_req}^{-1}$ denotes the events on required interfaces from α_i^{-1} (α_i^{-1} contains the events from α_i with $?$ replaced by $!$ and vice versa). For illustration, the proper environment protocol for (EX-A1) is $EP^{trig}'_{ca} = (!m1 \ ; \ !m2\{?t\{!b\}\} \ ; \ !m3) \mid ?r1 \mid ?r2 \mid ?r3$.

Similarly, an environment protocol for (B1) has to comply with the scheme

$$EP^{trig}_C = (\beta_{1_prov}^{-1} \ o p_1 \ !m\uparrow \ ; \ \beta_{2_prov}^{-1} \ ; \ ?t\uparrow \ ; \ ((!t\downarrow \ ; \ \beta_{3_prov}^{-1} \ ; \ (B2)$$

$$\ ; \ ?m\downarrow \ o p_2 \ \beta_{4_prov}^{-1}) \mid !b) \mid \beta_{1_req}^{-1} \mid \dots \mid \beta_{4_req}^{-1},$$

while an environment protocol for (C1) has to comply with the scheme

$$EP^{trig}_C = ((\gamma_{1_prov}^{-1} \ o p_1 \ !s \ ; \ \gamma_{2_prov}^{-1}) \mid ?a) \mid \gamma_{1_req}^{-1} \mid \gamma_{2_req}^{-1}. \quad (C2)$$

The proper environment protocol for (EX-B1) is $EP^{trig}'_{cb} = (!m1 \ ; \ !m2\uparrow \ ; \ ?t\uparrow \ ; \ ((!t\downarrow \ ; \ ?m2\downarrow \ ; \ !m3) \mid !b) \mid ?r3$, while the proper environment protocol for (EX-C1) is $EP^{trig}'_{cc} = (!m1 \ ; \ !s\uparrow \ ; \ ((?s\downarrow \ ; \ (!m2 \ + \ !m3)) \mid !a) \mid ?r3$.

3.2 Construction of Calling & Trigger Protocol

The algorithm for construction of a calling & trigger protocol (*CTP*) is based on the syntactical algorithm for derivation of a context protocol that was presented in [15] - the main difference is the newly added support for callbacks and autonomous activities. Only the basic idea is described here, i.e. technical details are omitted.

In general, the algorithm accepts frame protocols of all components (primitive and composite) in the given application and bindings between the components as an input, and its output are *CTPs* for all primitive components in the application. The frame protocols have to be augmented with identification of events that correspond to triggers for callbacks and autonomous activities.

The algorithm works in a recursive way: when executed on a specific composite component C , it computes CTP_{C_i} for each of its sub-components C_1, \dots, C_N , and then applies itself recursively on each C_i .

More specifically, the following steps have to be performed to compute the calling & trigger protocol CTP_{C_k} of C_k , a sub-component of C :

1) A directed graph G of bindings between C and the sub-components of C is constructed and then pruned to form a sub-graph G_{C_k} that contains only the paths involving C_k . The sub-graph G_{C_k} contains a node N_C corresponding to C and a node N_{C_i} for each sub-component C_i of C ; in particular, it contains a node N_{C_k} for C_k .

2) An intermediate version IP_{C_k} of CTP_{C_k} is constructed via a syntactical expansion of method call shortcuts during traversal of G_{C_k} in a DFS manner. The traversal consists of two phases - (i) processing synchronous reactions and autonomous activities on required interfaces, and (ii) processing callbacks. Technically, the first phase starts at N_C with CTP_C of C (inverted frame protocol is used for the top-level composite component) and ends when all the edges on all paths between N_C and N_{C_k} are processed (cycles are ignored in this phase); the second phase starts at C_k and processes all cycles involving C_k . When processing a specific edge E_{lm} , which connects nodes N_{C_l} and N_{C_m} (for C_l and C_m), in the first phase, the current version $IP_{C_k}^{lm}$ (computed prior to processing of E_{lm}) of IP_{C_k} is expanded in the following way: assuming that a required interface R_l of C_l is bound to a provided interface P_m of C_m , each method call shortcut on R_l in $IP_{C_k}^{lm}$ is expanded to the corresponding method body defined in the frame protocol of C_m .

For example, if $IP_{C_k}^{lm}$ contains "...; !R1.m1 ; !R1.m2 ;..." and the frame protocol of C_m contains "...; ?Pm.m1{prot1} ; ?Pm.m2{prot2 + prot3} ;...", the result of one step of expansion has the form "...; prot1 ; (prot2 + prot3) ;..."

3) CTP_{C_k} is derived from IP_{C_k} by dropping (i) all the events related to other sub-components of C and (ii) all events on the required interfaces of C_k with the exception of triggers for callbacks, which have to be preserved.

In general, these three steps have to be performed for each sub-component of each composite component in the given component application in order to get a calling & trigger protocol for each primitive component.

4 Tools and Experiments

In order to show the benefits of use of the calling & trigger protocol-based approach instead of a context protocol or an inverted frame protocol, we have implemented construction of a context protocol (via consent composition) and a calling & trigger protocol (Sect. 3.2), and performed several experiments.

	Inverted frame protocol-based EP	Context protocol-based EP	Calling & trigger protocol-based EP
Time to compute EP	0 s	3 s	0.1 s
Total time (EP + JPF)	n/a	1102 s	1095 s
Total memory	> 2048 MB	762 MB	748 MB

Table 3: Results for the `Store` component

Our implementation of construction of a calling & trigger protocol and a context protocol does not depend on a specific component system, i.e. it can be used with any component system that supports formal behavior specification via behavior protocols (currently SOFA [4] and Fractal [1]). Moreover, the automated environment generator for JPF (EnvGen for JPF) [13] is available in both SOFA and Fractal versions, and thus we provide a complete JPF-based toolset for checking Java implementation of isolated SOFA or Fractal primitive components against the following properties: obeying of a frame protocol by the component’s Java code [17] and all the properties supported by JPF out of the box (e.g. deadlocks and assertion violations).

