Assume-Guarantee Verification of Software Components in SOFA 2 Framework[1]

Pavel Parizek[1], Frantisek Plasil[1,2]

[1]Distributed Systems Research Group, Department of Software Engineering,
Faculty of Mathematics and Physics, Charles University in Prague
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{parizek,plasil}@dsrg.mff.cuni.cz
[2]Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodarenskou vezi 2, 182 07 Prague 8, Czech Republic

Abstract. A key problem in compositional model checking of software systems is that typical model checkers accept only closed systems (runnable programs) and therefore a component cannot be model-checked directly. A typical solution is to create an artificial environment for the component such that composition of them forms a runnable program that can be model-checked. While it is possible to create a universal environment that performs all possible sequences and interleavings of calls of the component's methods, for practical purposes it is sufficient to capture in this way just the use of the component in a particular software system–this idea is expressed by the paradigm of assume-guarantee reasoning.
In this paper, we present our approach to assume-guarantee-based verification of software systems in the context of the SOFA 2 component framework. We provide an overview of our approach to the construction of an artificial environment for verification of SOFA 2 components implemented in Java with the Java PathFinder model checker. We show the benefits of our approach on results of experiments with a non-trivial software system and discuss its advantages over other approaches with similar goals.

1 Introduction

There has been a general trend in software engineering towards construction of software systems in a modular manner, using components with well-defined interfaces as basic building blocks [1]. This trend is visible especially in enterprise software systems, since use of components as building blocks promotes reuse and makes code updates easier, thus reducing the cost of software development and maintenance.

A typical process of development of a software system from well-defined components (supported by formal verification) consists of the following five phases:

- design of the system's architecture and component interfaces,
- definition of a design-level model of behavior (behavior specification) of each component in the system,
- verification of compatibility (compliance) of components' behavior models,
- implementation of components in a programming language, and

[1] This paper is a preprint of a paper accepted by IET Software and is subject to Institution of Engineering and Technology Copyright. When the final version is published, the copy of record will be available at **IET Digital Library**.

- checking whether the implementation of each component satisfies its behavior specification and whether the whole system is free of statically detectable errors.

Behavior specification of a component may have the form of (i) contracts (preconditions and postconditions) for individual methods defined, e.g., in a formalism like JML [2], or (ii) finite state machine (FSM) or expression in a process algebra-based formalism that describes the valid sequences of calls of component methods [3][4][5]. In this paper, we focus on checking whether component implementations satisfy (obey) their behavior specifications that describe the valid sequences of method calls and whether they are free of low-level concurrency errors like deadlocks and race conditions.

The verification technique that is most suitable for checking concurrent software systems against temporal properties like compliance with FSM-based behavior specification and absence of concurrency errors is model checking [6]. It is based on exhaustive traversal of the state space of a software system model (implementation) to achieve systematic exploration of all execution paths of the system – in particular, it can detect subtle errors that occur only in a specific thread scheduling sequence.

The main limitation of software model checking is that it does not scale in terms of the size of software systems due to the well-known problem of state explosion. The state explosion problem manifests itself especially in model checking of a whole software system at once – therefore, in case of software systems built from components, a natural solution is to apply compositional techniques in order to improve scalability. The basic idea of compositional model checking [7] is to verify the behavior (determined by design model or implementation) of each component in isolation and infer global properties of the whole system from the results of verification of individual components. A single component typically has smaller state space than the whole system and therefore model checking of the component is less prone to state explosion.

Another problem in model checking of a component is that a typical software model checker accepts only a runnable program (closed system). A component is inherently an open system – its behavior depends on the context (environment) in which it is used – and therefore cannot be model-checked directly. We call this issue the *problem of missing environment*. A typical solution is to create an *artificial environment* which composed with the component yields a runnable program that can be model-checked [8] [9].

The artificial environment should be constructed in such a way that it exercises the component in various ways in order to discover as many errors as possible in the component's model (implementation). Specifically, the artificial environment has to perform various sequences of calls of component's methods (with "reasonably chosen" parameter values) in one or more concurrent threads. One option is to create a *universal environment* that performs all possible sequences and interleavings of calls of component's methods to challenge robustness of the component. However, model checking with a universal environment is obviously infeasible for non-trivial components due to state explosion. Nevertheless, a component is typically expected to work properly only in specific environments [10], e.g. those using the component in a way that is compliant with its behavior specification. Therefore, for practical purposes, it is sufficient to use an artificial environment that simulates the behavior of the set of actual environments, in which the component can be used.

The idea of using an artificial environment that represents the behavior of a particular set of actual environments is expressed by the paradigm of assume-guarantee reasoning (AGR) [11]. Using this paradigm, the general process of verification of a component consists of the following steps [12] (*AGR verification*):

1. An *environment assumption* is specified (constructed), which characterizes the behavior of a set of specific contexts (actual environments) in which the component is expected to work properly.
2. An artificial environment for the component is constructed on the basis of the environment assumption – a runnable program composed of the component and the artificial environment is created this way.
3. The runnable program is model-checked in order to find whether the component (implementation or its model) satisfies its behavior specification and contains no low-level errors, when used in compliance with the environment assumption.

Before the component can be used in an actual software system, it has to be also checked whether the rest of the software system (other components) satisfies the environment assumption, i.e. whether the system will interact with the component in a way compliant with the environment assumption. Only if both checks are successful, it is guaranteed that the component will work correctly (in conformity with its behavior specification) in the given system.

Both the environment assumption and artificial environment can be in principle written by hand, but this would be a daunting task even in simple cases. Therefore, key challenges of applying AGR verification include: (i) construction of such an environment assumption that model checking of the runnable program composed of the component and artificial environment is not prone to state explosion [13], (ii) to automate construction of the environment assumption based either on (a) behavior specification of the component (to fully exercise it in a way compliant with the specification–*standalone approach*) or (b) behavior models of actual environments (to provide their union–*context approach*), and (iii) automated generation of the artificial environment from the assumption.

In this paper, we present our solution to these challenges in the context of the SOFA 2 component framework [14]. We provide an overview of our approach to construction of an artificial environment for AGR verification of SOFA 2 components implemented in Java using the Java PathFinder model checker (JPF) [15], and we show benefits of our approach on results of its application to a non-trivial software system built of SOFA 2 components. We use the formalism of behavior protocols [3] for behavior specification of components and for definition of environment assumptions.

