Creating a Tool Chain for Automatic Aspected Model Generation

J.W. Koelewijn
Faculty of EEMCS
University of Twente
j.koelewijn@student.utwente.nl

ABSTRACT
Following the research of Krishnamurthi et al. [KFG04] a tool chain is created which can be used to create models of aspects which can be checked modularly. This is done by adapting the AspectBench Compiler [ACH+05] and using the Bandera [CDH+00] model checking framework.

Keywords
automatic aspected model generation, modular model checking, aspects, Bandera, AspectBench Compiler

1. INTRODUCTION
In the past, software development has undergone some drastic changes, from procedural or functional approaches to the object oriented approach. With the transition to object orientation it was possible to separate different concerns into different objects, instead of writing them down in a long procedural listing. This made it easier to reason about programs and made it easier to maintain the code. But still, there are concerns which cannot be separated. These concerns are tangled with many of the other concerns and for this crosscutting behavior they are called 'crosscutting concerns'.

Aspect Orientation is a relatively new paradigm to address these crosscutting concerns. By means of pointcuts to identify so called joinpoints, pieces of code (called advice) are woven into the base code. This allows for the definition of crosscutting behavior in stand-alone pieces of code, called aspects, which are super-imposed upon the base code before it is run. This way the crosscutting behavior is defined indirectly instead of the normal way of explicitly stating the behavior.

When using this method to define behavior, one can ask some serious questions, such as 'Will the resulting code still have certain properties?', or 'How does one know for sure that super-imposing aspects will not break the software?'

The solution to these questions is twofold. First, one will have to look at general methods to verify code in order to answer questions about properties and second it will be necessary to extend these methods to make them applicable for aspects.

Model checking has been used for years now on procedural and object oriented programs and it has proved its purpose. But to extend it to aspect orientation is somewhat more difficult, because an aspect is not a complete program of which a model can be created in the conventional way. Moreover, the model of an aspect should be interwoven with the model of the base program at the proper joinpoints, which are specified by a pointcut designator. A pointcut designator is an expression that is used to specify a certain point in the execution of a program, called a joinpoint. At this joinpoint code can be inserted. This insertion of code at a joinpoint is called weaving. The result of weaving is a new program in which the crosscutting concerns are woven with the basecode. Because weaving of one model with another is not a trivial task, model checking of aspects usually happens after weaving. This is a more convenient method because creating a model after weaving is just the same as creating a model from a normal object oriented program. This method however has some drawbacks which will be discussed in the following section.

2. PROBLEM STATEMENT
The creation of a model for a program has been a matter of creativity for a long time, but recently tools have become available to create them automatically. One of these model generators is Bandera [CDH+00]. Bandera can be used to create mathematical models from Java code, so it could also be used to create a model from a composed program (i.e. base program with its aspects woven), and this program could be checked against some properties, such as safety and liveness. However, this approach introduces some difficulties. Sometimes the base program is not available to the developer of the aspect, which means that the aspect cannot be verified prior to super-imposition. This could lead to some serious errors, which would have been found if model checking had been used. Another difficulty with this approach is that the verification time is proportional to the size (i.e. number of states) of the model of the system. When the advice is altered often (which is not too hard to imagine in a development cycle), this results in a reduction of productivity, because the verification of advice against a fixed program is inefficient.
To overcome these problems, Krishnamurthi et al. have proposed a method to model check aspects modularly in [KFG04]. This method describes that models of aspects can be checked modularly. This is achieved by first verifying the base code. During this verification process, interfaces are created. These interfaces describe for each joinpoint, which properties are true at that joinpoint. This allows for model checking of the aspects only and without the need of a model of the base program. This process is depicted in figure 1.

Before this method can be used, a model of the aspect has to be created at its own right. Currently a method to generate a model from an aspect automatically does not exist, and the models should still be created by hand. So Krishnamurthi’s method gains in efficiency in the model checking phase itself, but it still lacks efficiency in the model creation phase. This lack of efficiency is the reason Krishnamurthi et al. call for research to generate models of aspects automatically:

“Our approach currently assumes that the programs and advice are given as state machines. There are tools that currently consume Java source and generate state machines similar to those we need; we expect similar tools to eventually exist for aspect source, and thus regard this problem orthogonal to the work presented here.” [KFG04].