	Inverted frame protocol-based EP	Context protocol-based EP	Calling & trigger protocol-based EP
Time to compute EP	0 s	2 s	0.5 s
Total time (EP + JPF)	n/a	n/a	485 s
Total memory	> 2048 MB	> 2048 MB	412 MB

Table 4: Results for the `ValidityChecker` component

We have performed several experiments on the `Store` component (Sect. 2.1) and the `ValidityChecker` component, which forms a part of the demo component application developed in the CRE project [1] - frame protocol, context protocol and calling & trigger protocol of `ValidityChecker` are in appendix A. For each experiment, we measured the following characteristics: time needed to compute a particular environment protocol, total time (computation of `EP` and JPF checking) and total memory; the value “> 2048 MB” for total memory means that JPF run out of available memory (2 GB) - total time is set to “n/a” in such a case.

Results of experiments (in Tab. 3 and Tab. 4) show that (i) construction of a calling & trigger protocol takes less time and memory than construction of a context protocol for these two components and (ii) total time and memory of environment’s behavior model construction, environment generation and checking with JPF (against obeying of a frame protocol, deadlocks and race conditions) are the lowest if the calling & trigger protocol-based approach is used. Time needed to compute `EPctx` of both `Store` and `ValidityChecker` is also quite low, since frame protocols of other components bound to them (in the particular applications) do not involve very high level of parallelism and thus state explosion did not occur. The main result is that the whole process of environment construction and, above all, JPF checking has a lower time and space complexity for calling & trigger protocol than if the other approaches are used.

5 Evaluation and Related work

In general, our experiments confirm that although `EPtrigC` for a component `C` specifies an “additional” parallelism (a parallel operator for each method of the `C`’s required interfaces), the size of the JPF state space in checking `C` with an environment modeled by `EPtrigC` is not increased (i.e. state explosion does not occur because of that), since the “additional” parallelism is not reflected in the environment’s Java code explicitly via additional threads - the environment only has to be prepared to accept the call of any method from `C` (except triggers of callbacks) at any time and in parallel with other activities. On the contrary, modeling environment by `EPtrigC` has the benefit of low time and space complexity (i) of construction of the model with respect to use of `EPctxC`, and (ii) of JPF checking of component’s Java code with respect to the use of `EPinvC`.

There are many other approaches to modeling behavior of software components and their environment that can be used to perform compositional verification of component-based applications (e.g. [9], [10], [5] and [12]); in particular, [9] and [10] do so on the basis of the assume-guarantee paradigm. However, to our knowledge, none of them supports independent constructs for the following atomic events explicitly in the modeling language: acceptance of a method invocation (`?i.m↑` in behavior protocols), emitting a method invocation (`!i.m↑`), acceptance of a return from a method (`?i.m↓`), and emitting a return from a method (`!i.m↓`). Process algebra-based approaches ([9], [5]) typically support input (acceptance) and output (emit) actions explicitly in the modeling language, while transition

systems-based approaches (e.g. [10] and [12]) support general events. In any case, it is possible to distinguish the events via usage of different names (e.g. event names `m1_invoke`, resp. `m1_return`, for invocation of `m1`, resp. for return from the method); however, an automated composition checking may fail even for compliant behavior specifications in such a case, since the developer of each of them can choose a different naming scheme (e.g. `m1_invoke` versus `m1↑`). We believe that a formalism for modeling component behavior should support all the four types of atomic events, since:

- (a) independent constructs for method invocation and return from a method are necessary for proper modeling of callbacks and autonomous activities, and
- (b) independent input and output actions are necessary for compliance checking, i.e. for checking the absence of communication errors between components.

Program model checking of open systems (isolated software components, device drivers, etc) typically involves construction of an “artificial” environment - an open system subject to checking and its environment then form a closed system (a complete program). The environment typically has the form of a program, as in our approach [14] and in [10], where the environment is defined in Java, or in SLAM/SDV [2], where the model of the windows kernel (environment for device drivers) is defined in the C language. In general, each approach to model checking of open software systems involves a custom tool or algorithm for construction of the environment, since each program model checker features a unique combination of API and input modeling language (i.e. different combination than the other program model checkers).

As for automated construction of the model of environment’s behavior, one recent approach [7] is based on the L^* algorithm for incremental learning of regular languages. The basic idea of this approach is to iteratively refine an initial assumption about behavior of the environment for a component subject to checking. At each step of the iteration, model checking is used to check whether the component satisfies the property, and if not, the assumption is modified according to the counterexample. The iteration terminates when the component satisfies the given property in the environment modeled by the assumption. An advantage of our approach over [7] is lower time and memory complexity, since use of model checking is not needed for construction of EP^{+rig} .

6 Conclusion

In our former work, we introduced two specific approaches to modeling of environment’s behavior: inverted frame protocol and context protocol. However, JPF checking of a component with the environment determined by any of these modeling approaches is prone to state explosion for the following reasons: (i) Java code of the environment is complex, since it has to ensure proper interleaving of invocation and return events on the component’s provided and required interfaces, (ii) for the context protocol, the algorithm for its construction involves model checking, while for the inverted frame protocol, the environment involves high

level of parallelism. To address the problem of state explosion, in [16] we proposed to use a model of environment's behavior based on the calling protocol. Since the calling protocol-based approach models precisely only the events on component's provided interfaces, it does not allow to express C-E patterns properly in general (it is an overapproximation of the desired behavior).

