Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-011-0191-0

REGULAR PAPER

The common aspect proof environment

Shmuel Katz · David Faitelson

Abstract We describe the goals, architecture, design considerations and use of the common aspect proof environment (CAPE). The CAPE is an extensible framework integrating formal methods and analysis tools for aspect-oriented programs. It is intended both to encourage the use of formal methods and analysis tools for systems with aspects and to facilitate the development of new tools for aspect analysis. The general principles of the CAPE are first explained, and the components and their high-level organization are given. The user interfaces are described, providing both application oriented and tool oriented Eclipse views. A description of the tools already integrated into the CAPE is given, and some analysis and verification scenarios are shown for an example aspect system. The aid of the CAPE in tool evaluation and experimentation with innovative usage of tools is demonstrated. As one example, *verification aspects* are used to aid in the abstraction and specification needed for formal analysis in Java Pathfinder. The scenarios and strategies employed are steps towards a methodology that includes multiple formal methods tools in aspect-oriented software development.

Keywords Verification framework · Aspects · Formal methods tools · Tool evaluation

1 Introduction

Aspects are powerful programming constructs that allow modular treatment of concerns that otherwise would cut across usual class and package modules (see [6] for a variety of aspect languages). An aspect declaration generally includes a description of when it is to be activated (called a *pointcut descriptor*) and code segments to be executed (called *advice*). Aspects often introduce nonlocal effects which make them difficult to reason about. One possibility to help us reason about systems with aspects is to use formal verification and analysis techniques.

Unfortunately, using such techniques is not trivial, even for non-aspect systems. Indeed, full post-facto formal verification of implemented software systems has proven impractical. Inductive methods have floundered due to the difficulty in providing appropriate invariants, and model checking has proven unable to directly handle the huge state-space of complex software with asynchronous threads or processes. Thus, any attempt to introduce formal methods for aspects has to either deal directly with key abstract models, e.g., of individual aspects, or provide a methodology for abstracting an implemented system by isolating components for independent analysis and/or reducing the possible values of fields or variables.

A second problem is that even though there are attempts to develop tools that use formal techniques to verify aspects, there has not been any attempt to integrate the different tools into a single coherent framework.

A third difficulty is that existing verification tools for aspects differ widely in their robustness and maturity and may be showcased on example systems that do not reveal the limitations of the tool. Thus, a "neutral" environment is essential for evaluation of and experimentation with tools.

In order to improve this situation, we have been developing, as a part of the AOSD-Europe project, the common aspect proof environment (CAPE), which is a platform for the integration of verification and analysis tools for aspect-oriented languages. The purpose of the CAPE is to try and

S. Katz (✉) · D. Faitelson
Computer Science Department, The Technion, Haifa, Israel
e-mail: katz@cs.technion.ac.il

D. Faitelson
e-mail: davidfn@cs.technion.ac.il

Published online: 16 March 2011

bring the tools and techniques of formal verification closer to developers of systems with aspects, and to facilitate the evaluation of such tools, as well as to integrate the use of such tools into development methodologies for aspect systems.

In this paper, we summarize the work done on the CAPE environment and use a case study to demonstrate how the CAPE may be used to apply various tools to verify aspectual code. The scenarios that we demonstrate on the sample aspect system deal both with verifying an abstract model of an individual aspect relative to its specification, even before it is expressed in a programming language, and with the abstraction needed to formally verify a module of Java code with aspects in AspectJ. Both of these help to alleviate the first problem mentioned above, of abstraction. The scenarios themselves and typical chaining of methods facilitated by the uniform tool interface, along with the dual view provided by the CAPE (explained in the following Section), help with the second problem, of integration. The difficulties encountered in applying some of the tools to new examples, and experiments with new ways of applying the tools (some of which are described later), show the usefulness for tool evaluation.

In the following section, the structure and user interfaces of the CAPE are described, and a brief overview of the tools presently in the framework is given. Then, our view of integrated use of tools is explained, followed by a description of the elements in the case study, using a medium-sized aspect application called "Health Watcher" [26] and the application of several usage scenarios and tools to it. The paper ends with a description of related work, and conclusions.

2 CAPE

The CAPE, whose basic architecture was first described in [5], is an extensible platform for integrating verification and analysis tools of aspect-oriented programs. The CAPE provides a common interface to a variety of verification tools, manages the verification artifacts, and offers a standard interface for extending the platform with new tools. In addition, the CAPE provides a way to link various tools together to form proof strategies, exploiting the relative strengths of the different tools, and suggests use scenarios to demonstrate how different verification tasks can be approached. The CAPE is implemented as a set of plugins on top of the Eclipse platform. It is available for examination and download at http://www.cs.technion.ac.il/~ssdl/research/cape.

In the approach to tool integration seen here, modules of existing tools become the basis of the integration framework. Thus, a collection of existing static analysis and verification modules is used to construct specific aspect verification tools for a variety of programming and design languages. The generic collection includes data-flow and parsing components, as well as existing model checkers and type analyzers. For example, the MAVEN tool to check the correctness of individual aspects (and that is described in more detail later) uses the NuSMV model checker, and in turn is a component in another tool being developed to detect interference among multiple aspects.

The CAPE proof environment is intended for three types of users: *application developers and testers* who wish to apply the tools to aspect programs or designs, *tool developers* providing new verification or analysis tools for aspects, and *tool evaluators* who wish to evaluate the quality or robustness of a tool. Therefore, individual verification and analysis modules are intended to be accessible and separately included in the CAPE, both so that they can be individually activated or evaluated, and also so they can serve as building blocks for developing new tools. Besides standard code analysis modules, facilities are provided to construct finite-state models that are then input to model checkers.

The CAPE uses Eclipse's plug-in mechanism to achieve a flexible infrastructure of its various modules. The CAPE provides a framework in which the different modules can communicate, sharing resources and complementing each other's functionality. This is achieved through the Generic Aspect Representation (GAR): a collection of common representations of aspect-oriented source code and other properties. Among the representations are the abstract syntax trees of the component modules of implemented systems, abstract transition system representations intended for model checking, dataflow information (including *define-use* pairs and method summaries) for aspect systems, representations of pointcuts as regular expressions, and properties proven about the system.