The rest of the paper is structured as follows. We provide an overview of the SOFA 2 component framework, behavior protocols and JPF in Section 2. A software system that is used for illustration of presented ideas is introduced in Section 3. In Section 4, we provide an overview of our approach to AGR verification of SOFA 2 components implemented in Java with JPF. We evaluate our approach in Section 5, and discuss related work in Section 6. Then we conclude in Section 7.

2 Background

2.1 SOFA 2 Component Framework

The SOFA 2 component framework consists of a component model [16] and a runtime environment [17]. The component model defines (i) the concepts of interface and component, (ii) the way components can be composed to form a system/application, and (iii) other abstractions related to lifecycle of individual components and complete systems. The runtime environment reflects the SOFA 2 execution model and provides several development and administrative tools. While the component model is not specific to a particular programming language, the runtime environment currently supports only Java.

The SOFA 2 component model defines a component as a unit of composition with a set of external interfaces of two kinds–*provided interfaces* specify the services that the component provides to its clients and *required interfaces* specify the services that the component requires from its environment (i.e., from other components in the system). All external interfaces of a component form its *frame*. The key feature of the component model is that it is hierarchical and therefore supports nesting of components – *primitive components* are implemented directly in a programming language and represent leafs of a hierarchy, while *composite components* are composed of nested sub-components. Components at the same level of nesting are connected via *bindings* among interfaces.

The SOFA 2 component framework also supports formal specification and verification of component behavior. Each component can be equipped with a behavior specification defined in the formalism of behavior protocols (details in Section 2.2)– then it is possible to check whether (i) the Java implementation of each primitive component satisfies (obeys) its behavior specification and (ii) all sub-components of a composite component communicate without errors and comply with the parent's behavior specification.

2.2 Behavior Protocols

The formalism of behavior protocols [3] is a specific process algebra that we use for modeling and specification of behavior of SOFA 2 components. A behavior protocol *prot* is an expression that specifies a set of finite traces of method call-related events on component's provided and required interfaces. Four kinds of atomic events are supported by the formalism:

- `?interface.method↑` (acceptance of a method invocation on an interface),
- `!interface.method↑` (emit of a method invocation),
- `?interface.method↓` (acceptance of a return from a method), and
- `!interface.method↓` (emit of a return from a method).

More complex behavior protocols can be constructed from the atomic events using the operators for sequence (`;`), choice (`+`), finite repetition (`*`), and parallel composition (`|`). The parallel composition operator generates all interleavings of event traces defined by its operands such that no synchronization is assumed. An empty protocol is denoted by the expression `NULL`. Several syntactical shortcuts that enhance readability of protocols can also be used (they are supported by tools):

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

Here, the protocol *P* models a method body (it can be empty).

Given a component *C*, various behavior protocols modeling different aspects of *C*'s behavior can be defined in general – however, a behavior protocol with special meaning in the context of SOFA 2 is the *frame protocol*, which specifies the valid sequences of method calls on *C*'s provided interfaces and valid reactions of *C* in terms of method calls on required interfaces. This way, a frame protocol FP_C of the component *C* represents *C*'s behavior specification.

The key benefit of the formalism of behavior protocols is the built-in support for checking whether components equipped with frame protocols can communicate without errors. This can be done using the consent operator (∇) [18] for parallel composition of frame protocols – semantics of the consent operator is similar to the CCS parallel composition operator, i.e. it forces complementary events (e.g., `?i.m`↑ and `!i.m`↓) to synchronize and form an internal action (e.g., `τi.m`↑); however, it also identifies specific communication errors (deadlock and no response to a method call). We say that two protocols FP_1 and FP_2 are *compliant* if their composition via consent ($FP_1 \nabla FP_2$) yields no communication errors – components equipped with compliant frame protocols behave in a compatible way. The consent operator is implemented in the BPChecker [19] tool, which was developed in our group; technically, search for communication errors is done via exhaustive state space traversal.

2.3 Java PathFinder

Java PathFinder (JPF) [15] is a highly extensible and configurable explicit-state model checker for Java bytecode programs. It accepts a runnable Java bytecode program (with `main()`) and a set of properties as an input, and checks whether the program satisfies all the properties. By default, JPF checks low-level properties like freedom from deadlocks and assertion violations, but it can be extended via its API to check also high-level properties like compliance of Java code with a behavior specification.

A key feature of JPF is that it provides a powerful API, which allows (i) to extend it in various ways (e.g., with domain-specific properties), and (ii) to integrate it easily into development and verification frameworks. An important part of JPF's API is the `Verify` class that provides methods for non-deterministic data choice. It is supposed to be used in model checker-aware test drivers (and artificial environments) to systematically check the behavior of a fragment of Java code (a class) on all inputs from a given set – for example, a call of `Verify.getInt(0,3)` means that the code following the call is checked for each integer value in the range 0..3.

3 Running Example

The techniques presented in this paper are illustrated on a part of a software system in SOFA 2 [20] that was developed in our research group as a solution to the CoCoME assignment [21] (a fragment of its UML component diagram is in Figure 1) – further referred to as "CoCoME system".

