

Master Thesis
Resource Operations Model in a
Dynamic World

Software Engineering
Faculty of Electrical Engineering,
Mathematics and Computer Science

University of Twente

J.W. Koelewijn
Computer Science
Software Engineering
0007838
Witbreuksweg 379-007
7522 ZA
Enschede

December 31, 2009

RESOURCE OPERATIONS MODELS HAVE OFTEN BEEN USED AS AN ABSTRACTION OF SYSTEMS TO PERFORM BEHAVIOURAL PATTERN DETECTION. TRADITIONALLY, FOR THE ACTUAL PATTERN DETECTION AND CONSTRAINT SPECIFICATION, THESE SYSTEMS USE REGULAR EXPRESSIONS.

HOWEVER, REGULAR EXPRESSIONS SUFFER FROM POOR EVOLVABILITY, BECAUSE THE ALPHABET MUST BE KNOWN. THEREFORE IN THIS THESIS, TWO ALTERNATIVES ARE CONSIDERED: GRAPH TRANSFORMATION SYSTEMS AND THE VIBES SPECIFICATION LANGUAGE. BECAUSE CONSTRAINT VERIFICATION METHODS ARE ALMOST EXCLUSIVELY USED IN COMBINATION WITH STATICALLY TYPED LANGUAGES, WE INVESTIGATED THE ISSUES IN PERFORMING CONSTRAINT VERIFICATION ON DYNAMICALLY TYPED LANGUAGES. TRADITIONAL VERIFICATION SYSTEMS USE THE TYPE INFORMATION AVAILABLE IN THE ABSTRACT SYNTAX TREE (AST), BUT THIS TYPE INFORMATION IS MOSTLY NOT AVAILABLE IN THE AST OF DYNAMICALLY TYPED LANGUAGES, UNLESS COMPLEX TYPE INFERENCE ALGORITHMS ARE USED.

ALTHOUGH TYPE INFERENCE HELPS IN SOME CASES, MOST OF THE TIME ONLY A SET OF TYPES CAN BE INFERRED FOR VARIABLES, RATHER THAN AN EXACT TYPE. IT HAS APPEARED THAT THIS MAKES IT NECESSARY TO PERFORM CONSTRAINT VERIFICATION AT RUNTIME. THE SOLUTIONS THAT ARE PROPOSED IN THIS THESIS HAVE BEEN IMPLEMENTED AND TESTED IN THE DYNAMICALLY TYPED LANGUAGE SMALLTALK.

Acknowledgements

This thesis marks the end of an era for me. An era that begun when I first set foot on campus and in which I have learned very much, both professional and personal. I call it an era, because it seemed to last forever. But as with all good things, it comes to an end. Before I begin with this thesis, I want to take the possibility to show my gratitude to all people that have stood me by and helped in my professional and personal development.

First and foremost I would like to thank my family, who never failed to believe in me, even in times this thesis seemed to have no end. During my whole life as a student they have believed in me, and the only times they asked how long it would take until it was finished, was because they were genuinely interested in me.

I would also like to thank my girlfriend, who always believed in me, even in times I felt helpless and lost. She helped me find motivation and helped me relax and for this I am very thankful. Without you the last year would not have been so pleasant.

My friends and colleagues have also helped me find my way, personally and professionally, something for which I am very grateful.

Last but not least I would like to show my gratitude to my supervisors. Without the countless discussions and help with my text I would not been able to write this thesis.

To all the people I may have forgot to mention, but who undoubtedly helped me in some way this era, I would like to say thank you as well, without all of you I would not be where I am standing now.

Table of Contents

1	Introduction	1
1.1	Resource Operations Model	2
1.2	Aspectual conflict detection	5
1.3	Regular expressions	7
2	Problem Statement	10
3	Specification of behavioural patterns	14
3.1	Difference between specification and underlying model	14
3.2	Graph based	14
3.2.1	Introduction	14
3.2.2	Experiments	19
3.2.3	Evaluation	21
3.3	Vibes based	22
3.3.1	Introduction	22
3.3.2	Evaluation	26
3.4	Summary	28
4	Associating actions with behavioural patterns	29
4.1	Specifying actions in constraints	30
4.2	Actions and AOP	32
4.3	Multiple simultaneous actions	32
4.4	Evaluation	34
5	Detecting behavioural patterns in dynamically typed languages	36
5.1	Verification of static and dynamically typed languages	36
5.2	Keeping track of patterns	37
5.3	Resource designation	40
5.4	Tying it together	41
5.5	Multiple constraints	44
5.6	Summary	46
6	Implementation in Smalltalk	49
6.1	Why Smalltalk?	49
6.2	Deterministic Abstract Recognisers	49
6.3	Meta information	51
6.4	Constraints	52
6.5	Ordering constraints	53
6.6	SuperVisor	54
6.7	Message interception	55
6.8	Working out the example	57
6.8.1	Creating the constraints	57
6.8.2	Creating the annotations	62

6.8.3	Using the SuperVisor to tie it all together	62
6.8.4	Ordering the actions	66
6.9	Conclusions	67
7	Conclusions	69
7.1	Overcoming poor evolvability	69
7.2	Behavioural joinpoints	70
7.3	Verifying dynamically typed systems	71
7.3.1	Keeping track of patterns	71
7.3.2	Designating resources	71
7.3.3	Coupling patterns to constraints	71
7.4	Implementation	72
7.5	Lessons learned	72
7.6	Future work	72
7.7	Personal evaluation	73
7.8	Summary	75
A	CheckDesign	78
B	API	85
B.1	DAR	85
B.2	Annotations	88
B.3	Constraints	89
B.4	SuperVisor	90

Chapter 1

Introduction

A Resource Operations Model is a model in which the world is abstracted into resources and the operations that can be performed on them. On such a model constraints can be defined, which constrain the behaviour of the resource in terms of its operations. Using a Resource Operations Model to model a system can thus be seen as creating a behavioural model of the system.

This chapter gives background information about Resource Operations Models and how they have been used in the past and explains regular expressions and how they are used.

Chapter 2 gives the problem statement, which is broken down into three problems, that will be addressed in this thesis.

In Chapter 3 we give a short recapitulation of the shortcomings of regular expressions with respect to constraint specifications, especially on how susceptible regular expressions are to changing alphabets, and we give an evaluation of two alternative approaches, Graph Transformation Systems and Vibes. The two alternatives are evaluated on how they compare to each other and to regular expressions with respect to behavioural constraint specifications for resource operations models, and how they could be used for constraint verification for dynamically typed languages.

In Chapter 4 a novel way to further use information contained in behavioural constraint specifications is given. This chapter proposes to use the state information contained in constraints to add functionality to the system under verification. The chapter does not only propose a way to use this information, but will also discuss several implications that may be introduced, such as the addition of functionality from multiple constraints at the same time.

In Chapter 5 a method to perform behavioural constraint verification with respect to resource operations models on dynamically typed languages is introduced. Dynamically typed language cannot be verified using traditional (compile-time) verification techniques, because these techniques rely heavily on information in the Abstract Syntax Tree (AST) that is not available at compile-time in dynamically typed languages. This chapter proposes a solution to this problem and proposes methods to keep track of behavioural patterns and to designate pieces of code as resources. After this it combines the previous findings in a solution with which it is possible to perform behavioural constraint verification on resource operations models of dynamically typed languages.

Chapter 6 gives an overview of a reference implementation of this verification system, and will discuss problems found during implementation and how they have been solved.

In Chapter 7.7 we reflect on the process of writing this research and will discuss the lessons that were learned during it.

We conclude in Chapter 7, where we give an overview of the problems that were

solved in this thesis and how they have been solved.

1.1 Resource Operations Model

A Resource Operations Model is a model in which the world is abstracted in resources and the operations that can be performed on these resources. This abstraction can be compared with the concept of Object Orientation. The concept of Object Orientation was introduced to bring a higher level of abstraction to programming paradigms. This abstraction was introduced during a time that programming languages were still procedural, and it was hard to model the world using such a paradigm. Therefore the Object Orientation model was introduced, in which real world things were represented as objects. An object is the data that makes up this real world concept, and the operations that can be performed on this data. This way, a real world concept was encapsulated with its functionalities in one unit.

A Resource Operations Model can be compared to Object Orientation in that real world concepts are modelled as Resources, and the Operations that can be performed on these resources. However, Resource Operation Models are not used for programming behaviour in the first place, but rather to provide a tool to *reason about the behaviour* of a program. The Resource Operations Model is used to detect certain behavioural patterns in programs, rather than being used to specify the behaviour of programs.

Because Resource Operations Models are used to describe the behaviour of programs, their elements can be seen as abstractions of programming concepts. This means that the resources in the Resource Operations Model are abstract representations of concepts and that the operations in the Resource Operations Model can be seen as abstract representation of the methods of these concepts. There is one caveat though: A Resource could represent multiple concepts at once and an Operation could represent multiple methods at once. This is where the difference in abstraction becomes visible.

Consider a program in which multiple abstract data types are being used which have been created over time. Because multiple teams have worked on the project, naming of the abstract data types has not followed a clear and distinct standard, and some abstract types have been created multiple times. For instance, both teams have implemented a logical queue using for instance arrays as the underlying concrete data type (Buffer and Queue), and the operations of this logical queue can be abstracted with *add* and *remove* operations. Both implementations represent the same concept of a logical queue, but because the teams did not cooperate well, they allowed the two separate artefacts to be created independently of each other. Although both implementations share the same abstract operations, it is not possible to test whether their behaviour is the same.

To test their behaviour we introduce the Resource Operations Model and we define both Buffer and Queue to be a Queue-Resource and we specify that the *store* method of the Queue object and the *put* method of the Buffer object are abstracted in the *put* operation of the Queue-Resource. Now we have abstracted both objects into one Resource and we have defined the operations of this Resource, and we can test whether the behaviour is as expected. This example shows that a resource can be used to abstract *part* of the behaviour of a more complex concept.

Once a Resource Operations Model has been defined, constraints can be specified on the model. A constraint is defined in the New Oxford American Dictionary as a limitation or a restriction. In this context it means that a constraint is used to limit or restrict the behaviour of a program. In the Resource Operations Model, a constraint is used to test if the program behaves within the limitations of the constraints.

If we look back at the example of the Queue-Resource, we can specify a constraint that we cannot use the *put* operation if the Queue-Resource is full, i.e. has reached

its capacity. Another constraint can be specified stating that it is not possible to get something from an empty Queue-Resource. Of course, to fully specify these resources we have to define operations for getting something from a Queue-Resource, but this is not of importance now. What is important is that the Resource Operations Model is an abstraction of (part of) the behaviour of a program and that constraints can be specified on this behaviour. The behaviour of a program can be tested using the Resource Operations Model and the constraints.

Because the Resource Operations Model is an abstraction of the behaviour of a program, it is very useful for creating a common ground for behavioural conflicts. For instance, a constraint could be specified that it is not possible to retrieve an item from an empty collection. This constraint is an abstraction of the previous example, but it can be expressed using the Resource Operations Model as well. Instead of abstracting the Queue and the Buffer Objects as Queue-Resource, they could have been abstracted in a Collection-Resource. Other objects of the collection data type could then be abstracted as Collection-Resources as well, and only one constraint has to be used to test the behaviour of all these objects. However, this does not necessarily mean that the Buffer and Queue objects are now only abstracted into Collection-Resources, the model also allows for multiple abstractions of the same object. This means that a Queue can be abstracted into a Queue-Resource and into a Collection-Resource at the same time. This makes the Resource Operations Model a very powerful and expressive model, that can be used as an abstraction of (part of) the behaviour of Object Oriented systems. Note however, that the Resource Operations Model is not limited to Object Oriented systems, because it can be used in procedural systems as well, for instance to constrain the usage of variables. In this example the variables are abstracted in to resources, and reading and writing a variable can be abstracted into *read* and *write* operations.

The Resource Operations Model is an abstraction of the behaviour of a program and as such can be used to model the behaviour and to test if a system is compliant to a given set of behavioural constraints.

Another example in which the usage of the Resource Operations Model is illustrated is the analysis of a bank account. A bank account can be accessed by multiple systems at once. One of them can be an ATM where you are withdrawing money and another system is a bank back-end system transferring money to the account. If these two actions would take place at different moments in time and there would be no overlap, there is no problem. You withdraw money and some time later money is transferred to your account.

To see what problems can arise in this scenario, we first identify the bank account as a shared resource. An account can be seen as a shared resource, because the account is shared by both the ATM and the bank back-end system. Now we identify the operations that can be performed on the resource. The most obvious operations are withdrawing money and transferring money to it, but actually these operations consist of sub-operations. The basic operations are reading the amount of money, performing an arithmetic operation on this amount (addition or subtraction) and writing a new amount. This means that the operations that are performed consist of reading and writing the amount of money in the bank account.

The case in which withdrawal and transfer of funds to the account occur in different, non-overlapping moments in time can be considered as correct behaviour. The bank account holds the correct balance, and everything is in order. A schematic representation in the form of a sequence diagram is given in Figure 1.1.

Now we will focus on a different case, in which the withdrawal and transfer overlap. As will become clear, this can result in a balance that is not correct. Remember that the withdrawal and transfer processes are both made up of the operations reading the balance and eventually writing the new balance. What if one process reads the

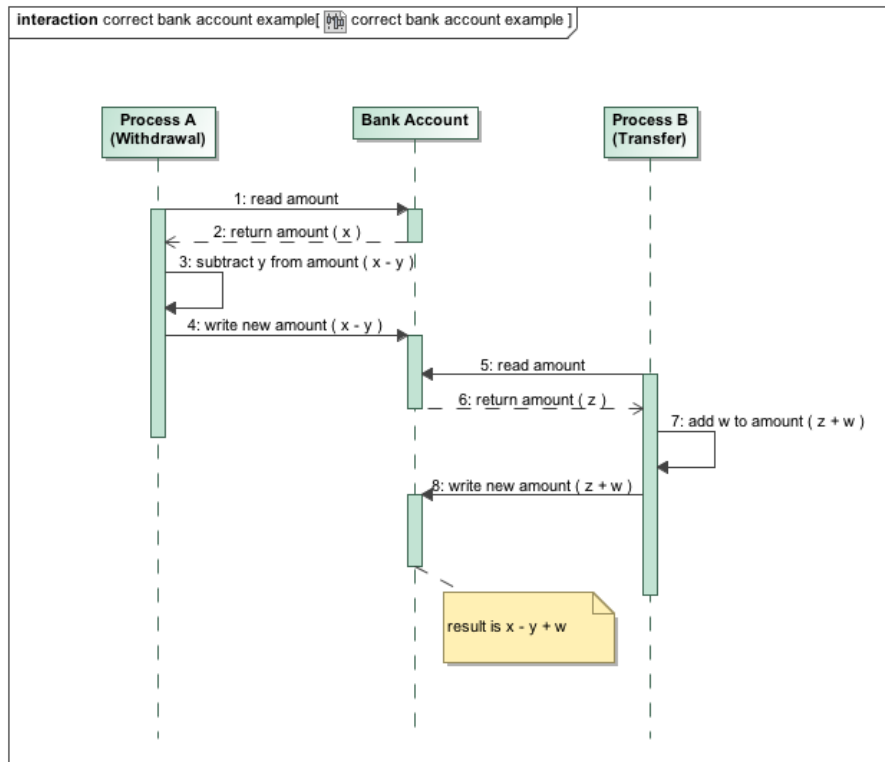


Figure 1.1: Representation of correct behaviour of processes A and B

balance and the other process reads the balance as well before the new balance is written? In this case, both processes have the same balance on which they perform their arithmetics, and from now on it does not matter in what order the rest of the operations are performed, because the balance will be stored wrong anyway. This wrong sequence is illustrated in the sequence diagram in Figure 1.2.

In this diagram we see that Process A (withdrawal) reads the amount of money in the bank account. Directly after this, Process B (transferring) reads the amount as well. Now both processes hold the same amount. Now Process A subtracts an amount and writes the result back to the amount. After this, Process B adds an amount and writes this amount to the account. This means that in the end only the transfer of money is recorded in the balance and that the result of the withdrawal is not recorded in the balance. In this case this would not be a big problem for the account holder, but imagine an ordering in which only the withdrawal is recorded...

This situation can be easily detected if a constraint would be used that specifies that a write operation should always immediately be preceded by a read operation. This simple constraint would be enough to detect that the second situation (Figure 1.2) would display unwanted behaviour.

In the previous example we implicitly introduced the Resource Operations Model, i.e. we identified the bank account as a resource and reading and writing the balance as operations. By using this resource and operations based approach it is possible to identify behavioural problems that would otherwise be hard to identify.

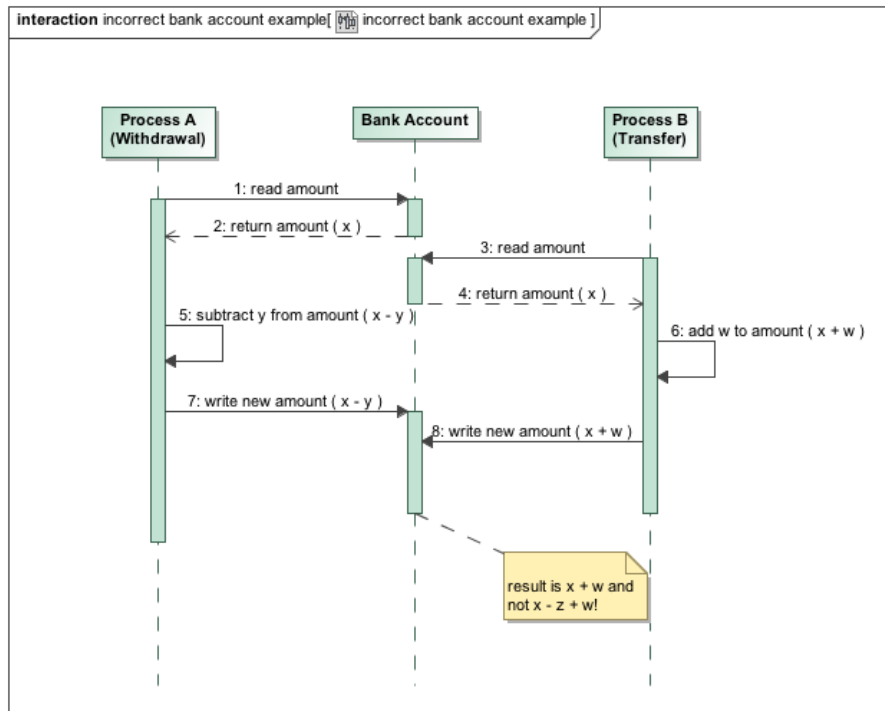


Figure 1.2: Representation of incorrect behaviour of processes A and B

1.2 Aspectual conflict detection

The Resource Operations Model has been used in the context of Aspectual Conflict Detections as well. Before we can explain how the Resource Operations Model is used, Aspect Oriented Programming is explained and we explain what Aspectual Conflicts are.