Therefore, in this paper we proposed a slightly modified idea - calling & trigger protocol, which models precise interleaving of events on provided interfaces and triggers of callbacks, and the "other events" models implicitly, similar to [16] with no threat of state explosion. The key idea is to impose certain constraints on the frame protocol of a component in terms of interleaving of C-E events with other events and to express the constraints via syntactical patterns the frame protocol has to follow, and then, if the constraints are satisfied, derive in an automated way the calling & trigger protocol. The experiments confirm that the idea is viable.

As a future work, we plan to create a tool for automated recognition of those component frame protocols that do not satisfy the constraints and to integrate it into the SOFA runtime environment.

Acknowledgments. This work was partially supported by the Grant Agency of the Czech Republic (project number 201/06/0770).

References

- [1] 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, 2006
- [2] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., Ustuner, A.: Thorough Static Analysis of Device Drivers, Proceedings of EuroSys 2006, ACM Press
- [3] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B.: The FRACTAL component model and its support in Java, *Softw. Pract. Exper.*, 36(11-12), 2006
- [4] Bures, T., Hnetyнка, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceedings of SERA 2006, IEEE CS
- [5] Brim, L., Cerna, I., Varekova, P., Zimmerova, B.: Component-interaction Automata as a Verification-oriented Component-based System Specification, Proceedings of SAVCBS 2005, ACM Press
- [6] Clarke, E. M., Long, D. E., McMillan, K. L.: Compositional Model Checking, Proceedings of LICS'89, IEEE CS
- [7] Cobleigh, J. M., Giannakopoulou, D., Pasareanu, C. S.: Learning Assumptions for Compositional Verification, Proceedings of 9th TACAS, LNCS, vol. 2619, 2003
- [8] CoCoME, <http://agrausch.informatik.uni-kl.de/CoCoME>
- [9] de Alfaro, L., Henzinger, T. A.: Interface Automata, Proceedings of 8th European Software Engineering Conference, ACM Press, 2001
- [10] Giannakopoulou, D., Pasareanu, C. S., Cobleigh, J. M.: Assume-guarantee Verification of Source Code with Design-Level Assumptions, Proceedings of 26th International Conference on Software Engineering (ICSE), 2004
- [11] Mach, M., Plasil, F., Kofron, J.: Behavior Protocol Verification: Fighting State Explosion, *International Journal of Computer and Information Science*, 6(2005)

- [12] Ostroff, J.: Composition and Refinement of Discrete Real-Time Systems, ACM Transactions on Software Engineering and Methodology, 8(1), 1999
- [13] Parizek, P.: Environment Generator for Java PathFinder, <http://dsrg.mff.cuni.cz/projects/envgen>
- [14] Parizek, P., Plasil, F.: Specification and Generation of Environment for Model Checking of Software Components, Proceedings of FESCA 2006, ENTCS, 176(2)
- [15] Parizek, P., Plasil, F.: Modeling Environment for Component Model Checking from Hierarchical Architecture, Proceedings of FACS'06, ENTCS, vol. 182
- [16] Parizek, P., Plasil, F.: Partial Verification of Software Components: Heuristics for Environment Construction, Proc. of 33rd EUROMICRO SEAA, IEEE CS, 2007
- [17] Parizek, P., Plasil, F., Kofron, J.: Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker, Proceedings of SEW'06, IEEE CS
- [18] Pasareanu, C. S., Dwyer, M., Huth, M.: Assume-guarantee model checking of software: A comparative case study, Proceedings of the 6th SPIN workshop, LNCS, vol. 1680, 1999
- [19] Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, 28(11), 2002
- [20] Tkachuk, O., Dwyer, M. B., Pasareanu, C. S.: Automated Environment Generation for Software Model Checking, Proceedings of ASE 2003, IEEE CS
- [21] Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs, Automated Software Engineering Journal, vol. 10, no. 2, 2003, <http://javapathfinder.sourceforge.net>

Appendix A

```

FPValidityChecker = (
  ?IToken.SetEvidence
  |
  ?IToken.SetValidity
  |
  (
    ?IToken.SetAccountCredentials {
      !ICustomCallback.SetAccountCredentials
    }
    +
    NULL
  )
)
;
?ILifetimeController.Start^
;
!ITimer.SetTimeout^
;
(
  (
    ?Timer.SetTimeout$
    ;
    !ILifetimeController.Start$
    ;
    (
      ?IToken.InvalidateAndSave {
        !ITimer.CancelTimeouts;
        (!ICustomCallback.InvalidatingToken + NULL);
        !ITokenCallback.TokenInvalidated
      }
    )
  )
)

```

```

    }*
    |
    ?IToken.InvalidateAndSave {
        !ITimer.CancelTimeouts;
        (!ICustomCallback.InvalidatingToken + NULL);
        !ITokenCallback.TokenInvalidated
    }*
)
)
|
?ITimerCallback.Timeout {
    (!ICustomCallback.InvalidatingToken + NULL);
    !ITokenCallback.TokenInvalidated
})*
)

```

```

EPinvValidityChecker = EPctxValidityChecker = (
    !IToken.SetEvidence
    |
    !IToken.SetValidity
    |
    (
        !IToken.SetAccountCredentials {
            ?ICustomCallback.SetAccountCredentials
        }
        +
        NULL
    )
)
;
!ILifetimeController.Start^
;
?ITimer.SetTimeout^
;
(
    (
        !Timer.SetTimeout$
        ;
        ?ILifetimeController.Start$
        ;
        (
            !IToken.InvalidateAndSave {
                ?ITimer.CancelTimeouts;
                (?ICustomCallback.InvalidatingToken + NULL);
                ?ITokenCallback.TokenInvalidated
            }*
            |
            !IToken.InvalidateAndSave {
                ?ITimer.CancelTimeouts;
                (?ICustomCallback.InvalidatingToken + NULL);
                ?ITokenCallback.TokenInvalidated
            }*
        )
    )
)
|
!ITimerCallback.Timeout {
    (?ICustomCallback.InvalidatingToken + NULL);
    ?ITokenCallback.TokenInvalidated
})*
)

