Extraction of Component-Environment Interaction Model Using State Space Traversal

Pavel Parizek
Department of Software Engineering
Faculty of Mathematics and Physics
Charles University, Malostranske namesti 25
Prague 1, 118 00, Czech Republic
parizek@dsrg.mff.cuni.cz

Nodir Yuldashev
Department of Software Engineering
Faculty of Mathematics and Physics
Charles University, Malostranske namesti 25
Prague 1, 118 00, Czech Republic
yuldashev@dsrg.mff.cuni.cz

ABSTRACT

Scalability of software engineering methods can be improved by application of the methods to individual components instead of complete systems. This is, however, possible only if a model of interaction between each component and its environment (rest of the system) is available. Since constructing formal models of interaction by hand is hard and tedious, techniques and tools for automated inference of the models from code are needed.

We present a technique for automated extraction of models of component-environment interaction from multi-threaded software systems implemented in Java, which is based on state space traversal. Models are captured in the formalism of behavior protocols, which allows to express parallel behavior explicitly. Java PathFinder is used to perform the state space traversal. We have implemented the technique in the Java2BP tool and applied the tool on two non-trivial software systems to show that our approach is feasible.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Documentation, Verification

Keywords

Software components, behavior protocols, Java PathFinder

1. INTRODUCTION

Many software systems are constructed in a modular manner, using reusable components with well-defined interfaces as basic building blocks [17]. An advantage of building software systems from well-defined components is that scalability of various software engineering-related methods can be improved by application of the methods to individual components instead of complete systems. This is important, for example, in verification and performance analysis.

Nevertheless, software engineering-related methods typically can be applied to a single component C of a software system only if a formal model of interaction between C and its *environment* E (rest of the system) is available — we denote it as the *model of C-E interaction* in this text. The model of C-E interaction is then used as an input to tools that implement the methods. Since construction of formal models of C-E interaction by hand is hard and tedious, techniques and tools for automated extraction (inference) of the models are needed.

1.1 Related Work

Several techniques that can be used for automated extraction of models of C-E interaction from the code of software systems were developed in recent years. The techniques can be divided into five groups according to (i) the target of an analysis — a component, an environment for the component, or the complete system composed of a component and its environment — and (ii) the underlying approach — learning of a finite automaton using the L^* algorithm [4] with a model checker as a teacher, static analysis or runtime analysis.

The first group includes techniques that use the L^* algorithm to infer a specification of valid usage of a component by any environment. For example, the technique presented in [2] aims at inference of a valid usage specification of a Java class in the form of a finite automaton that captures sequences of method calls that do not trigger an error (an exception) in the component. The main drawback of techniques in this group is that they involve invocations of a model checker, and therefore are prone to state explosion. On the other hand, use of model checkers as teachers allows to create precise specifications and models with respect to reachability of error states.

Techniques in the second group also aim at inference of a valid usage specification of a component, but they are based on static analysis of the component's implementation. Into this group belongs the technique presented in [12], which aims at analysis of Java components. The technique infers a set of predicates over the state of a component and history of method calls that determines how the component should be used in order to avoid an error state.

Techniques in the third group (e.g., [15]) use static analysis of the code of component's environments to extract an approximate model of component's valid usage. The tech-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'10 March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.

niques are based on the assumption that prevailing ways of component's usage by its environments are often correct — the techniques capture only the prevailing usage of the component, using statistical methods to identify it.

In general, techniques based on static analysis consider all execution paths in the code, but produce approximate models of C-E interaction due to the summarization of information from different execution paths.

The fourth group includes techniques based on runtime analysis of the whole system (a component and its environment). The techniques typically work in the following way: first, the set of traces of important events is recorded during execution of the system via runtime monitoring, and then the model in the form of a finite automaton is derived from the traces. Like in the case of the third group, the underlying assumption is that the typical usage of a component is correct. An example of this group is a technique presented in [3], which uses machine learning and stochastic methods to derive the specification of correct usage of a component from the set of traces recorded via runtime monitoring. Although runtime analysis techniques monitor concrete executions of a system and therefore are very precise, they capture only the traces that actually occured during a particular set of runs of the system — only a subset of execution paths of the system — and therefore suffer from limited coverage.

There are also hybrid techniques (the fifth group) that combine static and runtime analysis. For example, the technique proposed in [20] uses runtime analysis of the whole system to acquire traces of method calls during a program run, and static analysis of the component's code to identify the traces that may cause an exception to be thrown. The resulting model in the form of a finite automaton reflects only traces that do not cause an exception to be thrown.