Although the adoption of the Object Oriented programming paradigm did not take off to a great start when its first concepts were introduced with first Simula 67 [17] and later the first Object Oriented programming language Smalltalk [20], since the introduction of graphical user interfaces in the mid 1990s, object orientation has become one of the most widely used programming paradigms. This started with the introduction of C++ and accelerated with the introduction of Sun's Java [13] and Microsoft's .NET Platform [33]. With the introduction of the object oriented programming paradigm, focus shifted from modules with separated data and behaviour, to self-contained units which encapsulated both the data and the behaviour. These units are known as classes and instances of these classes are called objects.

This new notion brought a new layer of abstraction to programming languages, and it enabled programmers to model in a more data-centric manner rather than a procedure-centric manner. A table is an object on which things can be placed and is made up of a surface object and leg objects, which all have their own functionalities.

The introduction of object orientation also brought new modelling techniques, such as UML which is now widely used for designs of systems. However, there still remain parts of systems that are logically one concept, but which are impossible to implement in one place. An example of this is the well known Observer pattern [11], which can be used by objects to notify other objects that might be interested of changes. In a design

document it is often enough to mention that the Observer pattern is used, and other developers know what is meant with this. The design is clear, because the Observer pattern is in there in a self contained area in the diagram. In a perfect world, the same could be said for the code. One would expect that the code for the Observer pattern would be in a self contained set of classes as well, but this is not the case. Because of the nature of the pattern, the concept is used by many classes, and method calls are made from all these classes as well. This means that in order to implement the Observer pattern, lines of code have to be added to all classes that want to exhibit this observable behaviour.

The Observer pattern is not the only example in which object oriented programming does not seem to be able to deliver a good solution in which the functionality is contained in a limited set of classes. Other examples are for instance logging, tracing, auditing and security related concerns. Because these concerns have the tendency to cut through the interfaces of many other classes, they are called *crosscutting concerns*.

These problems could be solved by the concept of *meta objects* that was introduced by Pattie Maes in her paper Concepts and Experiments in Computational Reflection [24] and which was later extended in the book The Art of the Metaobject Protocol of Kiczales *et al.* [23]. Metaobjects are the objects that make up classes, and by modifying the objects that make up classes, one can change the behaviour of these classes. Although the concept is very expressive and useful, its concepts are hard to understand and it did not bring the abstractions needed to solve the problems of crosscutting concerns in a modular fashion.

Later these ideas were used to create a new paradigm specifically created to address crosscutting concerns. This new paradigm is called Aspect Oriented Programming [22]. In this paradigm, a piece of code can be executed on pre-specified moments during the execution of the code. These pre-specified moments are called joinpoints. A joinpoint is a point in the execution of a program at which new code can be inserted. Different implementations of Aspect Oriented Programming allow the usage of different joinpoints, but the commonly used joinpoints are method calls, method executions and field access.

The code that should be executed and the specification of the joinpoints where this should happen is called an aspect. An aspect is made up of the following parts:

- A name that is given to the aspect
- A pointcut expression selecting joinpoints on which the aspect should be applied
- Advice, the code that should run when the joinpoints that are selected by the pointcuts are encountered during the execution
- Methods that can be used in the advice
- Variables that can be used in the advice and (possibly)
- A Super aspect from which functionality and pointcuts can be inherited.

The process in which the advice is applied to the base code (i.e. non aspectual code) is called weaving. This can happen in different ways: the advice can be inserted into the original source code by a preprocessor or it can be inserted into the compiled code. When a virtual machine is used, there are other possibilities. One of them is weaving the advice when the original classes are loaded into the virtual machine, or to insert the advice into the bytecode. Another method of applying advice is by using so called Proxy classes. Because the term weaving can mean different things, we will use the term superimposition in this thesis.

In light of the example of the Observer pattern, using AOP enables a developer to create an aspect which has the Observer pattern specific code specified as advice and

which holds pointcut expressions to select those joinpoints at which the state of objects is changed; for instance, all setter methods of a specific collection of objects. This way the functionality provided by the pattern is contained in one unit, called the aspect, and the source is a clear representation of the design in which this functionality was already in a self contained area.

A good overview of Aspect Oriented Programming and an overview of multiple implementations can be found in [10]. Well known AOP implementations are AspectJ [1] and Compose*[2].

We have seen that Aspect Oriented Programming provides a method to modularise otherwise crosscutting code. This brings great power to developers, but with it new problems can be introduced as well.

Imagine for instance two aspects, one that is used to encrypt data that is written to a database and another aspect that is used to log all database access. Now consider that these two aspects are defined on the same joinpoints as well. Whenever two aspects are superimposed at the same joinpoint, there is the possibility that unwanted behaviour emerges. The problem is that this is not always obvious. Both aspects can work perfectly well when superimposed separately on the base code, but could introduce errors when superimposed together. This, for instance, can be the case when both aspects change the same fields in the base code.

Whenever such unwanted behaviour occurs, it is called an *aspectual conflict*. Aspectual conflicts happen when two aspects, when superimposed together, introduce unwanted and unanticipated behaviour.

Aspectual conflicts can be of two kinds: the first is structural and the second is behavioural. Structural conflicts arise when for instance two aspects introduce methods or fields with the same name, or structural conflicts can be introduced by changing the inheritance tree or introduce conflicting annotations. Behavioural conflicts arise when two aspects together change the behaviour of the program. For both kinds of problems research has been done, in [15] for instance, graph transformation systems are used to detect structural conflicts and in [29] a classification system for aspects is introduced to detect whether aspects interfere with the behaviour of the base program. In [29], Rinard *et al.* introduce the concept of scopes, with which they try to capture the behaviour of an aspect or method with respect to fields in the base code. The behaviour is being captured as being either read behaviour or write behaviour, and although it is not mentioned in the work itself, the Resource Operations Model is used in a very basic form.

In Durr [9] the Resource Operations Model is used to reason about behavioural conflicts between aspects. In this work the use of the Resource Operations Model is and is used in a wider sense than the basic, implicit use in Rinard. Where the work of Rinard *et al.* only focuses on classifying aspects, the work of Durr focusses on the detection of aspectual conflicts. A conflict is defined in either a constraint or in assertions, which both are given in the form of regular expressions. Before the aspects are superimposed on the base code, all possible orderings in which the aspects can be superimposed are generated, and traces of possible sequences of operations on resources are created. Then these sequences are tested to see if they match the regular expression that was used to specify the constraint (or do not match in the case of assertions).

1.3 Regular expressions

Regular expressions are widely used to detect certain patterns. Regular expressions are widely used in word processors and text editors, but nowadays they have been incorporated into programming languages as well. As stated, regular expressions are

used to specify certain patterns which can then be identified in a string of text. Before a regular expression can be used, however, the alphabet has to be known. The alphabet of a regular expression is defined as the collection of the characters, or tokens, that cannot be broken down into smaller constituents. A regular expression is then specified using the symbols from the alphabet. In other words, the alphabet defines the tokens that can be recognised by the regular expression. Because the alphabet of the expression is specified by the user, this means that regular expressions can be used for other purposes than identifying text patterns. For instance, regular expressions can be used to detect patterns in certain abstractions.

In [31] a regular expression over an alphabet Σ is defined recursively as follows:

i) Basis: λ and a , for every $a \in \Sigma$, are regular expressions over Σ

ii) Recursive step: Let u and v be regular expressions over Σ . The expressions

$(u \cup v)$

(uv)

(u^*)

are regular expressions over Σ

iii) Closure: u is a regular expression over Σ only if it can be obtained from the basis elements by a finite number of applications of the recursive step.

In this recursive definition, λ denotes the null element and Σ the alphabet of the regular expression. The regular expression $(u \cup v)$ is the union of u and v , (uv) is the concatenation and (u^*) denotes the Kleene star operation, meaning zero or more occurrences of u . Sometimes u^+ is used as a shorthand for uu^* .

An example, that can be used with the bank account example of section 1.1, is the following regular expression:

(read write)

Although it is very simple, this can be read as: **write** is preceded by **read**. However, this expression only accepts the sequence **read write**, and what we want to achieve is a regular expression that accepts more than only that sequence. To do this, we add the Kleene star operation, so that the sequence **read write** can occur zero or more times:

(read write)*

This regular expression accepts the sequences:

$\emptyset$

read write

read write read write

read write read write read write

read write read write read write read write

...

Regular expressions can be a very powerful tool to detect sequences and to see if a certain sequence adheres to a given pattern. That is why they are often used to specify constraints. The given example for instance can be interpreted as that a **write** operation can only occur if a **read** operation was performed immediately before that and that a **read** operation can only be followed by a **write** operation and cannot occur on its own. If this is not the case, it is an indication that the actual behaviour is not aligned with the required behaviour.

Another example where regular expressions are being used for constraints is the work of Durr [9], in which regular expressions are used to specify constraints and assertions. If the behaviour (represented in a Resource Operations Model) cannot be captured by the regular expression of a constraints, a warning is generated and displayed to the developer.

To test if a given sequence adheres to a regular expression, a deterministic finite automaton is created that accepts the regular expression.

Definition 1.3.1 A *deterministic finite automaton (DFA)* is a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ a finite set called the alphabet, $q_0 \in \Sigma$ a distinguished state known as the start state, $F \subset Q$ called the final or accepting states, and δ a total function from $Q \times \Sigma$ to Q known as the transition function [31].

A DFA can be represented graphically by using a special graph, consisting of nodes and edges. A state is represented by a circle with a label, the start state is indicated by a circle with an arrow head attached to it, and a final state is represented by two circles drawn inside each other. The edges have labels over them and the labels come from the alphabet. For every $\delta(q_i, a) = q_j$ from the transition function ($q_i, q_j \in Q$ and $a \in \Sigma$) there is an edge from the node with label q_i to the node q_j with label a .

For instance, the regular expression $(\text{read write})^*$ can be represented by the DFA in figure 1.3.

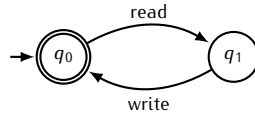


Figure 1.3: DFA of the $(\text{read write})^*$ regular expression

The DFA starts in q_0 , indicated by the arrowhead attached to it, and q_0 is an final (or accepting) state as well, which is indicated by the double circles. If a **read** symbol is encountered, a transition is made to q_1 , which is not a final state. This means that the sequence **read** itself is not accepted by the regular expression. Only if a **write** symbol is encountered another transition is made to q_0 , which is an accepting state and thus the sequence **read write** is accepted.

Although regular expressions are very powerful and are widely used to specify and detect patterns, they do have certain shortcomings as will become clear in the next chapter.

Chapter 2

Problem Statement

In the previous chapter regular expressions were introduced, but although they are often used, they do have certain shortcomings. We have seen that a regular expression is expressed using a certain alphabet Σ . This alphabet contains all symbols that are accepted by the corresponding regular automaton. In some cases, the fact that this alphabet Σ is predetermined, causes a problem. The problem is not apparent when a certain regular expression is used once, but it emerges over time. This becomes apparent when a regular expression is used to detect patterns with evolving alphabets. When the alphabet does not evolve, there is no problem. However, when the alphabet changes, it means that symbols are added or removed from the alphabet. When such a change in the alphabet occurs, it could mean that the regular expressions should be changed as well. This is especially the case when they are used in a context such as in [9]. In this work regular expressions are used to specify constraints and assertions on operations that are performed on certain resources. But what if the set of operations changes, for instance, if a certain operation is added? In this case the alphabet of the regular expression changes and because of this the constraints should change as well.

The need to change specifications if the alphabet changes places an extra burden on the developers of the code. The developers always have to keep the specifications in the back of their head and should always think about the constraints when they update the code. Because when they update the code, it could be that they add or remove symbols from the alphabet. When the alphabet changes, the regular expressions used to create the specification could change as well, otherwise constraints that are not violated, could be violated using the same specifications.

To illustrate this, consider the following example. In Figure 2.1 a regular automaton of a constraint is given. This constraint is specified on a buffer which only has the following methods:

$$\{put, get, isEmpty\}$$

This constraint states that something can only be retrieved from a buffer if it is not empty. To enforce this, a call to the method *isEmpty* should be made before the *get* method is invoked. Only if this is done the automaton accepts the input. If the *get* method is invoked without invoking the *isEmpty* method, the automaton makes a transition to state q_2 , which is not a final state and from which it is not possible to reach a final state. Such a non-final state from which it is not possible to reach a final state is called a *trap state*. Trap states are used to indicate failures of the modelled system.

Now consider that the interface of the buffer changes, and that a method *sortBuffer* is added. This does not change the semantics of the constraint, because it still means that the *isEmpty* method should be invoked before the *get* method is invoked. So our intuition tells us that the constraint does not need any adaptations. This is not the

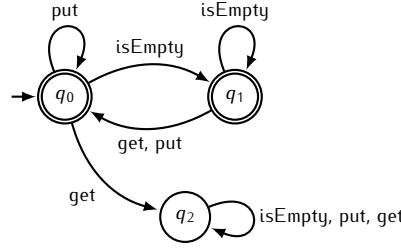


Figure 2.1: Buffer constraint expressed with an regular automaton.

case, however. Because the interface changed, the alphabet Σ changed as well, and this in turn means that the constraint should be changed as well to be able to handle calls to the introduced *sortBuffer* as well. In the case of the constraint of the example this means that new transitions should be added that point to the same state as where they originate. An adapted version of the same constraint is shown in Figure 2.2.

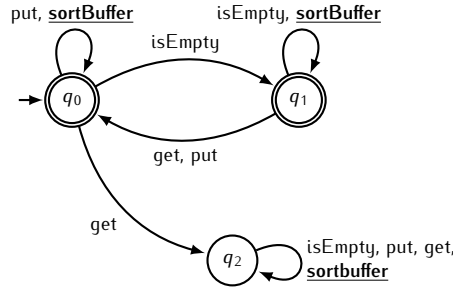


Figure 2.2: Buffer constraint adapted for the added *sortBuffer* operation.

The example shows that regular expressions can suffer from poor evolvability. This poor evolvability can become a serious problem when regular expressions are used to perform verification of constraints, such as in the area of conflict detection or when regular expressions are used to detect behavioural conflicts in code. This thesis will look for a way to overcome this problem of poor evolvability with respect to the Resource Operations Model and will investigate alternatives to regular expressions to specify constraints on resource operations.

The Resource Operations Model has been used in the context of Aspect Oriented Programming (AOP) to identify possible aspectual conflicts. However, the Resource Operations Model can be used for other purposes within the field of AOP, i.e. to specify points in the execution to insert code, so-called joinpoints.

In a keynote, Kiczales (one of the main contributors to AspectJ) made a strong case to improve joinpoint models as they exist today [21]. This because he feels that the current joinpoint models are not completely capable of capturing all events that can occur in the execution of a program. In [6] Cazzola et al. investigate the limitation of current mainstream AOP implementations, and they opt for a more semantic approach for joinpoint selection. They identify that the semantics of the joinpoint selection model can be *property* or *behavioural* based. With property based semantics they mean that joinpoints are based on properties of those joinpoints, for instance, whether the code at the joinpoint does not interfere with a specific property. Behavioural based semantics then deals with what the executed code displays as behaviour.

Because a constraint on a Resource Operations Model captures part of a behavioural pattern, we feel that a constraint can be used to define behavioural join-

points. This thesis therefore investigates how constraints on a Resource Operations Model can be used to define and select behavioural joinpoints at which functionality can be added to existing systems.

The detection of behavioural patterns has until now only been carried out on statically typed languages such as C, C# and Java. In a statically typed language, the type of a variable has to be specified explicitly, or can be inferred by a type inference system automatically. This way, information about variables in the code can be used by the compiler to detect certain type errors. Because information about variable types is available at compile-time, behavioural patterns can be detected before the program is run.

In dynamically typed languages, such as Smalltalk, Ruby or Python, values are typed, rather than variables. Because of this, a variable only has a type when a value is stored in it. Although there exist flow analysis algorithms to narrow the set of types a variable can have at any moment [19], no exact type information is available at compile-time and the exact type information of a variable is only available at runtime. Because the behaviour of a program is often expressed using operations that are performed on variables, and because the set of operations that can be performed on a variable is dependent on the type of the variable, behavioural patterns can only be detected at runtime.

This is the reason that the detection of behavioural patterns has mostly focused on statically typed programming languages (such as for instance the detection of aspectual conflicts in [9]) and dynamically typed languages have been left behind in such investigations. In this thesis we will investigate whether it is possible to detect behavioural patterns in dynamically typed languages, and if so, how to keep track of patterns that have occurred during the execution. We want the solution to be scalable in the sense that whenever the constraint specification grows in complexity, the verification system should not incur performance penalties.

To give an overview of what this thesis addresses, Figure 2.3 is given. This figure gives an overview of an hypothetical system, which is capable of verifying systems based on a Resource Operations Model and constraints on this model.

At the heart of this picture we see the Verification System, which will give a Verdict as output based on the Constraints and the Resources and Operations. We also see that the Constraints can be used to attach Actions. Actions are used to specify functionality that is to be added to the system under verification. The diagram also shows that the Resources and Operations are an abstraction of the system under verification, and that the Constraints are used to constrain the Resources and Operations.

To summarise, this thesis addresses the following problems in terms of Figure 2.3:

1. How can we overcome the poor evolvability of regular expressions, which are used to specify the Constraints?
2. How can the Resource Operations Model be used to specify behavioural joinpoints? In other words, how can Actions be added?
3. How can the verification system for Resource Operations Models be used for dynamically typed languages?

These problems will be addressed in the following chapters.

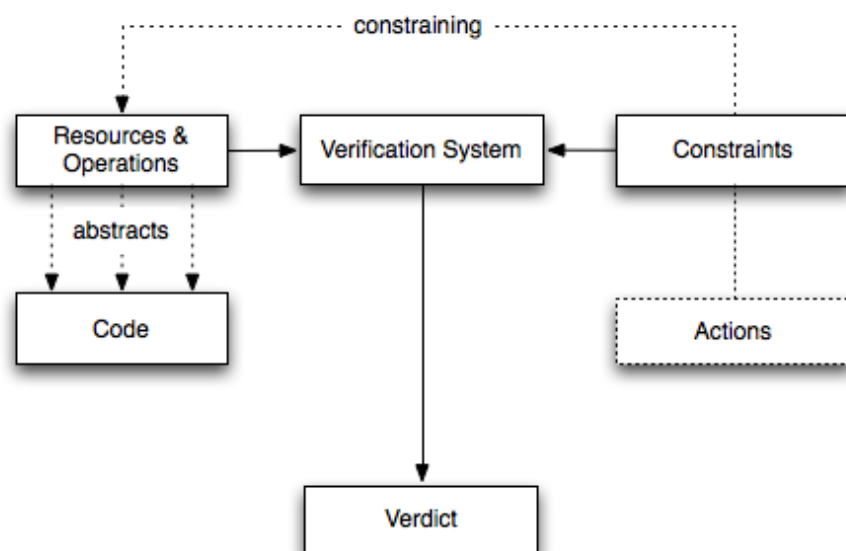


Figure 2.3: Ideal system that can be used to verify constraints on Resource Operations Models.