Since the GAR is implemented on top of the Eclipse resource management platform, we first provide a brief overview of a user view of Eclipse and then describe how the GAR is implemented. The basic unit of organization in Eclipse is a *project*. Each project consists of a collection of files and directories. In addition, each project has one or more *natures*. A nature defines the type of the project and associates to the project a *builder*. Thus for a Java application we may create a project with a Java nature that associates a Java compiler as the builder of the project. Whenever the build action is selected from the menu, the Java compiler takes all the Java source files in the project and compiles them.

A single project is homogeneous. You can create a Java project or a Ruby project, but you cannot have both Ruby and Java source code in the same project. In order to support applications built using more than one language, we can collect several projects into a *working set*. We then can have a working set for an application, containing a project with a Java nature for the Java code, a project with a Ruby nature for the Ruby code, and so on.

Another important property of Eclipse projects is that they can refer to the same files and directories using *linked*

resources. A linked resource is similar to a symbolic link in UNIX. This makes it possible to create some projects as views of other projects by making the files in the view projects link to the files in the other projects.

The CAPE's GAR is implemented as an Eclipse working set. A typical CAPE GAR consists of one or more projects that hold the source code of the application and one or more analysis projects that either hold abstract models of the application, or use linked resources to select the parts of the source code that are relevant for analysis. For example, to perform aspect data flow analysis using the AIDA tool (to be described later) we create an AIDA project (that is, a project whose nature is AIDA) and populate it with links to the source files that we would like to analyze. When we build the project, the AIDA tool (which is the builder of that project) compiles the files in the project and produces a graph that is added to the project's output directory.

In addition to the standard AspectJ Development Toolkit (AJDT), the compilers presently in the CAPE are the standard AspectJ [18] compiler, the extendable abc compiler [1], and a partial compiler for Composition Filters [2]. Moreover, there are five foundational tools (not especially oriented to aspects) available, namely

- Java PathFinder [10], a model checker for woven systems, which is explained in detail in Sect. 8.
- NuSMV [3], a well-known publicly available (Open Source) model checker.
- Cadence SMV [22], another version of an SMV model checker, with different properties.
- GROOVE [24], a tool set for creating and analyzing graphs, including an editor for creating graph production rules, a simulator for visually computing the graph transformations induced by such rules, a generator for automatically exploring state spaces, and an imaging tool for converting graphs to images.
- VPA package [23], a collection of decision procedures for Visibly Pushdown Automata that can be used to determine whether complex pointcuts from different aspects define overlapping joinpoints.

The tools specifically oriented to analyzing systems with aspects include

- MAVEN [7,8], a model analyzer for individual aspects relative to their specifications (that is further explained in Sect. 7).
- SECRET, a syntactic tool to detect conflicts (that originates in the Composition Filters compiler suite).
- AIDA [30], a static analysis tool that detects how aspects interact through changes they cause to the data flow of a system.
- AIA (Aspect Introduction Analyzer) [11], which analyzes and reports conflicts related to introductions. AIA automatically converts (part of) the abstract syntax tree of Java and AspectJ programs (source code) to a GROOVE graph-based model. In addition, it automatically converts the introduction specifications that are part of the AspectJ source code to GROOVE graph transformation rules, which can be applied to such a model.
- CNVA [12], a scenario-based tool to check that every execution sequence in a woven system is equivalent to one where aspect advice is done atomically immediately from states that are joinpoints (also briefly considered in Sect. 10).
- GRASS [27], a tool for detecting interference between aspects written in the Composition Filters language. The tool builds a set of graph transformation rules that models the behavior of the composition filters and then uses the GROOVE tool suite to generate a labeled transition system that represents all the possible executions of the filters. Interference between aspects is detected by looking for different final states in the graph, since they indicate that a different execution order of the aspects (at the join point) produces different results.

The structure of a typical CAPE application working set is seen in Fig. 1. In it, an application engineer for a security application has applied dataflow analysis using AIDA to detect whether the PasswordEncryption and PasswordRetrieval aspects potentially influence the same fields or variables, used AIA introduction analysis to check that there are no name conflicts in those aspects, and used Maven to check the correctness of state machine models of those components relative to a linear temporal logic specification.

The tools introduced into the CAPE are divided into the categories *Verifiers, Translators* and *Editors*. Verifiers analyze their input and provide the user with a viewable result of the analysis process. Examples include model checkers and data flow analyzers. They are implemented over Eclipse's *launching* mechanism, which allows the user to "debug" his/her input files with a verifier much like any other debugger in Eclipse.

Translators translate (or compile) their input into an output that other tools may use. Some examples are Graph Production System generators, finite state machine builders, or compilers. They are implemented as *builders*, and are used as described above.

Editors, as their name suggests, modify their own input files and are implemented as either built-in or external Eclipse editors.

In addition to the typical (for Eclipse) application-based view, the CAPE provides a tool-based Eclipse perspective, illustrated in Fig. 2, which offers a centralized location where