A drawback common to all the techniques we are aware of is that they do not capture parallel execution of methods in multiple threads and therefore cannot be used to extract precise models of C-E interaction from multi-threaded software systems. One of the reasons is that they extract models and specifications in finite state machine-based formalisms, which cannot express parallel behavior explicitly. Formalisms based on finite automata allow to express parallelism only implicitly, e.g. via an automaton that accepts a language (a set of traces) of words (traces) corresponding to all interleavings of threads running in parallel.

1.2 Contribution

We present a technique for extraction of models of C-E interaction from multi-threaded Java programs (software systems implemented in Java), which is based on explicit traversal of the programs' state space. The extracted models of C-E interaction are expressed in the formalism of *behavior protocols* [14], which allows to capture parallel behavior of multiple threads explicitly. Java PathFinder (JPF) [19] is used to perform the state space traversal of Java programs.

The advantages of state space traversal with JPF over the existing techniques are: comparable precision to runtime analysis and comparable coverage to static analysis-based techniques. On the other hand, state space traversal suffers from state explosion when applied to software systems that involve high number of threads running concurrently.

We focus on Java programs that are built from components with a well-defined boundary in the form of Java interfaces — this is the case, for example, of Java classes that implement some Java interfaces or large Java libraries with well-defined API. We also distinguish between a component (component instance) and a component type.

2. RUNNING EXAMPLE

Key concepts of the proposed technique will be illustrated on a simple MediaPlayer application (Fig. 1) that is implemented in Java. The application consists of three Java components: `Player`, `LocalFS`, and `ZipFS`. The `Player` component is responsible for presentation of media files, whose content it retrieves from the other two components. The `LocalFS` component provides access to a local file system on a harddisk and the `ZipFS` component provides access to the contents of a zip archive.

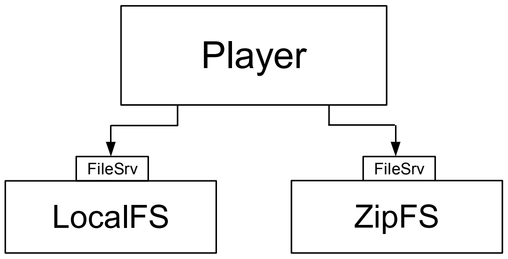

Figure 1: Architecture of MediaPlayer

The `LocalFS` and `ZipFS` components are of the same type `FileSystem`, and, in particular, both of them provide the `FileSrv` interface with the signature presented on Fig. 2.

```
public interface FileSrv {
  File openFile(String fileName);
  void closeFile(File f);
  String[] readDir(String dirName);
}
```

Figure 2: Java signature of the `FileSrv` interface

A fragment of a simplified implementation of the `Player` component in Java is on Fig. 3. The key characteristic of the implementation is that selection of files to be played is performed in the main thread, while each file is played in a separate thread (instance of the `PlayThread` class). Note also that at most two media files can be played concurrently.

3. BACKGROUND

3.1 Behavior Protocols

Behavior protocols [14] is a formalism for modeling of interaction between a software component and its environment (rest of a system) in terms of sequences and parallel interleavings of method calls. Data, including method parameters and return values, are neglected by the formalism.

A behavior protocol *prot* is an expression that specifies a set $L(prot)$ of finite traces of method call-related events on component interfaces. Four kinds of atomic events are supported — `?i.m^` (acceptance of invocation of method `m` on interface `i`), `!i.m^` (emit of a method invocation), `?i.m$` (acceptance of a return from a method), and `!i.m$` (emit of

```
public class Player {
  public void run() {
    while (!exit) {
      String[] curDir = new String[]{"/"};
      String path = selectPathInGUI(curDir);

      while (isDir(path)) {
        if (isZip(path))
            curDir = zipFS.readDir(path);
        else curDir = localFS.readDir(path);
        path = selectPathInGUI(curDir);
      }

      while (numberOfPlayThreads > 2) wait();

      if (isZip(path))
          new PlayThread(zipFS, path).start();
      else new PlayThread(localFS, path).start();
    }
  }
}

class PlayThread {
  public void run() {
    File f = fs.openFile(fileName);
    // load all data from file
    fs.closeFile(f);
  }
}
```

Figure 3: Implementation of the `Player` component

a return). More complex protocols can be constructed from the atomic events using the standard regular operators — `;` (sequence), `+` (choice), and `*` (repetition) — and the parallel composition operator `|`, which generates all interleavings of event traces defined by its operands. Several shortcuts that enhance readability are also supported: `?i.m` stands for `?i.m^ ; !i.m$` and `?i.m{`*prot*`}` stands for `?i.m^ ;` *prot* `; !i.m$`. An empty protocol is denoted by `NULL`.