Chapter 3

Specification of behavioural patterns

This chapter evaluates two alternatives to regular expressions that can be used to specify behavioural constraints. The focus of the investigation will be the usability and evolvability of the alternatives with respect to Resource Operations Models. The chapter begins with an overview of the differences between a specification and its underlying model and continues with describing graph transformation systems and the Vibes specification language as alternatives to regular expressions to specify constraints and verifying whether these constraints are not violated. The chapter concludes with an evaluation of the two alternatives.

3.1 Difference between specification and underlying model

Before we start the discussion of limitations that a specific specification method may have, it is important to distinguish between the specification and the underlying model that is used to verify whether a certain specification holds. It is often possible to get from a specification to its model through a form of translation, but to give a thorough overview of the problems a specification technique may have, it is important not to overlook its underlying model. The specification in itself is a certain language in which properties that should hold or should not hold can be expressed. This language can be a very formal looking language (such as for instance Z [18]), or a language that completely exists of graphical symbols, such as for instance the language of flow diagrams. The underlying model is an abstraction of the system against which the specification is checked and often there are multiple ways in which the underlying model can be described (e.g. the formal model of UML). The underlying model is therefore of great importance in discussing specification techniques, because often problems of a specification technique have their origins in the model.

3.2 Graph based

3.2.1 Introduction

A graph is a widely used mathematical formalism, which basically consists of vertices and edges between them. The vertices are used to represent entities and the edges between them are commonly used to represent relations between those entities. Graphs are widely used as an abstraction of real world problems, they are for instance used in satellite navigation systems to calculate the shortest route between two cities.

The abstraction used there is that the cities are represented by vertices and the roads between the cities as edges. One of the reasons why graphs are so commonly used, is because of the mathematical properties of graphs. Many algorithms exist for graphs, such as for the calculation of the shortest path between two points, or for the calculation of a cycle that visits each vertex.

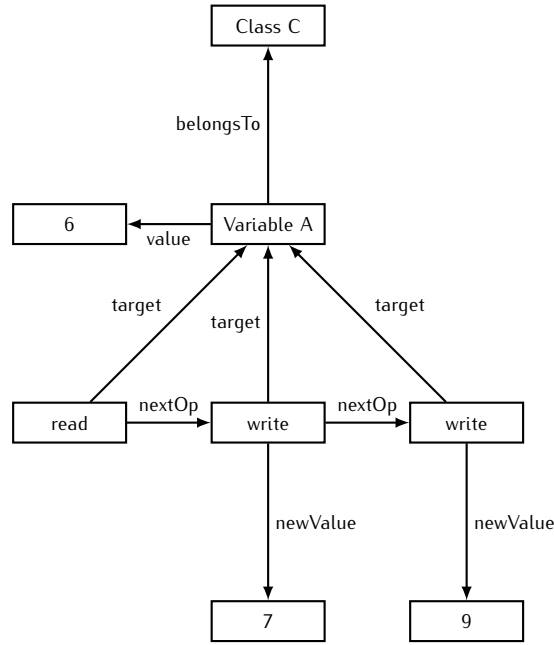


Figure 3.1: Graph representing operations performed on a variable.

An example of a graph is given in Figure 3.1. In this figure the rectangles with *Class C*, *Variable A*, *read*, *write*, *6*, *7* and *9* are vertices and the arrows between them represent relationships between them. The name of the relationship is given as a label next to the arrows. The example displays one of the ways graphs can be used to model the usage of a variable by a program. The variable is depicted as a node in the graph and as can be seen, Variable A belongs to Class A and has an initial value of 6. The first operation that is performed on the variable is a *read* operation. This can be seen because the *read* node has a *target* edge with Variable A as the target node. After the *read* operation, the next operations are two *write* operations, having *newValue*'s of respectively 7 and 9.

The graph given in Figure 3.1 is one way to model the behaviour of a program as a Resource Operations Model. Although this graph does not give a complete model, it already can be used to detect some behavioural patterns. However, before we explain how patterns can be specified using graphs and how they can be detected, we have to explain the concept of meta models.

To formally create a model, a so called *meta model* is needed. Generally, a meta model defines the language and processes from which to form a model. The meta model states what elements can occur in a model and what relations may exist between those elements. For instance, the meta model of the graph in Figure 3.1 is given in Figure 3.2.

This meta model expresses what relations can exist between different type of nodes in a graph. For instance, it makes clear that a Variable has a value and belongs to a class. It is also shown that *read* and *write* are both specialisations of *Operation*.

A graph can be used to model real world scenarios, however, in itself it cannot be used to detect the occurrence of patterns. For this, a method to specify the pattern is

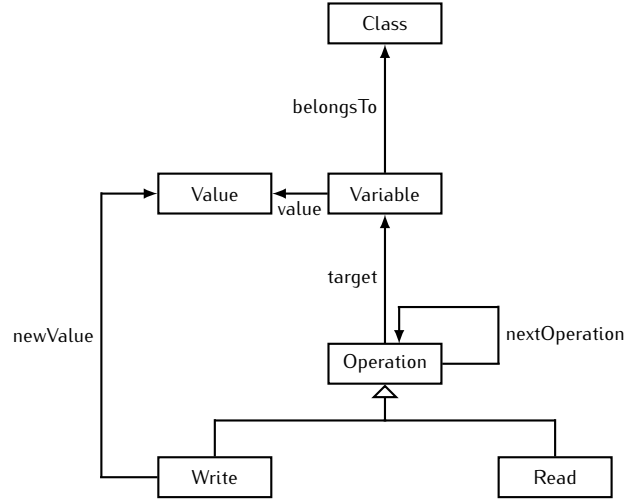


Figure 3.2: Meta model of the operational model of the graph in Figure 3.1.

needed. Of course, such a pattern can be modelled using a graph as well, but for this to be usable, there should be a method to find the pattern in the original graph.

Before explaining how two graphs can be compared, it is necessary to give a formal definition of a graph.

Definition 3.2.1 A *graph* G is a tuple $\langle V, E, L \rangle$, where

- V is a finite set of vertices (nodes)
- E is a finite set of edges and
- L is a finite set of labels

An edge $e \in E$ is a tuple (src, lbl, tgt) , where $src, tgt \in V$ and $lbl \in L$. This defines an edge between vertices src and tgt with label lbl . Labels on vertices are represented by an edge e where $src = tgt$.

To compare two graphs, the concept of *morphisms* is introduced. A morphism defines that there exist similarities in the form and structure of two graphs:

Definition 3.2.2 Given two graphs $G = \langle V_G, E_G \rangle$ and $H = \langle V_H, E_H \rangle$, a *morphism* $f : G \rightarrow H$ has two functions:

- $f_V : V_G \rightarrow V_H$
- $f_E : E_G \rightarrow E_H$

and preserves structure: $f_E(v, a, w) = (f_V(v), a, f_V(w))$ for all $(v, a, w) \in E$.

This definition states that there exist mappings from nodes of graph G to nodes of graph H and that there exist mappings from edges of graph G to edges of graph H . It also defines that the structure of the graph remains in the second graph, by stating that when an edge is mapped, the mapping of the source and target nodes of that edge are mapped as well and consistently.

When these mappings are bijective, the graphs are said to be *isomorphic*. Isomorphism states that the form and structure of two graphs are exactly the same, and a one-to-one mapping between the graphs can be made.

Now we can define how a pattern can be detected in a graph: if a certain pattern occurs as a subgraph in another graph, there must exist a morphism between the

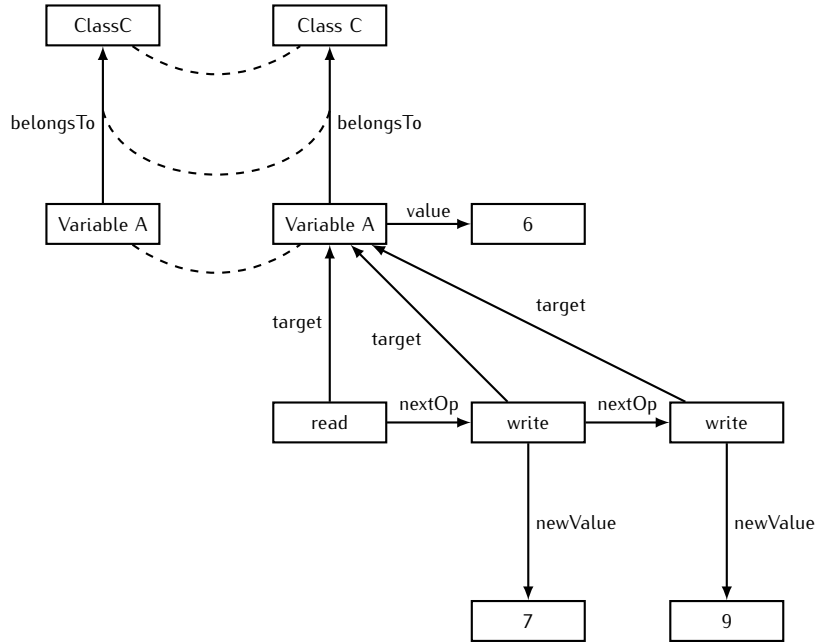


Figure 3.3: Two graphs and a morphism between them (represented by the dashed arcs)

two graphs. This process of finding such a morphism is often called *graph matching*, because we want to find a match of a certain graph in another one.

To make this more clear, consider the left hand side of Figure 3.3. Here we see a graph that represents the relationship between Class C and a Variable A, represented by the edge labeled *belongsTo*. This graph can be matched in the graph of Figure 3.1, because there exists a morphism between the two graphs. This morphism is represented by the dashed arcs between the graph in the left hand side and the graph in the right hand side. Note that the morphism does not only define a mapping between nodes of the two graphs, but of the edges involved as well, and that the morphism preserves the structure.

When graphs are used to detect a graph in another graph, the latter graph is called the *host graph*. The host graph is the graph for which a morphism has to be found.

Although morphisms make it possible to detect subgraphs in other graphs, being able to change the structure of the host graph once we found a match would be useful as well, especially in the context of behavioural conflict detection. This would make it possible to indicate in the host graph where a conflict was found and which operations are involved in this conflict. This is in contrast to using a special data structure to store this information. This would completely decouple the conflict detection from the underlying programming language, because all detection calculations are performed in the graph, rather than in a specific programming language and thus would make the pattern detection more portable.

Thankfully there already exists a concept to change the structure of a graph, so-called Graph Transformation Systems. With such a system, a graph is specified that should be matched in the host graph and another graph is given that specifies how the found subgraph is to be altered once a match is found. Not only additions to the host graph can be specified, but deletion of nodes and edges is possible as well.

In this thesis we will use the transformation rule notation of the Groove toolset [28], [3]. This toolset contains tools to create graph transformation systems and to simulate application of rules from these transformation systems. In Groove, transformation rules for pattern detection, addition and deletion and the specification of so called Negative

Application Conditions (NACs) are combined in one specification, opposed to the more commonly used format in which the pattern, items to be added or deleted and the NACs are specified using morphisms. Not only does this allow a more compact notation, it is more easy to see what a rule does as well.

Opposed to a graph that specifies which subgraph should be matched, a Negative Application Condition in a transformation rule specifies a pattern that should *not* be matched in the host graph. If the graph specified by a NAC can be matched in the host graph, the rule itself will not match and the transformations specified by the rule will not be performed. This makes it possible to match very specific subgraphs in a host graph and completes the expressiveness of graph transformation systems.

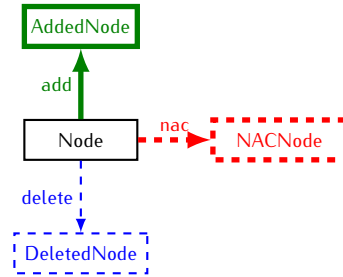


Figure 3.4: Notation of transformation rules in Groove

In Figure 3.4 a transformation rule is given with all features of a graph transformation system. First of all, there is the black, solid node with the *Node* label. This specifies the graph that should be matched in the host graph. Next to this the green edge and node with the *AddedNode* label are displayed, with a bigger line width. This represents a node and an edge that are added after the application of the transformation rule. There is also a dashed node *DeletedNode* and a dashed edge. These two represent a node and an edge that should be matched in the host graph as well, but these will be removed after the application of the rule. Finally, there is a dashed node *NACNode* and an edge connected to it with increased line width, which specify the Negative Application Condition. As stated before, these specify a pattern that should not be matched in the host graph.

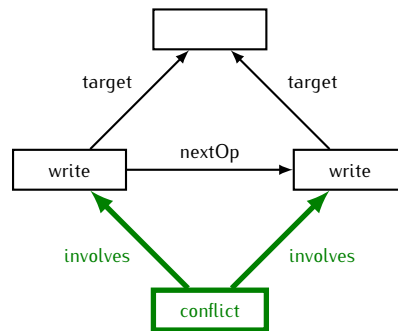


Figure 3.5: Graph transformation rule to indicate a conflict.

In Figure 3.5 a transformation rule is given that can be applied to the graph of Figure 3.1. This is an example of a rule that can be used to detect a certain pattern in the operations that are performed on the Variable A resource. The rule specifies that it is a conflict when two *write* operations are performed after each other on the same target. Whenever this pattern is detected, a new *Conflict* node is introduced which has edges to the operations involved in the conflict. In this example a node without a label is shown as well, this means that it does not matter what label the node has.

In other words, the rule states that it does not matter what the target of the *write* operation is, but it is considered a conflict whenever two consecutive *write* operations are performed on this target. Application of this rule on the graph of Figure 3.1 results in the graph of Figure 3.6.

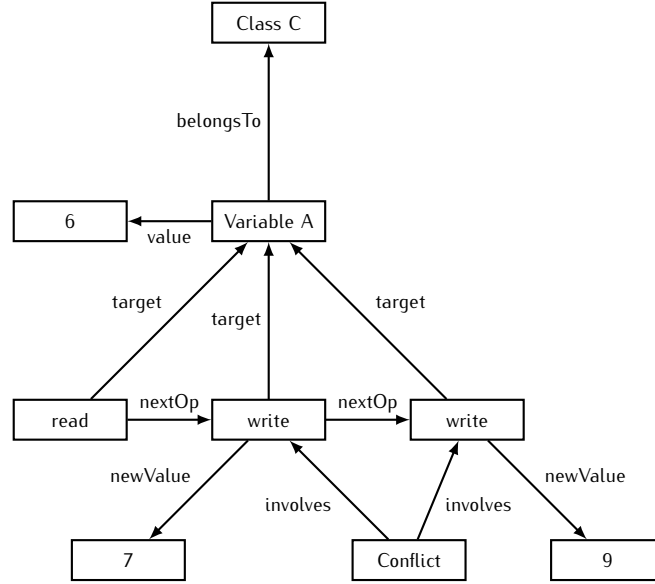


Figure 3.6: Graph representing operations performed on a variable after the application of the rule of Figure 3.5

The previous example shows that transformation rules can be used to specify constraints on a model. These transformation rules are always specified using the meta model, and are applied on an instance of this meta model. Because the rules are specified using elements from the meta model, that what can be expressed using a transformation system depends on what elements are available in the meta model. In other words, the expressiveness of the transformation system depends on the expressiveness of the meta model.

3.2.2 Experiments

In order to evaluate graph transformation systems for behavioural pattern detection, we have developed a meta model for Resource Operations. This meta model was used to model the operations that are performed on resources. To create instances of this meta model, we developed a system that creates a resource operations model of a program while it is executing. This resource operations model is expressed as a graph, and constraints are specified using transformation rules. When the program is finished, subgraph matching is performed to see whether one of the constraint graphs can be found in the model of the program. When such a match is found, it indicates that a constraint violation has been found.

With respect to constraint specifications, the applicability of graph transformation systems all depends on the meta model that is being used. When the meta model is not general enough it can be hard to create constraint specifications that apply to different variations of the same constraint and great care should be taken when developing such a meta model. One of the meta models that has been used in our experiments is given in Figure 3.7.

In this meta model the main entity is formed by the Resource. The Resource is at the heart of the meta model, because it is what the Resource Operations model

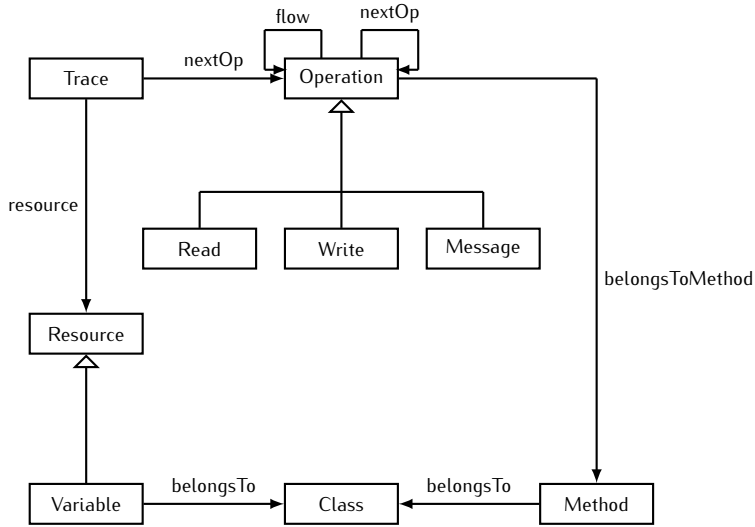


Figure 3.7: One of the meta models used during experiments with using Graph Transformation Systems for defining constraints for Resource Operation Models

is all about. A Resource is connected to a Trace. A Trace is the entity in the model from which all possible operations on a resource can be traced back. A Trace belongs to exactly one Resource. A specialisation of the Resource is given by the Variable. This specialisation is represented in the figure by the open arrowhead. A Variable belongs to a class (the *belongsTo* edge), and a Class can have multiple Variables. A Class can also have multiple Methods, but a Method can only belong to one Class. A Trace is the entity that binds a Resource to the Operations performed on it. The Trace has at most one edge to an Operation, denoted *nextOp*. The Operation that is pointed to by this edge is the first Operation performed on the Resource in the given Trace. Each following Operation in the Trace can be found by following consecutive *nextOp* edges. Each Operation belongs to a certain Method, which is demonstrated by the *belongsToMethod* edge. This edge is in the model for traceability purposes; using this, one can specify where in the code a certain conflict was detected.

The Operation entity is further specialised in Read, Write and Message entities, which denote what kind of Operation was actually performed on the Resource.

