Annales UMCS Informatica AI 2 (2004) 115-124

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

An algorithm and case study for the object oriented abstraction

Jakub Ratajczak*

Institute of Computer Science, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warszawa, Poland

Abstract

Model checking of software systems becomes more effective each day. However it still can not handle huge state spaces of real software. Particularly, concurrent systems are hard to verify. Abstraction techniques are one of the solutions aimed at managing the complexity problem. This paper describes the object oriented abstraction algorithm. It allows semi-automatic abstraction of real (i.e. Java) programs. A novelty method for constructing class abstractions is shown. It uses additional program annotations expressed in a formal manner. Proposed techniques are shown in a context of algorithms used in the Bandera toolset. A short case study of this approach is also shown.

1. Introduction

Model checking is one of the formal methods for software verification, which becomes more effective each day. This approach has proved its advantages in the hardware applications and now there is a lot of effort put into migrating this technology to the software domain. One of the goals to achieve is to verify programs written in the modern object oriented languages. Since the state space of such program is possibly infinite (or extremely large) it can not be examined straightforwardly by model checking algorithms. Additionally, the state explosion occurs when concurrent software is being verified. There are several ways to avoid the state space size and explosion problems. Abstraction algorithms appear to be the most promising approaches [1,2].

There are several methodologies and tools developed to aid the verification of software systems developed using ordinary programming languages like C or Java. Nevertheless, most of them can not handle real-size applications due to size and efficiency problems. The Bandera toolset [3,4] is one of more sophisticated model checking tools dealing with Java programs. It uses combination of slicing and abstraction [5] to reduce a state space of model checked programs. The slicing is a program reduction technique that allows for

* *E-mail address*: j.ratajczak@ii.pw.edu.pl

removing parts of a program that does not affect the property being examined. Unlike slicing, the abstraction does not remove anything. It reduces state space of the modeled system intensively by appropriate merging sets of states together in a way preserving desired properties of a system. Reduced programs have to be treated in a special way in order to prove their correctness but it is beyond the scope of this paper. Appropriate techniques are described in [5-7].

In the paper the abstraction algorithm for object oriented programs is described. It is an enhanced version to the one used in the Bandera toolset. The original one allows abstracting variables of a few types only – simple, built-in ones. The algorithm proposed in the paper allows abstraction of variables of any class or type, even API and user-defined classes. It opens model checking possibilities to a much wider range of applications.

Idea of the automatic abstraction presented in this paper covers Java programs, it uses PVS prover and it is interfaced with the Bandera toolset. However, it may be smoothly applied to other programming languages and formal verification tools. It does not require any particular model checker.

Section 2 describes briefly the program verification process. The way of preparing specifications of classes and methods is shown in Section 3. Section 4 explains a method for abstracting objects. Section 5 concludes abstraction and shows how to prepare an abstracted program. Section 6 presents a case study. The last sections present conclusions.

2. The program verification process

The process of the verification of program [3] consists of several tasks: i) preparing the property (set of properties) being verified Φ, ii) slicing the program in order to remove its irrelevant parts, iii) abstracting a program ($P \overset{\alpha}{\rightarrow} P'$) and an examined property ($\Phi \overset{\alpha}{\rightarrow} \Phi'$), iv) model checking of sliced and/or abstracted program, v) interpreting results. In this paper we focus on a program and property abstraction (iii). The original type abstraction algorithm used by the Bandera toolset allows to abstract Java programs by replacing program P variables of built-in types only with their abstractions. Here the enhanced version is proposed. It handles objects of any class.

The object abstraction algorithm produces an over-approximation of the original program *P'*: the abstraction process introduces extra activities to the program. In other words there are fewer states in the program but additional transitions appear. These transitions appear as a result of merging states (abstracting concrete variables) while all possible execution paths (transitions) are preserved. Therefore an abstracted state must be the starting point for all transitions that started in the concrete states that produced the abstracted state. Respectively the abstract state has to be an end point of all transitions that in the concrete domain ended in all states the new one is abstracted from [8].