For example, interaction of the `LocalFS` component with its environment from the perspective of `LocalFS` can be specified in the formalism of behavior protocols in this way:

```
?FileSrv.readDir*
|
(
  (?FileSrv.openFile ; ?FileSrv.closeFile)
  +
  NULL
)*
|
(
  (?FileSrv.openFile ; ?FileSrv.closeFile)
  +
  NULL
)*
```

The protocol states that methods of `LocalFS` can be called in three parallel threads. A finite number of calls to the `readDir` method can be performed in one thread, and the methods `openFile` and `closeFile` can be called in the correct sequence in the other two threads.

3.2 Java PathFinder

Java PathFinder (JPF) [19] is an extensible and customizable explicit-state model checker for Java bytecode programs. However, in general it can be used as a tool for state space traversal of Java bytecode programs that can gather specific information about program's execution during the traversal. It is implemented as a special Java virtual machine (JPF VM) that supports backtracking, state matching and non-deterministic choice — in particular, JPF VM executes the given Java program in all possible ways with respect to thread scheduling.

The state space of a Java program is constructed on-the-fly by JPF. A transition is a sequence of bytecode instructions that is terminated either by a *scheduling-relevant instruction* or by an instruction corresponding to non-deterministic data choice. A bytecode instruction is scheduling-relevant if its effects are visible to other threads in a system — this is the case, e.g., of the `PUTFIELD` instruction for write to an object field (variable on the heap). All instructions in a single transition are executed by the same thread; however, any two adjacent transitions on any path in the state space can be performed by two different threads — thread context switch can occur only at a transition boundary. A state in the state space is a snapshot of the current state of the checked Java program at the end of a transition, as viewed by JPF VM.

The key extension mechanisms of JPF with respect to the technique proposed in this paper are *listener API*, *choice generators* and *scheduler factory*. The listener API provides means for monitoring of the state space traversal and execution of a Java program by JPF VM — a *JPF listener* is notified about (i) state space search-level events like completion of a transition or backtracking from a state, and (ii) VM-level events like execution of a bytecode instruction or start of a thread. The mechanism of choice generators unifies all possible causes for a choice among different ways of program's execution from a particular state, including thread scheduling. A specific instance of choice generator is associated with each state and maintains the list of enabled and unexplored transitions leading from the state. The choice generator API also provides means for altering the set of enabled and unexplored transitions — it is possible, for example, to select specific transitions that should be explored. Using custom choice generators and custom scheduler factory, it is possible to reduce the number of instructions that are effectively considered as scheduling-relevant by JPF and therefore to reduce the size of the state space of a multi-threaded Java program.

4. EXTRACTION OF BEHAVIOR PROTOCOL FROM JAVA CODE

In this section, we describe the state space traversal-based technique for extraction of the model of C-E interaction in behavior protocols from a Java program. We assume that the program consists of two parts — a specific component C of type T_C that can be unambiguously identified among all instances of T_C in the program, and the rest of the program, i.e. the environment E of C. Moreover, a well-defined boundary between C and E in the form of a set of instances of Java interfaces must exist. The assumption is obviously fulfilled if there is only a single component of type T_C in the program. The case, when there are two or more components of type T_C in the program, is discussed in Sect. 4.3.

The input of the technique is a complete Java program with the `main` method, and a list of interfaces that form the boundary between C and E (*C-E boundary*), and the output is the model of interaction between C and E in the formalism of behavior protocols (*C-E protocol*). The C-E protocol produced by the technique is expressed from the perspective of C — calls of C from E are expressed by events of the form `?i.m^` and `!i.m$`, while calls of E from C are expressed by events of the form `!i.m^` and `?i.m$`.

The process of extraction of the C-E protocol from a Java program involves JPF, which is used for traversal of the Java program's state space, and three extensions of JPF — two JPF listeners and a custom scheduler. The C-E protocol is extracted from the given Java program in two steps:

1. For each thread T_i in the program, a *thread protocol* $tprot_i$ expressing all sequences of actions at the C-E boundary that can be performed by T_i is extracted.

2. The complete C-E protocol is constructed as a parallel composition of thread protocols $tprot_1, \ldots, tprot_N$ that reflects the interplay among threads in the given program (e.g., synchronization).

The algorithm for extraction of a thread protocol for a single thread is described in Sect. 4.1 — it can be used also for extraction of the complete C-E protocol for a single-threaded Java program. The algorithm for determination of parallel composition of thread protocols is described in Sect. 4.2.

We have configured JPF in such a way that it does not check any properties during the state space traversal of the given Java program and therefore the traversal cannot be terminated due to the occurrence of a property violation. This way it is ensured that the extracted C-E protocol reflects the "real behavior" of the Java program that can be observed during executions of the program outside of JPF — this means, for example, termination of the program upon an uncaught exception on one hand, but continuation of program's execution upon occurrence of a race condition on the other hand.