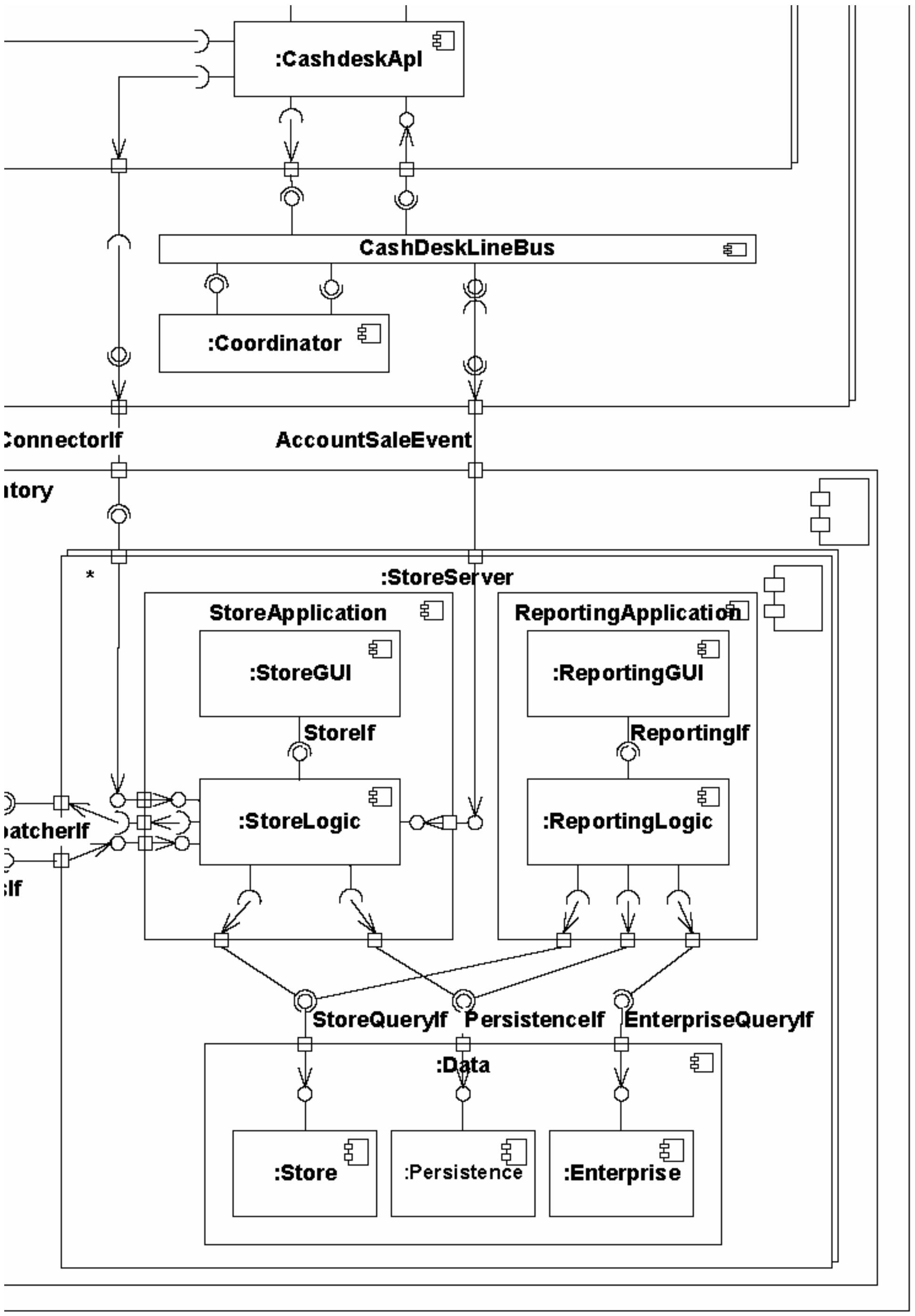

Figure 1: Architecture of the CoCoME system

We focus especially on the `Store` component, which provides the `StoreQueryIf` interface (a fragment of the corresponding Java interface is depicted on Figure 2) and has no required interfaces.

```
public interface StoreQueryIf
{
  StockItem queryStockItem(long storeId,
        long productbarcode, PersistenceContext pctx);
  Product queryProductById(long productId,
        PersistenceContext pctx);
  ...
}
```

Figure 2: The `StoreQueryIf` interface in Java

The frame protocol of `Store`, FP_{Store}, is depicted in Figure 3. It states that methods of the component can be called in all possible sequences and at most in four parallel threads, and that each method can be called a finite number of times.

```
(
  ?StoreQueryIf.queryProductById +
  ?StoreQueryIf.queryStockItem +
  # calls of other methods on StoreQueryIf
)*
|
# the fragment above is repeated here three more times
```

Figure 3: Frame protocol of the `Store` component

4 Construction of Artificial Environment for Verification of SOFA 2 Components in Java with Java PathFinder

Our approach to AGR verification of a SOFA 2 component *C* implemented in Java with JPF (*AGR$_{JPF}$ verification*) is an instance of the general three-step process described in Section 1.

The process of AGR$_{JPF}$ verification of a component *C* consists of the following three steps (illustrated also on Figure 4):

1. Construction of an environment assumption for *C*, which consists of two elements. The first element is a behavior protocol (*assumption protocol*) that specifies the desired sequences and parallel interleavings of method calls on *C*. It is created alternatively (a) from frame protocol of *C* (standalone approach) or (b) from frame protocols of components forming the actual environment (context approach). The second element is a Java class that contains a specification of data values, i.e. the possible values of parameters for methods of *C*s provided interfaces and possible return values from methods of *C*s required interfaces.
2. Automated generation of the artificial environment for *C* from the environment assumption – a runnable Java program that can be model-checked with JPF is then available.
3. Model checking of the runnable Java program with JPF to verify whether Java implementation of *C* satisfies (obeys) the *C*s frame protocol and contains no low-level concurrency errors like deadlocks and race conditions.

If model checking with JPF succeeds, it is guaranteed that *C* (Java code) obeys its frame protocol and does not trigger any low-level concurrency errors when used in a system that satisfies the environment assumption. The actual technique of the checking is based on combining JPF with BPChecker (described in detail in [22]).