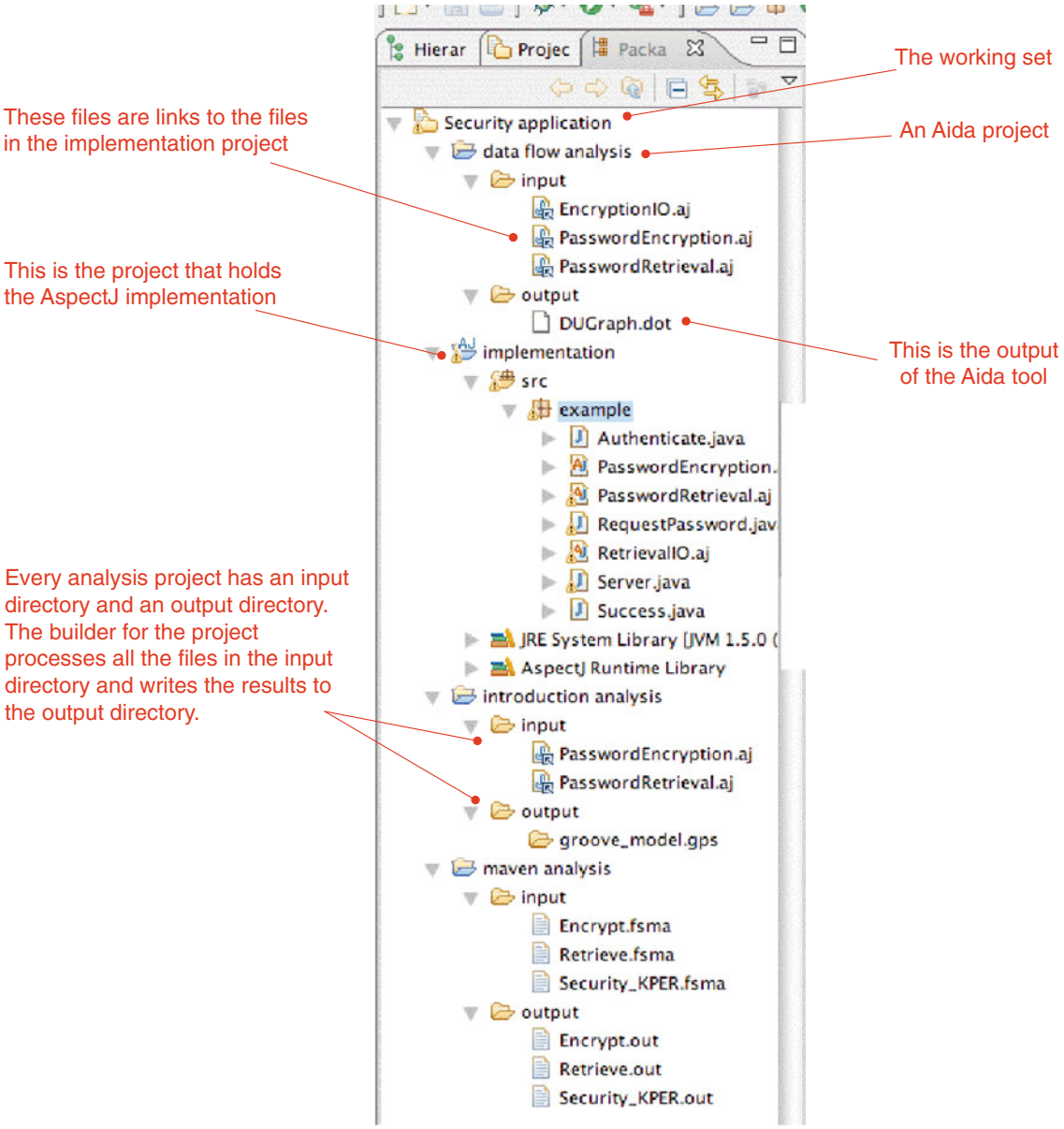

Fig. 1 The structure of a CAPE working set

the user can browse and activate the available tools on the applications to which that tool has been applied. This allows convenient evaluation of tools in the CAPE by gathering together examples of its application.

The organization of the CAPE facilitates addition of new tools to the CAPE, and a precise menu for integrating such tools is provided on the CAPE webpage (along with a video demonstrating the steps in adding a new tool seamlessly to both perspectives of CASE tools). This should encourage both new tool development, and evaluation of additional aspect verification tools.

3 The importance of integrated analysis

Effective analysis requires the integration of different tools and techniques at different stages of the development cycle, for three main reasons.

First, different tools have complementary (though often overlapping) capabilities. For example, because AIDA can only find problems that involve data flow dependencies it did not detect a race condition in our case study that clearly is related to control flow. On the other hand, even though a tool like JPF can find, at least in principle, any problem that

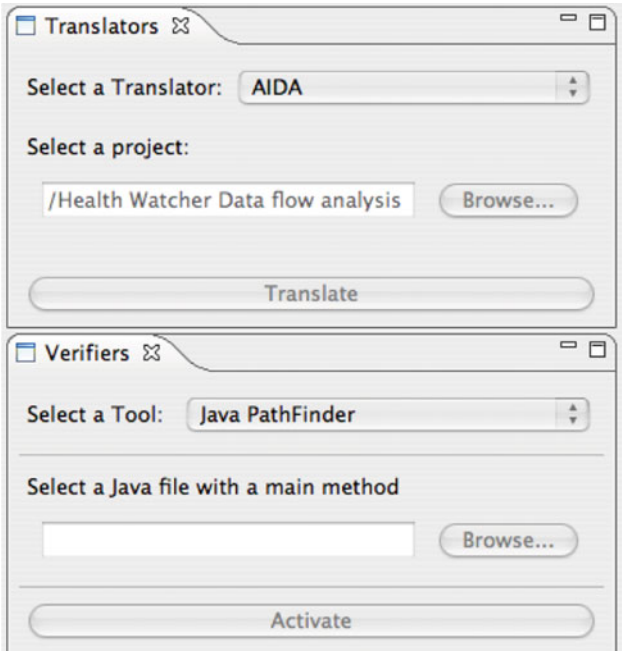

Fig. 2 The CAPE's tool perspective

AIDA can find, in practice we would prefer to use AIDA whenever possible because it operates entirely automatically on the source code, while with JPF we need to invest more time and effort to adapt the system and to design and insert appropriate assertions to locate the problem. In general, syntactic and dataflow tools (like AIDA and AIA) are easier to operate but are limited in the classes of problems they can detect, while general purpose tools (like JPF and MAVEN) can detect (almost) any class of problem but require (sometimes much) greater effort.

Second, many tools, in particular general purpose tools, follow a pattern of diminishing returns. Investing a small amount of effort finds most of the problems, but to find more problems requires investing much more time. For example, with JPF it is relatively easy to find violations of simple invariant assertions. However, when we are looking for more general temporal properties, as will be described, we must translate them to simple assertions by augmenting the state space of the system with auxiliary variables updated at various points. At some point the complexity of this mechanism might be greater than the system under test, and it then may be better to try another tool (for example, MAVEN which natively supports checking full LTL temporal logic properties).

Finally, different tools operate at different levels of abstraction, trading time and effort against precision. Even if an abstracted model satisfies desired properties, the full implemented system has to be checked, because the abstracted model might not fully reflect the details of the implementation. An example of this phenomenon will be shown.