4.1 Extraction of Thread Protocol

A thread protocol $tprot_i$ for a specific thread T_i is constructed by a JPF listener on-the-fly during traversal of the Java program's state space by JPF. Only the transitions executed by T_i are taken into account in construction of $tprot_i$. All the logic of the construction algorithm is defined in handlers of specific notifications from JPF. The key idea of the algorithm is the following:

- During processing of a yet unexplored transition by JPF, a sequence (trace) of method invocations and returns from methods on interfaces forming the C-E boundary (i.e. a sequence `e1`, ..., `eN` of atomic events of the protocol) that are performed during the transition is recorded via inspection of all executed Java bytecode instructions. When the transition is terminated, the corresponding protocol of the form `e1 ; ... ; eN` is associated with the transition.

- During backtracking, the resulting thread protocol is created from the sequences of atomic events associated with transitions using protocol operators. The form of the thread protocol depends on the structure of the state space. If JPF backtracks to a state that has only one successor then the sequence operator `;` is used, while if JPF backtracks to a state having more than one successor (e.g. as a consequence of non-deterministic data choice) then the choice operator `+` is used — the resulting protocol will have the form `p1 + p2 + ... + pN`, where symbols `p1`, `p2`, ..., `pN` represent the sub-protocols associated with transitions starting in the state to which JPF backtracks.

The idea is illustrated on Fig. 4 — Fig. 4a shows the case of a state having just one successor, while Fig. 4b shows the case of a state that has two successors.

Note that the repetition operator `*` is not used in construction of a thread protocol, since it abstracts away the number of iterations, which may be important in some cases (e.g. for performance analysis). A possible approach to recognition of loops in a protocol is to detect repeated executions of the same instruction (instruction at the same position in the same Java method) and to capture event traces recorded between two successive executions of such an instruction as an operand of the `*` operator.

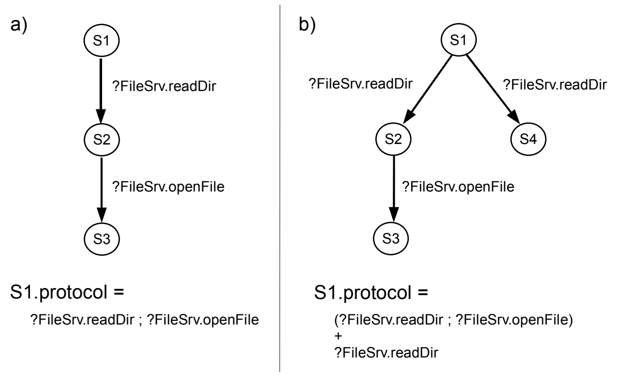

Figure 4: Merging of protocols associated with transitions and states

For illustration, thread protocols for threads involved in interaction between the `Player` and `LocalFS` components (Sect. 2) may have the form that is depicted on Fig. 5 — `M` stands for the main thread, and `P1` and `P2` stand for instances of the `PlayThread` class.

```
M:  ?FileSrv.readDir ; ?FileSrv.readDir ;
    ?FileSrv.readDir
P1: ?FileSrv.openFile ; ?FileSrv.closeFile ;
    ?FileSrv.openFile ; ?FileSrv.closeFile
P2: ?FileSrv.openFile ; ?FileSrv.closeFile
```

Figure 5: Thread protocols for interaction between the `LocalFS` component and its environment (`Player`)

Since for each thread T_i only the transitions executed by T_i are taken into account in construction of $tprot_i$, it is sufficient to traverse only those state space paths that correspond to a single interleaving of all threads in the given program in order to identify thread protocols $tprot_1, \ldots, tprot_N$, and therefore this step is not prone to state explosion. Technically, all thread protocols are extracted in a single run of JPF. This optimization is implemented by exploring only one choice for each choice generator (CG) related to thread scheduling; other choices in such a CG are ignored. The restriction to one choice for each thread scheduling-related CG is correct with respect to identification of actions performed by individual threads, since the choices that were not selected will be enabled at the next scheduling point (i.e., when the next thread scheduling-related choice is to be made). Sets of choices associated with CGs related to non-deterministic data choice are not altered at all.

4.2 Parallel Composition of Thread Protocols

The key challenge in construction of a parallel composition of thread protocols $tprot_1, \ldots, tprot_N$ is to capture the interplay among threads, where by interplay we mean (i) starting and termination of other threads in one thread and (ii) synchronization among multiple threads. However, in this work we focus only on starting and termination of threads — we do not consider synchronization among threads via monitors and calls of the `wait` and `notify` methods.

A precise parallel composition of thread protocols with respect to starting and termination of threads can be constructed only if the following information is available:

- the earliest possible start time and the latest possible termination time of each thread T_i with respect to interplay of all threads in the program, and
- ordering of start and termination times of all threads.