In the meta-model there is also a *flow* edge from operations. These edges are used to specify the flow between Operations. If only one Resource is being tracked, this *flow* edge will be the same as the *nextOp* edge, but in case multiple Resources are being tracked, they will diverge. The *flow* edges are merely there to keep track of the flow of control, and can, together with the *belongsToMethod* edges, be used to create a stack trace to a detected conflict.

Using this meta model, several experiments have been carried out to evaluate graph transformation systems for use in resource operations models and constraints on this model.

One observation that was made during these experiments showed that instead of specifying what behaviour was expected in a constraint, specifying constraints has to be done the other way around. With transformation rules constraint violations are modelled, instead of modelling expected behaviour. If for instance a constraint should specify that a *write* operations should always be preceded by a *read* operation, the transformation rule would specify that it is a violation of the constraint if this is not the case, i.e. if the operation preceding the *write* operation is *not* a *read* operation. An example of this is given in Figure 3.8, which gives a transformation rule that can

be used to detect this constraint violation.

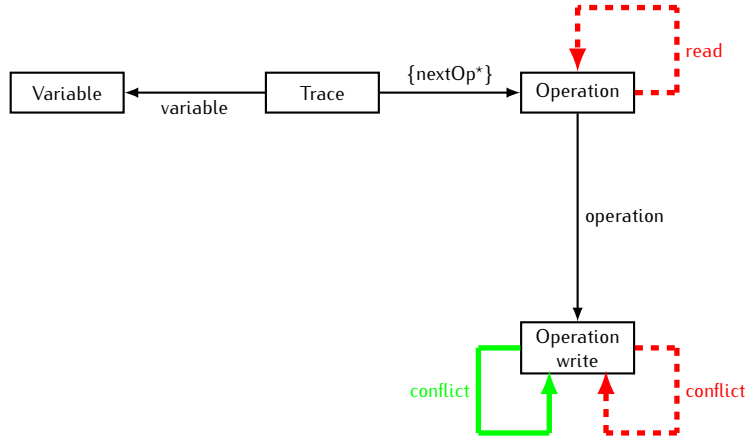


Figure 3.8: Transformation rule defining a constraint violation.

In this example transformation rule, a conflict edge is introduced whenever a node with the *write* label is preceded by a node that does not have a *read* label. This example shows that constraints are specified by giving the exceptions to the expected behaviour. This could also mean that in order to fully test whether an implementation conforms to a constraint, multiple exceptions to the expected behaviour have to be specified, and that one constraint could exist of several rules.

If this is compared to regular expressions, this is a disadvantage, because with regular expressions the constraint itself is specified, rather than the exceptions to that constraint, and a constraint can be specified using one regular expression. The regular expression either matches, or does not match and when it does not a constraint violation is detected. However, as will become clear in the next section, there are methods with which it is possible to specify constraints using Graph Transformation Systems instead of specifying the exceptions to the expected behaviour.

3.2.3 Evaluation

We have seen that graph transformation systems are based on the relatively simple concepts of graphs. The simplicity of graphs is one of the advantages of graph transformation systems, because they are easy to understand by domain experts. This is of importance, because often constraints on the behaviour of a system are developed by domain experts, or the constraints are at least verified by these experts. We have also seen that graphs, although simple in nature, form a powerful modelling tool, because the expressiveness only depends on the meta model that is used. Because of their relative simplicity and expressiveness, they have been used in several areas such as internet routing and for protocol verification.

During the experiments we encountered that the constraints had to be expressed in terms of conflicts rather than the wanted behaviour. Compared to regular expressions, this could be a disadvantage. However, these problems can be solved by changing how a conflict is detected.

In the experiments a conflict was detected whenever a rule would match, and therefore the rule had to specify the exceptions to the expected behaviour. If this conflict detection method is changed, i.e. a conflict is detected when a rule does *not* match anymore, the rule could be used to specify the expected behaviour, rather than specifying the exceptions.

With respect to the evolvability of constraints, graph transformation systems suffer from the same problems as regular expressions. The creation of constraints and a one-to-one mapping of message names to operation names introduced the same evolvability problems as with regular expressions; when a method name changes, the constraint specifications has to be changed as well. However, when new methods are added, it is not necessary to change the constraint specifications, because when a call to this introduced method occurs, in the worst case the constraint specification graph will not be found.

One way to overcome the problem of changing method names is by introducing an intermediate layer in which method names are mapped to operation names. These mappings could be many-to-one mappings and using this it is possible to update the implementation without the need to update the constraint specifications. This solution however, is not uniquely usable for graph transformation systems, but could be used with regular expressions as well. Such a mapping could be implemented using an annotation system, with which method names can be annotated as operation names and thus creating a mapping from methods to operations. There is one pitfall however: the mapping between method names and operation names has to be updated as well, meaning that the introduction of the mapping only shifted the evolvability question to another level.

There is another property of graph transformation systems that poses a problem with respect to this thesis. To test whether a rule can be applied, a match between the rule R_1 and the graph G_1 must be found. The algorithm for matching is known to be NP complete in the size of R_1 [12]. This means that although it is relatively simple to check whether a solution is correct, finding such a solution cannot be done in polynomial time. For small graphs this is not a problem, but whenever the pattern grows, the time to find a match grows exponential in the size of the pattern. Note however, that this is a worse case complexity, it does not mean that this will always be the case. There exists for instance a method that performs incremental pattern matching, with which it is not necessary to perform a complete graph matching whenever the host graphs changes. In most cases this means that a complete match does not need to be found, because a partial match was already found. However, even these methods still have the same worst case complexity, which means that in some cases the pattern matching will take a very long time.

Although this does not pose a problem with respect to the expressiveness nor to the evolvability of specifications, this does pose a problem for runtime checking of resource operations constraints. Because in the worst case, every time a resource operation has been performed, a new subgraph match has to be found. Because the algorithm for subgraph matching is NP complete, this means that it will considerably slow down the analysis process. In Chapter 2 we stated that we wanted that the solution would be scalable and that complex constraint should not incur performance penalties. Therefore graph transformation systems are not considered to be an alternative for regular expressions in the context of this thesis.

3.3 Vibes based

3.3.1 Introduction

Vibes is a constraint specification language for testing whether implementations are compliant with their design specifications. It was introduced in the work of Güleşir [14] along with several tools to test the consistency between several constraints and automated tools to test the compliance of the implementation with the constraint specifications. It introduced a graphical syntax to define constraint specifications and

the most important contribution of his work consists of a so-called *context sensitive wildcard*. The meaning of the wildcard is dependent on the context in which it appears, and before we explain this more clearly, we will give an introduction to the syntactical elements of Vibes.

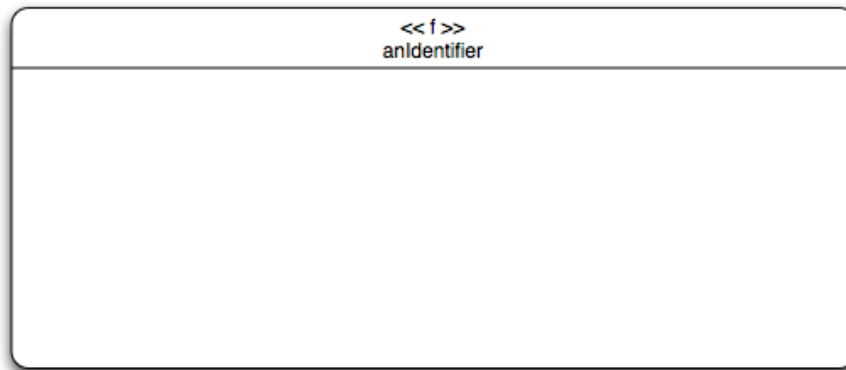
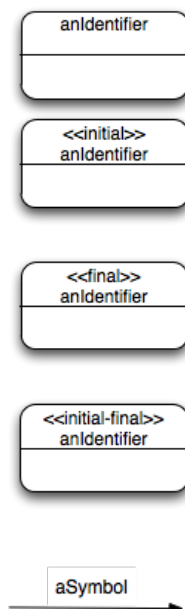


Figure 3.9: Container node

We begin by giving the semantics of the **container node**, of which an example is given in Figure 3.9. The container node defines the boundaries of a specification, i.e. all elements belonging to the specification are drawn inside the container node. The container node has a name, in the example this is *anIdentifier*, and a regular expression, called the **scope expression** of the specification. This scope expression is used to identify the set of procedures on which the specification should apply. In the example this scope expression is $\langle\langle f \rangle\rangle$, meaning that this specification is meant for all procedures that are called *f*.



In a specification a normal **node** is given by a rectangle with rounded corners. A node has an identifier which is used to specify its name. In this example the name is given by *anIdentifier*.

In a specification an **initial node** is also given by a rectangle with rounded corners, but this node is stereotyped with $\langle\langle initial \rangle\rangle$. This node represents the unique initial state of the constraint specification.

A **final node** is given by a normal node with the $\langle\langle final \rangle\rangle$ stereotype. This is a node that when reached, is accepted by the specification, meaning that the program can always be in that state without violating the constraint given by the specification.

An **initial-final node** is a node that is both an initial and a final node at the same time. This means that the specification starts at this node and that it will never be a constraint violation when it is in this state.

In a specification a **transition** is given by an arrow between a source node and a target node with a label written above it. The label is used to specify procedures that can be called from within the constrained procedure. If such a procedure is performed by the system under verification, the specification will make a transition from the source node to the target node.



A special kind of transition is given by the **wildcard transition**. A wildcard transition is given by an arrow between a source node and a target node with a \$ label. The wildcard transition represents all transitions that are not defined explicitly to leave the source node. As a result, the wildcard transition is dependent on all other outgoing transitions from the same state, because it is used to designate all possible, non-specified operations that can be performed on the resource in a specific state. In other words, the transition is context sensitive.

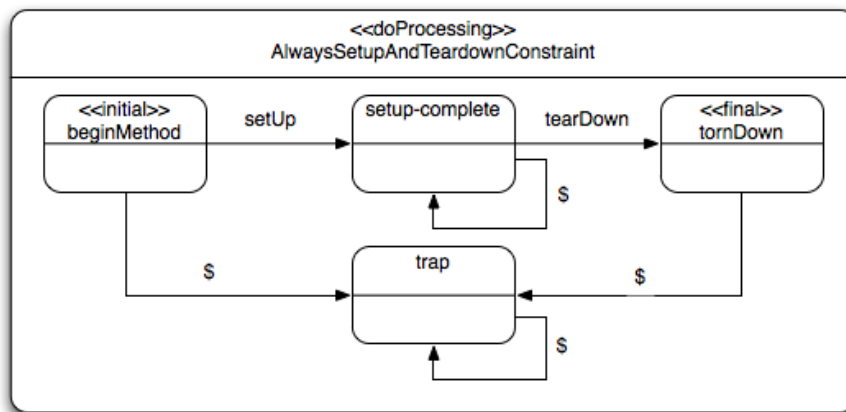


Figure 3.10: Constraint on method *doProcessing* stating that a method should always start with a call to *setUp* and should end with a call to *tearDown*.

As an example look at Figure 3.10, in which a Visual constraint is given. This constraint has as scope expression `<<doProcessing>>`, meaning that it is applicable for all methods with that name. The identifier of the constraint is *AlwaysSetupAndTearDownConstraint*. This constraint consists of four states, *beginMethod*, *setup-complete*, *tornDown* and *trap*. The constraint can be interpreted as follows: The only possible way in which an implementation is accepted is when it first executes the *setUp* method, then makes zero or more other method calls and finally makes a call to the *tearDown* method. When it tries to call any other method before the *setUp* method, or when a call to any method is made after the *tearDown* method was called, the *trap* state is reached. This state is a non-final state and it is not possible to reach a final state either. This implies that the system is in a conflicting state.

However, when the final call is a call to *tearDown*, and no other calls are made afterwards, the system is in a state that complies to the specification and therefore it can be said that the system is behaving according to its specification.

```

1 void doProcessing ()
2 {
3     setUp ();
4     executeAlgorithm ();
5     tearDown ();
6 }

```

Listing 3.1: Source code example for design compliance verification

Consider the source code in Listing 3.1. This is an implementation of a *doProcessing* method, and therefore should comply with the constraint of Figure 3.10. Now let

us check whether this is really the case.

When we start, we are in state *beginMethod*, because this is the initial state of the Vibes specification. Then, in line 3, we make a call to *setUp()* and by doing this, we make a transition to state *setup-complete* of the constraint. In line 4, we call the *executeAlgorithm()* method. This method call does not appear as a transition in the specification, but because we are in state *setup-complete* and the only outgoing transitions are the *tearDown* and the *\$* transition, we automatically map the call to *executeAlgorithm()* to the **context sensitive wildcard** transition *\$*. We can do this, because we have encountered a method call whose name does not appear as an outgoing transition from the state we are in (i.e. *setup-complete*). Now, in line 5, we make a call to the *tearDown()* method, and because we are still in *setup-complete* and there is an outgoing transition with the same name, we go to state *tearDown*. Because *tearDown* is a final state, and there are no more method calls in the source code, we know that this implementation complies with the constraint.

```
1 void doProcessing ()
2 {
3     initialize ();
4     setUp ();
5     executeAlgorithm ();
6     tearDown ();
7 }
```

Listing 3.2: Another (wrong) source code example for design compliance verification

Now consider the code of Listing 3.2. When we start, we are in state *beginMethod* of the *AlwaysSetupAndTeardownConstraint* as well, because this always is the initial state of the constraint. In line 3, we begin by making a method call to the *initialize()* method. Because we have a *\$* transition from state *beginMethod* to the *trap* state, and we are not calling the *setUp()* method, we enter the *trap* state. From here it does not matter what we do, we will never reach a final state. This means that the implementation did not comply to the constraint and that a constraint violation was detected.

The thing that jumps out from the examples is the power of the context sensitive wildcard. Due to this wildcard, the constraints are more robust, because in comparison with regular expressions, we do not need the whole alphabet to be defined. In other words, a regular expression with alphabet Σ only accepts or rejects finite sequences from Σ^* whereas Vibes accepts any finite sequence, even sequences that are not in Σ^* .

Although relative simple in nature, the context sensitive wildcard brings great power to Vibes specifications, because there is no need to change constraint specifications whenever implementations change. All newly introduced methods or symbols are caught with the *\$*-transition.

In this thesis we try to find an alternative to regular expressions to create constraint specifications for resource operation models. Vibes, in contrast, was introduced to check whether an implementation in a procedural language conforms to its design. Therefore we have to map Vibes specification to the Resource Operations Model.

In order to be able to use Vibes specifications as constraints on Resource Operations Models, we have to make some changes to how certain pieces of a Vibes specification are interpreted:

- Where the container node in the original Vibes specification language was used to specify a constraint on methods, we now use the container node to represent a constraint on the operations that are performed on a resource.

- The scope expression is now used to specify what resource is constrained by the constraint.
- Transitions in the Vibes specifications are now used to represent the operations that are performed on the constrained resource.

These three small changes in how Vibes specifications are interpreted, enable us to use Vibes to create constraints on resource operations. We only made minor changes to the semantics, and we did not change the semantic of the context sensitive wildcard, nor did we change the semantics of states. Because of this, all properties of Vibes specifications still stand, i.e. there is still no need to specify all possible symbols in the constraint specification. Because the context sensitive wildcard is still used as it was originally intended, it still stands for all transitions from the current state which are not explicitly stated in the specification.

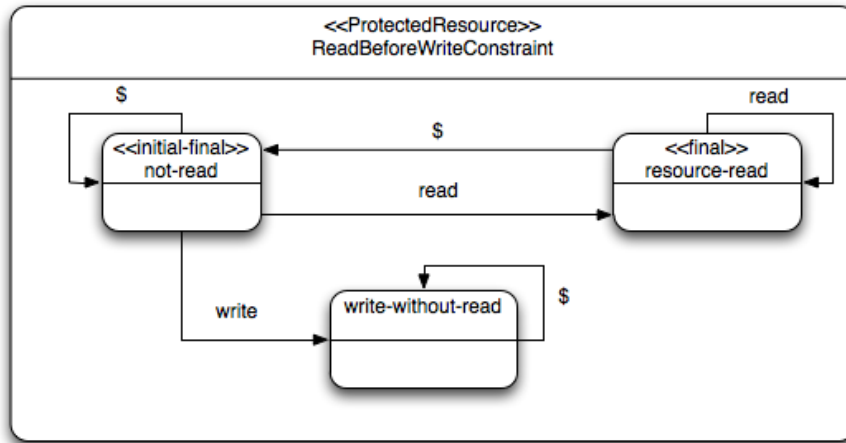


Figure 3.11: Constraint on *ProtectedResources* stating that a *write* operation should always be preceded by a *read* operation.

As an example of a constraint specification for resource operations, take a look at Figure 3.11. This specification constrains *ProtectedResources* and it is meant to enforce that each time a *write* operation is performed, it is preceded by a *read* operation. When a *write* operation is performed without a preceding *read* operation, we immediately make a transition to state *write-without-read*, which is a trap state, because it is not possible to reach a final state from it.

3.3.2 Evaluation

In contrast to regular expressions, Vibes specifications are easier to understand for domain experts, because of their graphical representation. Because Vibes specifications are almost the same as deterministic finite automata, they are not computationally complex, they can be implemented using an adapted form of state machines. If no transition exists for a given input symbol, the wildcard transition is followed. This means that a constraint specification can be implemented at relatively low costs and checking constraints is nothing more than offering a symbol for consumption. Because the state machine knows its current state, it only has to check all outgoing transitions from that state and follow the correct one. Compared to graph matching Vibes specifications are more attractive as an alternative to regular expressions for the specification of resource operation constraints.

Because Vibes specifications have the added context sensitive wildcard transition, they are less susceptible to changes in the implementation than regular expressions and they can very well be used instead of regular expressions.

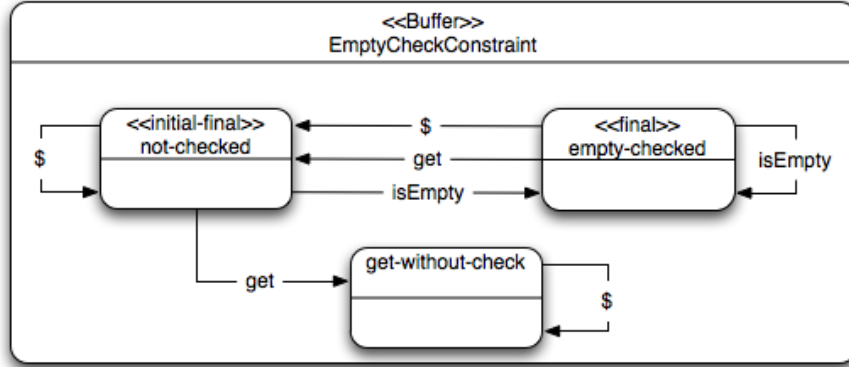


Figure 3.12: Vibes constraint for a Buffer resource.