3. APPROACH

This paper proposes a tool-chain to automatically generate models from aspects which can be checked modularly. It will do so by using Bandera [CDH+00] and the AspectBench Compiler [ACH+05] to generate models of aspects written in AspectJ. These models can be checked modularly as described in [KFG04]. The verification itself can be done using a verifier based on the Bogor framework, which is part of the Bandera tool chain. The process has been developed in an incremental manner and has resulted in an adaptation of the AspectBench Compiler and a Session file for Bandera.

3.1 Bandera

Bandera is a model generator for Java, which first translates Java files to an intermediate representation called Jimple [VRH], on which it is easy to apply transformations and optimizations. Bandera creates the Jimple representation by using the Soot toolkit [VRCG+99]. Soot is an analysis and transformation tool for Java which creates Jimple to reason upon. The Jimple representation is then transformed into the Bandera Intermediate Representation (BIR), which is used as input for the Bogor model checking framework.

3.2 AspectJ

This paper focuses on the automatic creation of models for aspects written in AspectJ. AspectJ is a widely used aspect-oriented language, which uses a special pointcut expression language to express the joinpoints on which the advice in the aspect has to be applied. AspectJ is used as aspect-oriented language in [KFG04], so to optimize compatibility AspectJ will be used here. This, combined with the availability of the abc toolkit (which will be explained further on) are the main reasons to use AspectJ.

3.3 AspectBench Compiler

Normally, aspects written in AspectJ are compiled and woven with the compiler that comes with AspectJ, i.e. ajc (AspectJ Compiler). This compiler was developed to support fast and incremental compilation and to interact
closely with the Eclipse tool-set. Extensibility was not a requirement for the development of abc. That is why in this paper the abc toolkit will be used [ACH+05, ACH+06]. Abc stands for AspectBench Compiler and it is a compiler build for extensibility. It uses a toolkit called Polyglot for the front-end and Soot for the back-end. The usage of Soot means that in the compilation process an aspect representation in Jimple exists. This is displayed in the left block of figure 1, which represent the AspectBench compilation process [ACH+06]. Figure 2 represents the proposed tool chain. In the figure it is clearly visible that aspects are represented in Jimple at several stages in the compilation process. This aspect representation is used to create a model of the aspect in Bandera and it is this model that will be transformed by the verifier created by Krishnamurthi et al., to make modular model checking possible.

4. ASPECTBENCH COMPILER
4.1 Polyglot

The AspectBench Compiler is an alternative to the standard compiler used by AspectJ. As stated earlier, it uses Polyglot as a frontend. Polyglot is the parser which creates the abstract syntax tree of the Java files and aspects. It is created as a highly extensible frontend for Java. It performs semantic checks on the language and it allows for easy extensibility of grammar of the base language. Using Polyglot, the abc developers were able to extend the Java language with all of the AspectJ constructs [ACH+06].

4.2 Jimple

As stated before Soot serves as a backend to the AspectBench Compiler. Soot is a Java bytecode analysis and optimization toolkit, that uses an intermediate representation of the Java bytecode, called Jimple. Because most of the compiler optimizations known today are three-address based, Jimple is a three-address, stack-less language. In a three-address language, all operations are expressed in terms of explicit variables, i.e. all variables of the expression are known. As a result, the weaving process becomes less complex, because there is no need to unwind the computation stack to keep track of implicit operations, which is the case with bytecode manipulation.

In Jimple, jointpoint information can be extracted fairly easy. It is at this jointpoint extraction phase in the compilation process that relevant information regarding the aspect can be captured.

4.3 Extracting Information

To automatically create a model of an aspect, the pointcut designators (PCD) and the advice applicable at these PCDs are needed. So basically what has to be captured is the pointcut information and the methods called for the applicable advice. This information is being stored in what is called the Aspect Info [ACH+06]. This is seen in figure 2, where the Aspect Info is shown. This information does not reside in one single class, but is made up of an entire package, which is called abc.weaving.aspectinfo. In this package the most relevant class for extracting the information is the AdviceDecl class. In it there is a method called makeWeavingContext in which all the advices of the aspect are being processed with the corresponding pointcut designators. This information needs to be stored for all advices, and should be stored in a file. The most comprehensible format for this file is an XML representation, which will be easy to parse in the consecutive phases.

