Specification and Generation of Environment for Model Checking of Software Components*

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

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

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

Abstract. Model checking of isolated software components is inherently not possible because a component does not form a complete program with an explicit starting point. To overcome this obstacle, it is typically necessary to create a model of the environment of the component which is the intended subject of model checking. Two solutions to generating of environment are compared in this technical report: the Bandera Environment Generator tool [2] and our approach based on behavior protocols [1] employed for model checking of components by the Java PathFinder tool [8].

1 Introduction

1.1 Background

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

Nevertheless, a component usually cannot be checked in isolation, because it does not form a complete program inherently needed to apply model checking. It

*The work was partially supported by the Grant Agency of the Czech Republic (project number 201/03/0911) and by the Czech Academy of Sciences (project 1ET400300504).

is, therefore, necessary to create a model of the environment of the component under the test, and then check the program that consists of the component and the environment.

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

1.2 Goals and Structure of the Text

The main goal of this technical report is to compare two approaches to specification and automatic generation of environment for model checking of software components. The first approach is the one in the Bandera Environment Generator tool [2] and the second approach is using behavior protocols [1] for specification of the environment, provided the behavior of a component is specified as well.

The remainder of the technical report is organized as follows. Sect. 2 provides a motivational example and Sect. 3 introduces the Bandera Environment Generator (BEG) [2]. Sect. 4 starts with an overview of behavior protocols [1] and then presents the key contribution - namely the description of our approach to specification of environment and model checking of software components. The rest of the text contains evaluation, related work and conclusion.

2 Motivation

In this section, we present a simple class `DatabaseImpl` and a handwritten environment for the class, assuming `DatabaseImpl` is the intended subject of model checking.

The class implements one interface and requires two internal references - it can be also looked upon as a component with formally defined one provided and two required interfaces.

Source code of the class in the Java language is:

```
public interface Database
{
    public void start();
    public void stop();
    public void insert(Object key, Object value);
    public Object get(Object key);
}

public class DatabaseImpl implements Database
{
    private Logger log;

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

```
    }

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

    public void insert(Object key, Object data)
    {
        ...
    }

    public Object get(Object key)
    {
        ...
    }
}
```

The environment for the class should create at least two threads of control to call methods of the class in parallel. The main reason is to allow a model checker to search for concurrency errors.

The environment could take the following form (we present only fragments of the source code):

```
public class EnvThread extends Thread
{
    Database db;

    ...

    public void run()
    {
        db.insert(getRandomString(), getRandomString());
        Object res = db.get(getRandomString());
        ...
    }
}

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

    db.start();

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

    db.stop();
}
```

In general, creating an environment by hand is hard and tedious work even in simple cases. An obvious solution to this problem is to automatically generate the environment from a higher-level abstraction than the code provides.

3 Environment Generator in Bandera

3.1 Bandera

Bandera [6] is a tool set designed for model checking of complete Java programs, i.e. featuring the `main` method. It is composed of several modules - model extractor, model translator, and model checker, to name some of them. The model extractor extracts a finite model from Java source code and the model translator translates the internal model of the Bandera tool into the input language of a target model checker. Originally, the Bandera tool set supported the Spin and Java PathFinder model checkers, but currently it employs Bandera specific model checker (Bogor [7]).

3.2 Bandera Environment Generator

The Bandera Environment Generator (BEG) [2], a part of the Bandera tool set, is a tool for automated generation of environment for the purpose of model checking of Java programs. It is based on the assume-guarantee style of reasoning, which means that the model of the environment is determined by assumptions provided by the user.

User of the BEG tool must first decompose the Java program into two parts - the tested *unit*, i.e. the classes to be tested, and its *environment*. The tool itself automatically discovers the interfaces (i.e. method calls in one way or the other) between the unit under analysis and the environment.

The model of the environment can be constructed

i) either from environment assumptions provided by the user, or

ii) as a result of code analysis of the environment classes if they are available.

The BEG tool supports two forms of environment assumptions specification -

(1) LTL, and

(2) regular expressions;

both of them use program actions (method calls, assignments, etc) - LTL as atomic propositions, and regular expressions as the underlying alphabet.

An environment specification for the component presented in Sect. 2, defined in the input language of the BEG tool, could take the following form:

```
environment
{
    instantiations
    {
        1 Logger log;
        Database db = new DatabaseImpl();
        db.setLogger(log);
    }
```

```
    regular assumptions
    {
        T0: (db.get() | db.insert())*
        T1: (db.get() | db.insert())*
    }
}
```

The specification of the environment, enclosed in a block marked by the `environment` keyword, contains the `instantiations` and `regular assumptions` sections.

The `instantiations` section allows the user to specify how many instances of a certain type should be allocated and to give names to some of those variables and objects for the purpose of referencing them. In our example, two named objects are instantiated - one instance of the `Logger` class and one instance of the `DatabaseImpl` class.

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

As you can see, calls to the `start` and `stop` methods of the `Database` interface that should occur before, resp. after, the threads are started, resp. stopped, are not specified in the model of the environment. The reason for this is that the input language of the BEG tool allows to define threads of control only at the outermost level, i.e. no method call can be specified to occur before or after the threads are executed.

The BEG tool also allows to specify values of parameters to methods that are called on tested classes. If the value of a parameter is not specified, as in the example above, then it is non-deterministically selected from all available values of a given type (e.g. from all allocated instances of a given class, etc). It is even possible to use a variable that is defined in the `instantiations` section as a parameter to a method call.

Since the most recent release of the Bandera model checker [6] is an alpha version only, not being fully stable yet, we have decided to use the Java PathFinder model checker (JPF) [8].

4 Environment Generator for Java PathFinder

Since the Java PathFinder does not have an environment generator, we have built our own environment generating tool for model checking of components. Our approach to model checking of code of software components stems from the assumption that components are during the design specified at the ADL level, which in particular includes specification of their provided and required interfaces

and also specification of their behavior. The latter is done via behavior protocols [1]. In this section we show how this behavior specification can be advantageously employed for generating a component environment necessary for model checking of the component.

4.1 Behavior protocols

A behavior protocol is an expression that describes the behavior of a software component in terms of a set of atomic events. The term *behavior of a component* denotes the activity on interfaces of a component, i.e. accepted and emitted method calls on those interfaces.

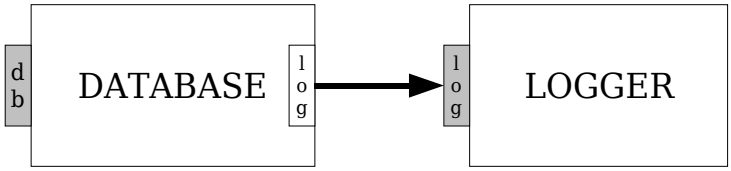

Figure 1: The DATABASE and LOGGER components, defined in Sect. 2

An example protocol for the component from Fig. 1 follows:

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

Each event has the following syntax: `prefix interface.method suffix`, where the suffix is optional - events that have no suffix are syntactical shortcuts. The prefix `?` denotes an accept event and the prefix `!` denotes an emit event. The suffix `^` character stands for a method call and the suffix `$` stands for a response. An event of the form `!i.m` is a shortcut for `!i.m^;?i.m$`, an event of the form `?i.m` is a shortcut for `?i.m^;!i.m$` and a protocol of the form `?i.m{prot}` is a shortcut for `?i.m^;prot;!i.m$`. Events that have a suffix are considered to be atomic events.

The example above presents also several operators. The `;` character stands for a sequence operator, the `*` character stands for a repetition operator and the `||` word represents an or-parallel operator. Behavior protocols support also an alternative operator `+` and an and-parallel operator `|`. The or-parallel operator is only a shortcut - `a || b` stands for `a + b + (a | b)`.

Semantics of a behavior protocol is defined in terms of Labelled Transition System (LTS), where the transitions are labeled by events. Each protocol can define an infinite set of traces, each of them being finite.