For all of the aforementioned reasons, the integration of different tools is essential for an effective analysis and verification process. Such a process should be flexible (allowing us to accommodate real world limits of time and effort), economical (allowing us to extract from each tool the greatest benefit with the least effort), and finally allow compensating for the limitations of each individual tool with the capabilities of the other tools.

4 The case study

Health Watcher [26] is a web application that manages health-related records and complaints. It contains about 100 classes and 14 aspects with a total of about 10K lines of code.

The application is essentially a database of various health-related records: disease types, complaints, health units, employees of these units, and so on. For each such problem domain entity there is a repository that can be stored either in a relational database or in memory (usually for testing purposes). The decision whether to use an external database or to store everything in memory is global to the system, and is determined by the initialization of a constant.

The functionality of the system is collected under a single facade object which contains a method for every operation and query that the system supports: creating new employees, updating complaint records, retrieving complaints and so on. Users interact with the system (like any traditional web application) by filling a form in the browser and sending it to the web server. Next, the web server forwards the HTTP Get or Post request to the servlet, which parses the request and calls the appropriate Health Watcher facade operation. The servlet also translates the results of the call (including exceptions) into HTML which is sent back to the browser. The aspects in the system are divided into packages:

- Synchronization aspects, of which there are three, convert the record repositories into thread-safe versions by protecting their operations with a Java `synchronize` keyword.
- Distribution aspects provide an option for running the system in a client-server mode by changing the servlet to contain a Java RMI client and creating an RMI server that runs the application logic.
- Exception handling aspects are used to translate exceptions into HTML messages. This is required because unless caught, application level exceptions such as transaction aborts will cause internal web server errors. The exception aspects catch these exceptions and convert their content into an HTML message.
- Persistence aspects replace calls to the creation of repository objects by calls that decide whether to create an in-memory or a persistent implementation of the repository

interface. The decision is based on the value of a global constant.
– A transaction management aspect turns the public methods of the facade into transactions by invoking a begin-transaction operation before every facade method call, a commit-transaction operation after a successful return, and an abort-transaction operation after an exceptional return.

We applied several CAPE tools to the Health Watcher system. The static analyzer AIA and dataflow tool AIDA were used to identify possible areas of interest. Using AIA gave us a visual view of aspect introductions which highlighted some that under further investigation were found to create naming conflicts. Applying AIDA created a graph of data flow dependencies between aspects which acted as a map of dangerous spots that we then examined more closely. We used MAVEN to determine the correctness of models of specific aspects and also applied an extension for detecting aspect interference to analyze an interaction between the exception handling and transaction management aspects. Finally, JPF was used on modules of bytecode after weaving both Health Watcher aspects and additional aspects to specify and abstract the module. Below we describe the application of these tools in greater detail, how the CAPE can be used both to verify an application, and to evaluate the tools. The difficulties encountered in using the tools are summarized in Sect. 9.

5 Aspect introduction analysis (AIA)

The Aspect Introduction Analyzer tool (AIA) analyzes and reports conflicts related to introductions. AIA automatically converts (part of) the structure (abstract syntax tree) of Java and AspectJ programs (source code) to a GROOVE graph-based model. In addition, it automatically converts the introduction specifications that are part of the AspectJ source code to GROOVE graph transformation rules.

The tool explicitly models several kinds of composition conflicts as graph matching patterns. Using GROOVE as a subsystem to match these patterns against the program models, the tool can automatically detect the occurrence of composition conflicts. The same tool is used to apply the introductions (represented by graph transformation rules) to the program model.

AIA enables us to detect situations in which the transformations can be applied in different orders, leading to potentially different transformed program structures. In practice, this would mean the (normally arbitrary) order of compilation may lead to differently structured compiled programs—a highly undesirable effect. It can, for example, detect "holes" in the type-checking of AspectJ, not detected by the regular compiler, where the same field name is introduced by two aspects, but given different types (and used for different purposes). In practice, such conflicts were not found in the original Health Watcher system, but when errors were inserted in order to check the tool, it indeed succeeded in identifying them. Once the user creates an AIA project, and selects Java and AspectJ code files to be analyzed, the tool automatically executes and creates a folder with results that can be viewed using GROOVE's graph visualization package.

6 Aspect interaction detection analysis (AIDA)

The AIDA tool [30] detects how aspects interact through changes they cause to the data flow of a system. For example, one aspect may read a variable x while another aspect (perhaps introduced later) writes to this variable. The earlier aspect which assumed that x was updated by the original program is now reading values that were updated by the new aspect. This interaction may lead to unexpected (and often undesired) modifications to the behavior of the program. The AIDA tool can analyze the code before it runs and locates such interactions. It performs an iterative bottom-up dataflow analysis on AspectJ programs, based on summaries of methods and aspect advice.

We used AIDA to look for possible problems in the Health Watcher case study. In some variants of the application system, AIDA indeed detected potential Read/Write interference among aspects. For example, additional aspects that monitor execution may be influenced in unwanted ways by aspects that increase the distributiveness of the application. Once the potential interaction among aspects is detected by the dataflow, it can be considered in greater depth to determine whether errors could result in the woven system.

Of course, this tool is useful for identifying potential interference based on dataflow, and cannot detect problems such as race conditions based purely on control flow, or semantic interferences related to the intended properties of each aspect.

7 MAVEN

MAVEN [7,8] is a model checking tool that determines whether an aspect is correct relative to its specification. The specification is given in two parts: an *assumption* relating to an underlying system to which this aspect should be woven, and a *guarantee* about any system to which the aspect has been woven. The aspect is *correct* relative to its specification if whenever it is woven to a system satisfying the assumption, the woven result will satisfy the guarantee. A user has to describe the assumption and guarantee in temporal logic, and the model of the aspect advice as a transition system in the language of the SMV model checker. The pointcut is

also identified in terms of predicates that are true for pointcut states.

The tool then automatically builds a transition system that represents every computation of a system that satisfies the assumption, and weaves the aspect advice transition system into it, by transferring control from a state representing a pointcut to the initial state of an advice segment and transferring control back to the underlying system in the appropriate state after the advice segment. The result of this construction is then passed to the NuSMV model checker that determines whether the guarantee of the aspect holds for this model. If it does, the aspect is correct relative to its specification.