The second form of information needed is the Jimple representation of the advice. This can be achieved by using one of the compiler flags for Soot, which can be given before starting the compile process. The command to compile the aspects and classes then becomes: abc +soot -f J -soot -d destinir *.*. When this is used, the aspects are compiled and woven with the basecode and the compiled code is written to Jimple files. However, this is not what is needed, because now we have a Jimple representation of woven code. When this will be transformed into a model in one of the next steps, it will be transformed into a model of the woven code. That is why a next step is needed, which is the compilation of the basecode itself. This is done by using the following: abc +soot -f J -soot -d destinir *.java. This results in the compilation of only the Java files. The result is a collection with a compiled aspect and with normal basecode classes. When using this method, the aspect can be checked, because all the objects it refers to are available as well. Moreover, the basecode classes are still runnable without the aspect, because they do not refer to the aspect in any sense.

Using this approach has one drawback however: The around advice can not be used. To see why, we will explain a bit about the weaving process at around advice.

When around advice is used, the weaving process first processes the applicable advice and when it encounters the proceed statement, it will insert a method call to a non-existing method in the baseclass. This non-existing method is inserted into the baseclass in the next stage, and it will contain the same statements pointed to by the pointcut designator. These methods are called shadow-methods, because they are oblivious to the end-user. In order to create a model to check modularly, the aspect representation needs to be checked against the unaltered basecode. But these unaltered classes do not have the shadow methods, so the model creation fails. One way to overcome this would be to monitor the weaving process more closely and detect when around advice is used. More on this can be found in the Discussion in section 7.

5. BANDERA

Bandera is a tool set for the creation and checking of models of Java source code. It delivers a method to extract a model from a program. This extraction process can be guided using slicing and abstractions. It can create models for SPIN and SMV, but it is designed with extensibility in mind. The current release is 0.3, but there exists a closed beta version which was used for this research [SG]. The extensibility is provided by the Tool API, which opens up the possibility to connect to different pieces of Bandera and to extend possibilities. The Tool API and its use are explained in the following section.

5.1 Tools and Sessions

By using the provided Tool API, Bandera can be extended to virtually any domain. The API is used to create new Tools which can be inserted into the chain of tools executed by Bandera. This chain of tools is specified using a XML file (called the Session file), in which Tools can be instantiated and configured. It is also possible to specify in and outputs of Tools. This way the Tools can be instantiated with a correct configuration and communicate with each other.
A Tool is created by implementing the edu.ksu.cis.bandera.tool.Tool interface and communication to the console can be achieved by overriding the edu.ksu.cis.bandera.util.BaseObservable class. The Tool interface specifies methods to deal with the creation, initialization and running of the tool, while the notifyObservers() method in the BaseObservable class makes it possible to send notifications to classes which implement the Observer interface. The main process of Bandera itself implements this interface, so every notification is logged to the standard output.

To help developers in the model checking phase, Bandera provides some standard tools, which are all you need for simple model generation and checking. These standard tools will be explained next, and some tailoring to our needs is explained as well. One of these tools is the ProjectTool, which makes it possible to create a project for your Session. Using this tool, one can state the classes of the project and the main class (with its optional arguments). This is the starting point of each session checking more than one class. The next Tool is the SootCompilerTool. This is a Tool which can compile Java files into Jimple, but which can also load Jimple files itself. It uses some inputs from the ProjectTool and creates a Soot Scene. The Soot Scene is a data structure representing the program. This Scene is used in the consecutive phases. After the compiling and linking by the SootCompilerTool, it is time to inline the main method into the Scene and the optional arguments to this main method. This is done by using the MainSootInlinerTool, also provided by Bandera. This Tool expects the Soot Scene from the previous phase as well as the name of the class containing the main method and its arguments. These latter two are configured in the ProjectTool discussed earlier. When this Tool is finished, the J2BTool (Jimple to Bandera Tool) can be used to transform the Soot Scene into the Bandera Intermediate Language. BIR is developed as a highly extensible intermediate representation, which can be used to model all the classes of Java. When a model is created, all the methods and classes it refers to are translated and inserted into the model. This means as of now the model is completely Java independent. This representation can be written to a file using the output of the J2BTool as an input for the BIRWriterTool. This Tool only expects one input and a configuration option to tell it to which file the representation should be written.