Each component has two protocols associated with it - a frame protocol and an architecture protocol. The frame protocol of a component describes only its external behavior, what means that it can contain only events on external interfaces of a component. On the other hand, the architecture protocol describes the behavior of a component in terms of composition of its subcomponents.

4.1.1 Protocol Checker

For the purpose of checking compliance of a frame protocol with an architecture protocol, and also for checking of compositional compliance for subcomponents, we use the protocol checker [10] that we have developed in our research group.

4.2 Modeling the Environment with Inverted Frame Protocol

The environment of a component is modeled by the inverted frame protocol of the component, which is derived from the frame protocol by replacing all accept events with emit events and vice versa. The inverted frame protocol constructed in this way forces the environment

(1) to call a certain method of a certain provided interface of the component right at the moment the component expects it,

(2) and to expect a certain method call on a certain required interface when the component should call the method.

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

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

In order to minimize the increase of state space size, our environment generating tool simplifies the inverted frame protocol of the target component before it generates the environment. More specifically, it translates the alternative operator between some subprotocols to a sequence of those subprotocols and the repetition operator to two copies of given subprotocol in a sequence. The and-parallel operator between three or more subprotocols is replaced with all distinct pairs of subprotocols, where each pair is connected with the and-parallel operator - this optimization reduces the number of interleavings of instructions that the Java PathFinder model checker must process during state space traversal.

4.3 Integration of the Java PathFinder and the Protocol Checker

The integration of the Java PathFinder tool with our protocol checker [10] makes it possible to verify that a primitive component obeys its frame protocol.

Communication between the JPF and the protocol checker during checking of the `Database` component is depicted on Fig. 2. The left part of the schema shows the JPF traversing the code (state space) of the component and the right part shows the state space of the protocol checker, which is determined by the frame protocol of the component. A plugin for the Java PathFinder tool, which we have developed,

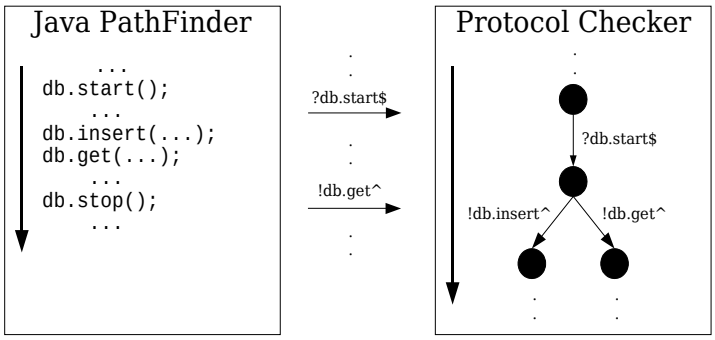

Figure 2: Java PathFinder and Protocol Checker

traces execution of invoke and return instructions that are related to methods of provided and required interfaces of the target component, and notifies the protocol checker of those instructions in the form of atomic events. The protocol checker verifies that the implementation of the target component obeys the frame protocol of the component. When the protocol checker encounters an unexpected event or a missing event, it tells the JPF to stop the search through the state space and to report an error to the user.

5 Evaluation

5.1 Comparison of Approaches to Modeling of the Environment

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

The main differences between them are

(1) in that the BEG tool allows to specify parallelism only at the outermost level of regular expressions that specify behavior of the environment,

(2) and in that behavior protocols have no support for method parameters.

We elaborate on these two differences below. There is also a difference in that the BEG tool targets plain Java classes with informally specified provided and required interfaces, while our approach is aimed at software components that have provided and required interfaces defined in an explicit way.

The input language of the BEG tool allows the user to specify several threads of control, which are executed in parallel, only at the outermost level, while a behavior protocol can contain parallel operators at any level. As an example, a description of an environment in the language of the BEG tool, when translated to a behavior protocol, looks like this:

```
(!i1.m1;..;!i1.mN) | (!i2.m1;..;!i2.mN)
```

On the other hand, the input language of the BEG tool allows the user to specify values of method parameters, while the behavior protocols have no support for parameters of methods. Therefore, our tool generates the environment in such a way that the values of all method parameters are non-deterministically selected during model checking.

Another advantage of support for, possibly partial, specification of parameter values is that it enables the environment generator to select a proper version of an overloaded method - or to generate a code that will non-deterministically invoke all versions of the method that conform to the partial specification.

5.2 Proof of Concept

We have created an implementation of the environment generator that uses the simplified inverted frame protocol of a component as a model of the environment's behavior. It aims at components that use the Fractal Component Model [3] and expects that the Fractal ADL is used to define components.

6 Related work

The Zing model checker [4] does not need to generate any environment because it is used to check models of complete programs, which are extracted from source code in the C and C# languages.

SLAM [9], a part of the SDV tool for verification of device drivers for the Windows operating system, is a model checker that creates boolean abstractions of the source code. The environment for device drivers is defined by interfaces that are provided by the Windows kernel. The SLAM tool models the environment via training [5]. Here, the basic principle is that, for a certain procedure P that is to be modeled, it first takes several drivers that use the procedure P, then it runs the SDV tool on those drivers and therefore gets several boolean abstractions of the procedure P, and finally merges all those abstractions and puts the resulting boolean abstraction of the kernel procedure P into a library for future reuse.

Our tool for environment generation is partially based on [11]. The tool that is described in the thesis creates an environment that is driven by an automaton derived from the inverted frame protocol of the target component. It is too aimed at components that use the Fractal Component Model [3]. We decided not to use

this tool mainly because it generates an environment that increases the state space size quite significantly.

7 Conclusion

Model checking of software components in isolation is usually not possible because model checkers accept only complete programs. Therefore, it is necessary to create an environment for each component that is a subject of model checking.

In this technical report, we have two approaches to generating environment for components, resp. classes - namely the Bandera Environment Generator (BEG) tool [2] in Sect. 3, and our approach that is based on behavior protocols [1] in Sect. 4. Main differences between the two approaches lie in the level of support for parallelism, in support for specification of parameter values, and in the fact that the BEG tool is aimed at plain Java classes while our approach targets software components with explicitly defined provided and required interfaces.

What remains to be done is automated derivation of sets of values that will be used as sources for non-deterministic choice of values of method parameters. It is motivated by the fact that non-deterministic selection from all possible values of a given type usually results in an unnecessarily big increase of state space size. The sets of values should be generated in such a way that will let the model checker to check all the code paths in a target method. A viable approach to the derivation of possible parameter values could be to use static analysis of source code or byte code.

References

[1] Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002

[2] Tkachuk O., Dwyer M.B., Pasareanu C.S.: Automated Environment Generation for Software Model Checking, In the Proceedings of ASE 2003, May 2003

[3] Fractal Component Model, http://fractal.objectweb.org/

[4] Zing Model Checker, http://research.microsoft.com/zing/

[5] Ball T., Levin V., Xie F.: Automatic Creation of Environment Models via Training, TACAS 2004

[6] Bandera tool set, http://bandera.projects.cis.ksu.edu

[7] Bogor model checker, http://bogor.projects.cis.ksu.edu

[8] Java PathFinder, http://javapathfinder.sourceforge.net

[9] SLAM Project, http://research.microsoft.com/slam/

[10] Mach, M., Plasil, F., Kofron, J.: Behavior Protocol Verification: Fighting State Explosion, IJCIS Vol.6, Number 1, ACIS, ISSN 1525-9293, pp. 22-30, Mar 2005

[11] Potrusil T.: Specifying Missing Component Environment in Bandera, Master Thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2005