Over-approximation preserves universal propositions. Informally, a universal proposition states that the given proposition holds on all paths of the program. An over-approximation is not conservative. It is only weak property-preserving:

$$P \models \Phi \Leftarrow P' \models \Phi' .$$

Unfortunately, because of extra transitions the over-approximated program and property do not preserve existential properties. Therefore, the over-approximation may lead to false alarms while universal propositions are examined. Some techniques of avoiding this problem are described in [6]. They apply as well to the proposed enhanced abstraction algorithm.

3. Formal specification of a class

Formal specification of a class is the description of its behavior expressed in a formal language. Specification is used later on in the reasoning about abstracted class methods. It is not necessary to write specification of a class if one does not want to preserve fully formal path leading from original to abstracted class. In such case abstraction may be done using other class description (i.e. documentation or just source code).

Formal specification of a class consists of three steps:

- translating relevant class members to a formal description,
- providing necessary class invariants concerning the above fields,
- providing specifications of methods of a class.

Specification of a class has to be done manually. However, when all methods are well specified it should be a relatively easy task. There are successful projects on proving correctness of such specifications basing on real Java code [9]. Any formalism can be used for the specification. Here the PVS [10] was chosen because of its powerful proving engine.

3.1. Translating class members

Translating class members is rewriting them (either abstracted or not) in a formal way. Class fields have to be translated to allow defining methods specifications. Each method is to be specified in terms of how it acts on these fields. Only fields that are vital for a class behavior (and for behavior of its abstraction) should be translated. There is no universal recipe how to decide which ones are relevant and which are not.

Types of fields of the class should be translated to types of a formal language in a way allowing for defining methods behavior later on. If a given type is not abstracted then the closest possible type of a formal language is chosen. When a field is an object its class formal definition should be used.

3.2. Class invariants

Class invariants are used as auxiliary lemmas for proving falsifications (see the next section) during the phase of deducing abstract methods. Their usage eliminates repeating conditions they cover in each method definition.

Class invariants should operate on fields translated in the previous step. Invariants do not have to hold inside method bodies. They should be valid always when no operation is performed on an object.

3.3. Methods definitions

Method definition defines how method acts depending on the object state and method actual arguments. It defines also if and when an exception is thrown by a method.

Method description should be as detailed as possible. The better description is the more accurate abstractions it will produce.

4. Class abstraction algorithm

The program abstraction algorithm uses i) a set of abstract domains, ii) a set of mappings of concrete variable values to abstract values, iii) a set of abstracted methods and operators operating on abstract domains. Abstract domains (i) depend on a property being checked, on types of semantics and on the use of variables. Mappings from concrete values to abstract values (ii) are derived both from concrete and abstract type semantics. A developer prepares them. Abstracted operators (iii) are prepared automatically from the semantics of equivalents of original operators present in the used theorem prover (e.g. PVS [10]) [5,11]. Bodies of abstracted methods (iii) are built using formal annotations of original methods – if they exist. Formal annotating of Java class is described in Section 3. If formal specification of a class was omitted abstract methods (iii) have to be prepared manually. Abstract methods used in the case study below were prepared manually.

Most elements from the sets above i), ii) and iii) are reusable and thus may improve the efficiency of the whole verification process. They may be prepared in advance as well. Such an approach seems to be extremely useful when applied for API classes. Nevertheless usefulness of the algorithm is not restricted only to API classes. Each ordinary class can be annotated and its abstraction (together with abstracted methods) can be introduced to the abstraction library and reused.

4.1. Class abstraction

To abstract a class we have to complete the following tasks:

– an abstract domain has to be specified,

– a function mapping from concrete domain to the abstract one has to be specified,
– all methods (used in the verified program) of the class have to be abstracted.

They are described in [3] apart from the third one which is here substantially modified.

Specifying an abstract domain

An abstract domain is a set of values that objects of abstracted class can take. An abstract domain has to be chosen carefully. During the abstraction process we lose information about original program. However, we have to preserve enough to decide about properties of a program. Choosing proper abstractions is the key issue.