5.2 Bogor
The Bogor framework is the model checking heart of Bandera. Bogor is a model checking framework that allows for new types, expressions and commands. This way the framework can be adapted for virtually all domains. It also makes it possible to create a verifier based on the research of Krishnamurthi et al. [KFG04]. Such a verifier was not available yet at the time of this research, so the standard Bogor implementation has been used to explore the statespace of the models.

To use the Bogor framework, a tool called BogorTool has to be added to the bandera session. This tool requires as input the BIR System delivered by the J2BTool. Using the standard configuration it will check the model for errors and during this process it explores the total statespace. One of the outputs of the BogorTool is the number of states visited, which can be used to compare the checking of a woven program and a unwoven program. This comparison has been carried out and the results will be evaluated later on.

A verifier based on the work of Krishnamurthi et al. should be inserted at the place of the BogorTool. Such a tool should accept the AspectRepresentation gathered from the AspectBench Compiler and the BIR of the aspect. Using this representation it can construct an interface of the joinpoints and advice as stated by Krishnamurthi and use this information to check the BIR model.

6. RESULTS
During the process of creating aspectual models, some unforeseen limitations arose. These adaptations and limitations will be explained in this section.

6.1 abc Adaptations
In order to extract the joinpoint information of the aspect, the AspectBench Compiler had to be adapted. The modifications made are very local, and all the work done to extract it has been done in one method: AdviceDecl.makeWeavingContext() in the abc.weaving.aspectinfo package, of which the adapted version is shown in figure 3.

In line 12 the advice-name is added to the AspectRepresentation object and in line 13 the pointcut designators are added to this advice. This way, the pointcut designators and the advice methods called at those pointcuts are saved in the AspectRepresentation. Because the AspectRepresentation class is a Singleton class, all the pointcut information of an aspect is captured in one class, which is saved to an XML file at the end of the compilation process.

6.2 Bandera Session
As stated before, Bandera uses a Session file in which the connections between tools are being specified. Examination of the documentation of Bandera convinced us that the generation of a model was possible by using only standard Bandera tools, because tools such as a Jimple to BIR compiler are provided. As a result of this, the resulting session file is just a standard one, crafted to our needs. Every tool is initialized and configured and the resulting model is written to disk using only standard tools.

6.3 Limitations
The implementation of the model generation tool chain led to some unforeseen limitations. Some of these limitations are a result of the model checking technique being applied at the end of the chain, but some others are a result of choices made during implementation. The limitations resulting from the model checking technique proposed by Krishnamurthi et al. are recognized in their paper [KFG04]:

• According to Krishnamurthi et al. the model is only applicable to speculative advice, i.e. advice that does not change the basecode. It permits advice to read the data of the program, which is already sufficient to implement standard aspects as tracing and logging, but it does not permit an aspect to modify a program’s data. However, Katz proves in [Kat06] that even weakly invasive advice is verifiable with
the method of Krishnamurthi et al. This means that regulative advice is verifiable as well.

- The model form is not well-suited to programs which use many recursive calls (i.e. programs with a heavily recursive control structure). However, programs with a heavily recursive structure are often ill-suited for model-checking in general, because of the statespace explosion resulting from the recursive calls.

Limitations introduced by our method are:

- The around advice can not be used anymore. As explained earlier this is the result of the lack of shadow methods in the basecode. To overcome this limitation, more research is needed.
- In order to automatically create a model of an aspect to verify modularly, the basecode is needed. This is a result of the references from the aspect to the basecode classes. Without the basecode, the model can not be created.

6.4 Resulting model

The model resulting from the tool chain is a model suitable for modular model checking. The model has been checked with the standard Bogor Tool of Bandera, without any properties to check. When no properties are given, all states are visited by the verifier, but no actions are taken. This leads to a model checker which states that every model is correct. The only use for this is to check the size of the state spaces being checked. This has been done with woven code and unwoven code of a very simple Aspect which is given in figure 4 and the results of the model check are given in figure 5 and 6.

Remarkable is the difference in the size of the state space (max depth). The size of the state space of the modular aspect model is up to 50 times smaller than that of the woven aspect model. This can be explained by the fact that there are only references from the aspects to the basecode when modular model checking is performed, this in contrast with references from and to the basecode when the code is woven. As a result of the smaller state space, the model checking process itself can be performed in less time, while using less memory.