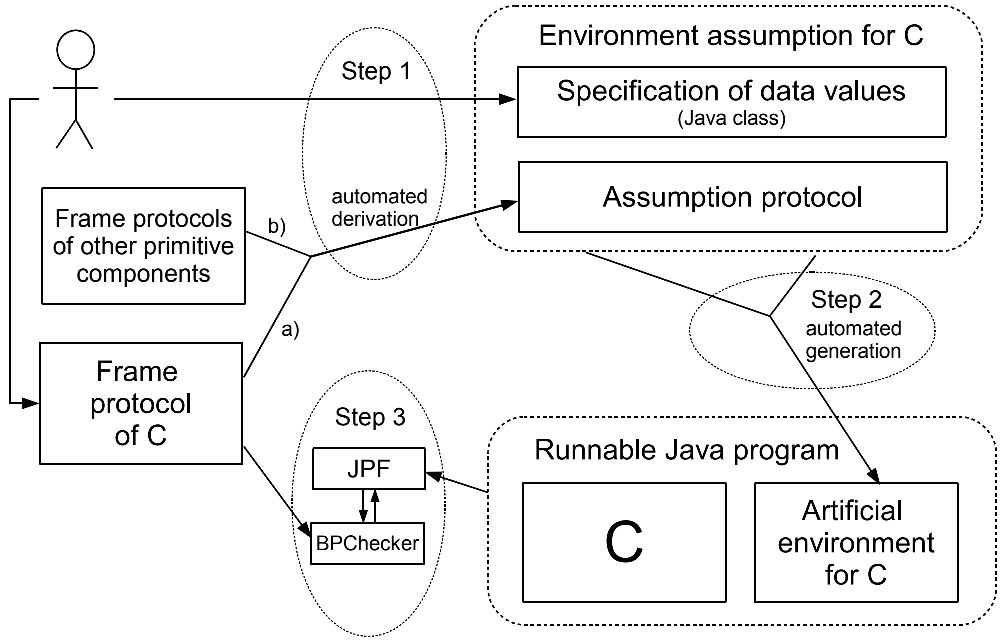

Figure 4: The process of $\mathrm{AGR_{JPF}}$ verification of the component C: a) standalone approach; b) context approach

When *C* is to be used in a particular software system, it is also necessary to check whether the rest of the software system satisfies the environment assumption (as mentioned in Section 1). In our case, this can be done via checking behavior compliance between the assumption protocol and composition of the frame protocols of other components at the same level of nesting.

The rest of this section provides an overview of our approach to construction of an assumption protocol (Section 4.1), specification of data values (Section 4.2), and automated generation of an artificial environment (Section 4.3) – more details can be found in [23] and [24].

4.1 Construction of Assumption Protocol

Formal definition of the concept of assumption protocol is based on the idea that an environment for a component *C* can be considered as another component E_C that is bound to *C* – an assumption protocol AP_C of *C* is a frame protocol FP_E of E_C. Nevertheless, AP_C has to be compliant with *C*'s frame protocol FP_C, since an artificial environment for *C* cannot exercise *C* in a way that violates its frame protocol.

The key characteristics that have to be considered in constructing an assumption protocol are: coverage of component's functionality, and, in order to fight state explosion, time and space complexity of construction process and the complexity of checking the resulting runnable Java program with JPF. Practically, the construction process has to be based on a trade-off between the coverage and complexity. Therefore, we have designed three specific variants of assumption protocol with the aim to reflect typical scenarios of usage of components in real-world software systems – *inverted frame protocol* (AP^{inv}), *context protocol* (AP^{ctx}), and *calling & trigger protocol* (AP^{trig}). AP^{inv} can be algorithmically derived from the frame protocol of *C* (standalone approach), and the other two can be constructed from the frame protocols of other components forming actual environments (context approach).

An inverted frame protocol $AP^{inv}{}_{C}$ of a component *C* models the use of *C* in all ways allowed by FP_C, i.e. it has the maximal possible coverage of *C*s behavior. An artificial environment $E^{inv}{}_{C}$ modeled by $AP^{inv}{}_{C}$ performs all the sequences and parallel interleavings of method calls on *C* that are allowed by FP_C and accepts method calls from *C* at the moments specified in FP_C. The protocol $AP^{inv}{}_{C}$ is constructed directly from FP_C via syntactical transformations (replacement of `!` with `?` and vice versa), and thus time and space complexity of its construction is linear with its length. On the other hand, model checking of *C* in $E^{inv}{}_{C}$ with JPF may be prone to state explosion, since it calls methods of *C* in all ways allowed by FP_C.

A context protocol $AP^{ctx}{}_{C}$ models (covers) the actual use of *C* in a particular software system – it is determined by the composition of the frame protocols of all other primitive components in the software system. In general, the software system may use only a subset of *C*s functionality, and therefore the behavior specified by $AP^{ctx}{}_{C}$ is a subset of the behavior specified by $AP^{inv}{}_{C}$. Consequently, checking of *C* with JPF may be less prone to state explosion if the artificial environment is determined by $AP^{ctx}{}_{C}$ instead of $AP^{inv}{}_{C}$. On the other hand, $AP^{ctx}{}_{C}$ is computed via exhaustive traversal of a composed state space of the frame protocols of all other primitive components in the software system, and therefore the process of construction of $AP^{ctx}{}_{C}$ has a high time and space complexity, being thus prone to state explosion.

A calling & trigger protocol $AP^{trig}{}_{C}$ is an optimization of $AP^{ctx}{}_{C}$. Again, it models (covers) the actual use of *C* in a particular software system, but it is constructed via syntactical transformations of the frame protocols of all components in the software system (including *C*). The key idea of the construction algorithm is inlining of method bodies specifications – for each binding between a required interface *R* of a component C_1 and a provided interface *P* of a component C_2 in the system's architecture, all the method calls on *R* are replaced by corresponding method bodies specifications that are defined in the frame protocol of C_2. Nevertheless, $AP^{trig}{}_{C}$ assumes that frame protocols of all components in a software system comply with specific syntactical patterns that are related to interleaving of the events corresponding to callbacks and autonomous activities (performed by inner threads of *C*) with other specified events. In principle, $AP^{ctx}{}_{C}$ captures interleaving of all events accurately, while $AP^{trig}{}_{C}$ does it precisely only for the events on *C*s provided interfaces and for triggers of callbacks, over-approximating interleavings of other events on *C*s required interfaces. Consequently, the corresponding artificial environment $E^{trig}{}_{C}$ is simpler than $E^{ctx}{}_{C}$, and therefore AGR_{JPF} verification of *C* is less prone to state explosion. Moreover, the process of construction of $AP^{trig}{}_{C}$ (via syntactical transformations of frame protocols) has lower time and space complexity than construction of $AP^{ctx}{}_{C}$, which involves exhaustive traversal of the composed state space of frame protocols of all other primitive components. A more detailed explanation of $AP^{trig}{}_{C}$ construction is beyond the scope of this paper – a full-fledged description and a claim that assuming the patterns is viable are in [23].