This information is, as in the first step, determined by a JPF listener during traversal of the Java program's state space by JPF. Nevertheless, since the interplay of threads depends on thread scheduling, it is not sufficient to traverse only the state space paths corresponding to a single thread interleaving — times of thread start and termination are computed over all paths in the state space. Whole state space of the Java program has to be traversed and thus this step is prone to state explosion.

The times of thread start and termination can be measured only if common clock ("reference temporal dimension") are available. We use virtual clocks that are defined on the basis of flow of control in the *main* thread — current value of the virtual clock on a path p in the state space is the number of the last transition performed by the main thread in p. A valid clock value is always available during the program's lifetime, since the main thread runs from the start of the program until its termination.

All the necessary time-related information is determined by the JPF listener in the handlers of specific notifications from JPF. The information includes the absolute time of start and termination of each thread, which is based on the virtual clock, and the start and termination times of each thread T_i with respect to interplay of all threads, which are encoded via special marks in the thread protocols. During traversal of the program's state space by JPF, the current value of the virtual clock is maintained and the state space of each thread protocol is traversed, so that it is possible to insert the marks at proper points in the thread protocols.

At the end of exhaustive traversal of the program's state space, the information about the earliest possible start time and the latest possible termination time of each thread with respect to interplay of all threads is represented by the marks in thread protocols. The ordering of start and termination times of all threads may not be linear — the incomparable times can be ordered, e.g., on the basis of thread numbers.

The thread protocols from Fig. 5 annotated with time marks may look like those depicted on Fig. 6 and the ordering of events may be `M_s` < `P1_s` < `P2_s` < `P2_t` < `P1_t` < `M_t`. A mark of the form `[T_s]` denotes the start time of a thread T, and a mark of the form `[T_t]` denotes the termination time of a thread T.

Although the algorithm for determination of thread start and termination times employs exhaustive state space traversal and therefore it is inherently prone to state explosion, the size of the state space traversed by JPF can be significantly reduced if instructions for accesses to shared variables are not considered as scheduling-relevant. Technically, this optimization is implemented by a custom scheduler for JPF.

Using all the time-related information, the parallel composition of thread protocols $tprot_1, \ldots, tprot_N$, i.e. the resulting C-E protocol, is constructed via syntactical operations upon thread protocols in the following five steps:

1. A sequence TI of intervals is defined on the basis of the ordering of thread start and termination times. E.g., for the ordering `M_s` < `P1_s` < `P2_s` < `P2_t` < `P1_t` < `M_t` we get a sequence $TI =$ `{<M_s, P1_s>, <P1_s, P2_s>, <P2_s, P2_t>, <P2_t, P1_t>, <P1_t, M_t>}`.

2. Each thread protocol $tprot_i$ annotated with marks is decomposed into a set TPF_i of *thread protocol fragments* such that marks corresponding to elements of TI form the boundaries of the fragments — i.e., there can be no mark in a fragment except at its beginning and end. Then, a union TPF of sets TPF_i for all thread protocols is formed.

3. For each interval $int \in TI$, a set TPF_{int} of thread protocol fragments whose boundaries are formed by int is selected from TPF. The set TPF_{int} for an interval int contains thread protocol fragments that specify sequences of method calls on the C-E boundary that can occur in the interval int.

4. For each interval $int \in TI$, thread protocol fragments in the set TPF_{int} are composed using the parallel operator `|`, yielding a fragment of the C-E protocol.

5. All fragments are composed using the sequence operator `;`, yielding the resulting C-E protocol.

Given the annotated thread protocols on Fig. 6 and the ordering `M_s` < `P1_s` < `P2_s` < `P2_t` < `P1_t` < `M_t`, the resulting C-E protocol will take the form as in Fig. 7.

```
?FileSrv.readDir ; (
  ?FileSrv.readDir
  |
  (?FileSrv.openFile ; ?FileSrv.closeFile)
) ; (
  ?FileSrv.readDir
  |
  (?FileSrv.openFile ; ?FileSrv.closeFile)
  |
  (?FileSrv.openFile ; ?FileSrv.closeFile)
)
```

Figure 7: C-E protocol modeling interaction between the `LocalFS` component and its environment

```
M:  [M_s] ?FileSrv.readDir ; [P1_s] ?FileSrv.readDir ; [P2_s] ?FileSrv.readDir [P2_t][P1_t][M_t]
P1: [M_s][P1_s] ?FileSrv.openFile ; ?FileSrv.closeFile ; [P2_s] ?FileSrv.openFile ; ?FileSrv.closeFile [P2_t][P1_t][M_t]
P2: [M_s][P1_s][P2_s] ?FileSrv.openFile ; ?FileSrv.closeFile [P2_t][P1_t][M_t]
```

Figure 6: Thread protocols annotated with time marks

However, the five-step algorithm described above does not give correct result if a thread T_i creates another thread T_j only in some execution paths, i.e. if the thread protocol $tprot_i$ for T_i specifies that T_j is created only in some operands of a choice operator + occuring in $tprot_i$. In this case, a top-level choice operator with two operands is added to the C-E protocol such that the thread T_j is created in one of the branches and not in the other.