If we look back at the example of Chapter 2, in which we gave a regular expression to constrain a Buffer resource, we will address the problem that existed with regular expression using Vibes specifications to specify the constraint.

Remember that in Chapter 2 we consider a resource Buffer for which the following operations are defined:

$$\{put, get, isEmpty\}$$

and consider the constraint specification given in Figure 3.12. This constraint specifies that the *get* operation can only be performed if the *isEmpty* operation is performed immediately before it. If the *get* operation is performed without a preceding *isEmpty* operation the constraint will be violated, because the non-final state *get-without-check* is reached from which it is not possible to reach a final state. Note that in the example of Figure 3.12 the *get* transition between *empty-checked* and *not-checked* has been made explicit for clarity; this transition could have been omitted because of the *\$* transition between *empty-checked* and *not-checked*.

Now consider that a new operation *clearBuffer* is added. This has no impact on the constraint specification of Figure 3.12, because every time the new *clearBuffer* operation is performed this is handled by the *\$* transition in the specification.

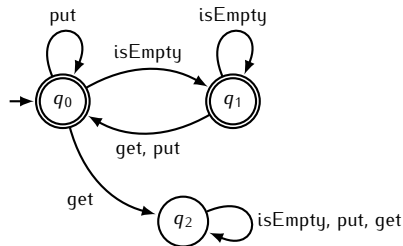


Figure 3.13: Buffer constraint expressed with an DFA.

If, however, one of the approaches utilising regular expressions would have been used, the same constraint on performing the *get* operation would be given by the deterministic finite automata (DFA) of Figure 3.13. This DFA does not have the wildcard transition, instead all symbols of the alphabet are in the specification in order to be

able to accept those symbols. If we now add the *clearBuffer* operation, this would mean that in the DFA of Figure 3.13 new transitions have to be added. For the semantics of the constraint to stay the same, we have to add this new symbol to all looping transitions. For the DFA given in Figure 3.13 this would result in the DFA given in Figure 3.14.

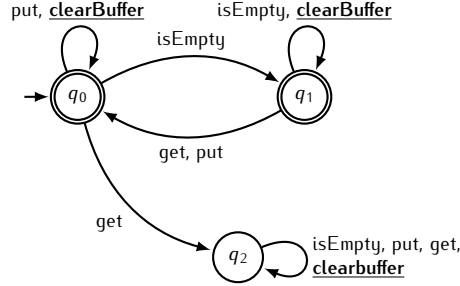


Figure 3.14: Buffer constraint adapted for the added *clearBuffer* operation.

This demonstrates that Vibes specifications are less susceptible to changes than regular expressions and thus have higher evolvability than regular expressions.

3.4 Summary

In this chapter we have investigated two alternatives to regular expressions in the context of resource operation constraint specifications. We have seen that although very expressive and easy to understand, graph transformation systems are rather expensive to implement. Because of the NP completeness of subgraph matching, graph transformation systems are not a good alternative for runtime verification of these constraints.

Although Vibes specifications were originally intended to test whether procedural implementations conform to their design, they can be used for resource operations constraint specifications if we change the interpretation of three of its elements. By changing how these elements are interpreted, Vibes specifications can be perfectly used for specifying and checking constraints on resource operations, and because of the context sensitive wildcard they are less susceptible to changes in the implementation than regular expressions. Because Vibes specifications have the same computational complexity as regular expressions, we feel that Vibes specifications are a good alternative for regular expressions in the context of this thesis. In the remainder of this thesis we will therefore use Vibes specifications instead of regular expressions for the definition of resource operation constraints.

Chapter 4

Associating actions with behavioural patterns

In the previous chapter we have discovered that Vibes can be used as a constraint specification language for Resource Operation Models. We also discovered that Vibes in a somewhat adapted form can be used to detect behavioural conflicts in other areas than in which the model is mostly used, i.e. concurrent computing and aspect oriented programming. With the Resource Operations model it is possible to capture multiple implementations of the same abstract concept into one Resource, which allows users to reason about the more abstract implications of this abstract concepts. An example of this is that both a buffer and an array can be captured by a Collection resource. Constraints can then be applied on the collection resource, regardless of the actual implementation of the collection. In object oriented systems this is comparable with polymorphism, but with polymorphism there is the limitation that the interface of the objects must be the same. In contrast, with an Resource Operations model it is also possible to create abstractions of similar methods into one operation.

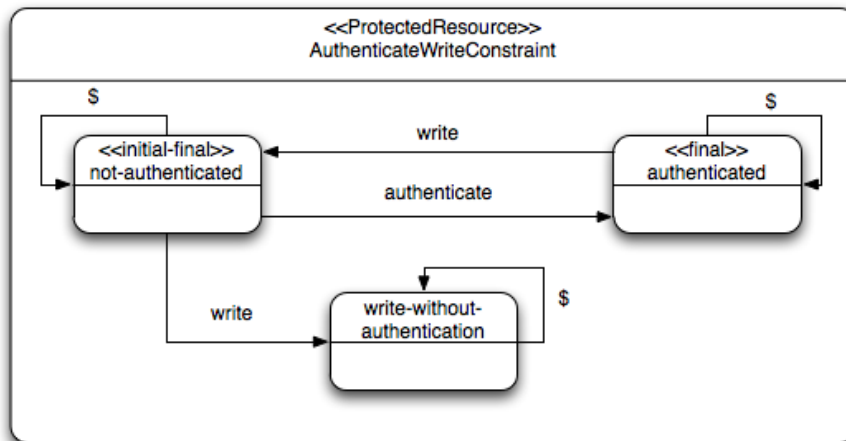


Figure 4.1: Constraint on *ProtectedResources*. When in state *authenticated* we know that authentication has taken place.

The possibility to define constraints on high level concepts, in a state machine based manner such as Vibes, can be very useful, especially when we look beyond the constraint specifications. When we look at a Vibes specification, what we really see is

an adapted finite state automaton capturing the state of a given resource. For instance, consider the constraint specification of Figure 4.1. If we consider state *authenticated*, we know one thing: The only way to get in state *authenticated* is by having executed the *authenticate* operation. In other words: in state *authenticated* authentication has taken place.

In this case we do not only use Vibes to create behavioural constraints, but we can use the information contained in the specification as well. Because we know that a state in a constraint also represents the state of the resource, we can use this information to reason about the state of the resource. But if we can reason about it, we could also use it to perform actions.

This notion is also present in the original work on Vibes [14], where state information can be used to perform extra tasks. In this work the coupling between a non-reactive part and a reactive part of a wafer scanner is taken as an example. This coupling is done in what is called TransformSource.

In the following sections we will introduce a method to attach actions to constraint specifications, and we will take a look at action precedence in case multiple actions from different constraints should be performed on the same resource.

4.1 Specifying actions in constraints

We have identified that states in Vibes constraints represent states of resources and that this could make it possible to perform extra actions. However, before we can create a model to specify these actions, we have to identify where these actions could be performed, and at what specification element they should be bound.

When we look at a constraint specification we can distinguish two important objects: states and transitions. Two events can be identified for each of those: a state can be entered and left, and we can begin or end a transition. This is represented graphically in Figure 4.2. The numbered circles in this figure represent each of the four different events that occur in a constraint specification.

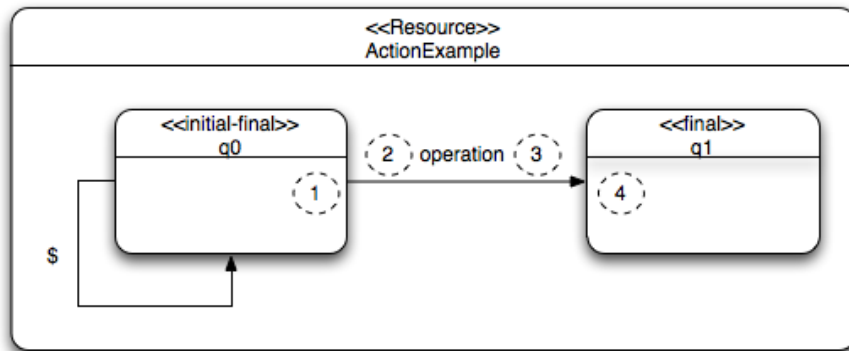


Figure 4.2: Example of events that occur in a Vibes constraint: 1. a state is left, 2. a transition is started, 3. a transition ends, 4. a state is entered.

We propose to use these events to attach actions. This means that the actions should be performed at those moments as well. When a transition is made, first all actions defined on the state leave event are performed. After this all transitions defined on the start transition event are performed, then the actions on the transition end event and finally all actions on the enter state event.

When these events are studied more closely, it becomes apparent that instead of using four different events, the same could be accomplished by using two events.

Semantically leaving a state is always the same as beginning a transition and the same goes for entering a state. However, we choose to keep the four different events at the syntactic level, because being able to define actions on state leaving or entering removes the need to copy the action to each outgoing or incoming transition.

To specify actions, blocks are used. A block is a piece of code that is attached to an event. This means that there can be four different kinds of blocks in a specification: *leave* a state, *before* a transition, *after* a transition, and *enter* a state.

We propose to write actions on states inside the state itself. In the state, two types of actions can be specified for the two different events that can occur in a state, by writing the actions either in an *enter* or *leave* block. These blocks indicate that the action is performed when the state is entered, or left.

Actions that are associated with transitions are added in the specification by writing them in a comment that is attached to a transition by a dotted line. For actions on transitions two blocks can be used, either *before* or *after*. These blocks indicate that the action is performed before or after the transition is made.

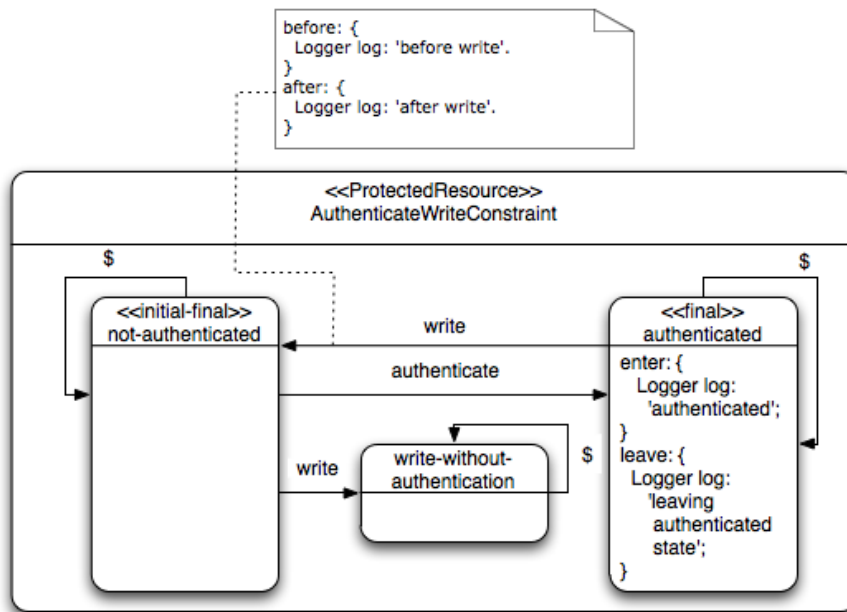


Figure 4.3: Example of a Vibes constraint with added actions.

An example of a Vibes constraint that has actions attached to it is given in Figure 4.3. This example shows four actions that are executed at each of the four discovered events. When a transition is made to state *authenticated*, the action is to log that the resource is now in an authenticated state. When this state is left, this is logged as well. When the *write* transition from state *authenticated* to state *not-authenticated* is performed, this is logged both before and after the actual transition is made.

Note that the actions performed when state *authenticated* is entered or left could be placed on the incoming and outgoing transitions respectively as well. This would mean that the logging of entering the state could be specified on the *after transition* event of the *authenticate* transition and on the *after transition* of the *\$* transition. The logging that happens when the state is left could be attached to the *before transitions* of both the *\$* and the *write* transitions.

4.2 Actions and AOP

With the introduction of actions in Vibes constraint specifications, we have actually introduced something similar to advice in Aspect Oriented Programming (AOP) [22].

AOP is often used to move crosscutting code to one specific location. Crosscutting code is code that appears in multiple places in a implementation. Consider for instance a system where a user must be authenticated to perform certain operations. In the implementation of this functionality a check is performed to see whether the user is authenticated. The code to perform the test to see whether the user is authenticated is located in multiple places in the source code, because it is needed for each operation for which the user has to be authenticated. Because this code appears at multiple places, and its functionality is not limited to one place, it is called scattered crosscutting code.

With AOP, this scattered code is placed in an advice and then the advice is superimposed on the implementation. To perform this superimposition, a pointcut expression is used to specify at which joinpoints the advice should be executed. A pointcut expression is an expression that selects a subset of all joinpoints in the system and a joinpoint is a point in the execution of a program at which code can be inserted.

Different implementations of Aspect Oriented Programming allow the usage of different joinpoints, but the commonly used joinpoints are method calls, method executions and field accesses. In AOP joinpoints are mostly specified on a syntactic level, and although AOP can be used to change the behaviour of a program, joinpoints are usually not specified in terms of behaviour. Although AspectJ [1] has a construct to specify joinpoints based on the call stack, these joinpoints are still very much bound to the syntax and structure of the program.

In a keynote on the AOSD, Kiczales (one of the main contributors to AspectJ) made a strong case to improve joinpoint models as they exist today [21]. This because he feels that the current joinpoint models are not completely capable of capturing all events that can occur in the execution of a program. In [6] Cazzola *et al.* investigate the limitation of current mainstream AOP implementations, and they opt for a more semantic approach for joinpoint selection. They identify that semantics in the context of AOP can mean two different things, being *property* and *behavioural* based. With property based semantics they mean that joinpoints are based on properties of those joinpoints, for instance, whether the code at the joinpoint does not interfere with a specific property. Behavioural based semantics then deals with what the executed code displays as behaviour.

The Vibes constraint specifications with added actions can be seen as superimposing advice at behavioural joinpoints. The joinpoints are selected by selecting a state in the constraint specification, and these states represent the behaviour of the code on an abstract level. This introduces a joinpoint model at a much higher level than just at the syntactic level.

4.3 Multiple simultaneous actions

When multiple constraints with actions have been specified that constrain the same resource, multiple actions from different constraints can coincide. This means that multiple actions have to be performed at the same time. Sometimes this can lead to conflicts and these need to be resolved.

Look for instance at the constraints in Figure 4.4. Both constraints have a scope expression stating that they constrain ProtectedResource and both constraints have actions defined. When both constraints are in their respective initial states and a *write* operation is performed, both constraints have actions that should be performed at the same time. The first constraint has a *leave* action and the second constraint defines a

before action that should be performed. Both actions should be performed at the same time, but it is not possible to determine which one should be performed first.

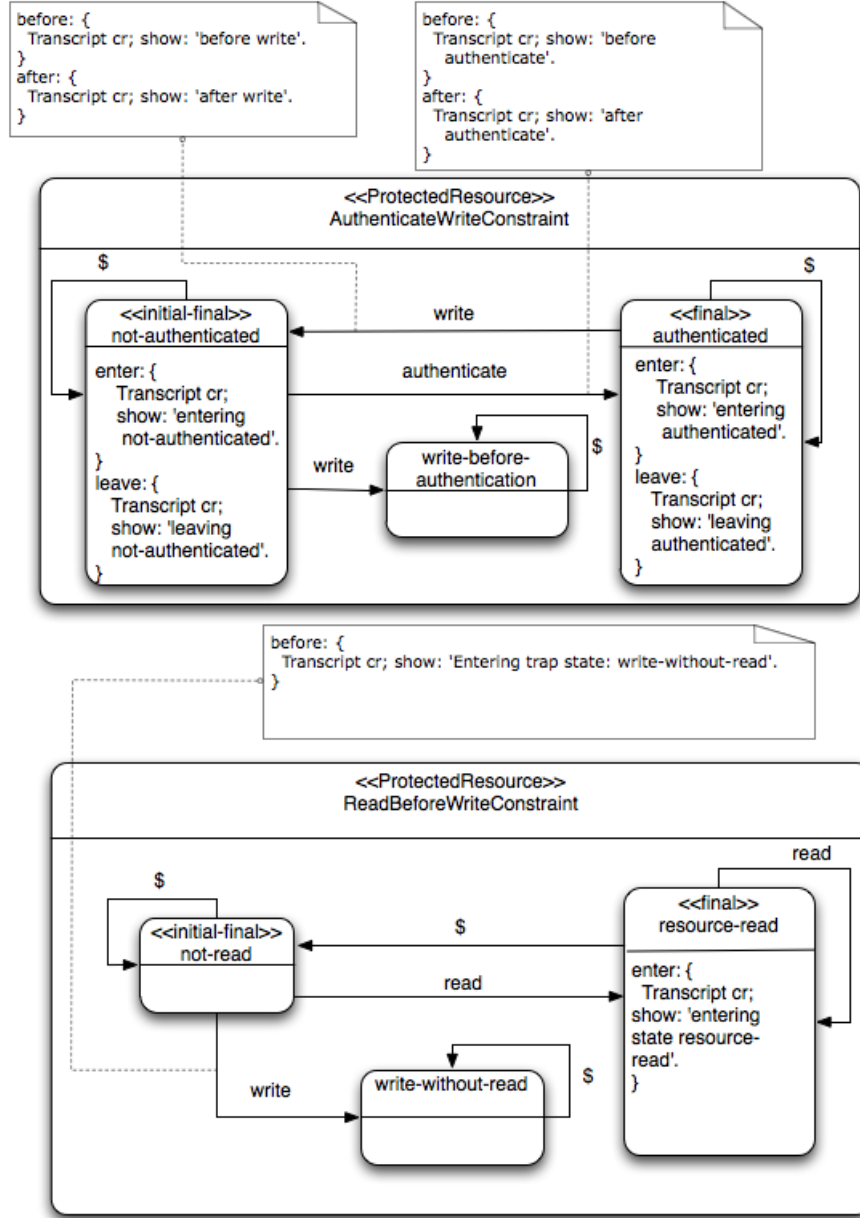


Figure 4.4: Two constraints with actions, both constraining ProtectedResource.

The problem of ordering actions that are specified on the same transition is comparable to multiple aspects being superimposed on the same shared joinpoint in Aspect Oriented Programming. If it is not possible to specify an ordering for the aspects, it could very well be that the newly composed program does not behave as expected. This is why for instance in AspectJ it is possible to declare a partial ordering by using the *precedence* keyword [1].

In Nagy [25], the need for ordering of aspects is identified and a general, AOP language-agnostic way to define precedence and conditional application of actions is

introduced. An action is defined as a piece of code that should be performed on a so called shared join point. A shared join point is a point in the execution of a program where multiple actions should apply. If multiple actions should be applied on the same shared join point, it can be the case that the ordering of these applications becomes important. It is therefore important that it is possible to specify an ordering.