All three assumption protocols for the `Store` component are depicted in Figure 5. $AP^{ctx}{}_{Store}$ captures a subset of behaviors specified by $AP^{inv}{}_{Store}$, since other components in the CoCoME system (e.g., `StoreApplication`) use only a subset of functionality of `Store`. Specifically, the `StoreApplication` component calls (via the `Data` component) the methods `queryProductId` and `queryStockItem` in three parallel threads, and all other methods of the

`StoreQueryIf` interface in two threads or not at all in parallel. The protocols AP^{trig}_{Store} and AP^{ctx}_{Store} are equal, since `Store` has no required interfaces and therefore no interleaving of events on required interfaces has to be captured in AP^{ctx}_{Store}.

```
APinv Store = (
   !StoreQueryIf.queryProductById +
   !StoreQueryIf.queryStockItem +
   # calls of other methods on StoreQueryIf
 )*
 |
 # the fragment above repeated three more times

APctx Store = APtrig Store =
 ( !StoreQueryIf.queryStockItem* ; ... )*
 |
 (
   !StoreQueryIf.queryProductById*
   +
   !StoreQueryIf.queryStockItem*
 )*
 |
 !StoreQueryIf.queryProductById*
 |
 (
   ... ;
   (
     ( !StoreQueryIf.queryProductById*; ... )
     +
     ( ... ; !StoreQueryIf.queryStockItem* )
     +
     ...
   )
 )*
```

Figure 5: Assumption protocols of the `Store` component

4.2 Specification of Data Values

The specification of data values has to be provided by the user in the form of a Java class that works as a container for the values. Given a component *C* subject to AGR verification, the user has to specify (i) a set of possible values for each method parameter in a provided interface of *C* and (ii) a set of possible return values for each method of a required interface of *C*. The key requirement on these sets is that they should cover all paths in the control-flow graph (CFG) of Java code of each method of *C*–in other words, for each path *p* in the CFG of any method *m* of *C*, there should be at least one combination of values defined in these sets that triggers *p* when *m* is called by the artificial environment.

For illustration, a fragment of the specification of data values for `Store` may take the form depicted in Figure 6. The fragment states that (i) the set of possible values for the first parameter (of type `long`) of the `queryProductById` method of the

StoreQueryIf interface is {1, 2}, and (ii) the value null should be used for the second parameter (of the type cocome.PersistenceContext) in all calls of the method. By calling the constructor of StoreDataValues, the specified values are stored in an internal data structure.

```
class StoreDataValues {
 ...
 public StoreDataValues() {
  putLongSet("Store", "StoreQueryIf",
         "queryProductById", 1, new long[]{1,2});
  putObjectSet("cocome.PersistenceContext", "Store",
         "StoreQueryIf", "queryProductById", 2,
         new cocome.PersistenceContext[]{null});
 }
 ...
}
```

Figure 6: Specification of data values for the Store component

4.3 Generation of Artificial Environment

The artificial environment E_C for a component C is generated by a tool – Environment Generator for Java PathFinder (EnvGen) [24]–that we have developed. The input of EnvGen includes a particular assumption protocol AP_C, specification of data values, and the signatures of C's provided and required interfaces. The output of EnvGen is the artificial environment in the form of a set of Java classes that contains (i) a "driver" class with the main method, which calls the methods on C's provided interfaces in line with AP_C and with parameters taken from the specification of data values, and (ii) stub implementations of the required interfaces of C (which accept the calls issued by C). The driver class employs the JPF's API for non-deterministic data choice (the Verify class) in order to ensure that each method of a provided interface of C is called with all combinations of method parameter values that can be derived from the specification of data values–specifically, a non-deterministic choice from the set of possible values is made for each parameter of each called method. Moreover, the driver class interacts with the stubs in order to ensure proper sequencing and interleaving of method call-related events (invocations and returns) triggered by E_C with events triggered by C– for example, a callback triggered by C (via a call on a required interface) has to be performed by E_C at the moment(s) specified in AP_C.

A fragment of the artificial environment E_{Store} for the Store component is in Figure 7. The behavior of E_{Store} is specified by the inverted frame protocol AP^{inv}_{Store} of Store (Figure 5)–therefore, the environment calls methods of the StoreQueryIf interface in four parallel threads and, in each thread, the method to be called is selected non-deterministically (Verify.getInt(8)) in each iteration of the loop. Calls of the get method on the instance of StoreDataValues retrieve the specified method parameter values from an internal data structure.

```
public class StoreEnv {
  public static void main(String[] args) {
    StoreDataValues dataValues = new StoreDataValues();
    StoreQueryIf store = new StoreImpl();

    EnvThread th1 = new EnvThread(store, dataValues);
    // three more threads are created in the same way

    th1.start(); th2.start(); th3.start(); th4.start();
    th1.join(); th2.join(); th3.join(); th4.join();
  }
}

class EnvThread extends Thread {
  public void run() {
    while (true) {
      switch (Verify.getInt(8)) {
        case 0: store.queryProductbyId(
          dataValues.get(..., "queryProductById", 1),
          dataValues.get(..., "queryProductById", 2));
        case 1: store.queryStockItem(...);
        ...
      }
    }
  }
}
```

Figure 7: Artificial environment for the `Store` component

5 Evaluation

We have implemented the process of AGR$_{JPF}$ verification of a SOFA 2 component in the COMBAT toolset [25]. The toolset is built on top of JPF and includes standalone tools that implement the individual techniques described in Section 4. The flow of control and data among parts of COMBAT and JPF is shown on Figure 8. Technically, COMBAT can be used in two modes−either as a standalone tool, or via the Cushion tool for development of SOFA 2 applications, which is a part of the SOFA 2 framework [14].

In order to evaluate applicability of our approach to AGR verification of real-world software systems built of SOFA 2 components, we have applied the COMBAT toolset on the CoCoME system, which consists of 18 primitive components, each featuring tens to hundreds of lines of Java code, and 9 composite components.