Function mapping from concrete to abstract domain

Given the abstract domain, we have to specify how real objects are to be mapped to abstract ones. We need to prepare a mapping from a concrete to an abstract domain (surjection).

4.2. Abstracting methods

Applying a method *m* on an object *o* in the concrete domain changes its state from os_i (old value) to os_j (new value). Applying abstracted method *m'* (where $m' = \alpha(m)$) in the abstract domain should change state of an object os_i' to the other state os_j', which is the abstraction of os_j ($os_j' = \alpha_c(os_j)$):

$$\begin{array}{ccc} os_i & \xrightarrow{m(o)} & os_j \\ \downarrow{\scriptstyle \alpha(o)} & & \downarrow{\scriptstyle \alpha(o)} \\ os'_i & \xrightarrow{m'(o')} & os'_j \end{array}$$

Because abstraction function α_c is narrowing, abstract methods are indeterministic. In fact, each method that in abstract domain leads to all abstract values of an object (we may say is fully indeterministic) is a well-defined abstract method. It is a trivial case. The goal is to decrease its indeterminism in order to use the model checking on abstract programs practically.

Preparing abstract methods can be automated with a theorem prover as shown in [12].

While operators have only one or two arguments methods may have multiple arguments. Moreover, some of them can be of different type or class from the class the method belongs to. These arguments require abstraction as well. There are two ways to manage this problem:

1. To provide a method abstraction separately for each combination of abstract argument types. Unfortunately, it causes exponential growth of the library due to the increasing number of abstraction domains.
2. To annotate the abstracted method in such a way that its behavior for a particular argument abstraction can be determined just in time. Methods abstracted in this way are stored in the library.

5. Abstracting program

Abstracting a program is a clue of the abstraction process. After this step an abstracted program is generated and, hopefully, it is small enough to be the model checked.

Because the whole abstraction algorithm is compatible with the Bandera toolset the program abstraction steps are similar. The difference is that while originally only types and operators are replaced with their abstract equivalents now also classes and methods have to be replaced. For a full description see [5].

Abstracted program may be expressed in the original language (i.e. Java).

6. Case study

This case study is described in details in [13]. The algorithm proposed in the paper was tested on a real application. The application being verified is a Java ME (CLDC/MIDP) program for mobile devices. It has more than 2000 code lines with 1 to 3 tasks are running concurrently. It is a part of a language teaching system – Mnemonius [14].

During the final test phase on a real cellular phone (Nokia 6310i) there was a bug noticed. It caused that one of the texts showed on a screen appeared in a wrong place: at the top of a screen, while it should appear in the middle (see Fig. 1). It is a scrolled hint for a user. This malfunction was not notified in a development environment and it was noticed only once during all tests.

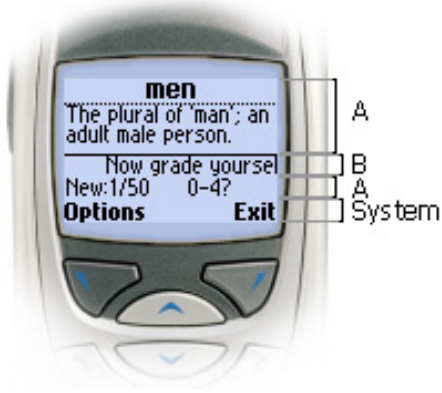

Fig. 1. The Mnemonius application screenshot

There are 2 concurrent tasks involved in the screen rendering. Thread A is responsible for the main part of a screen showing definition of a learned word at the top and the status line at the bottom. Thread B is responsible for scrolling hint for an user in the middle. The very bottom of a screen is maintained by the system and shows menu options.

Mobile applications have to be extremely small and stable. They run on devices with very limited resources. Because of that an application can not use sophisticated programming constructs and many tasks are simplified. In our case screen access among different tasks can not be fully separated. Tasks have to cooperate, exchange data and synchronize themselves.