To determine what actions should be performed first, Nagy et al. introduce amongst others so called *ordering constraints*. To define an ordering constraint, the $pre(A, B)$ operator is used, to specify that the actions in the advice of aspect A should be performed before those in the advice of aspect B are performed.

This $pre(A, B)$ operator can be used in the context of Vibes constraints as well, and can be used to define an ordering on the actions of multiple Vibes constraints. However, to be able to define a precise ordering, a method to uniquely identify an action in a resource is needed. Therefore we introduce the following syntax to uniquely identify actions belonging to states:

ConstraintName.statename.[enter | leave]

and for actions belonging to transitions

ConstraintName.(sourcestate, transitionname, targetstate).[before|after]

We now can use the $pre(action_1, action_2)$ function to specify that $action_1$ should be performed before $action_2$ is performed.

We can apply the $pre(A, B)$ operator to define the order of actions of the constraints of Figure 4.4, by specifying the following:

*pre(AuthenticateWriteConstraint.not-authenticated.leave,
ReadBeforeWriteConstraint.(not-read, write, write-without-read).before)*

This ordering constraint states that the actions associated with the *leave* event of state *not-authenticated* in the *AuthenticateWriteConstraint* should be performed before those associated with the *before* event of the *write* transition between states *not-read* and *write-without-read* of the *ReadBeforeWriteConstraint*.

The $pre(A, B)$ ordering constraints can be used to define a partial ordering of actions to be performed when an operation is performed on a Resource. This partial order is created by creating a dependency graph of the actions, and using this dependency graph an ordering can be determined.

Although the $pre(A, B)$ operator can be used to specify the ordering of actions in detail, we feel that it can be cumbersome to specify the order for all actions in a constraint. Often it can be the case that all actions from one constraint should be performed before the actions of another constraint. Instead of using the $pre(A, B)$ operator for each distinct combination of actions, we introduce the $C_1 < C_2$ operator. The meaning of this operator is that all actions in constraint C_1 are performed before the actions of constraint C_2 .

If, for instance, we wanted to state that all actions of the *AuthenticateWriteConstraint* from the example should be performed before the actions of the *ReadBeforeWriteConstraint*, we could easily specify this with the $<$ operator:

AuthenticateWriteConstraint < ReadBeforeWriteConstraint

4.4 Evaluation

One thing that can be noted when evaluating actions and the events where they occur, is that leaving a state can be seen as beginning a transition. If we look at the example

in the beginning of this chapter, the first event of leaving a transition, is the same as beginning a transition. The same is true for entering a state, which is the same as ending a transition. Although this observation is theoretically correct, we have chosen to distinguish leaving a state and beginning a transition and to distinguish ending a transition and entering a state. We have opted for this interpretation, because the distinction minimises the required effort from an end-user; instead of specifying actions on each transition, only one state related action has to be specified. Note however, that whenever a distinction is made between the events, this distinction should be made in the implementation as well, because logically one expects the actions of a *leave state* event to be executed before the actions of a *begin* transition.

The Vibes specification itself is in theory programming language agnostic. This means that for the specification itself it does not matter which programming language is used to specify actions. However in our implementation the language depends on what programming language the Resource Operations Model is used. For instance, if the model is used to constrain the behaviour of resources in Smalltalk, the actions should be written in Smalltalk as well.

A method to specify ordering constraints of actions in graphical constraint specifications has not been specified, but this could be achieved simply by drawing arrows between two constraint specifications. The direction of this arrow could then specify which constraint has precedence. At the moment the set of ordering constraints has to be specified separately in text. The exact method and location where these ordering constraints should be specified should be investigated further.

Chapter 5

Detecting behavioural patterns in dynamically typed languages

One of the goals of this thesis is to perform detection of behavioural patterns in dynamically typed programming languages. Therefore, in this chapter we introduce a method of performing the detection of behavioural patterns in dynamically typed languages. We begin by giving an introduction to the problems encountered during behavioural pattern detection, after which methods to keep track of the state of behavioural patterns are described. We then describe a method that can be used to indicate which objects we want to keep track of and finally we give an overview of the general solution that binds everything together into a solution to perform detection of behavioural patterns in dynamically typed programming languages.

5.1 Verification of static and dynamically typed languages

Compilation is the process in which a program in human readable form is transformed (compiled) into machine readable instructions. This compilation process consists of several stages. First, the source code is read and transformed into a stream of tokens. A token is a single atomic unit of a programming language. For instance, in the Java programming language the keyword “do” is a token. The process of creating a token stream is called scanning or lexical analysis. The token stream is used as input for the next phase, the syntactic analysis phase. In this phase the tokens are parsed and the linear structure of the program is transformed into a tree structure, which represents the formal language defining the programming language. This tree structure is often called the Concrete Syntax Tree (CST), because all elements of the syntax are in it. The Concrete Syntax Tree is then transformed into a more abstract form from which unnecessary tokens, like braces and semi colons, are removed. This form is called the Abstract Syntax Tree or AST. The AST is used in the next phase, the semantic analysis phase, in which the semantics of the program are checked. In this phase variables are bound to their definitions as well.

The scanning and syntactic analysis phases are the same when statically and dynamically types programming languages are compared. In both types of languages the source code has to be transformed into a token stream and this token stream has to be transformed into a CST and AST. The semantic analysis phase, however, differs between statically and dynamically typed languages. In a statically typed language, the AST is updated with information about the types of arguments and variables. This information is used to analyse whether parameters passed to a function or values assigned to variables have the right type. On the other side of the scale, dynamically typed language compilers do not attach this type information to the AST, because this

type information is not available. In dynamically typed languages, variables do not have to be declared with a type, this type is only inferred at runtime.

Static verification of source code means that the verification itself can be done before the program is run. This kind of verification is often performed at compile-time. Static verification techniques use the type information provided in the AST to perform the verification. In the work of Güleşir [14] this is the case as well. The AST is analysed and based on the information in the AST the actual verification of Vibes constraints is performed.

The information that is used at compile-time is especially of importance for variables. In a statically typed language, a variable is first declared and then used. Part of the declaration is specifying what the type of the variable will be. This type restricts what operations can be performed on the variable. This type is also often used in compile-time verification, because it is the type that gives a description about how the variable can be used. In terms of the Resource Operations Model, the type of an object is mapped to a given Resource, and the operations that are performed on that object are mapped to resource operations. This mapping can only occur because the variable was given a type when it was declared. When dynamically typed languages are considered, variables can also be declared, but it is not necessary to also define the type of the variable. During the lifetime of the variable, it can take on any type, and it can only be determined at runtime whether certain operations can be performed on it. Because in the context of Resource Operations Models the type is mapped to a resource, it means that when no type information is available, it is not possible to determine as what resource kind the variable is to be regarded. Because this type information is only available at runtime, it is only possible to determine the kind of resource at runtime as well.

Because dynamically typed languages do not have the type information available at compile-time, the static verification approach cannot be used for dynamically typed languages. This means that in order to use the static verification technique, complex algorithms like type inference have to be used, which could lead to multiple type possibilities for a variable. All these possible types have to be taken into account, and the verification process becomes far more complex than in cases where the precise type of a variable is known.

To reduce the possible different flows of control, we have chosen to perform runtime verification. This will lead to only one flow of control, which in turn will lead to precise type information of variables. However, runtime verification introduces some challenges which have to be solved first. These challenges involve keeping track of what has been encountered and finding a way to couple the actual program state to the state of the verification of the constraints.

5.2 Keeping track of patterns

The most important aspect of keeping track of behavioural patterns is knowing how much of the pattern has been detected thus far, because without keeping track of what happened thus far, it is impossible to detect patterns at all. Detecting patterns is all about recording what is encountered until the complete pattern has been encountered. In order to test whether an implementation violates a certain Vibes constraint, it must be possible to detect and record what has happened before.

Traditionally pattern detection has been carried out using Finite State Machines (FSMs). A FSM consists of a collection of states and transitions between them. A specialisation of an FSM is a Labeled Transition System (LTS)¹, in which the transitions between states are labeled.

¹For a more thorough introduction of FSMs and LTS's, see [31]

Although LTS's can be used to keep track of the state of a Vibes constraint, they suffer from the same shortcomings as regular expressions and FSM's. LTS's, FSM's and regular expressions all need to have their alphabet defined up front (see Chapter 2). However for keeping track of the state of a constraint this does not pose a real problem, because a Vibes constraint could be transformed into an LTS every time it is used for runtime pattern detection. Because it is transformed every time it is needed, the transformation could use all available runtime type information to create a LTS with a correct alphabet.

Instead of using LTS's, in the original work on Vibes[14], an extension to LTS's is proposed that does not require a complete finite alphabet. This extension is called a Deterministic Abstract Recogniser (DAR), which builds on the concepts introduced in the Vibes specification language, especially context sensitive wildcards.

Definition 5.2.1 A *Deterministic Abstract Recogniser (DAR)* D is given by:

$$D = \langle Q, \Sigma_a, \delta, q_0, F, \Xi, \eta \rangle$$

where:

- $Q = \Omega \cup \{q_t\}$ is a finite set of states, with Ω the set of user-defined states, q_t the default trap state and $q_t \notin \Omega$. A trap state is a non final state from which it is not possible to reach a final state.
- $\Sigma_a = \Sigma_b \cup \{\#\}$ is the abstract input alphabet, where Σ_b is a finite set of symbols such that $\# \notin \Sigma_b$. Σ_b is called the base input alphabet. $\#$ is a reserved symbol that stands for all symbols not defined in the base alphabet.
- $\delta : Q \times \Sigma_a \rightarrow Q$ is a total function that is called the transition function. $\forall a \in \Sigma_a (\delta(q_t, a) = q_t)$, which defines the behaviour of the default trap state.
- $q_0 \in \Omega$ is the initial state
- $F \subseteq \Omega$ is the set of final states
- Ξ is the scope expression and is used to specify on what resource the constraint should be applied.
- η is the name of D .

A Deterministic Abstract Recogniser (DAR) is a deterministic finite state machine with an additional *scope expression*, name and an extra symbol $\#$ that matches for any symbol not in the base alphabet.

Definition 5.2.2 The *base alphabet* of a DAR with alphabet Σ is:

$$\Sigma_{\text{BASE}} = \Sigma \setminus \{\#\}$$

At first glance the distinction between the \$ symbol in a Vibes specification and the # symbol in a DAR is not clear, but there is a difference between the two. The \$ symbol represents a context sensitive transition, whose meaning depends on other transitions originating in the same state, and the # label is used for all inputs not part of the alphabet of the DAR. In other words, the # symbol is a single symbol that is part of the alphabet, where the \$ symbol can represent multiple symbols from the alphabet.

A DAR can be directly created from a Vibes specification. For clarity this process is outlined using an example. In this example DAR will be created of the Vibes specification of Figure 5.1, which constraints *ProtectedResource*'s in such way that a *write* operation should always be preceded by a *read* operation.

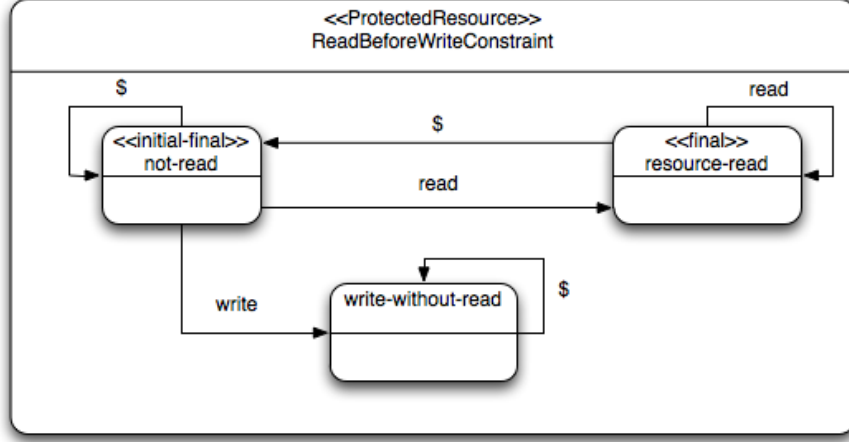


Figure 5.1: Constraint on *ProtectedResources* stating that a *write* operation should always be preceded by a *read* operation.

The first step of creating a DAR from a Vibes specification consists of initialising the new DAR. This is done by setting the name and the scope expression of the DAR to the name and the scope expression of the Vibes specification. Further, we initialise the set of states and initialise the alphabet to be the alphabet of the Vibes specification with the added $\#$ symbol. The transition function is not defined yet. To be more precise, the DAR M^e (the superscript e is used to denote the example DAR) is defined as $M^e = \langle Q^e, \Sigma_a^e, \delta^e, q_0, F^e, \Xi^e, \eta^e \rangle$, where

- $Q^e = \Omega^e \cup \{q_t\}$ and $\Omega^e = \{not-read\}$ (Initialise the set of states to only contain a start and a trap state).
- $\Sigma_a^e = \Sigma_b^e \cup \{\#\}$ and $\Sigma_b^e = \emptyset$ (Initialise the alphabet to be the empty *base* alphabet and the $\#$ symbol)
- δ^e is not defined yet (transition function is not defined yet).
- $F^e = \emptyset$ (Initialise set of final states to the empty set).
- The scope expression of the DAR is $\Xi^e = ProtectedResource$
- The name of the DAR is $\eta^e = ReadBeforeWriteConstraint$

Transforming a Vibes specification into a DAR, is nothing more than creating states and defining transitions between these states. The states are the same as in the Vibes specifications, and the transitions are taken from the Vibes specification as well, with the exception of the $\$$ -transitions. The $\$$ -transitions are expanded into $\#$ -transitions, which means that a new transition with the same source and destination nodes of the $\$$ -transition is added. The label of this transition will be $\#$. Then for each symbol for which no outgoing transition is defined from the source state of the $\$$ -transition, a new transition is added from that source state to the destination state of the $\$$ transition. As an example look at Figure 5.2, in which the Vibes constraint of Figure 5.1 was translated into a DAR.

The DAR of Figure 5.2 is a state machine that can be used to keep track of patterns detected thus far. Keeping track of a pattern now consists of starting in the initial state *not-read* and for each operation that is performed on the resource constrained by this DAR, we take the corresponding transition. If the transition is not defined, follow the

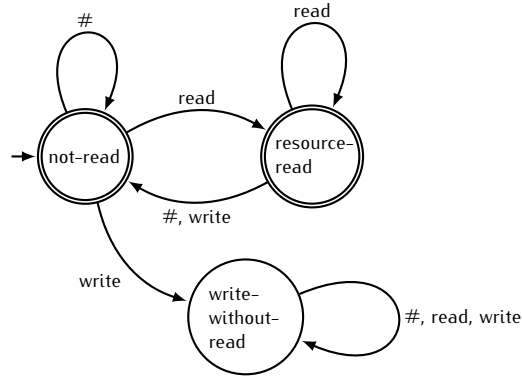


Figure 5.2: Deterministic Abstract Recognisers after the final step.

transition. When the DAR is in a state from which it is not possible to reach a final state, the constraint represented by the DAR is violated.

To actually keep track of the pattern, the current state of the DAR is stored. The current state of the DAR represents the current state of the constraint and thus how much of the pattern has been encountered so far. When a new symbol is encountered, the corresponding transition from the current state will be taken, and the current state will be updated to the target state of this transition.

Verifying Vibes constraints now consists of first creating a corresponding DAR and then following transitions when operations on the constrained Resources are performed. However, a method to bind a DAR to a runtime object has not been defined yet; this will be defined in the following section.

5.3 Resource designation

Being able to record how much of a pattern has been recognised so far is useful, but in order to use constraints on Resource Operations, there must be a method to define what runtime entities are defined as resources.

In [26], Nagy *et al.* claim that it is important to decouple semantic information (called design information in [26]) from the code itself. This makes it possible to reason about the code in ways not anticipated by the original developer of the code. This possibility is important, because a developer cannot be expected to know in advance how the code he develops will be used in the future or how requirements for certain parts may change. Therefore, Nagy reasons that it is important that semantic information is as decoupled from the code as possible, preferably by using some sort of superimposition mechanism to impose design information on the existing code. This superimposition mechanism is proposed especially for crosscutting annotations, i.e. annotations that should be applied on multiple places in the source code.

This reasoning led us to believe that in the context of resource operation verification it is important to decouple the code from what is to be defined as a resource. Just as with design information, a developer cannot be expected to know in advance in what possible ways his code will be regarded as a resource. This means that it is necessary that there is a way to designate pieces of code as resources.

To this end we propose the use of *meta information* to specify that a piece of code is to be regarded as a resource. A common way to define meta information in code is using an annotation system, with which pieces of code are annotated. However, we are not proponents of annotation models of for instance Java [27] or .Net (called *Custom Attributes*) [33], because we feel, like Nagy *et al.*, that meta information and

code should be decoupled as much as possible. In Java and in .Net the annotations are specified within the source code, coupling the code with the semantic information and making it hard to see whether an annotation crosscuts multiple runtime entities.

We therefore propose a completely decoupled meta information system in which it is possible to attach a resource name to classes. Using this decoupled system it is possible to state that an instance of a class is to be regarded as a certain resource and thus let it be subject to verification.

We propose a meta information system with which it is possible to superimpose meta information onto classes. This makes it possible to specify the meta information in one central place, without having to change the implementation. The superimposition could be achieved by using pointcut expressions such as for instance in AspectJ [1] or like the expressions in Compose* [2]. Using these approaches it is possible to select classes using the pointcut expressions and superimpose the meta information. This approach would work well in cases where the meta information should be available at compile time, because the meta information could for instance be added to the Abstract Syntax Tree (AST). For cases where the meta information should only be available at runtime the approach could be different. In this case a simple dictionary would suffice, in which classes are mapped to meta information. In both cases the meta information system is decoupled from the source code and can be specified in a modularised fashion.

The meta information system can be developed as a stand alone component, with methods to add and remove meta information. This meta information is stored in a table structure, in which classes are mapped to so called *class* meta information. First of all this meta information holds the name of a resource as which the class can be regarded. This class meta information also holds a table mapping method names to resource operation names. This way it is possible to regard a given method as a particular operation on the given resource.

Using this proposed meta information system it is possible to designate objects in the code as being Resources, and thus creating a Resource Operations Model.

5.4 Tying it together

In the previous sections we have proposed solutions to keep track of patterns and to designating objects in the code as resources. However, these two solutions in itself are not enough to facilitate pattern detection at runtime. For this to work there must be a way to tie the two together. The resources have to be coupled to Deterministic Abstract Recognisers that keep track of the patterns.