```

```

EPtrigValidityChecker = (
  (
    !IToken.SetEvidence
    |
    !IToken.SetValidity
    |
    (
      !IToken.SetAccountCredentials
      +
      NULL
    )
  )
  ;
  !ILifetimeController.Start^
  ;
  ?ITimer.SetTimeout^
  ;
  (
    (
      !Timer.SetTimeout$
      ;
      ?ILifetimeController.Start$
      ;
      (
        !IToken.InvalidateAndSave*
        |
        !IToken.InvalidateAndSave*
      )
    )
    |
    !ITimerCallback.Timeout*
  )
)
|
?ICustomCallback.SetAccountCredentials*
|
?ITimer.CancelTimeouts*
|
?ITimer.CancelTimeouts*
|
?ICustomCallback.InvalidatingToken*
|
?ICustomCallback.InvalidatingToken*
|
?ICustomCallback.InvalidatingToken*
|
?ITokenCallback.TokenInvalidated*
|
?ITokenCallback.TokenInvalidated*
|
?ITokenCallback.TokenInvalidated*

```

Chapter 10

Evaluation and related work

10.1 Method

In this thesis, we addressed the following three challenges (Goals G1-G3 in Chapter 3) related to automated formal verification of primitive components implemented in Java:

- Modeling and construction of artificial environment for isolated components with the goal of feasible verification via model checking (G1).
- Absence of support for high-level properties like obeying of an event trace-based behavior specification in the state-of-the-art model checkers and verification frameworks for Java programs (G2).
- State explosion in discovery of concurrency errors in Java code of real-life components via model checking (G3).

The rest of this section provides a comprehensive comparison of our solution to the challenges with related techniques proposed by other researches. Detailed evaluation of the individual parts of our contribution was already published in the included papers (Chapters 5-9).

Checking component implementation against behavior specification

There exist several frameworks for verification of programs that use model checking as the core technique and static analysis or theorem proving as complementary techniques. Most of the frameworks are based on model checkers that work only for complete programs (with `main` in case of Java) and support only low-level properties like absence of deadlocks and assertion violations by default — this is true, e.g., for JPF [60], Bandera [25] and SLAM [9]. Nevertheless, frameworks like JPF and Bandera can be used for verification of isolated primitive components against the property of obeying an event trace-based behavior specification (defined, e.g., as an LTS), since extensions and tools that provide support for construction of an artificial environment and property specification are available [27][59][30]. More specifically, the Bandera toolset supports automated generation of environment from a model of its behavior via the Bandera Environment Generator (BEG) [59] tool, and the

properties defined as finite state automata (regular expressions) via an extension to its core model checker Bogor [27]. The extension is responsible for management and traversal of an automaton, and for reporting of property violations.

In case of JPF, there is a technique [30] for checking Java code of isolated components against behavior specifications defined in LTS; the events correspond to method calls and locking-related operations upon objects. The technique is based on the assume-guarantee paradigm — for a given isolated component, an assumption in the form of an LTS is generated automatically via the L* algorithm [23] and then an artificial environment (Java code) is constructed from the assumption. According to [30], the code of the environment is written by hand, but tools like BEG [59] and our Environment Generator for JPF [49] could be used to generate the code automatically. Support for the property (obeying a behavior specification defined in LTS) is provided via instrumentation of the component’s Java code — technically, a Java class that encapsulates the LTS (i.e. keeps track of event history and performs transitions) is added to the program and checks for validity of event traces (histories) are encoded into assertions.

Besides JPF and Bandera, which aim at verification of Java code, there is also the MAGIC [19] tool for verification of C procedures against behavior specifications in the form of an LTS. As input, MAGIC accepts the C code of a procedure P subject to verification and a behavior model of P ’s environment. The environment for P is formed by a set of LTSs that model the behavior of other procedures invoked by P . Verification of the procedure P consists of two steps: (i) the LTS model of P is derived from its C code, and (ii) simulation relation between the LTSs corresponding to implementation and specification of P is checked.

The main advantage of our approach — cooperation of JPF with the BPChecker (Chapter 5) — over the other is that we have reused an existing tool in order to provide support for the property (obeying a frame protocol). Contrary to [30], we avoided the necessity to instrument the component’s Java code with mechanisms for traversal of an LTS and detection of error states that are already implemented in BPChecker, and we also didn’t have to create a specific extension to JPF similar to Bogor. Moreover, our solution is portable with respect to new versions of JPF, since it uses only the standard extension points of JPF (listeners) — no modification of JPF’s core is necessary.

Modeling behavior of artificial environment for isolated components

The key idea behind majority of approaches to modeling behavior of component’s environment, i.e. to specifying the assumption A in the context of the assume-guarantee paradigm, is that a component is expected to satisfy the required properties only in the environments that behave in a specific way [3]. In particular, the component may not satisfy the properties in a universal environment (calling each method of the component at any time and in parallel with any other method), while satisfying them in an environment that, e.g., reflects use of the component by a particular application (a real environment). The approaches to construction of a model of environment’s behavior (environment assumption) can be divided into two groups — completely automated and (partially) manual. For example, the BEG

tool [59] generates an artificial environment for a Java component from a model that can be written by hand or derived automatically from the code of a particular real environment via static analysis (if such an environment is available); the model specifies both the control-flow (sequences of calls of component's methods) and values of method parameters.