7. DISCUSSION

The intention of this research was to create a tool chain for the automatic creation of aspectual models which can be checked modularly. This goal has been met, but there are some remarks to make. The lack of support of around advice is an example of this. While most use of the around advice can be simulated using before and after advice, there are cases where this cannot be done. This area should be researched further.

Another remark is that the results of this research should be extended to a framework which can generate models of aspects truly automated. While an outline has been given in this paper, there is still much left to be automated. At the moment the resulting Jimple files from the AspectBench Compiler need to be copied to a location Bandera can use. This is still a cumbersome action, so improvements can be made here. This could be achieved by creating a Bandera tool, which uses the AspectBench Compiler to compile aspect code to Jimple. This tool can then be incorporated in the tool chain by using the Session file.

Another problem of the approach taken in this paper is the lack of support for around advice. This could be overcome by careful analysis of the compilation process. When the usage of around advice is detected, the advised statements could be copied to the advice of the aspect. This way the basecode does not need to be altered and the model creation process can continue. This approach will probably give rise to new problems, such as the application of advice over advice. Therefore new research is needed to improve the findings of this paper.

8. CONCLUSIONS

Using the AspectBench Compiler and Bandera a tool chain for the automatic creation of modular aspectual models has been created using only tools which are proven to work. This has been done by adapting the AspectBench compiler, and by using only standard tools provided with Bandera. The models are smaller in state space and are ready to be checked by a verifier created on the Bogor framework based on the work of Krishnamurthi et al. Models can only be created for speculative aspects without around advice. Most of the time this will not trouble the model creation process, because the behavior of around advice can be simulated by using before and after advice. The automatic model creation can result in an increase of productivity when used in conjunction with a model checker based on the work of Krishnamurthi et al. The tool chain proposed in this paper is not fully automated yet, because of there is no coupling between the AspectBench Compiler and Bandera yet. This could be improved in the future by the creation of an aspect compilation tool for Bandera, such that the whole process can be controlled by the Bandera Session file.

REFERENCES


[Kat06] Shimuel Katz, Aspect categories and classes of temporal properties, Lecture
public WeavingContext makeWeavingContext() {
    NormalUnitPrinter printer;
    for (Iterator iter = getAspect().getInstanceClass().getSootClass().getMethods().iterator();
        iter.hasNext();)
        SootMethod method = (SootMethod) iter.next();
        for (Iterator iter2 = method.getActiveBody().getUnits().iterator(); iter2.hasNext();)
            Unit unit = (Unit) iter2.next();
    Vector arglist = new Vector(nformals, 2);
    arglist.setSize(nformals);
    AspectRepresentation.getInstance().addAdvice(impl.toString());
    AspectRepresentation.getInstance().getCurrentAdviceObject().addPoincutDesignator(spec + pc.toString());
    return new AdviceWeavingContext(arglist);
}

Figure 3: Implementation of the makeWeavingContext method


public aspect TracingAspect {

    pointcut setPoint(Point p):
        target(p) && set(private * *.*);  

    pointcut movePointPC(Point p):
        target(p) && call(public void move(..) );

    pointcut moveLinePC(Line l):
        target(l) && call(public void Line.move(..) );

    before(Line l): moveLinePC(l){
        System.out.println("Moving the line: " + l.toString());
    }

    after(Line l) returning: moveLinePC(l){
        System.out.println("Moved the line: " + l.toString());
    }

    before(Point p): setPoint(p){
        System.out.println("Going to set a new coordinate to: " + p.toString());
    }

    after(Point p) returning : setPoint(p){
        System.out.println("I just set a new coordinate: " + p.toString());
    }

    after(Point p) returning : movePointPC(p){
        System.out.println("Moved point: " + p.toString());
    }

    public static void main(String[] args){
        System.out.println("Started");
        Line line = new Line(new Point(0,0), new Point(4,4));
        line.move(4,5);
        line.move(-4,-5);
        System.out.println("Stopped");
    }
}

Figure 4: Implementation of the TracingAspect

Figure 5: Checking result of a woven model
Figure 6: Checking results of an unwoven model