4.3 Multiple Instances of Component Type

Multiple components of the same type can be represented in the program's architecture in two ways: (i) as undistinguishable elements of a single collection of components, or (ii) as separate entities (subsystems) that are explicitly specified in the architecture. Note that the components have the same environment in the first case and possibly different environments in the second case.

Given a system with multiple components C_i of the same type T_C, first a *component instance protocol* $cprot_i$ has to be extracted for each C_i separately using the algorithm described in Sections 4.1 and 4.2, and then the C-E protocol can be created from the component instance protocols depending on the way the components are used in the system.

The resulting C-E protocol has to reflect the architectural view on the components. If the components are undistinguishable elements of a single collection, then they should be used in the same (or very similar) way by the rest of a program. The C-E protocol should model the interaction of any component in the collection with the common environment E in such a case. An obvious and viable solution is to create the C-E protocol as a union of all component instance protocols $cprot_i$, such that the set $L(CEprot)$ of event traces specified by the C-E protocol is a superset of the set of traces $L(cprot_i)$ for each C_i.

On the other hand, if the components are separate entities in the architecture, then they are typically used in different ways — specifically, their component instance protocols differ to a great degree. In such a case, the two most appropriate solutions are: (i) to return a set of C-E protocols (one for each component), or (ii) to create a single C-E protocol such that individual component instance protocols are syntactically composed using the choice operator +.

5. IMPLEMENTATION: JAVA2BP

We have implemented the proposed technique on top of JPF in the `Java2BP` tool. The tool can be applied only to complete Java programs (featuring `main`) with fully specified inputs, since JPF works only for closed systems. Given a Java program whose behavior depends on external input (e.g., on data entered by a user via GUI or received over a network) and behavior of external entities (e.g., on sequences of actions performed by the user via GUI), then a fragment of Java code that provides the external input and simulates the behavior of external entities — a simulator — has to be provided together with the program. Methods for nondeterministic data choice, which are provided by the `Verify` class (a part of the JPF's API), can be used in the simulator to capture "random" behavior of users.

Application	Time	Memory	States
CRE demo	707 s	235 MB	120998
CoCoME	3751 s	243 MB	1987872

Table 1: Results of experiments with Java2BP

The Java2BP tool has limited support for programs that include calls of native methods, e.g. via Java libraries for file I/O or networking, since JPF cannot handle native methods in general and, in the current version, provides wrappers only for selected classes from the corresponding Java packages (`java.io` and `java.net`). Given a Java program that calls native methods for which the wrappers are not available in the current distribution of JPF, then calls of such methods have to be abstracted before it is possible to apply Java2BP to such a program.

5.1 Experiments

In order to find whether the proposed technique is feasible for non-trivial multi-threaded Java programs built from components, we have applied the Java2BP tool on two applications for the Fractal component model [8] — the demo application developed in the CRE project [1] ("CRE demo" for short) and a software system [7] developed in our group as a solution to the CoCoME assignment [9] ("CoCoME").

The CRE demo (1700 loc) is a prototype of a software system for providing WiFi internet access at airports. It supports, for example, payment via a credit card and assignment of IP addresses via DHCP. It also includes a simulator — the `Simulator` class — that was developed in the CRE project for the purpose of runtime checking of the application. Simulator exercises the components in the application in all ways that are allowed by their behavior specifications and, in particular, it calls methods of the components in two threads running in parallel. Java code of the CRE demo does not contain any calls of native methods and therefore Java2BP can be applied on the CRE demo directly.

CoCoME (2800 loc) is a prototype of a trading system for supermarkets. Architecture of CoCoME has two parts: (i) an inventory management system, which is responsible for management of databases of products and items, and (ii) a cash desk line formed by a set of cash desks. Each cash desk is represented by several components that control cash desk hardware (e.g., bar code scanner and credit card reader). Although the number of cash desks in a system can be arbitrary, for the purpose of experiments we used a configuration with two cash desks. Like in the case of CRE demo, a simulator is available that we developed for the purpose of testing and performance evaluation, and the Java code of CoCoME does not contain calls of native methods. Simulator runs the main thread and one additional thread per cash desk — that makes three threads in total in the case of configuration with two cash desks.

The results of experiments on CRE demo and CoCoME are listed in Table 1. Value of the "Time" column expresses the total time needed for extraction of a C-E protocol for any component in an application, which is equal to the running time of the application in JPF, and value of the "Memory" column expresses the memory needed for the extraction. Value of the "States" column expresses the number of states traversed by JPF during computation of threads' start and termination times via state space traversal.