Several recent approaches to automated construction of environment assumption are based on the L^* algorithm for learning of regular languages [6]. The basic idea, common to all those approaches, is the iterative refinement of an initial assumption using the teacher represented by a model checker. The initial assumption typically expresses the available knowledge about an environment — it can be empty, corresponding to a universal environment, or created by hand characterizing a particular real environment. The output of the process is the weakest environment assumption that guarantees satisfaction of the given properties by the component. In each iteration, the two premises of the A-G rule (Sect. 2.2) are checked by a model checker upon the current assumption (result of previous iteration); if the model checker reports a counter-example during one of the checks, the current assumption is modified (strengthened or weakened) on the basis of the counter-example. This way, the current assumption becomes more precise in each iteration — the sequence of assumptions computed during the iterative process converges to the weakest assumption. If the model checker fails to provide an answer due to state explosion in a specific iteration, the current assumption is used as a result of the whole process.

The L^* -based approaches differ mainly in the specific model checking technique and the style of communication between a component and its environment that is modeled. In [23], the focus is on communication via method calls and the LTSA model checker [44] is used, while in [56] and [5] the focus is on communication via shared variables (shared memory) and SAT-based model checking, resp. symbolic model checking, is used.

In our case (Chapter 6), an environment assumption consists of two distinct parts: (i) a model of interaction between a component and its environment via sequences of method calls, which is defined in the formalism of behavior protocols, and (ii) specification of the possible values for method parameters in the form of a Java class. Currently only the construction of the model of interaction is automated — the specification of values has to be created by hand.

The main advantage of our approach over [23], [56] and [5] is the low time and space complexity — we use a syntactical algorithm for construction of the model of interaction between a component and its environment (Chapter 9), while the other approaches involve the use of a model checker as the teacher for L^* . Although the number of calls of a model checker during the learning process can be significantly reduced (see, e.g., [20] and [31]), still the model checker has to be called several times.

Detection of concurrency errors in program code

The techniques and tools for detection of concurrency errors in software systems can be divided into four groups based on the main paradigm they use — static analysis, runtime analysis, model checking and testing.

Static analysis-based techniques typically combine several control-flow and/or data-flow analyses. For example, the Jlint [7] and FindBugs [39] tools aim at automated detection of predefined error patterns (fragments of code that represent potential errors) in Java code via combination of linear scan of methods' bytecode, several control-flow analyses (including traversal of a control-flow graph) and data-flow analyses.

Examples of runtime error detectors are the Eraser [54] tool, which implements the well-known lockset algorithm for discovery of race conditions, and Java PathExplorer (JPaX) [36], which supports discovery of race conditions and deadlocks in Java programs.

In case of model checking, heuristics for state space traversal (see [32] for heuristics in JPF) are often used to mitigate the state explosion with the goal of discovery of errors in reasonable time and memory - e.g. for discovery of concurrency errors with JPF, a heuristic that prefers aggressive thread scheduling [32] can be used.

Detection of concurrency errors via testing typically involves a careful (deterministic) control of thread scheduling and interaction. For example, the ConAn [41] tool tests a predefined set of thread interleavings using synchronization via a clock.

Combination of different approaches with the goal of better precision and performance is also very popular. Often a particular static or runtime analysis technique is used to identify potential concurrency errors and a model checker then checks whether the errors are real or spurious. For example, in [35], the authors proposed a combination of runtime analysis and model checking on the basis of JPF, where the model checker (JPF) is guided by the counter-example reported by the runtime detector (JPF in simulation mode); more specifically, JPF focuses on those threads that are involved in the potential errors identified by runtime analysis.

Our approach to discovery of concurrency errors in Java code of isolated components combines static analysis (search for suspicious patterns in Java bytecode of pairs of methods) and model checking (Chapter 8). Since the goal of the static analysis is only to identify potential concurrency errors so that a reasonable artificial environment can be constructed, we use an analysis technique that (i) has low time and space complexity and (ii) may report false warnings (performance/precision tradeoff). Nevertheless, no false warnings (spurious errors) are actually reported to the user, since JPF precisely detects real concurrency errors in the Java code composed of a component and its reasonable environment. The main drawback of the technique is that it supports only pre-defined patterns involving pairs of Java methods — naturally, the pre-defined patterns do not cover all possible concurrency errors in Java. Currently we are working on a technique for detection of potential errors in arbitrarily-sized sets of Java methods, which is based on a software metric that measures degree of interaction among Java methods via concurrency-related constructs of Java (e.g., accesses to shared variables and `synchronized` blocks).

Note also that we could not use runtime analysis for identification of potential errors like it is done in [35], since runtime analysis works only for complete programs (components with artificial environment), i.e. it expects that the artificial environment already exists — in our case the artificial environment is constructed on the basis of the information about potential errors that is acquired via the static analysis.

10.2 Tools

We have implemented all techniques proposed in this thesis — each in a standalone tool — and integrated all the tools in the COMBAT toolset (Fig. 10.1), which is available at <http://dsrg.mff.cuni.cz/projects.phtml?p=combat>. In particular, the toolset contains:

- a plugin for JPF that allows checking of Java code against the property of obeying a frame protocol (Chapter 5),
- the Environment Generator for Java PathFinder (EnvGen for JPF) [49] in the SOFA and Fractal versions (Chapter 6),
- implementation of construction of all the specific models of environment’s behavior, including calling & trigger protocol (Chapter 9), and
- a detector of potential concurrency errors in Java code that is based on search for suspicious patterns (Chapter 8).