The bug appeared only once. It was almost impossible to debug it traditionally, so we decided to try model checking. We decided to use the JavaPath Finder [16] model checker. The original application is a concurrent one and it was too big to verify it straightforward. It had to be reduced. The application was reduced as postulated in this paper. Irrelevant parts were omitted (slicing), classes (incl. API) were abstracted and finally some concurrent blocks of code were atomized. Atomization allowed to decrease significantly the state explosion problem.

There were 8 API classes with 30 methods abstracted. Also some application methods were abstracted. The abstraction was made manually, without preparing formal specification. It was based on documentation and no special tool used. Slicing and abstraction left only about 400 lines of code from the original program.

```
protected void paint(Graphics g) {
  Verify.beginAtomic(); //BEGIN ATOMIC BLOCK
  ...  Verify.endAtomic();//END ATOMIC BLOCK
  if (m_iScreenW==0) {
     m_iScreenW=getWidth();//ERROR
     m_iStatusLineTop=iScreenH-font.getHeight()-2;
     if (m_bShowHelpScroll)
        m_iBottomLine=m_iStatusLineTop-font.getHeight();
     else
        m_iBottomLine=m_iStatusLineTop;
  }
  Verify.beginAtomic(); //BEGIN ATOMIC BLOCK
  ...
  Verify.assertTrue(m_iStatusLineTop>50);//ASSERTION
  g.drawString(sText,m_iScreenW-
m_iVScrollHelpPos,m_iStatusLineTop+1,
  ...
  Verify.endAtomic();   //END ATOMIC BLOCK
 }
```

Fig. 2. Source code with the assertion line, atomic blocks and the error line

The question we asked (property being verified) was: Under what circumstances is the variable responsible for determining place for a hint line too low? There was no need to prepare a temporal logic predicate. Introducing assertion into the source code was enough. Reduction had to take into account what variable is to be preserved.

```
Assertion Violated
================================== path to error (88 steps):
Step #0 Thread #0
Step #1 Thread #0
  jr\testscreen\Environment.java:12 Memo m=new Memo();
...
Step #36 Thread #0
  jr\testscreen\NokiaLearnScreen.java:45 m_iStatusLineTop=0;
Step #56 Thread #0 Random #0
  ...
  jr\testscreen\NokiaLearnScreen.java:86 m_scrollTask.start();
...
Step #66 Thread #0
  jr\testscreen\NokiaLearnScreen.java:251 if (m_iScreenW==0)
Step #67 Thread #0
  jr\testscreen\NokiaLearnScreen.java:253 m_iScreenW=getWidth();
Step #68 Thread #0
  jr\testscreen\javax\microedition\lcdui\Canvas.java:51 return
100;
Step #69 Thread #0
  jr\testscreen\NokiaLearnScreen.java:253 m_iScreenW=getWidth();
Step #70 Thread #0
  jr\testscreen\NokiaLearnScreen.java:254
                    m_iStatusLineTop=iScreenH-
font.getHeight()-2;
Step #71 Thread #0
  jr\testscreen\javax\microedition\lcdui\Font.java:35 return
m_iHeight; Step #72 Thread #1
...//thread #1 is being executed
Step #86 Thread #1
  jr\testscreen\NokiaLearnScreen.java:251 if (m_iScreenW==0)
Step #87 Thread #1
  jr\testscreen\NokiaLearnScreen.java:260 Verify.beginAtomic();
  ...
  jr\testscreen\NokiaLearnScreen.java:285
          Verify.assertTrue(m_iStatusLineTop>50);
==================================
  1 Error Found: Assertion Violated
```

Fig. 3. Fragments of a detailed counterexample path

Also the artificial execution environment was prepared because model checking applies only to closed systems – no input can be provided from outside. It simulated starting the application and user input leading to the moment when the bug appeared. It was 1 class with about 20 lines of code.