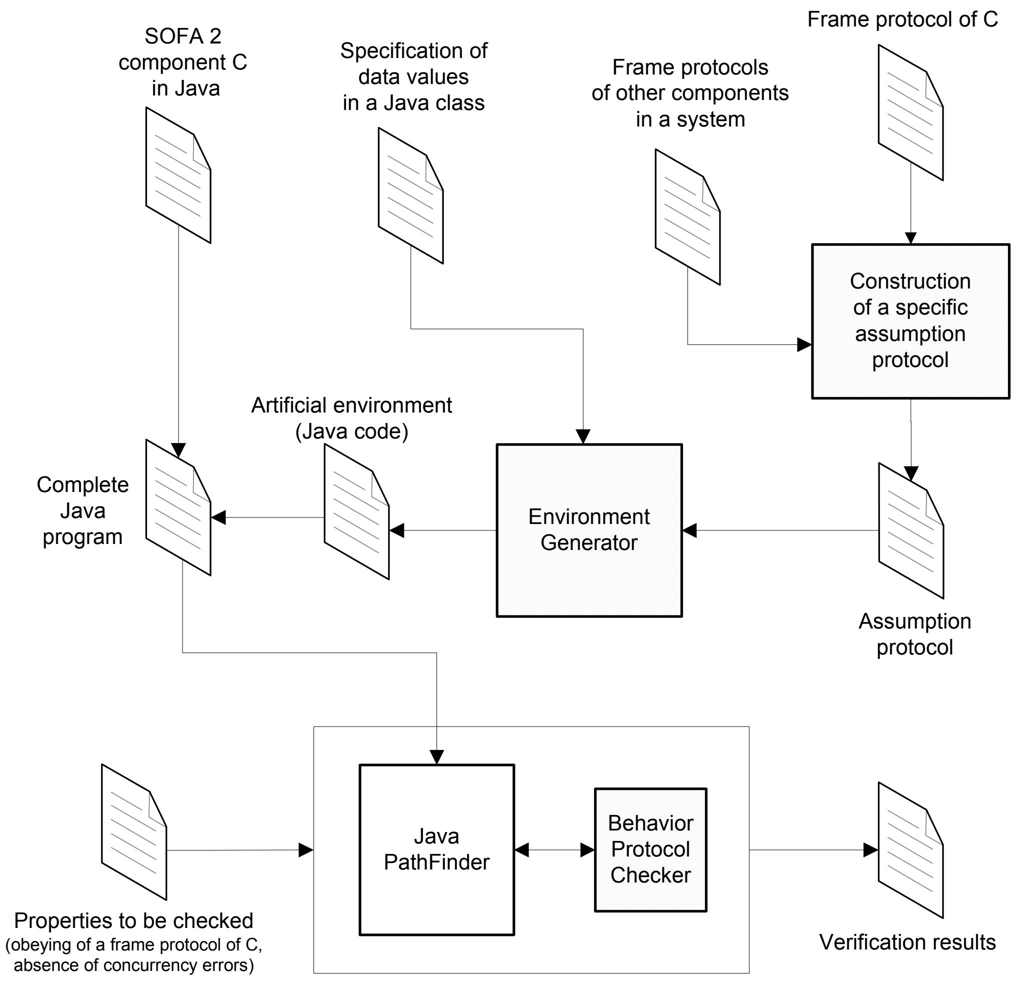
SOFA 2 component C in Java
Specification of data values in a Java class
Frame protocols of other components in a system
Frame protocol of C
Construction of a specific assumption protocol
Artificial environment (Java code)
Complete Java program
Environment Generator
Assumption protocol
Java PathFinder
Behavior Protocol Checker
Properties to be checked (obeying of a frame protocol of C, absence of concurrency errors)
Verification results

Figure 8: The COMBAT toolset

For illustration, we present results of experiments for two components of the CoCoME system – for `Store` (250 lines of code in Java) in Table 1 and for the `CashDeskApplication` component (500 loc in Java) in Table 2. The frame protocol of `CashDeskApplication` is available at [26]. Unlike `Store`, the `CashDeskApplication` component has several required interfaces so that AP^{ctx} and AP^{trig} differ. Presenting here the results for these two components was motivated by the fact that `Store` is of a typical complexity among all of the 18 primitive components and `CashDeskApplication` is the most complex one.

In each experiment, we measured the following characteristics: the time needed to compute a particular assumption protocol, the memory needed to compute an assumption protocol, the number of Java code lines of the generated artificial environment, the time needed for checking with JPF, the memory needed for checking with JPF, and the number of unique states traversed by JPF (equal to the number of non-deterministic choices). The entry '> 2048 MB' means a run-out of available memory (2 GB). Intentionally, the Java code of both components did not violate the checked properties in order to enforce traversal of the whole state space of the runnable Java program during AGR_{JPF} verification (unless a run-out of memory occurred). The results of experiments show that the whole process of AGR_{JPF} verification based on AP^{trig} has the lowest time and memory requirements.

As to comparison of the assumption protocols AP^{inv}, AP^{ctx}, and AP^{trig}, while all of them can differ in general, our experience shows that typically either $AP^{\text{inv}} = AP^{\text{ctx}}$, or $AP^{\text{ctx}} = AP^{\text{trig}}$ (as in case of `Store`), or even all of them equal. Specifically, $AP^{\text{inv}} = AP^{\text{ctx}}$ if the component is used in all the ways allowed by its frame protocol, while $AP^{\text{ctx}} = AP^{\text{trig}}$ if the component has no required interfaces. Nevertheless, AGR_{JPF} verification is the most efficient in case of E^{trig}, since (i) the process of construction of AP^{trig} has low time and space complexity, and (ii) the state space of the runnable Java program composed of the component and its artificial environment is the smallest in this case.

	AP^{inv}	AP^{ctx}	AP^{trig}
Time to compute AP	0 s	2.5 s	0.2 s
Memory to compute AP	103 MB	386 MB	107 MB
Java LOC of artificial env.	213	173	166
Time needed by JPF	n/a	3987 s	3987 s
Memory needed by JPF	> 2048 MB	693 MB	693 MB
Unique JPF states	n/a	5980056	5980056

Table 1: Results of experiments for the `Store` component

Overall, the results of verification experiments with the CoCoME system show that AGR_{JPF} verification with COMBAT is really feasible for non-trivial software systems built of SOFA 2 components. From the point of view of a user, a benefit is the complete integration of the verification process into the SOFA 2 framework and, in particular, into its development tools. Even though verification with COMBAT may still be prone to state explosion in case of a highly complex component or an environment triggering highly parallel activities, our experience with verification of non-trivial components in the CoCoME system shows that the method is practically applicable.