Such verification of an abstract aspect model is valuable in order to check that the assumptions and guarantee are well understood and that the pointcut and advice model achieve the desired result. However, it does not cover problems that might arise from the use of multiple aspects, where one aspect may interfere with the correct operation of another. To treat this problem, an Interference Detection tool [13] was used. This tool itself uses MAVEN as a subsystem. Given two aspects, A and B, with assumptions P_A and P_B respectively, for each possible weaving order of the aspects, two checks are performed: the KP_{AB} check determines whether aspect A, when woven into a system satisfying both P_A and P_B, keeps the assumption of B. The KR_{AB} check determines whether aspect B, when woven into a system satisfying the guarantee of A, R_A, keeps this guarantee. If these checks are passed for each pair of aspects in a library, any collection of aspects from the library can be woven without causing interference.

As an example of applying these to the Health Watcher system, consider the transaction management and exception handling aspects. We modeled the aspects as transition systems, and their specification was given in the assume-guarantee form described above. This is the input format of both MAVEN and the Interference Detector. The exception handling aspect (*Ex*) is applied when an exception is thrown by a facade method. This event is modeled by a state-predicate $throw$ (thus the pointcut of the aspect is $throw = true$). The aspect catches any transaction exception, creates an appropriate report, and returns it as an HTML message to the web server. Other exceptions are rethrown. The code and full advice model are not shown here.

We modeled by the *throw_trans* predicate the event of throwing a transaction exception, and by the *msg_send* predicate the event of sending the HTML message. We needed to express that throwing a transaction exception is indeed a type of throwing. Formally,

$$P_{Ex} = \mathsf{G}(\mathit{throw\_trans} \rightarrow \mathit{throw})$$

The specification of the exception handling aspect (*Ex*) guarantees that if a transaction exception is thrown, then an appropriate message will be sent:

$$R_{Ex} = \mathsf{G}(\mathit{throw\_trans} \rightarrow \mathsf{F}\, \mathit{msg\_send})$$

We have verified the aspect *Ex* in MAVEN according to the above specification, and found it to be correct.

The transaction management aspect (*TM*) is applied in three cases: when a facade method is about to be executed (modeled by the *before* predicate), when a facade method has just finished successfully (modeled by *return* predicate), and when an exception is thrown during the execution of a facade method (the *throw_trans* predicate mentioned above). Executing a facade method means performing a transaction; thus, when such a method is executed, the *in_progress* predicate is true. The specification of the transaction management aspect guarantees that whenever a transaction finishes successfully, its results are committed to the database (this event is modeled by the *commit* predicate), while if an exception was thrown during the execution, the results will be discarded (*rollback* predicate). The transaction management aspect should also guarantee that every transaction execution is preceded by announcing its beginning to the database (the *begin* predicate). Formally,

$$\begin{aligned} R_{TM} = [&\mathsf{G}(\mathit{return} \rightarrow (\neg \mathit{rollback}\ \mathsf{U}\ \mathit{commit})) \wedge \\ &\mathsf{G}(\mathit{throw} \rightarrow (\neg \mathit{commit}\ \mathsf{U}\ \mathit{rollback})) \wedge \\ &\mathsf{G}(\mathit{in\_progress} \rightarrow (\mathit{in\_progress}\ \mathsf{S}\ \mathit{begin}))] \end{aligned}$$

The assumption of the transaction management aspect expresses our external knowledge of the semantic relationships between the predicates of the base system. It contains statements about the connection between the *in_progress* predicate and all the rest, and also about mutual exclusion between the *throw*, *return* and *before* predicates. Although several such relations were overlooked in early attempts, leading to false counterexamples, ultimately the following predicate was used:

$$\begin{aligned} P_{TM} = [&\mathsf{G}(\neg(\mathit{return} \wedge \mathit{throw})) \wedge \\ &\mathsf{G}(\neg(\mathit{begin} \wedge \mathit{return})) \wedge \\ &\mathsf{G}(\neg(\mathit{begin} \wedge \mathit{throw})) \wedge \\ &\mathsf{G}(\mathit{in\_progress} \rightarrow \\ &\quad (\mathit{in\_progress}\ \mathsf{U}\ (\mathit{return} \vee \mathit{throw}))) \wedge \\ &\mathsf{G}(\mathit{in\_progress} \rightarrow \\ &\quad (\neg \mathit{before} \wedge \neg \mathit{return} \wedge \neg \mathit{throw} \wedge \\ &\quad (\mathit{in\_progress}\ \mathsf{S}\ \mathit{before})))] \end{aligned}$$

We have verified the transaction management aspect in MAVEN according to the aforementioned specification and also found it to be correct.

Though both the aforementioned aspects were correct, we detected an interference among them in one abstract variant of the Health Watcher system using the Interference

Detection procedure. We found that when the exception handling aspect was woven before the transaction management aspect, the latter does not preserve the guarantee of the exception handling aspect. The counterexample showed that a transaction exception is caught by the transaction management aspect which aborts the transaction but does not rethrow the exception and thus the exception handling aspect is not activated and the notification message is not sent. Moreover, we found that the opposite ordering of the aspects is also problematic: when the transaction management aspect is woven before the exception handling aspect, the exception handling aspect does not preserve the guarantee of the transaction management, for similar reasons.

8 Java Pathfinder

Java PathFinder (JPF) [29] is a model checker [4] that is implemented as a Java virtual machine. That is, JPF executes Java's virtual machine code, but unlike a regular virtual machine, JPF explores all the possible states that the program may reach.

JPF keeps track of every state (the content of the heap and the stack of each thread) the program has visited. When the program reaches a point that may lead to several possible executions (for example when two threads are executing in parallel) JPF reruns the program on all the possible executions. As a result, any assertion we embed in the code will be checked for all the possible executions of the program, under the assumption that the program always terminates. This means that if the assertion never triggers an exception then we are sure that the program can never violate the assertion. As an additional benefit, if the assertion is violated then JPF creates a report that consists of the entire history of the execution up to the point of the violation.

However, there are obstacles to the use of JPF. In order to apply it successfully to a realistic system we must both describe the properties we wish to check, and ensure that the state space of the program or module to be checked is small enough to fit within the memory of the computer.