The tools do not depend on a particular version of JPF, and thus the most recent stable version of JPF can be used with the toolset — nevertheless, a minor modification of some of the tools may be necessary when porting the toolset to a new JPF version in order to reflect changes in JPF API. Currently the COMBAT toolset supports only the SOFA [18] and Fractal [15] component platforms, but it can be easily ported to any component platform that uses (i) behavior protocols for component behavior specification and (ii) Java as the implementation language.

It should also be emphasized that although the COMBAT toolset and some of the techniques it implements are currently specific to JPF, they can easily be ported to another model checker for Java programs like Bogor [27].

10.3 Experiments

We have successfully applied parts of the COMBAT toolset on two realistic (real-life) component applications — the demo component application developed in the CRE project with France Telecom [1] (“CRE demo” for short) and the solution to the CoCoME contest [24] created in our group (“CoCoME”). Both applications involve approximately 25 components (20 of them being primitive), where each primitive component contains tens to hundreds lines of Java code. Since JPF does not work for Java programs that use native code, e.g. via libraries, we did not apply the COMBAT toolset to components that involve usage of a SQL database or GUI (in case of CoCoME), or low-level network communication over sockets (in case of CRE demo).

The results of experiments on selected components from CRE demo and CoCoME were published in the included papers (Chapters 5, 8 and 9 for CRE demo and Chapter 9 for CoCoME) and in [16] and [17]. All the results show that the proposed techniques in general work well and are feasible for realistic component applications. Nevertheless, verification of Java code of a highly complex component

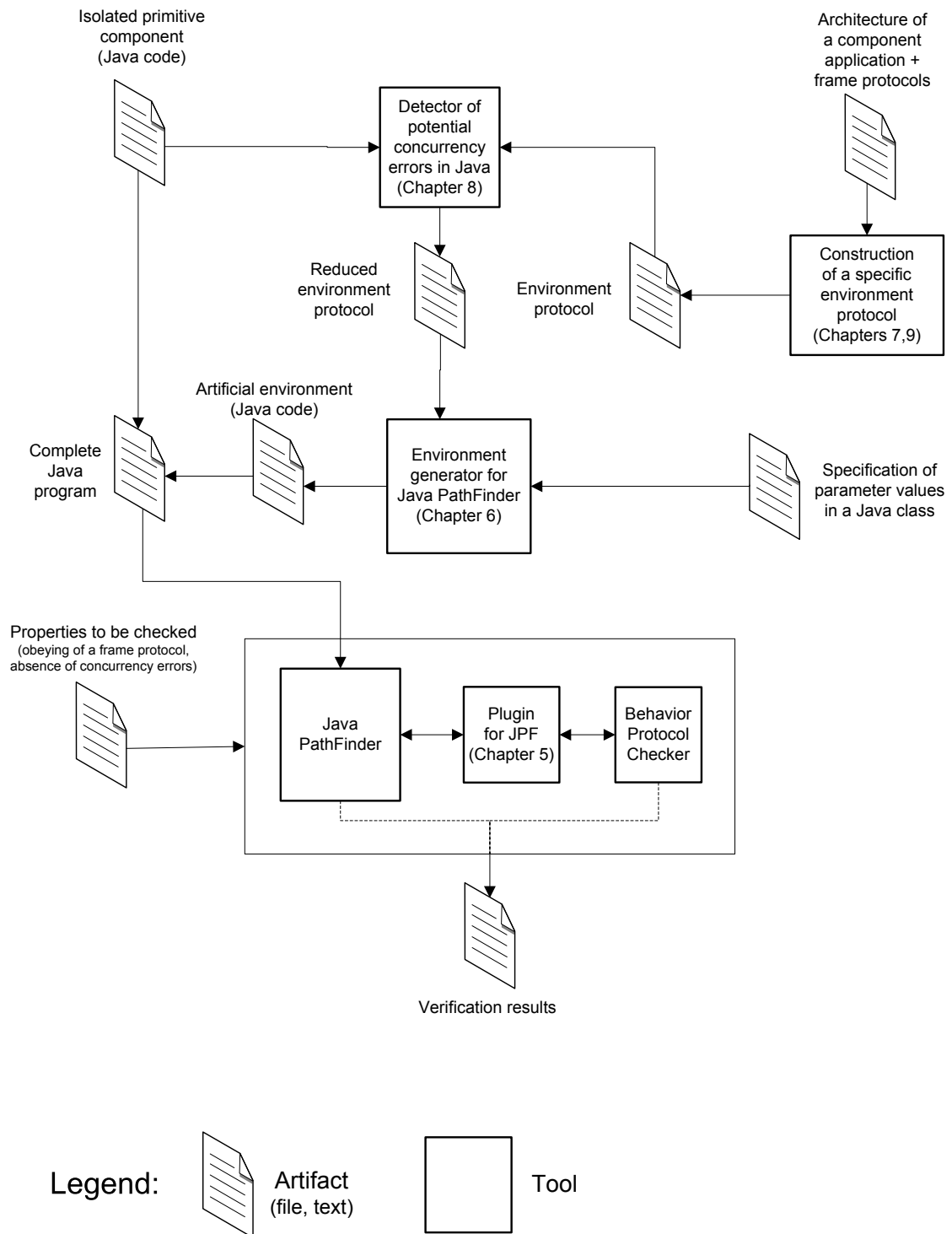


Figure 10.1: COMponent Behavior Analysis Toolset (COMBAT)

with COMBAT (and JPF) may still be infeasible due to state explosion — in particular, if an environment calls component’s methods in a high number of parallel threads, and/or there is a high number of control-flow paths in the component’s code. Addressing this issue is a part of our future work, as indicated below.