	AP^{inv}	AP^{ctx}	AP^{trig}
Time to compute AP	0 s	n/a	0.5 s
Memory to compute AP	126 MB	> 2048 MB	109 MB
Java LOC of artificial env.	2225	n/a	335
Time needed by JPF	n/a	n/a	320 s
Memory needed by JPF	> 2048 MB	n/a	336 MB
Unique JPF states	n/a	n/a	233271

Table 2: Results of experiments for the `CashDeskApplication` component

6 Related Work

Although software verification via model checking is a very active research area, the only other approach to AGR verification of code of components we are aware of is the one proposed in [12]. Similar to our solution, it uses JPF for checking whether a Java component obeys its behavior specification defined via LTS. The key differences are the following: (i) environment assumptions in LTS are constructed automatically via iterative learning and (ii) artificial environment is constructed by hand from the assumptions. Nevertheless, tools like Bandera Environment Generator (BEG) [8], which is a part of the Bandera toolset [27], can be used to generate the environment in an automated way.

There are, also, several approaches and tools aiming at generating artificial environment for components defined as collections of Java classes–most notably BEG and the method proposed in [28]. BEG is a tool for automated generation of an artificial environment for a collection of Java classes. As an input, it accepts an environment assumption in the form of regular expressions over the alphabet of method names with parameter values (allowing parallel activities at the highest syntactical level only). BEG supports extraction of the environment assumption from the implementation of an actual environment via static analysis – if no actual environment is available, then the environment assumption has to be provided by the user. In [28], the authors propose a technique for modular verification using JPF. The key idea is that, given a component *C* (collection of Java classes) subject to checking, stubs for all other components connected to *C* are generated in an automated way from assumptions specified as context-free grammars. The grammars determine valid interaction among *C* and the other components in terms of method call sequences – unlike behavior protocols and LTS, the power of context-free grammars allows specifying nested method calls without over-approximation. The generated stubs also check whether *C* interacts with the other components correctly. The key drawback of this method is that the grammars have to be provided by the user – no automated construction of them is supported by this technique.

There exist also approaches to compositional model checking of distributed Java applications with JPF, in which individual components (collections of Java classes) communicate over a network [29][30]. In this case, the environment of a component is represented by other components (not running in the scope of JPF) and the network infrastructure. All of these approaches are based on stubs implementing the Java network API and capturing the state of the network during verification – in particular, the stubs avoid repeating of network operations which would be triggered by backtracking.

A lot of research has also been done in automated construction of environment assumptions in finite state machine-based formalisms. For example, the method proposed in [9] aims at construction of the weakest environment assumption for a component modeled in LTS. The assumption is constructed in two steps: (1) the component is model-checked with an unrestricted environment modeled also in LTS, and (2) the model of environment in LTS is modified such that no error states are reachable in the component when it is model-checked with the resulting assumption.

Recently, several approaches to automated construction of environment assumptions via iterative learning (refinement) were proposed [31][32]. The basic idea of all these approaches is to iteratively refine an initial assumption on the basis of counterexamples reported by the model checker serving as a teacher – the goal is to derive an environment assumption guaranteeing that the component subject to checking satisfies the required properties. The initial assumption may be empty, or it may characterize the behavior of an actual environment. At each step of the iteration, the whole process of AGR verification is performed with the current assumption, and if some of the checks are not successful, then the assumption is refined with respect to the counterexample. The iteration terminates when the model checker (teacher) reports no error or when it is found that the component does not satisfy the required properties in the specific environment. The actual approaches based on iterative refinement of assumptions differ in the model checking technique used and in the anticipated styles of communication between the component and its environment. For example, the approach proposed in [31] aims at communication via method calls and uses the LTSA model checker [33], while [32] focuses on communication via shared variables (accesses to shared memory) and uses symbolic model checking.

The main advantage of our approach to automated construction of environment assumptions over those mentioned above is that it uses an algorithm which does not involve use of a model checker for constructing an environment assumption. Specifically, AP^{inv} and AP^{trig} are computed by an efficient syntactical algorithm, while construction of an environment assumption via iterative learning involves several calls to a model checker and therefore is more prone to state explosion.

In our view, the efficiency of our AGR_{JPF} verification is gained by (i) employing syntactical algorithm for constructing environment assumption (which works for behavior specification formalisms with syntax based on process algebra expressions) and (ii) making dependency requirements in code explicit (required interfaces). The latter is the feature of software component models like SOFA 2 and Fractal [34]. In comparison, the approaches of [12] and [31] aim at collections of Java classes (without explicit requirements specification) and use LTS for defining behavior specifications and environment assumptions (so that syntactical manipulation cannot be applied).

7 Conclusion

In this paper, we presented our approach to AGR_{JPF} verification of SOFA 2 components−we focus on Java components with provided and required interfaces and we use the formalism of behavior protocols for definition of component behavior specifications and environment assumptions. The process of AGR_{JPF} verification of a SOFA 2 component consists of three steps – construction of an environment assumption (assumption protocol and specification of data values), automated generation of an artificial environment for the component from the environment assumption, and JPF model checking of the runnable Java program composed of the component and its artificial environment.

We have implemented the approach in the COMBAT toolset and successfully applied it to the CoCoME system, which is a non-trivial software system in SOFA 2. Results of experiments with selected components from the CoCoME system show that our approach is feasible in practice.

The main advantage of our approach over the other ones with similar goals is that it uses syntactical algorithm for constructing an environment assumption, which is more efficient than constructing it via iterative learning, inherently involving calls to a model checker. A limitation of our approach is that it requires (i) a behavior specification in the form of a process algebra-like expressions and (ii) components with explicit provided and required interfaces. Nevertheless, it can be easily ported to similar component models like Fractal and OSGi [35], if a finite-state process algebra-like behavior specification is provided.

In future, we plan to develop an automated technique for extracting specification of data values (method parameters and return values) from the code of an actual environment, or from JUnit tests. Another option, which we may also pursue in the long-term, is to use symbolic execution in JPF [36] for calculating the sets of data values.

Acknowledgements. This work was partially supported by the Czech Academy of Sciences project 1ET400300504 and by the European Union under the ICT priority of the 7th Research Framework Program contract FP7-215013.