In the original application of JPF to the case study, we followed standard practice of inserting assert statements whenever needed in the source code to specify needed properties. This means that we had to change the program every time we decided to check for a different property. Moreover, often a single property cannot be checked in a specific place, but we must distribute assertions at several places in the code. To treat the needed state reduction, we followed the traditional approach of manually separating the parts of the program we wish to verify from the rest of the system, and reducing the domains of variables. This may be done either by writing a model that is entirely separate from the original program or by manually modifying the original program.

As part of JPF's integration into the CAPE, we investigated how best to apply it to complete systems with aspects after weaving. Since it works on the Java Bytecode level, this is not in principle difficult, but we sought to alleviate the above obstacles to use of JPF. In particular, we experimented with using aspects themselves to give a cleaner and more modular usage of JPF. By using *verification aspects* we can solve problems in annotating and specifying systems before applying JPF. We can write a different aspect for each new property and then weave it into the program without having to manually edit it. And by using the appropriate pointcut patterns we can easily distribute the different assertions to their appropriate locations. It also becomes trivial to remove the assertions when we have finished with the verification task. Finally, the connection between the pointcuts —representing where assertions are made—and the advice—representing the assertion itself—is immediate and direct.

Some of the properties we wanted to test were general temporal properties (namely the properties defined in the MAVEN model). Because JPF does not support full temporal logic, an aspect was written that augmented the state of the program by a set of variables that remembered key events and updated those auxiliary variables at appropriate places. Using these variables we formulated the properties as simple assertions for programs that terminate.

For example, to express

$$R_{Ex} = \mathsf{G}(\mathit{throw\_trans} \rightarrow \mathsf{F}\, \mathit{msg\_send})$$

we record the occurrence of *throw_trans*, set another OK flag to false, and then begin testing whether *msg_send* has occurred. When that happens, OK is set to true. When such an execution terminates, it is simple to check the value of OK to see whether the temporal formula held for that computation. Thus, the new claim is that OK is true whenever the program terminates (and the tool is only used for programs that by assumption always terminate).

To treat the needed state reduction, the traditional approach is to manually separate the parts of the program we wish to verify from the rest of the system and reduce the domains of variables. This may be done either by writing a model that is entirely separate from the original program or by manually modifying the original program. However, such an approach is tedious and error prone. As a result, verification is rarely preformed more than once, even though the system undergoes many changes.

Aspects also were used to cleanly decouple state reductions from the program. It then becomes easier to reapply the abstraction when the program changes. In addition, it becomes easier to understand the nature of the abstraction. By looking at the aspect we can immediately see which concrete elements of the system are replaced by abstract versions.

First, we must take control of the inputs to the program and replace them with JPF's special nondeterministic choice

operator. This is easily done with aspects that replace input by the needed nondeterministic choices. Second, we must disconnect the parts of the application that we wish to test from the entire application. In our case, we created a "driver" environment for running the Health Watcher system outside of a web application server because it would have been impractical to run the entire web server through JPF.

Another part of the environment is the database management system (DBMS) that underlies Health Watcher's persistence mechanism. As usual in testing, a "stub" DBMS is needed that also simulates exceptions to be tested. For example, here is the implementation of the abstract commit operation:

```
public void commitTransaction()
   throws TransactionException {
  if (Verify.getBoolean())
    throw new
      TransactionException(``STUB_COMMIT_EX'');
}
```

Such an implementation not only removed a problematic component from the test environment but also forced JPF to test the system through both the normal and the exceptional branches.

We then can create a verification aspect, as seen earlier, with three boolean variables:

- *txex_thrown* is set to true when a transaction exception is thrown.
- *msg_sent* is set to true when an HTML error message is sent to the web server.
- *tx_abort* is set to true when a transaction is aborted.

For each variable we create a pointcut that catches all the relevant events that the variable is intended to model. We then use an advice to set the variable to true at these points. For example, we set the *tx_abort* variable to true whenever the persistent mechanism's rollback operation is called:

```
pointcut transactionAborts() :
  call(void IPersistenceMechanism.
                rollbackTransaction());

after() : transactionAborts() {
  tx_abort = true;
}
```

After the program completes executing (either normally or abnormally) we check that if a transaction exception was thrown then it was reported and aborted:

```
pointcut mainProgram() :
    execution(void Main.main(..));

after() : mainProgram() {
  assert(!txex_thrown || msg_sent);
  assert(!txex_thrown || tx_abort);
}
```

We can place the assertion at the end of the program because we have created a version of the main method that runs the system only once. However, because JPF checks all the possible paths through the program and because we have replaced the persistence mechanism with nondeterministic versions, the program will be tested through all the possible combinations of normal and abnormal execution.

Recall that when we created the abstract MAVEN model of the transaction and exception aspects, we assumed that operations such as begin, commit, and abort a transaction were atomic. However, in practice they are compound operations that may fail and in particular may throw transaction exceptions. The JPF worked directly from the woven bytecode of the implementation, and exactly such a scenario was caught, detecting additional errors.

This example demonstrates the importance of using a combination of approaches. The abstract model detected key errors at an early stage of development, but it contained hidden assumptions that were not true of the actual program. In contrast, adapting the implemented system to run on top of JPF was complicated and time consuming, but it revealed hidden assumptions and errors that were not detected in the abstract models.

9 Tool evaluation using the CAPE

In addition to providing tools for application developers using aspects, and to aid in producing new aspect analysis tools, the CAPE provides an environment for evaluating the tools in it. Both students and industrial adopters can use the dual view of the CAPE to see which tools have been applied to a particular aspect system and which examples exist for each tool. The outputs can be evaluated for readability, effectiveness, and scalability, and the effort involved in applying the tool can be measured. Such evaluations were applied to the Healthwatcher case study, as well as to other CAPE applications.

Early developments of analysis tools are often demonstrated on examples chosen to feature the strengths of the tool. The present output of AIA is a graph that represents the syntactic structure of an AspectJ program. This output does not scale well even to relatively small programs such as our case study. The analysis itself is relatively fast, about the same time it takes to compile the program, but the resulting graph is too large. There is an option to zoom into the graph but it is slow, and once we have zoomed, we have to scroll the display in order to look for the highlighted occurrences. This is a

long and laborious process. However, it should not be difficult to analyze this graph automatically to detect problematic situations, with only these being displayed to the user as error or warning messages.