Using the COMBAT toolset, we also discovered several previously unidentified errors in the Java code of primitive components from CRE demo and CoCoME — specifically, a race condition in the `TransientIpDb` component from CRE demo (see Chapter 8 for details) and a race condition in the `CashDeskApplication` component from CoCoME. The toolset was also able to find a violation of a frame protocol of `CashDeskApplication` by its implementation, if the frame protocol was created according to the informal reference specification of the component’s behavior that was provided by organizers of CoCoME as a part of the assignment (see [16] and [17] for details).

Chapter 11

Conclusion

Summary of contribution. In this thesis, we presented a set of techniques related to behavior analysis and verification of primitive software components implemented in Java and equipped with a behavior specification defined in the formalism of behavior protocols. We have used the Java PathFinder model checker (JPF) as a core verification tool and focused on the properties of obeying an event trace-based behavior specification (frame protocol) and absence of concurrency errors (deadlocks and race conditions).

The overall goal of the thesis was to address the key issues of formal verification of Java components with JPF (Chapter 3): lack of support for the high-level property of obeying a frame protocol, applicability of JPF only to complete Java programs (problem of missing environment), and state explosion. We addressed the issues in the following way — we have:

- created an extension to JPF that allows checking of Java code against the property of obeying a frame protocol via combination of JPF with our model checker for behavior protocol compliance (BPChecker);
- solved the problem of missing environment via automated generation of an artificial environment for a primitive component from a model of the environment’s behavior (three specific models were proposed);
- addressed the state explosion problem in search for concurrency errors in Java code with JPF via reduction of the number of parallel threads in an artificial environment on the basis of static analysis of Java bytecode and heuristics.

We have implemented all the proposed techniques in the COMBAT toolset and evaluated them on two real-life component applications: CRE demo [1] and Co-CoME [17][16] — results of the experiments show that the proposed techniques in general work well and are viable for realistic components. Moreover, we have successfully used our extension to JPF (i.e., a part of COMBAT) in the BPEL checker tool [48] that aims at verification of BPEL code against behavior protocols [50].

Future work. As a future work, we plan to increase the degree of automation and performance of the whole process of verification of component’s Java code. In particular, we would like to address (i) the necessity to manually construct the

specification of possible values of method parameters (a part of component's artificial environment), and (ii) the potential infeasibility of checking Java code of a primitive component against its frame protocol with JPF in case of highly complex components or components that have complex environment.

Ad (i) Automated derivation of method parameter values could be done via symbolic execution in a similar way to [40]. Nevertheless, a problem common to state-of-the-art symbolic execution-based techniques is that they provide only limited support for complex data types (like lists and trees) and heap objects that are accessed via references — this has to be solved too in order to make automated derivation of values applicable to realistic Java components.

Ad (ii) In order to improve the performance of checking Java code against a frame protocol (and further address the state explosion problem), we plan to use static analysis of Java source code or bytecode. We have identified two options — construction of an abstracted Java program that is verified with JPF, and direct extraction of a behavior protocol (behavior model of the Java code) that is then checked against the frame protocol using the BPChecker.

We would also like to extend our approach to construction of a reasonable environment to address other kinds of errors and properties of Java code like assertion violations. This would involve use of different static analyses and heuristics than those used for concurrency errors — e.g. in case of assertion violations, values of program variables are typically more important than the number of parallel threads.

Bibliography

- [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] L. de Alfaro and T. A. Henzinger. Interface Automata, In *Proceedings of 8th European Software Engineering Conference*, ACM Press, 2001.
- [4] R. Allen and D. Garlan. A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, 1997.
- [5] R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions, In *Proceedings of 17th Conference on Computer-Aided Verification (CAV)*, LNCS, vol. 3576, 2005.
- [6] D. Angluin. Learning Regular Sets from Queries and Counterexamples, *Information and Computation*, 75(2), Nov. 1987.
- [7] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs, In *Proceedings of 13th Australian Software Engineering Conference (ASWEC)*, IEEE CS, 2001. <http://artho.com/jlint>
- [8] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers, *Proceedings of EuroSys 2006*, ACM Press.
- [9] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces, In *Proceedings of the 8th SPIN Workshop on Model Checking of Software*, LNCS, vol. 2057, 2001. <http://research.microsoft.com/slam>
- [10] J. A. Bergstra, A. Ponse, S. A. Smolka. *Handbook of Process Algebra*, Elsevier, 2001.
- [11] A. Bertolino and E. Marchetti. Software Testing, Chapter 5 in the *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, 2004 Version, IEEE Computer Society, 2004. <http://www.swebok.org>

- [12] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering, *International Journal on Software Tools for Technology Transfer*, 2007. <http://mtc.epfl.ch/software-tools/blast/>
- [13] S. Berezin, S. Campos, and E. Clarke. *Compositional Reasoning in Model Checking*, *Lecture Notes in Computer Science*, vol. 1536, 1998.
- [14] J.P. Bowen and M.G. Hinchey. *Formal Methods*, In *Computer Science Handbook*, 2nd edition, Section XI, Chapman & Hall/CRC, ACM, 2004.
- [15] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J. B. Stefani. The FRACTAL Component Model and Its Support in Java, *Software - Practice and Experience*, 36(11-12), 2006.
- [16] L. Bulej, T. Bures, T. Coupaye, M. Decky, P. Jezek, P. Parizek, F. Plasil, T. Poch, N. Rivierre, O. Sery, and P. Tuma. CoCoME in Fractal, Accepted for publication in *Proceedings of the CoCoME project*, LNCS, Jun 2007.
- [17] T. Bures, M. Decky, P. Hnetynka, J. Kofron, P. Parizek, F. Plasil, T. Poch, O. Sery, and P. Tuma. CoCoME in SOFA, Accepted for publication in *Proceedings of the CoCoME project*, LNCS, Jun 2007.
- [18] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, In *Proceedings of SERA 2006*, IEEE CS.
- [19] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C, *Transactions on Software Engineering (TSE)*, vol. 30, no. 6, June 2004
- [20] S. Chaki and O. Strichman. Optimized L*-based Assume-Guarantee Reasoning, In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, vol. 4424, 2007.
- [21] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*, MIT Press, 2000.
- [22] E. Clarke, D. Long, and K. McMillan. *Compositional Model Checking*, In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, 1989.
- [23] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning Assumptions for Compositional Verification, In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, vol. 2619, April 2003.
- [24] CoCoME: The Common Component Modeling Example, <http://agrausch.informatik.uni-kl.de/CoCoME>.
- [25] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zhueng. Bandera: Extracting Finite-state Models from Java Source Code, In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, June 2000. <http://bandera.projects.cis.ksu.edu>