References

1. Szyperski, C.: 'Component Software: Beyond Object-Oriented Programming', 2nd edition, Addison-Wesley, 2002.
2. Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E.: 'Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2', In Formal Methods for Components and Objects (FMCO), LNCS, vol. 4111, Springer-Verlag, 2006, pp. 342-363.
3. Plasil, F., and Visnovsky, S.: 'Behavior Protocols for Software Components', IEEE Transactions on Software Engineering, 28(11), 2002, pp. 1056-1076.
4. Yellin, D. M., and Strom, R. E.: 'Protocol Specifications and Component Adaptors', ACM Transactions on Programming Languages and Systems (TOPLAS), 19(2), ACM Press, 1997, pp. 292-333.
5. Olender, K. M., and Osterweil, L. J.: 'Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation', IEEE Transactions on Software Engineering, 16(3), 1990, pp. 268-280.
6. Clarke, E., Grumberg, O., and Peled, D.: 'Model Checking', MIT Press, 2000.
7. Clarke, E., Long, D., and McMillan, K.: 'Compositional Model Checking', In Proceedings of 4th Annual Symposium on Logic in Computer Science, IEEE Press, 1989, pp 353-362.
8. Tkachuk, O., Dwyer, M. B., and Pasareanu, C. S.: 'Automated Environment Generation for Software Model Checking', In Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), IEEE CS, 2003, pp. 116-129.
9. Giannakopoulou, D., Pasareanu, C. S., and Barringer, H.: 'Assumption Generation for Software Component Verification', In Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), IEEE CS, 2002, pp. 3-12.
10. de Alfaro, L., and Henzinger, T. A.: 'Interface Automata', ACM SIGSOFT Software Engineering Notes, 26(5), ACM Press, 2001, pp. 109-120.
11. Pnueli, A.: 'In Transition from Global to Modular Temporal Reasoning about Programs', In Logics and Models of Concurrent Systems, vol. 13, Springer-Verlag, 1984, pp. 123-144.
12. Giannakopoulou, D., Pasareanu, C. S., and Cobleigh, J. M.: 'Assume-guarantee Verification of Source Code with Design-Level Assumptions', In Proceedings of the 26th International Conference on Software Engineering (ICSE), IEEE CS, 2004, pp. 211-220.
13. Tkachuk, O., and Rajan, S. P.: 'Application of Automated Environment Generation to Commercial Software', In Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2006), ACM Press, 2006, pp. 203-214.
14. SOFA 2 component framework, http://sofa.ow2.org, accessed Feb 2009.
15. Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F.: 'Model Checking Programs' Automated Software Engineering, 10(2), 2003, pp. 203-232.
16. Bures, T., Hnetynka, P., and Plasil, F.: 'SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model', In Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications (SERA 2006), IEEE CS, 2006, pp. 40-48.
17. Bures, T., Hnetynka, P., and Plasil, F.: 'Runtime Concepts of Hierarchical Software Components', International Journal of Computer and Information Science, 8(S), 2007, pp. 454-463.

18. Adamek, J., and Plasil, F.: 'Component Composition Errors and Update Atomicity: Static Analysis', Journal of Software Maintenance and Evolution: Research and Practice, 17(5), John Wiley & Sons, 2005, pp. 363-377.
19. Mach, M., Plasil, F., and Kofron, J.: 'Behavior Protocol Verification: Fighting State Explosion', International Journal of Computer and Information Science, 6(1), 2005, pp. 22-30.
20. Bures, T., Decky, M., Hnetynka, P., Kofron, J., Parizek, P., Plasil, F., Poch, T., Sery, O., and Tuma, P.: 'CoCoME in SOFA', In The Common Component Modeling Example: Comparing Software Component Models, LNCS, vol. 5153, Springer-Verlag, 2008, pp. 388-417.
21. CoCoME: Common Component Modeling Example, http://www.cocome.org, accessed Feb 2009.
22. Parizek, P., Plasil, F., and Kofron, J.: 'Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker', In Proceedings of 30th IEEE/NASA Software Engineering Workshop (SEW-30), IEEE CS, 2006, pp. 133-141.
23. Parizek, P., and Plasil, F.: 'Modeling of Component Environment in Presence of Callbacks and Autonomous Activities', In Proceedings of 46th International Conference on Object, Components, Models and Patterns (TOOLS EUROPE 2008), LNBIP, vol. 11, Springer-Verlag, 2008, pp. 2-21.
24. Parizek, P., and Plasil, F.: 'Specification and Generation of Environment for Model Checking of Software Components', In Proceedings of Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA 2006), ENTCS, 176(2), Elsevier, 2007, pp. 143-154.
25. COMBAT toolset, http://dsrg.mff.cuni.cz/projects/combat, accessed Feb 2009.
26. CoCoME in SOFA, http://dsrg.mff.cuni.cz/projects/cocome/sofa/sofa.tgz, accessed Aug 2009.
27. Corbett, J., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zhueng, H.: 'Bandera: Extracting Finite-state Models from Java Source Code', In Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), ACM Press, 2000, pp. 439-448.
28. Hughes, G., and Bultan, T.: 'Interface Grammars for Modular Software Model Checking', In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007), ACM, 2007.
29. Artho, C., Leungwattanakit, W., Hagiya, M., and Tanabe, Y.: 'Efficient Model Checking of Networked Applications', In Proceedings of TOOLS EUROPE 2008: 46th International Conference on Objects, Components, Models and Patterns, LNBIP, vol. 11, 2008.
30. Barlas, E., and Bultan, T.: 'NetStub: A Framework for Verification of Distributed Java Applications', In Proceedings of the 22nd International Conference on Automated Software Engineering (ASE 2007), ACM, 2007.
31. Cobleigh, J. M., Giannakopoulou, D., and Pasareanu, C. S.: 'Learning Assumptions for Compositional Verification', In Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), LNCS, vol. 2619, Springer-Verlag, 2003, pp. 331-346.
32. Nam, W., Madhusudan, P., and Alur, R.: 'Automatic Symbolic Compositional Verification by Learning Assumptions', Formal Methods in System Design, 32(3), 2008, pp. 207-234.

33. Magee, J., and Kramer, J.: ‘Concurrency – State Models and Java Programs’, John Wiley, 1999.
34. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., and Stefani, J. B.: ‘The FRACTAL Component Model and its Support in Java’, Software–Practice and Experience, 36(11-12), 2006, pp. 1257-1284.
35. OSGi Alliance: OSGi Service Platform Release 4, http://www.osgi.org/, accessed Aug 2009.
36. Anand, S., Pasareanu, C. S., and Visser, W.: ‘JPF-SE: A Symbolic Execution Extension to Java PathFinder’, In Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), LNCS, vol. 4424, 2007.