For the AIDA tool, evaluation of an early version provided feedback to the developers that made it much easier to be fully exploited in practice. First, in the first version the graphs generated by AIDA contained not only dependencies between different aspects (inter-aspect dependencies) but also dependencies between different parts of the same aspects (intra-aspect dependencies). The problem is that the intra-aspect dependencies are much more frequent than the (more interesting) inter-aspect dependencies and tend to dominate the graph. As for the previous tool, it becomes very difficult to locate the interesting dependencies in the graph. This problem was easily solved by adding options for selective analysis to hide the intra-aspect dependencies.

The case study showed that the MAVEN tool can check semantic correctness relative to a specification, and can detect subtle interferences among aspects, but is difficult to use because of the need to accurately model the aspects and give precise specifications. The user must write the specification in temporal logic (including relations among the predicates), and the model in the SMV input language, which are significant obstacles for many users. The difficulty in writing formal specifications has motivated continued research on semi-automatically refining temporal logic specifications [14].

As seen in the case study, there is also the problem that the abstract model that is checked by MAVIN might not entirely reflect the implemented system, a typical difficulty when abstract models are not automatically refined into implemented code. Thus, both verification of an abstract model and a verification derived directly from code are needed (or else the code should be reliably generated from the abstract model).

The difficulties of JPF, in expressing the needed augmentations to the system and adding parts to "remember" key events, were already noted and were treated through a methodological approach that used aspects over JPF.

10 Other tools

Not all of the tools in the CAPE were applied to the Health Watcher system, nor is it reasonable to apply every tool to every application. The tools to be used depend on the specific verification tasks deemed most important, be it modeling, considering individual aspects intended for reuse, verifying a particular woven system, or treating aspect interactions.

For example, yet another CAPE use scenario investigates the weaver itself, using the CNVA tool. In particular, it can check whether it is justified to consider the series of steps taken in the woven system as if the advice instructions were done atomically immediately at a joinpoint. (For reasons of optimization, they might in practice be interleaved with instructions from other aspects or the base program, without damaging the correctness.) For the case study, it is reasonable to consider two clients and a single server, and desired typical computations where the steps in each aspect advice are executed sequentially with no other interleaving of operations from other threads, immediately when the appropriate joinpoint is reached. These cases are described using a regular expression language. The CNVA system then automatically checks whether every computation in a woven model of the entire system (that does include various orderings of operations from different threads) is equivalent to one of the desired typical ones. Two computations are defined as equivalent if they differ only in that independent operations were executed in a different order. Thus, it is also necessary to check which operations are independent of each other.

In other situations, where specifications are not available, but the system is supposed to be deterministic (at least in giving the same response to the same query), the GRASS tool can be used. The effect of multiple aspects applied at the same joinpoint is considered with different possible orderings and any differences in the results are identified.

11 Related work

Of course, there are numerous other works on formal methods or analysis for aspects, in addition to those presently included in the CAPE. The first work to separately model check the aspect state machine segments that correspond to advice is [19,20], where the verification is modular in that base and aspect machines are considered separately. The treatment there is for a particular aspect woven directly to a particular base program. It shows how to modularly extend properties which hold for that base program to the augmented program (using branching-time logic CTL).

In [16], model checking tasks are automatically generated for the augmented system that results from each weaving of an aspect. That approach has the disadvantage of having to treat the augmented system, but offers the benefit that needed annotations and set-up need only be prepared once. That work first introduces *verification aspects* intended to accompany a reusable application aspect (and applied in this report to JPF). In [16], such aspects were used in conjunction with the Bandera [9] assertion language to annotate a base or woven system with assertions and predicates prior to model checking. Bandera generates input to model checking tools directly from Java code, and can be extended to, for example, the aspect-oriented AspectJ language.

In [15] a semantic model based on state machines is given, and the treatment of aspects and joinpoints defined in terms

of transitions is described. Categories of aspects are defined that automatically preserve classes of temporal logic properties, or have easier proofs. Some categories are shown to be syntactically identifiable, or to require only dataflow, thus linking syntactic analysis, dataflow, and model checking for aspects.

To our knowledge, besides the CAPE, no other general framework for analysis and verification of aspects exists. In fact, most other combinations of verification tools, even for non-aspect systems, do not have the variety of approaches seen here. Some tools use a combination of different techniques to solve a specific problem. For example, ACL2 [17] uses BDDs to efficiently solve large propositional problems, in addition to usual theorem proving techniques. Similarly, the PVS theorem prover has been extended to use the Yices SMT solver. However, such combinations are tightly integrated into the tools and do not provide an extensible platform.

The CAPE framework has similar goals to the ETI [28] tool integration platform and its continuation jETI. That framework, associated with the STTT journal, provides an Internet-based integration platform for experimentation and support in particular domains. Thus, Bio-JETI [21] provides a framework for biological analysis. ETI/JETI and the CAPE have in common their ideas of combining tools and components in order to produce new tools and to use the combination of appropriate tools to treat difficult tasks in the domain under consideration. The jETI framework is Internet based, and thus provides quite loose coupling, and supports remote execution. The CAPE attempts to provide a uniform Eclipse-based environment, with application and tool-based views, along with guidelines on typical usage and chaining, through examples.

The CAPE can also be seen as an implementation, for aspect oriented software development, of the Evidential Tool Bus, presented as a position paper at a meeting about the Verification Grand Challenge [25]. That paper advocates loose integration and tool chaining as the best way forward to achieve practical analysis and verification.

12 Discussion and future work

We have described the CAPE and illustrated how we may use its tools to reason about aspects, both in woven systems and individually. The case study shows how the CAPE encourages evaluation of existing tools and helps identify remaining areas for improvement. Thus for AIA and AIDA, we saw that further development is needed to make the results accessible for larger systems. The ease of developing new tools is seen in the extension of MAVEN (itself built over the NuSMV model checker) to a tool for checking semantic interference among aspects. This Interference Detector tool is presently under advanced development, and has been significantly expedited by using the CAPE framework to create tool chains and to provide a uniform interface and system organization. Finally, for JPF we have described how aspects themselves can be used to specify and abstract a model before applying the tool. The verification-aspect technique we described reduced the effort and the overhead paid when switching the case study system between a real database and a stub version in memory used for testing, as well as for state reductions.