6. APPLICATIONS IN SOFTWARE ENGINEERING

While the Java2BP tool can be applied on fully implemented software systems in Java, provided that native calls unsupported by JPF are abstracted and a simulator is available, we envisage usage of the tool during development of a software system especially in the following way:

1. During initial stages of implementation (prototyping), Java2BP is applied on a prototype of the software system to extract a C-E protocol for each component.
2. C-E protocols are then used in the latter stages of the development process as an input for tools and methods that perform various software engineering tasks upon the full implementation of the system in a compositional manner, i.e. upon each fully implemented component of the system separately.

The advantage of applying Java2BP on prototypes is twofold: (i) they typically contain only a few calls of native methods (or none at all), which can be easily abstracted by hand, and (ii) simulators and test harnesses are created anyway for the purpose of testing and other analyses of the prototype. Moreover, prototypes typically have smaller state space than fully implemented systems and therefore the chance that state explosion occurs during extraction is much lower. Nevertheless, a limitation of this approach to use of Java2BP is that both the component interfaces and interaction among components cannot be changed during development of the full implementation of the system, otherwise C-E protocols extracted from a prototype of the system may not be valid.

The list of software engineering tasks, where C-E protocols extracted by Java2BP could be useful, includes compositional verification and performance analysis. In compositional verification, C-E protocol for a particular component C can be used as an environment assumption of C for the purpose of assume-guarantee reasoning [10], i.e. as a behavior model of the C's abstract environment [18][13]. In performance analysis, C-E protocol of C can be used as a basis of the C's usage profile — the full usage profile can be created by manual annotation of operands (branches) of each choice operator (`+`) in the C-E protocol with probabilities that the branches will be taken [5].

C-E protocols extracted by Java2BP can be used also for the purpose of code comprehension and debugging — for example, to find which methods are actually called during a run of a software system and at what time. This way, unexpected or errorneous sequences of method calls in prototypes of complex systems with many components can be discovered during initial stages of implementation.

Nevertheless, we would like to emphasize that the C-E protocols extracted by Java2BP are supposed to be used as an input for tools, since they will not be readable by humans except for simple Java programs. Syntactical post-processing of some kind could be used to make protocols more concise and readable.

7. EVALUATION

The proposed technique addresses some of the drawbacks of existing techniques for inference of behavior specifications and models from code (Sect. 1.1), while preserving their important advantages. Specifically, the technique

- has precision comparable to existing runtime analysis-based techniques, since JPF performs no summarization of information from different execution paths and also no abstraction of Java code, and
- considers all execution paths in a program, i.e. not only those recorded in a particular run (execution) of the program, and therefore has coverage comparable to static analysis-based techniques.

Unlike the existing approaches, the proposed technique captures concurrent execution of multiple threads, and therefore can be used also for multi-threaded software systems. The results of experiments on CRE demo and CoCoME (Sect. 5.1) show that the Java2BP tool can be successfully applied on non-trivial multi-threaded software systems built from Java components.

Note also that while the proposed technique is currently specific to behavior protocols, it can certainly be generalized to any formalism that allows to specify a set of traces of events (e.g., method calls) and supports sequential composition, choice and parallel composition. The list of suitable formalisms includes various process algebras [6]. Similarly, the idea of extraction of interaction models from code using state space traversal is not specific to Java and JPF — it can be applied to software systems implemented in any programming language, provided there exists a tool for state space traversal of programs written in the language that supports monitoring of the traversal.

In the rest of this section, we discuss limitations of the proposed technique from the perspective of the following aspects: (i) scalability, (ii) precision, and (iii) automation.

Ad (i) The main limitation of the proposed technique from the perspective of scalability is that it is prone to state explosion. Although the technique scales better than verification with JPF, since much smaller state space has to be traversed by JPF during extraction than during verification due to optimizations described in Sect. 4, still use of the Java2BP tool is not feasible for complex multi-threaded software systems. Note, however, that state explosion can be avoided for most single-threaded systems — big data domains are not a problem with respect to state explosion, if a simulator employs the JPF's API for non-deterministic data choice (the `Verify` class) in a reasonable way.

Ad (ii) From the perspective of precision, the main limitation is that C-E protocols extracted by Java2BP may over-approximate the actual behavior of programs with respect to synchronization among threads, since the algorithm for construction of parallel composition of thread protocols (Sect. 4.2) neglects synchronization via monitors and calls of the `wait` and `notify` methods. This is an issue for programs, in which some methods on the C-E boundary are called inside `synchronized` blocks — some sequences of method call-related events on the C-E boundary are executed atomically in such a case, while the C-E protocol specifies concurrent execution and thus over-approximates the actual behavior of such programs.

