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 <prefix><interface>.<method><suffix>. The prefix ? means accepting, ! emitting, the suffix ↑ means a request (of a method call) and ↓ a response (return from a call). Several shortcuts are defined: ?i.m is a shortcut for ?i.m↑ ; !i.m↓ and !i.m stands for !i.m↑ ; ?i.m↓. 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

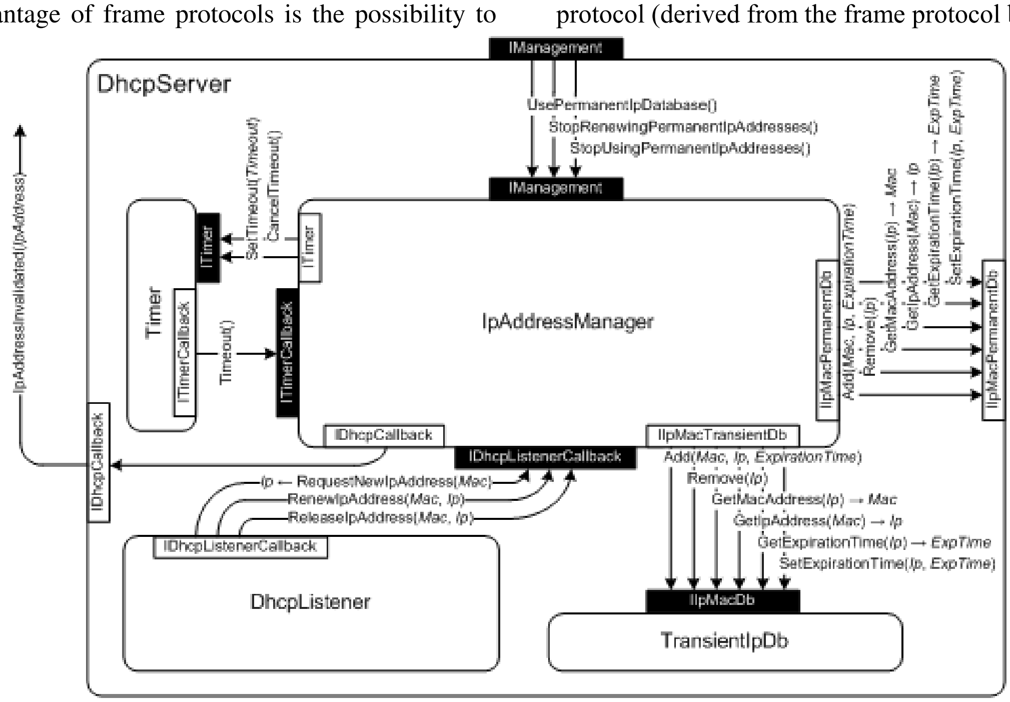

Figure 1: Architecture of the DhcpServer component

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:

```
!IIpMacDb.Add*  |  !IIpMacDb.Remove*  |
!IIpMacDb.GetMacAddress*        |
!IIpMacDb.GetIpAddress*         |
!IIpMacDb.GetExpirationTime*    |
!IipMacDb.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 outcoming 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 outcoming 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].

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

$$InitP\ ;\ (p_1 \mid p_2 \mid ... \mid p_N)\ ;\ FinishP \qquad (I)$$

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: $(p_1 \mid p_2 \mid ... \mid p_N)$ in (I) is replaced by

$$(...;p_{k-1};p_k;p_{k+1};...;\ (p_i \mid p_j)) + (...;p_{\underline{k}-1};p_{\underline{k}};p_{\underline{k}+1};...;\ (p_{\underline{i}} \mid p_{\underline{j}})) + ... + (p_1;...;p_N) \quad (II)$$

where an alternative with the parallel operator is introduced for any protocol tuple $<p_i, p_j>$ 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 $...;p_{k-1};p_k;p_{k+1};...$ contains all of the protocols $p_1, ..., p_N$ 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 $(p_i \mid p_j)$. This reflects the fact that the only "sensitive" (likely producing concurrency errors) protocols in the alternative are p_i and p_j. The sequence $...;p_{k-1};p_k;p_{k+1};...$ is intentionally chosen as a prefix (not a postfix) of $(p_i \mid p_j)$ to exclude this sequence from JPF backtracking triggered by execution of all interleavings of $(p_i \mid p_j)$. Sensitive

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

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

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

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

$$p_{k+1};\dots;p_N;(p_1 \mid \dots \mid p_k) \qquad \text{(III)}$$

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 $p_N;(p_1 \mid \dots \mid p_{N-1})$. Reduction is performed as long as the proportional weight of p_m is lower than a user-defined threshold; this assumes that ordering of $p_1 \mid \dots \mid p_N$ 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 $(p_1 \mid \dots \mid p_N)$ by

$$(p_i \mid p_j) + (p_i \mid p_j) + \dots \qquad \text{(IV)}$$

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 $(p_1 \mid p_2 \mid \dots \mid p_N)$; 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 ; (p_1 | p_2 | ... | p_N) ; 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 `TransientIpDb` (*project: CRE, 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 `IpAddressManager` (*project: CRE, 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 `Pessimistic Concurrency Manager` (*project: Perseus, 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 `OrderProcessor` (*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 `Pessimistic Concurrency Manager` (*project: Perseus, 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, IJCIS, 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