- [26] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, In 4th Symposium on Principles of Programming Languages, 1977.
- [27] M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building Your Own Software Model Checker Using The Bogor Extensible Model Checking Framework, In Proceedings of 17th Conference on Computer-Aided Verification (CAV), LNCS, vol. 3576, 2005. <http://bogor.projects.cis.ksu.edu>
- [28] Enterprise Java Beans Specification, version 2.1, Sun Microsystems, Nov 2003. <http://java.sun.com/products/ejb/>
- [29] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption Generation for Software Component Verification, In Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE), 2002.
- [30] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh. Assume-guarantee Verification of Source Code with Design-Level Assumptions, In Proceedings of the 26th International Conference on Software Engineering (ICSE), 2004.
- [31] M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. Refining Interface Alphabets for Compositional Verification, In Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 4424, 2007.
- [32] A. Groce and W. Visser. Heuristics for Model Checking Java Programs, International Journal on Software Tools for Technology Transfer (STTT), vol. 6, no. 4, December 2004
- [33] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives, In Proceedings of the 1999 International Symposium on Static Analysis (SAS), LNCS, vol. 1694, September 1999.
- [34] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An Integrated Development, Analysis and Verification Environment for Component-based Systems, In Proceedings of the 25th International Conference on Software Engineering (ICSE), 2003.
- [35] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs, In Proceedings of the 7th SPIN Workshop on Model Checking of Software, LNCS, vol. 1885, 2000.
- [36] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer, In Proceedings of the 1st Workshop on Runtime Verification, ENTCS, vol. 55, 2001.
- [37] C. A. R. Hoare. Communicating Sequential Processes, Prentice Hall International (UK) Ltd., 1985.

- [38] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*, Addison-Wesley, 2003. <http://spinroot.com/spin/whatispin.html>
- [39] D. Hovemeyer and W. Pugh. Finding Bugs is Easy, *ACM SIGPLAN Notices*, vol. 39, 2004. <http://findbugs.sourceforge.net>
- [40] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing, In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, vol. 2619, April 2003.
- [41] B. Long, D. Hoffman, and P. Strooper. Tool Support for Testing Concurrent Java Components, *IEEE Transactions on Software Engineering*, vol. 29, no. 6, 2003.
- [42] M. Mach, F. Plasil, and J. Kofron. Behavior Protocol Verification: Fighting State Explosion, *International Journal of Computer and Information Science*, vol. 6, no. 1, ACIS, Mar 2005.
- [43] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures, In *Proceedings of the 5th European Software Engineering Conference (ESEC)*, LNCS, vol. 989, 1995.
- [44] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs*, John Wiley, 1999. <http://www.doc.ic.ac.uk/ltsa/>
- [45] K. McMillan. *Symbolic Model Checking*, Kluwer Academic Publishers, 1993. <http://www.cs.cmu.edu/~modelcheck/smv.html>
- [46] F. Nielson, H. R. Nielson, and Chris Hankin. *Principles of Program Analysis*, Springer, ISBN 3-540-65410-0, 2005.
- [47] OMG: CORBA Components, version 3.0, OMG document formal/02-06-65, Jun 2002.
- [48] P. Parizek. BPEL checker, <http://dsrg.mff.cuni.cz/projects/bpelchecker>, 2007.
- [49] P. Parizek. Environment Generator for Java PathFinder, <http://dsrg.mff.cuni.cz/projects/envgen>.
- [50] P. Parizek and J. Adamek. Checking Session-Oriented Interactions between Web Services, Accepted for publication in proceedings of 34th EUROMICRO SEAA conference, IEEE Computer Society, 2008.
- [51] C. Pasareanu, M. Dwyer, and M. Huth. Assume-Guarantee Model Checking of Software: A Comparative Case Study, In *Proceedings of the 6th SPIN Workshop*, LNCS, vol. 1680, 1999.
- [52] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components, *IEEE Transactions on Software Engineering*, vol. 28, no. 11, 2002.

- [53] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs, Logics and Models of Concurrent Systems, vol. 13, 1984.
- [54] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs, ACM Transactions on Computer Systems, vol. 15, issue 4, 1997.
- [55] M. I. Schwartzbach. Lecture Notes on Static Analysis, BRICS, Department of Computer Science, University of Aarhus, Denmark, 2006.
- [56] N. Sinha and E. Clarke. SAT-Based Compositional Verification Using Lazy Learning, In Proceedings of the 19th International Conference on Computer Aided Verification (CAV), LNCS, vol. 4590, 2007.
- [57] C. Szyperski. Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, 2002.
- [58] F. Tip. A Survey of Program Slicing Techniques, Journal of Programming Languages, vol. 3, no. 3, September 1995.
- [59] O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. Automated Environment Generation for Software Model Checking. In Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE), 2003.
- [60] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs, Automated Software Engineering Journal, vol. 10, no. 2, April 2003. <http://javapathfinder.sourceforge.net>