Ad (iii) The main limitation from the point of view of automated application of the Java2BP tool is that the tool can be directly applied only to complete Java programs that have fully specified inputs and call only native methods supported by JPF — this is typically the case of prototypes, for which test harnesses and simulators are available. Although the calls of native methods that are not supported by JPF can be abstracted manually, it is possible to automatize the abstraction of such calls using tools based on libraries for transformation of Java source code and bytecode (e.g. SOOT [16]). Technically, calls of selected methods from the system libraries (e.g., from the packages `java.io`, `java.net` and `javax.*`) can be removed and non-deterministic choice can be used at program points, where control-flow depends on results of removed method calls.

8. CONCLUSION

We proposed a technique for automated extraction of models of component-environment interaction from multi-threaded Java programs, which is based on state space traversal. The technique uses Java PathFinder to perform the state space traversal of Java programs. Extracted models of C-E interaction are expressed in behavior protocols.

We have implemented the technique in the Java2BP tool and applied the tool on two non-trivial multi-threaded Java programs built from components. Results of experiments show that use of Java2BP is feasible for prototypes of complex multi-threaded software systems. Nevertheless, it is still prone to state explosion when applied to programs that involve high number of threads running in parallel. The key benefit of our technique is that, unlike the existing approaches to inference of behavior models and specifications from code, it captures parallel behavior of multiple threads and therefore can be used to extract models of C-E interaction from multi-threaded programs.

In future, we plan to improve the precision of extraction by considering synchronization among threads, and to design (or use) optimizations and heuristics that would make it possible to efficiently extract C-E interaction models from more complex multi-threaded systems in Java. We also plan to develop a tool for abstraction of calls of native methods in near future and then to evaluate our approach on large software systems. Moreover, we would like to extend the proposed technique towards the formalism of threaded behavior protocols (TBP) [11], which supports also state variables and control-flow based on values of the state variables.

9. ACKNOWLEDGMENTS

This work was partially supported by the Q-ImPrESS research project (FP-215013) by the European Union under the ICT priority of the 7th Research Framework Programme, and by the Grant Agency of the Czech Republic project 201/08/0266.

10. REFERENCES

[1] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil. Component Reliability Extensions for the Fractal Model, `http://kraken.cs.cas.cz/ft/public/public_index.phtml`, 2006.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes, ACM SIGPLAN Notices, 40(1), 2005.

[3] G. Ammons, R. Bodik, and J. R. Larus. Mining Specifications, In Proc. of the 29th Symposium on Principles of Programming Languages, ACM, 2002.

[4] D. Angluin. Learning Regular Sets from Queries and Counterexamples, Information and Computation, 75(2), 1987.

[5] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for Model-Driven Performance Prediction, J. of Systems and Software, 82(1), 2009.

[6] J. A. Bergstra, A. Ponse, and S. A. Smolka. Handbook of Process Algebra, Elsevier, 2001.

[7] 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, In the Common Component Modeling Example: Comparing Software Component Models, LNCS, vol. 5153, 2008.

[8] 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.

[9] CoCoME: Common Component Modeling Example, `http://www.cocome.org`, accessed in April 2009.

[10] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh. Assume-Guarantee Verification of Source Code with Design-Level Assumptions, In Proceedings of the 26th ICSE, IEEE CS, 2004.

[11] J. Kofron, T. Poch, and O. Sery. TBP: Code-Oriented Component Behavior Specification, Accepted for publication in Proceedings of SEW-32, IEEE CS, 2009.

[12] M. G. Nanda, C. Grothoff, and S. Chandra. Deriving Object Typestates in the Presence of Inter-object References, In Proceedings of the 20th OOPSLA, ACM Press, 2005.

[13] P. Parizek and F. Plasil. Modeling of Component Environment in Presence of Callbacks and Autonomous Activities, In Proceedings of TOOLS EUROPE 2008, LNBIP, vol. 11, 2008.

[14] F. Plasil and S. Visnovsky, Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, 28(11), 2002.

[15] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static Specification Mining Using Automata-based Abstractions, Proc. of ISSTA'07, ACM Press, 2007.

[16] Soot: a Java Optimization Framework, `http://www.sable.mcgill.ca/soot/`, accessed in April 2009.

[17] C. Szyperski. Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, 2002.

[18] O. Tkachuk and S. P. Rajan. Application of Automated Environment Generation to Commercial Software, Proceedings of ISSTA'06, ACM Press, 2006.

[19] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs, Automated Software Engineering Journal, 10(2), 2003.

[20] J. Whaley, M. C. Martin, and M. S. Lam. Automatic Extraction of Object-Oriented Component Interfaces, In Proceedings of ISSTA'02, ACM Press, 2002.