The case study and additional usage scenarios demonstrate tool chaining and integration among the tools, as well as delineating verification and analysis tasks appropriate for different tools at different stages of system development. However, fully automatic chaining, using the output of one tool as the input of another, has proven impractical for the tools in the CAPE due to the different notations and goals of each tool. Still, the scenarios show valuable strategies that are useful for many applications. These include (1) using fully automatic code analysis tools to identify problem areas, (2) modeling aspects with abstract models and specifications that are shown correct for key protocols or unclear properties (sometimes identified by the automatic tools), and (3) using verifiers directly from code to check delicate implementation issues, such as interference among aspects at shared join points, even when abstract modeling has also been done.

Work on the CAPE is continuing. In the future we expect to add additional tools and to integrate the existing tools more tightly. The CAPE has already proven valuable to us, its developers, in clarifying connections and assumptions about the tools it includes. It should also aid application system developers who use aspects by increasing the accessibility and uniform interface of a wide variety of formal methods tools. The case studies and use scenarios illustrate the diverse ways in which formal methods tools can be used during aspect-oriented software development.

Acknowledgments This research was partially supported by the New York Metropolitan Research Fund at the Technion. The support of AOSD-Europe, an EU Network of Excellence in the 6th Research Framework, is gratefully acknowledged. We also wish to thank Shahar Dag, Eyal Dror, Wilke Havinga, Yael Kalachman, Emilia Katz, Ha Nguyen, Tom Staijen, and Nathan Weston for their help in developing the CAPE and its tools.

References

1. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an extensible aspectj compiler. Trans. Aspect-Oriented Softw. Dev. **1**, 293–334 (2006). LNCS 3880
2. Bergmans, L., Aksit, M.: Composing crosscutting concerns using composition filters. CACM **44**, 51–57 (2001)
3. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: an OpenSource tool for symbolic model checking. In: Proceedings

of International Conference on Computer-Aided Verification (CAV 2002). LNCS, vol. 2404, Copenhagen, Denmark. Springer, July 2002
4. Clarke, E.M. Jr., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
5. Dror, E., Katz, E., Katz, S., Staijen, T.: The revised architecture of the cape. Technical report, AOSD Europe, August 2006
6. Filman, R., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley, New York (2005)
7. Goldman, M., Katz, E., Katz, S.: Maven: modular aspect verification and interference analysis. Form. Methods Syst. Des. **37**, 61–92 (2010)
8. Goldman, M., Katz, S.: Maven: modular aspect verification. In: Proceedings of 13th TACAS 2007. LNCS, vol. 4424, pp. 308–322. Springer, New York (2007)
9. Hatcliff, J., Dwyer, M.: Using the Bandera Tool Set to model-check properties of concurrent Java software. In: Larsen, K.G., Nielsen, M. (eds.) Proc. 12th Int. Conf. on Concurrency Theory, CONCUR'01. LNCS, vol. 2154, pp. 39–58. Springer, New York (2001)
10. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. Int. J. Softw. Tools Technol. Transf. **2**(4) (2000)
11. Havinga, W., Nagy, I., Bergmans, L., Aksit, M.: A graph-based approach to modeling and detecting composition conflicts related to introductions. In: AOSD '07, pp. 85–95. ACM Press, New York (2007)
12. Katz, E., Katz, S.: Verifying scenario-based aspect specifications. In: Proceedings of Formal Methods: International Symposium of Formal Methods Europe (FM05). LNCS, vol. 3582, pp. 432–447. Springer, New York (2005)
13. Katz, E., Katz, S.: Incremental analysis of interference among aspects. In: Proceedings of Foundations of Aspect Languages Workshop (FOAL08) (2008)
14. Katz, E., Katz, S.: User queries for specification refinement treating shared aspect join points. In: Proceedings of International Conference on Software Engineering and Formal Methods (SEFM) (2010)
15. Katz, S.: Aspect categories and classes of temporal properties. Trans. Aspect-Oriented Softw. Dev. **1**, 106–134 (2006). LNCS 3880
16. Katz, S., Sihman, M.: Aspect validation using model checking. In: Proceedings of International Symposium on Verification. LNCS, vol. 2772, pp. 389–411 (2003)
17. Kaufmann, M., Strother Moore, J., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer, Norwell (2000)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten M., Palm J., Griswold, W.G.: An overview of AspectJ. In: Proceedings ECOOP. LNCS, vol. 2072, pp. 327–353 (2001)
19. Krishnamurthi, S., Fisler, K.: Foundations of incremental aspect model-checking. ACM Trans. Softw. Eng. Methodol. **16**, Article 7 (2007)
20. Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying aspect advice modularly. In: Proceedings of SIGSOFT Conference on Foundations of Software Engineering, FSE'04, pp. 137–146. ACM (2004)
21. Margaria, T., Kubczak, C., Steffen, B.: Bio-jeti: a sevice integration, design, and provisioning platform for orchestratesd bioinformatics processes. BMC Bioinformatics **9**(S-4) (2008)
22. McMillan, K.L.: Getting Started With SMV. Cadence Labs, March 1999
23. Nguyen, H., Sudholt, M.: Aspects over vpa-based protocols. In: Proc. Intl. Conf. Software Eng. and Formal Methods (SEFM). Computer Science Press (2006)
24. Rensink, A.: The groove simulator: a tool for state space generation. In: AGTIVE 2003. LNCS, vol. 3062, pp. 479–485 (2003)
25. Rushby, J.: An evidential tool bus. In: Verification Grand Challenge Workshop, Jan 2006
26. Soares, S., Borba, P., Laureano, E.: Distribution and persistence as aspects. Software: Practice and Experience, Jan 2006
27. Staijen, T., Rensink, A.: A graph-transformation-based semantics for analysing aspect interference. In: Workshop on Graph Computation Models, Jan 2006
28. Steffen, B., Margaria, T., Braun, V.: The electronic tool integration platform: concepts and design. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 9–30 (1997)
29. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)
30. Weston, N., Taiani, F., Rashid, A.: Interaction analysis for fault-tolerance in aspect-oriented programming. In: Workshop on Methods, Models and Tools for Fault Tolerance (2008)