There were several tries undertaken to verify the application. A few first attempts failed because the program was too big for the verifier. The verifier either used all possible memory or it took too long to finish. It was caused by a state explosion problem. Thus we had to reorganized atomic blocks. Proper atomization decreases unnecessary checking interleaved tasks.

The final verification pass took about 18 s on a 1 GHz Intel. The full counterexample path leading to an error was shown (see Fig. 3). Detailed path allowed quick error detection and removal.

7. Conclusions and future work

We proposed the idea for improving object oriented program abstractions in this paper. It appeared to have practical application. The proposed algorithm handles abstracting not only built-in types and operators but also user defined classes and their methods. It allows for abstracting (and hence for the verification by model checking) of much wider range of Java programs. The proposed method allows compositional approach to the model checking. As classes abstractions can be prepared in advance (and reused) a programmer may focus on verification of his/her program itself. It seems to be particularly useful when applied for API classes for which the abstractions can be specified (and proved) just by their developers.

Articles [9] and [15] present the way to prove specification of real Java programs. Also the rest of abstraction algorithm is sound according to [5]. Therefore the presented algorithm allows the completely formal examination of Java programs.

The case study exemplifies a way of verification object programs and it is a practical application of the proposed algorithm.

The algorithms presented above can be the valuable and practical contribution in the field of software verification. However, they are still under development and their testing on more examples is required.

Acknowledgements

This work has been supported by grant No. 7 T 11 C 013 20 from Polish State Committee for Scientific Research (Komitet Badań Naukowych).

References

[1] Cousot P., Cousot R., *Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints*, Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (1977) 238.

[2] Clarke E.M., Grumberg O., Long D.E., *Model checking and abstraction*, (1994).

[3] Hatcliff J., Dwyer M., *Using the Bandera tool set to model-check properties of concurrent Java software*, Proc. CONCUR 2001, (2001) 39.

[4] Dwyer M.B., Hatcliff J., Joehanes R., Laubach S., Pasareanu C.S., Robby, Zheng H., Visser W., *Tool supported program abstraction for finite-state verification*, Proc. 23rd International Conference on Software Engineering, (2001) 177.
[5] Dwyer M.B., Hatcliff J., Joehanes R., Laubach S., Pasareanu C.S., Robby, Zheng H., Visser W., *Tool supported program abstraction for finite-state verification*, Proc. 23rd International Conference on Software Engineering, (2001).
[6] Pasareanu C.S., Dwyer M.B., Visser W., *Finding feasible counter-examples when model checking abstracted Java programs*, Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 2031, (2000).
[7] Corbett J.C., Dwyer M.B., Hatchliff J., Roby, *A language framework for expressing checkable properties of dynamic software*, Proc. 7th International SPIN Workshop, LNCS, 1885 (2000) 205.
[8] Chechik M., *On interpreting results of model-checking with abstraction*, CSRG Technical Report, 417 (2001).
[9] Huisman M., Jacobs B.P.F., van den Berg J.A.G.M., *A case study in class library verification: Java's Vector class*, STTT, 3 (2001) 332.
[10] http://pvs.csl.sri.com , PVS homepage
[11] Graf S., Saidi H., *Construction of abstract state graphs with PVS*, Proc. 9th Conference on Computer Aided Verification (CAV'97), Springer Verlag, (1997) 72.
[12] Ratajczak J., *An Algorithm for Object Oriented Abstraction*, ICS WUT Technical report, (2004).
[13] Ratajczak J., *Weryfikacja modelowa – stadium przypadku*, Software 2.0, 11'2003, (2003), in Polish.
[14] http://www.mnemonius.com , Mnemonius homapage
[15] Jacobs B., Poll E., *A logic for the Java Modeling Language JML*, Fundamental Approaches to Software Engineering (FASE'2001), LNCS 2029 (2001) 284.
[16] http://ase.arc.nasa.gov/visser/jpf , JavaPathFinder homepage
[17] http://www.cis.ksu.edu/santos/bandera , Bandera homepage
[18] http://www.cs.iastate.edu/~leavens/JML.html , JML homepage