Therefore we propose to use a central book keeping facility that maps resources to the DARs used to detect the patterns. However, this is not a trivial task. If a straightforward mapping would be made from resources to DARs, this would introduce problems. The problem lies in the fact that a difference should be made between the concept of a resource and an instance of a resource.

For instance, a buffer can be seen as a “collection” resource. The abstraction of the buffer is the concept of a collection. However, keeping track of usage patterns cannot be done on the concept of a resource. This would introduce problems if multiple different objects are abstracted into the same conceptual resource. As an example think about a vector that is also abstracted into a collection resource.

Take a look at Figure 5.3, in which these abstractions are depicted. When we do not make a difference between a resource concept and a resource instance, what we actually do is define an ‘*is a*’-relationship between the concepts and the objects. This means that operations performed on any of the objects are performed on the resource concept. In the example we can see that both the *add* and the *remove* operation are mapped on the Collection resource. The problem lies therein that it is not possible

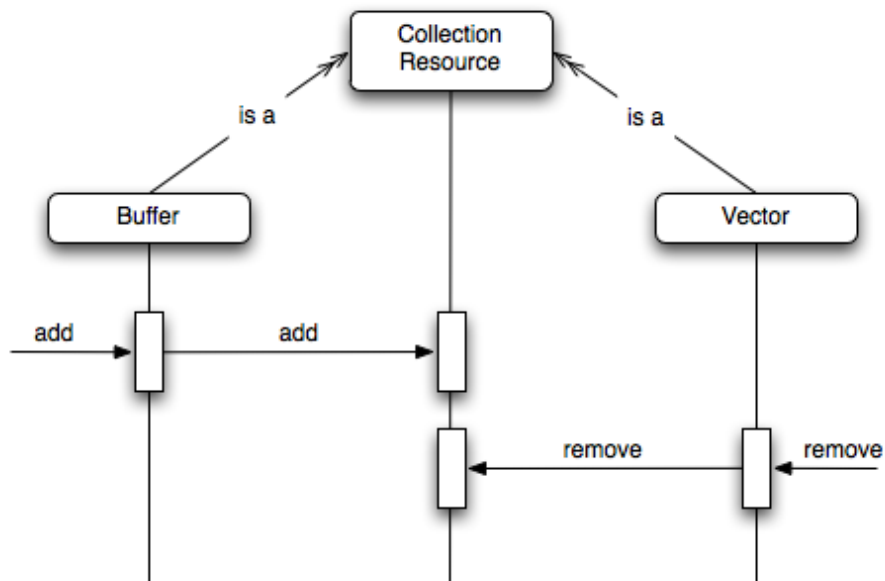


Figure 5.3: Constraining the resource concept.

to detect all constraint violations. Take for instance a constraint that specifies that a *remove* operation should not occur before an object is added to the Collection resource. In the example this constraint is violated, but because we used an '*is a*'-relation, this violation was not detected.

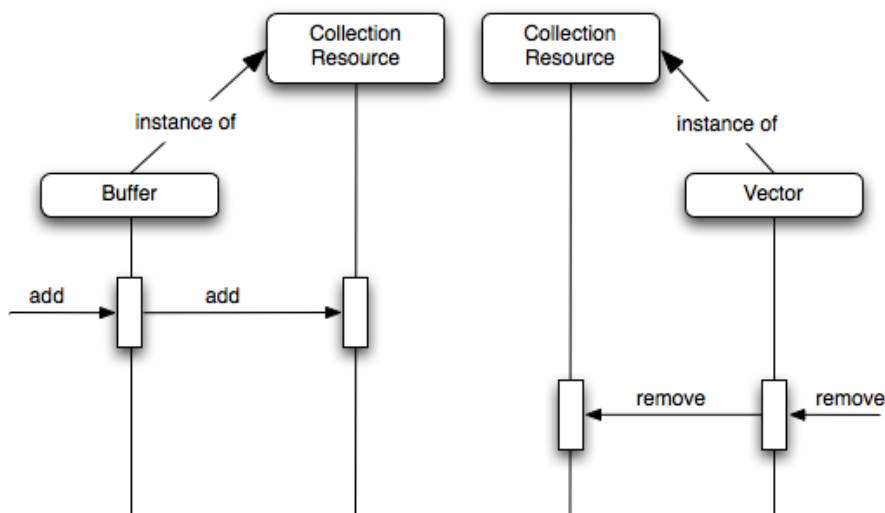


Figure 5.4: Constraining instances of the resource.

If, on the other hand, resource *instances* would be constrained, it is possible to detect these constraint violations. In Figure 5.4 the objects are abstracted by an '*instance of*'-relationship with the Collection resource. When this kind of relationship is used, an instance of the resource is created and constraints are defined on these instances of the resources. This means that the constraint violation from the example can indeed be detected.

Because of this, the central book keeping facility would have to keep a mapping from resource instances to the DAR of the pattern we want to detect. But this mapping would not be from one resource instance to one DAR, but from one resource instance to possibly more DARs, because it is possible that multiple constraints should be applied to a single resource. This results in a mapping from a resource instance to a collection of DARs and every time an operation is performed on the resource instance, this operation is given as input to all DARs mapped onto the resource instance.

We now have a way to keep track of the patterns of a resource instance, but one big problem still needs to be solved. How does the central book keeping facility know that an operation has been performed on a resource instance? In theory it would be possible to tap into the interpreter or virtual machine and keep track of all method calls that happen and then get the target of the method call. This would enable the capturing of operations that are performed on resources, but would require an adapted virtual machine or interpreter. Because the solution of this thesis aims to be as non-intrusive as possible with respect to already existing systems, adapting a virtual machine or interpreter is not preferable.

Another way of keeping track of what method calls are performed on resource instances, is by changing the resource instances, so that whenever a call is made they notify the central bookkeeping facility. Changing the instances however should be automated, because doing it by hand would first require the source code of the system to be available and would secondly be very much work. This is why we propose the usage of Aspect Oriented Programming (AOP) to achieve this. Using any form of AOP would enable us to change the behaviour of the resource instances in such a way that the instances would notify the central bookkeeping facility whenever a method is invoked on them. This could be achieved using AspectJ[1], Composition Filters[?], or even Proxy classes[11].

The problem however, is how to define which classes should have aspects superimposed on them. In AspectJ pointcut expressions are used to superimpose the aspects and in Compose* the superimposition is performed using the *superimposition* block of a concern. The creation of such pointcut expressions or superimposition blocks should be automated to enable pattern detection in dynamically typed languages.

To overcome this final hurdle, we return to the meta information system introduced in the previous section. Using meta information, classes can be designated as resources. The knowledge contained in the meta information system can be used for the superimposition phase as well, because it knows which classes are designated as resources. Because it knows this, it knows on which classes the added behaviour of notifying the central book keeping facility should be superimposed. This knowledge can be used to automatically create superimposition directives for Compose* or pointcut expressions for AspectJ.

For instance, consider a system in which annotations are used to add the meta information, which are super imposed using the following AspectJ pointcut expressions:

```
1 declare @type: Vector: @CollectionResource;
```

Listing 5.1: Pointcut expression to annotate the Vector class as a CollectionResource

This would inject the annotation information (meta information) into the Abstract Syntax Tree, and using this pointcut expression we know that a Vector should be regarded as a resource. Therefore we know that to all methods of the Vector behaviour should be added that informs the central bookkeeping facility that the method was called. This behaviour could be added using the following AspectJ aspect:

```

1 public aspect CollectionNotificationAspect {
2     protected CentralFacility centralFacility =
3         CentralFacility.getInstance();
4
5     pointcut methodCall()
6         : execution( * (@CollectionResource *).*(..));
7
8     Object around(): methodCall() {
9         centralFacility.beforeMethodCall(...);
10        Object returnedObject = proceed();
11        centralFacility.afterMethodCall(...);
12        return returnedObject;
13    }
14 }

```

Listing 5.2: AspectJ aspect adding behaviour that notifies the central bookkeeping facility when a method is called on a resource

This aspect first defines a pointcut based on the `CollectionResource` annotation and then defines an *around* advice. This means that the code in this advice is executed when a method call is made.

The behaviour that should be superimposed on the resource instances consists only of adding a notification sending mechanism to the central bookkeeping facility as the first line that is executed in all methods defined in the class. The result of this is that the central book keeping facility receives notifications whenever a method is invoked on the instance of the class. Information that should be added to the notification consists at least of the object on which the method was invoked and the method signature.

The central bookkeeping facility has a dictionary that maps object instances to the DARs that constrain the object. The first time a notification is sent to the central book keeping facility, it tries to lookup the collection of DARs for the object that notified it. If this does not exist, it means that this collection should be created. This is done by looking up the resources in which the object got abstracted. Once the resources are known, all constraints for those resources are looked up and the instances of corresponding constraint DARs are created and added to the collection of DARs for that object. Then each DAR is offered the method signature of the method call for consumption, or if applicable, the resource operation as which the method was abstracted using meta information for methods.

Later invocations of methods of the same resource instance result in that the collection of DARs is retrieved and all are offered the operation as input. When one of the DARs reaches a trap state (i.e. a state from which it is not possible to reach a final state), the corresponding constraint was violated and an alert can be generated to inform the user.

5.5 Multiple constraints

The central book keeping facility can be used to verify whether constraints on resources are violated at runtime, but there is one problem. As explained, per resource instance, the central book keeping facility holds a list of DARs representing the constraints on that resource instance. Every time an operation is performed on that resource instance, each DAR has to be given the operation as input. Depending on the amount of constraints concurrently imposed on the resource, this could slow down the application considerably, because each DAR has to be looked up, and has to react on the input (advance the DAR) and for every new state it has to be checked whether it is a trap state (i.e. a non-final state from which it is not possible to reach a final state).

Although the calculation of trap states could be moved upfront, for instance when a DAR is specified, this does not remove the need for looking up and advancing each DAR constraining the resource.

This slowdown is unwanted and in essence unnecessary, because it could be solved by combining all DARs constraining a resource into one DAR. Although the combination of DARs results in a bigger DAR (in terms of number of states) than the original DARs, the central bookkeeping facility only has to keep a reference to this DAR per resource instance. This DAR would be the only one that has to be offered the operation as an input, and because the current state would be known, the advancement of the DAR would only consist of looking up the next state. Moreover, per performed operation there would only be need for one test to see whether the new state is a trap state, or, when the calculation of trap states is performed at forehand, this trap state calculation has only have to happen once.

Combining multiple DARs consists of several steps, which are all outlined in [14], and which are given in Appendix A for completeness as well. In [14] DARs are combined to test whether multiple constraints are consistent. This consistency test is performed to make sure that certain constraints to not cause other constraints to be violated.

This consistency check is the most compelling reason to combine DARs for the verification as well. Because the DARs have to be combined for the consistency check, using a combined DAR would not introduce new calculations, and would not slow down the process.

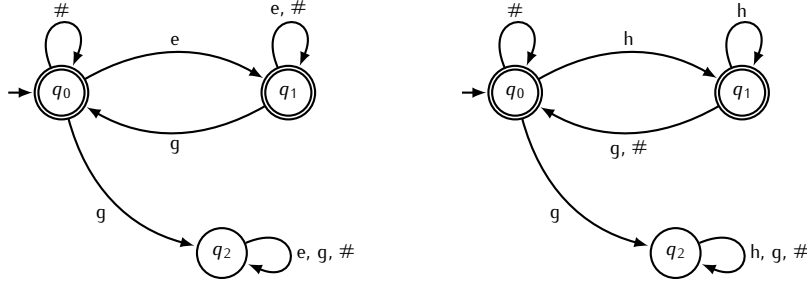


Figure 5.5: Deterministic Abstract Recognisers of the `e_somewhere_before_g` and the `h_directly_before_g` constraints.

However, combining constraints does introduce a problem. When a trap state is reached in the combined DAR, it is not possible to determine which original DAR reached the trap state. See for instance Figures 5.5 and 5.6. The latter represents the DAR that is a result of combining the two DARs of Figure 5.5. From the two DARs in Figure 5.5 it can be concluded that when the input sequence `e g` is given, this poses no problem for the left DAR, but the right DAR will end up in the trap state (for input `e` the `#`-transition is taken).

If we now look to the combination of DARs in Figure 5.6, we still see that the input sequence `e g` still causes the DAR to end up in a trap state, but it is impossible to determine which of the original DARs was violated. So the fact that a constraint violation occurred was detected, but there is no way to find out what precise constraint was detected.

A solution for this would be to keep references to the states of the original DARs after the DARs have been combined into one. This way it would be possible to detect which DARs were violated. However, when the algorithm from Appendix A is used, this solution would not work. This because the last step of this algorithm consists of reducing states by identifying equivalent states and combining them into one state. This often means that the trap states from the original DARs are combined into one

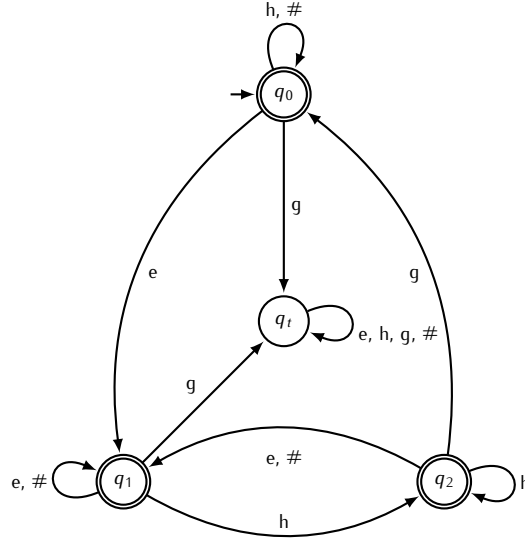


Figure 5.6: Combination of the DARs of Figure 5.5.

trap state. Even if this single trap state would have references to the original trap states, it would have references to all trap states of all original DARs and it would still not be possible to determine which constraint was violated.

Therefore we propose to use a combination of two approaches: combine the DARs (already needed for checking the consistency of multiple constraints), but also keep references to the original DARs. Next to this we keep a trail of encountered inputs, which is used as an input to the separate original DARs when a constraint violation was detected. The DAR(s) that generate a constraint violation, will indicate which constraints were violated and what input sequence caused it.

This hybrid solution performs better when no constraint violations are detected, but is somewhat slower when a constraint violation is detected, because it will search for the violated constraint. Because most of the time a constraint violation marks the end of a run, we feel that the improved performance during normal operation outweighs the somewhat reduced performance of finding violated constraints.

5.6 Summary

With dynamically typed languages the AST is of no use to perform behavioural pattern detection, because the AST does not contain enough type information. Therefore another method has to be used. In this chapter we have proposed a solution with which it is possible to detect behavioural patterns at runtime. For this solution, first a method to capture the state of the pattern has been proposed in the form of Deterministic Abstract Recognisers (DARs). DARs have been introduced in the original work on Vibes [14], and are essentially Labeled Transition Systems with an added wildcard symbol. Using DARs, a Vibes constraint can be represented in a mathematically sound manner.

The next step in the solution involves designating pieces of code as resources. For this we have introduced a meta information system, with which it is possible to designate resources. The meta information system is preferably as decoupled as possible from the original system on which to detect patterns [26], because one does not always have the source code available, and code can be used in other ways than anticipated when the code was written.

The final part of the solution consists of creating a central bookkeeping facility, which is signalled whenever an operation on a resource is performed. To enable the signalling, the as resource designated classes have signalling functionality superimposed on them by a form of Aspect Oriented Programming.

In Chapter 2 we have given a diagram showing how a conceptual constraint verification system for Resource Operations Models would look like. This diagram is given again in 5.7.

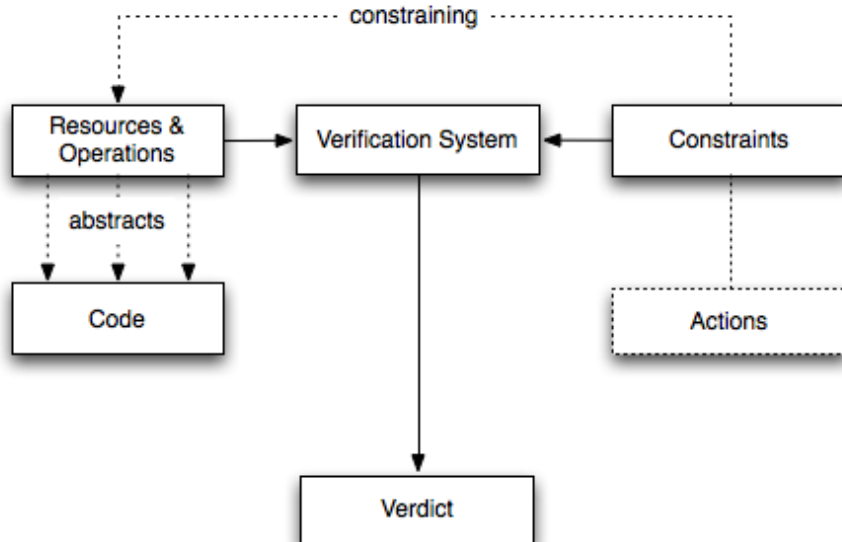


Figure 5.7: Ideal system that can be used to verify constraints on Resource Operations Models.

If we look at an overview of the runtime verification system described in this chapter (given in 5.8), we can clearly map the two diagrams. The Verification System is implemented by the Central Bookkeeping Facility, which takes Constraints and Resources and Operations as inputs to come to a Verdict. The Central Bookkeeping Facility however, takes one extra input, the Ordering Constraints, to define an order in which the Actions are executed. The Ordering Constraints are not depicted in Figure 5.7, because the need to order actions of constraints did not arise as a problem before. The abstraction of the code into a Resource Operations Model is performed by means of the Meta Information system also introduced in this chapter.

We have also shown a method to increase the runtime performance of the solution, but have also shown that with this performance increase loss of detail is introduced, which makes reporting on the cause of constraint violations harder. Therefore we proposed the use of a hybrid solution that performs better when no constraint violations are detected, but which performs a bit less when a constraint is violated.

The complete solution enables behavioural pattern detection at runtime, and can therefore be used for dynamically typed languages as well. Although the system has been created with dynamically typed languages and runtime verification in mind, it is certainly possible to use these ideas to perform runtime verification of statically typed languages as well.

