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.

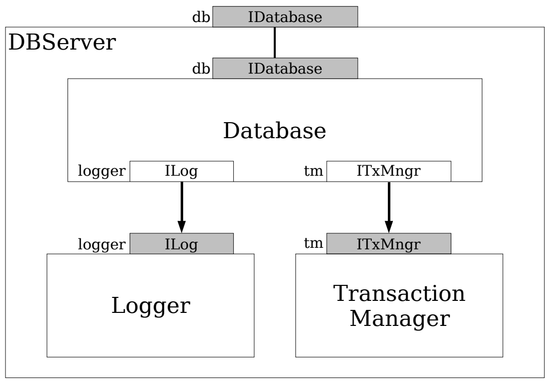

Figure 1: Architecture of the `DBServer` component

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.

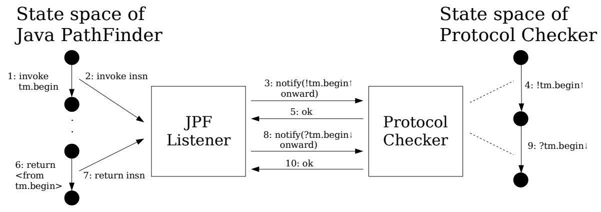
State space of Java PathFinder
State space of Protocol Checker
1: invoke tm.begin
2: invoke insn
3: notify(!tm.begin↑ onward)
4: !tm.begin↑
5: ok
6: return <from tm.begin>
7: return insn
8: notify(?tm.begin↓ onward)
9: ?tm.begin↓
10: ok
JPF Listener
Protocol Checker

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.

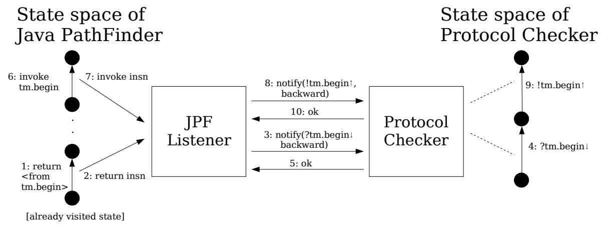
State space of Java PathFinder
State space of Protocol Checker
1: return <from tm.begin>
2: return insn
3: notify(?tm.begin↓ backward)
4: ?tm.begin↓
5: ok
6: invoke tm.begin
7: invoke insn
8: notify(!tm.begin↑, backward)
9: !tm.begin↑
10: ok
JPF Listener
Protocol Checker
[already visited state]

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 unnecessary 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 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 tests[1] 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.

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

(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);`

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

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

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 µcke model checker is presented in [14]. The Isabelle tool employs the µcke 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. Mehlitz, 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