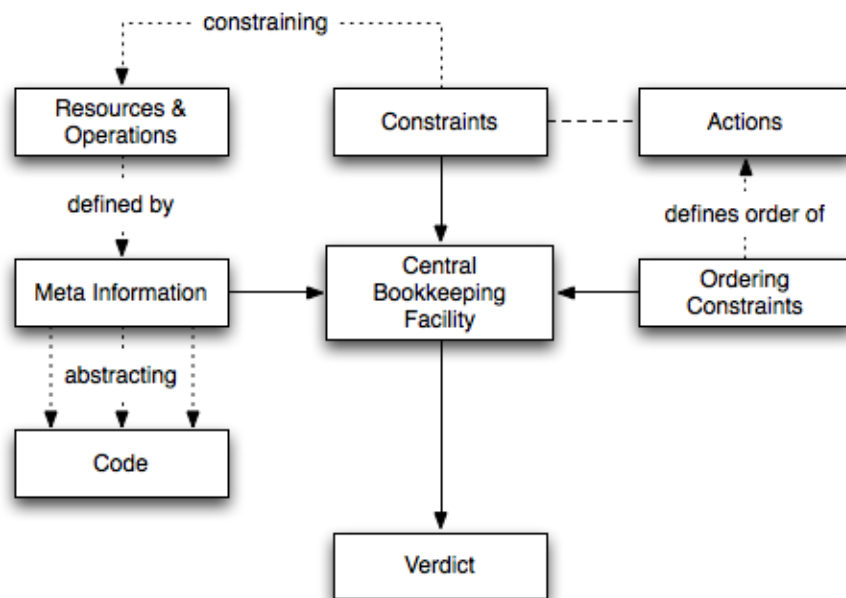


Figure 5.8: Proposed system that can be used to verify constraints on Resource Operations Models for dynamically typed languages.

Chapter 6

Implementation in Smalltalk

To demonstrate that the proposed techniques can be implemented and used, an implementation of the solutions described in this thesis has been made in Smalltalk. In it DARs have been implemented, as well as an annotation system and a central bookkeeping facility. It also incorporates the ideas from Chapter 4, so it is possible to add actions and to specify orderings in which these actions should be performed.

This chapter explains how the solution outlined in this thesis was implemented in Smalltalk. This chapter begins by explaining the choice for Smalltalk, after which an overview is given of the implementation of DARs, the meta information system and the central bookkeeping facility.

This chapter only focusses on the high-level implementation of the solutions, a more in-depth overview of the implementation can be found in the form of an API specification in Appendix B.

6.1 Why Smalltalk?

Because we want to demonstrate that the techniques can be used and implemented in a dynamically typed language, an implementation should be created in a dynamically typed languages such as Python, Lisp, Ruby or Smalltalk. From these languages we have chosen the latter, because of its large research heritage. Although Lisp has been used often in research, we wanted a language that is easy to adopt, but still is quite readable for people without intimate knowledge of the language. This is one of the reasons why we have chosen Smalltalk as the implementation language for this thesis. Another reason is that Smalltalk is still being used as a research language, for instance for topics like reflection [30][16] and runtime bytecode adaptation [8][32].

6.2 Deterministic Abstract Recognisers

In Chapter 5 the need to keep track of patterns was explained, and the usage of Deterministic Abstract Recognisers (DARs) was proposed. Remember that a DAR is in essence a Labeled Transition Systems with an added #-symbol, representing all symbols not in the base alphabet. Although a DAR also has a scope expression and a name, in the implementation we have chosen to move the scope expression and name to a more abstract Constraint object. The DAR can then be implemented just like a LTS would be implemented, but with added transitions for the #-transitions. The DAR contains a collection of states and each of these states has a dictionary mapping a symbol to a new transition. This dictionary is used to represent the collection of transitions originating from that state. Apart from a collection of states, the DAR holds

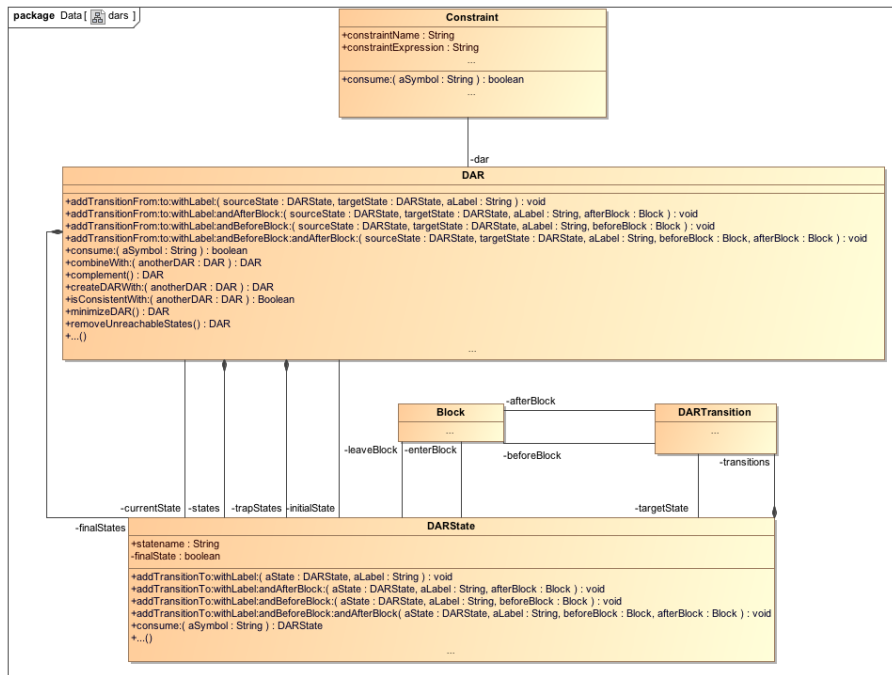


Figure 6.1: Implementation of DARS

references to the initial state, has a collection of final states and a collection of trap states. This latter collection can be calculated automatically.

The DAR class has methods to add new states and transitions with a given label and it is also possible to give a Block to execute before or after a transition is made or when a state is entered or left. Blocks are a Smalltalk specific implementation of Lambda expressions[7], and can be used as anonymous functions which can be passed around. This can also be implemented in languages without support for Lambda expressions by using the Command pattern [11].

When a transition is added to a DAR, first the source state is looked up. If this state cannot be found, the state is added to the collection of states of the DAR. For the destination state the process is analogous. After this a new transition object is created. A transition object consists of the label of the transition, the destination state and optional Blocks that are executed either before or after the transition is taken. After initialisation the transition is added to the transition dictionary of the source state.

A DAR makes a transition when its *consume*: method is called, with the name of the operation as a parameter. The DAR class looks up its current state, and looks up the transition belonging to the operation name. Then the transition is made to the target state of the transition, by setting the *currentState* variable of the DAR to the new state.

The DAR class also contains some methods that aid in determining whether two DARS are consistent with each other. The consistency test consists of first testing whether the original DARS are consistent (i.e. can reach a final state) and then combining the two DARS and testing whether a final state is reachable in the combined DAR. The algorithm to combine two DARS consists of several steps (explained in Appendix A) which are represented by the helper methods *complementDAR*, *createDARWith*: and *minimizeDAR*. These combinations are also performed to create one DAR per

resource instance.

6.3 Meta information

In the previous chapter we explained that we needed a way to designate which objects are to be regarded as resources. Following the argumentation of the work of Nagy[26], we stated that it is important to decouple this resource designation information from the code, because developers cannot be expected to know how their code will be used in the future. We therefore proposed the use of a meta information system that is completely decoupled from the source code, which can be used to attach meta information to objects and methods to designate them as resources and operations respectively.

Note that for historic reasons, that what is called meta information in Chapter 5, is called Annotations in this implementation.

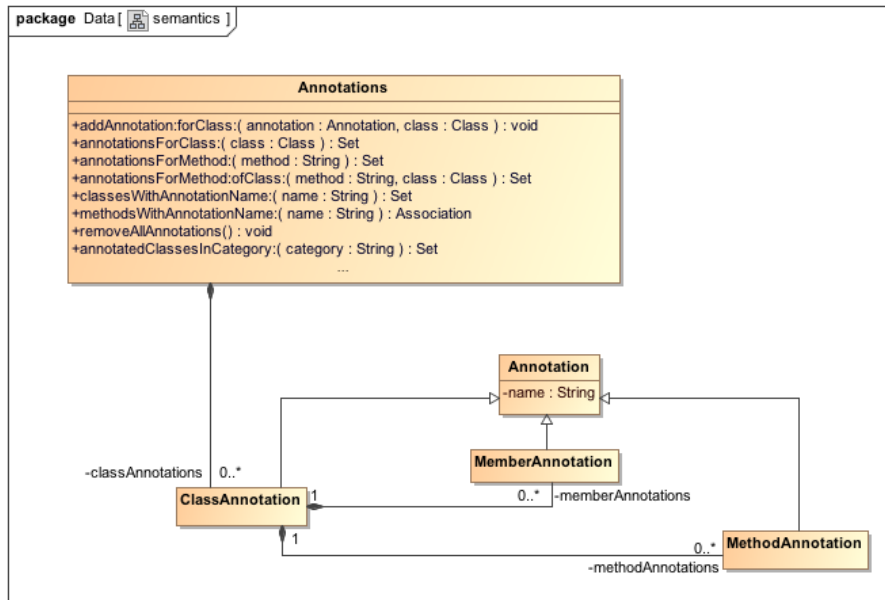


Figure 6.2: Overview of the meta information model

If a class is designated as a Resource, it can be the case that its methods have meta information as well. Methods can have meta information in this model, because it can be the case that a method name in a class should be regarded as a resource operation with another name. Using meta information for methods, the mapping from method to operation name can be decoupled in a similar way as the normal (class based) meta information.

However, using the Resource Operations Model it can happen that a method with name x is to be regarded as a resource operation y . However, in the constraint an operation named x could be defined as well. The question in this case is what should happen if a message is sent to the method with name x . Should this result in the offering of operation y to the DAR or should just x be offered?

In the implementation we have chosen for the following approach. If a method call is performed, first it is checked if meta information is defined for the method. If this is the case, the symbol for the method is replaced by the operation name from this meta information. This operation name is then offered as input to the DAR corresponding to the constraint. This approach was favoured, because we feel that in a Resource

Operations model, the operation names used in the model are more important than the names in the original implementation, because the operation name represents the name of a resource operation.

6.4 Constraints

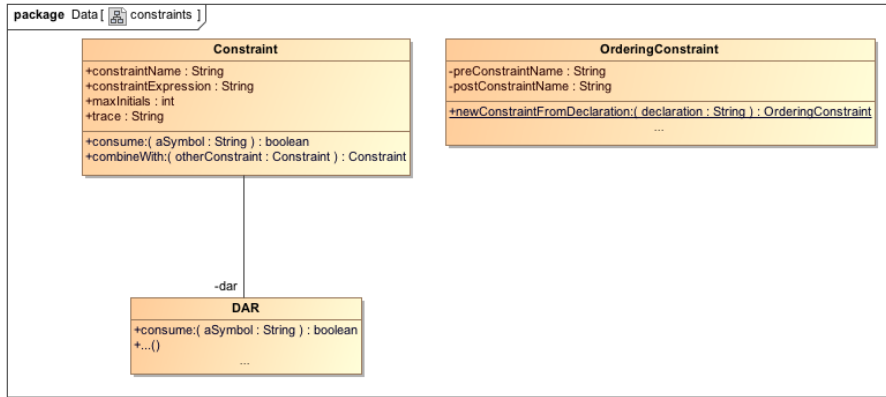


Figure 6.3: Overview of constraints in the implementation

Within the verification framework proposed in this thesis there are two different sorts of constraints. One of them is given by the Vibes constraints, which are used for the actual verification of systems and the second form of constraints is given by ordering constraints. These constraints are used to specify a partial order in which actions added to states or transitions should be performed. Both constraints are given in the class diagram of Figure 6.3.

The Vibes constraints are implemented by the *Constraint* class, which has a field for the name of the constraint, a field for the scope expression, a field to keep track of what inputs have been encountered and a field which represents the maximum number of times the initial state can be encountered before the trace is reset. This last field can be used to limit to the length of the trace. This is used in cases where the trace can grow very long, but encounters the initial states many times. Because encountering the initial state again can be interpreted as beginning a new trace, it is possible to reset the trace after a specified amount of initial state encounters.

The *Constraint* class also contains a reference to a *DAR* class. This instance variable is used to keep track of the state of the constraint.

When a message is sent to the *consume*: method of the *Constraint* class, the *Constraint* instance delegates this message to its internal *DAR* instance. The *Constraint* class is actually a wrapper around a *DAR* instance which holds extra information that is important for the Vibes constraint.

The *Constraint* class also has a method called *combineWith*:. This method is used to combine the current *Constraint* with the supplied one. This is done by creating a new *Constraint* and let its *dar* field reference the *DAR* that is a result of the combination of the original *DARs* from the original *Constraints*. The name of this new *Constraint* consists of the concatenation of the names of the original constraints, and the expression is set to be the disjunction of the two original expressions.

6.5 Ordering constraints

In section 4.3, we proposed a method to deal with situations in which multiple actions should be performed simultaneous. To this end we looked at solutions dealing with ordering constraints at shared joinpoints in the field of Aspect Oriented Programming [25]. Inspired by this work, we proposed the use of the $pre(A, B)$ operation. This operator can be used to specify that action A should be performed before action B is performed. As a convenience, we also proposed the $C_1 < C_2$ construct, that can be used to specify that all actions from constraint C_1 should be performed before all actions of constraint C_2 .

The results of [25] cannot be used to the fullest extent in the context of this thesis, because of how multiple Vibes constraints are combined to form one constraint per resource instance (see section 5.5, in which we proposed to use a hybrid solution for the verification). Because the Vibes constraints are combined, it is not possible anymore to determine which action originally belonged to a certain Vibes constraint. This information is lost when the DARs of the constraints are combined.

However, in the creation of the combination of DARs lies the key to the ordering of actions. To indicate a combination of two DARs we will use the $\oplus$ operator. $D_1 \oplus D_2$ means that D_1 is combined with D_2 . Now consider that this operation is symmetric when no actions are involved, meaning that $D_1 \oplus D_2 = D_2 \oplus D_1$. This behaviour is what we would expect when no actions are considered; the order in which the DARs are combined does not make any difference for the outcome of the operation.

Now let us consider the case in which actions are considered. We can alter the meaning of $D_1 \oplus D_2$ and say that it means that D_1 and D_2 are combined, *and* that the actions of D_1 are always performed before the actions of D_2 . This small adaptation results in $D_1 \oplus D_2 \neq D_2 \oplus D_1$.

If we now extend this to multiple Vibes constraints with multiple actions all performed on the same resource, we are able to specify the ordering of the actions by the order in which the DARs of the constraints are composed. However, to specify the order of composition we need an operator. This is why we introduce the $<$ operator. $C_1 < C_2$ means that the actions of constraint C_1 are always performed before the actions of constraint C_2 . In terms of $\oplus$ this means:

$$D_{C_1} \oplus \dots \oplus D_{C_2}$$

To find an ordering that is consistent with the partial ordering, a so-called dependency graph is used. In this graph the constraints are modelled by nodes and the partial ordering given by the $<$ operator are modelled by edges. When $C_1 < C_2$, an edge is added with C_2 as a source node and C_1 as a target node, to indicate that C_2 is dependent on C_1 . Once this graph is fully created, with independent constraints only dependent on an extra inserted start node, an ordering can be created. This is done by first marking the start node as completed. After this, the first node of which all its parents are marked as completed, is marked. This procedure either stops when all nodes are marked, or when such an ordering cannot be found. In the former case, the order in which the nodes are marked gives an ordering that is consistent with the partial ordering. In the latter case the algorithm cannot find an ordering, because not all nodes could be marked. If it is not possible to mark all nodes, it is an indication that there is a circular dependency somewhere in the graph, and thus somewhere in the specifications.

In the implementation, the *OrderingConstraint* class is merely a placeholder for the information that is contained in an ordering constraint. It has two fields: the *preConstraintName* and the *postConstraintName* and they both are used to denote

the two parts of an ordering constraint:

$$preConstraintName < postConstraintName$$

The *OrderingConstraint* class has a class method (also known as a *static* method) to create a new instance: *newConstraintFromDeclaration*:. This method can be used to create a new *OrderingConstraint* object by specifying the constraint declarative:

```
OrderingConstraint newConstraintFromDeclaration:
    'ReadBeforeWriteConstraint < AuthenticateWriteConstraint'.
```

6.6 SuperVisor

In 5.4 the observation was made that although a solution was found to keeping track of the state of a constraint and to designating objects as resources, a method to tie these solutions together to form an actual verification system was still lacking. As a solution, we proposed to use a central bookkeeping facility, which keeps track of the states of resource instance constraints and which is notified whenever an operation is performed on a constrained resource. This central bookkeeping facility is called *SuperVisor* in the implementation and it is responsible for tying together the annotation system and the constraint specifications.

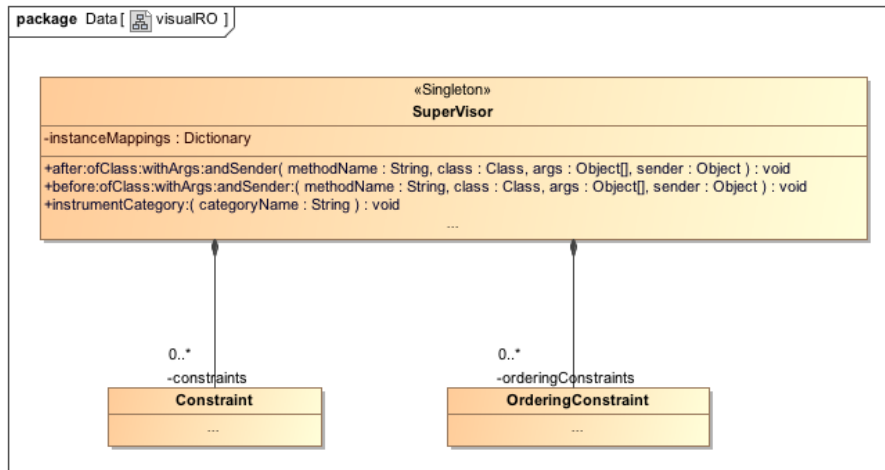


Figure 6.4: Overview of the SuperVisor implementation

The *SuperVisor* class (Figure 6.4) lies at the heart of the constraint verification mechanism. The *SuperVisor* class is a Singleton class [11], which basically means that at all times there is only one instance of this class. This instance keeps track of all constraints, both Vibes constraints and ordering constraints. Once the constraints are added to the *SuperVisor* instance, and the *instrumentCategory*: method is called, the *SuperVisor* collects all annotated classes in the given category and installs hooks on all methods of these classes. These hooks are used before or after a method in an instrumented class is called and the hooks call either the *before:ofClass:withArgs:andSender*: or the *after:ofClass:withArgs:andSender*: method of the instance of the *SuperVisor* class.

When the *before:ofClass:withArgs:andSender*: method is first called for a given instance of a resource, all constraints that have been defined for that resource are composed into one DAR. This DAR is then stored for that particular instance of the

resource and the Blocks that should be executed when the initial state is entered are executed. Every consecutive time this method is called, the composed DAR is retrieved and its current state is looked up and the transition that should be taken for the given operation. When this transition is retrieved, all actions (in the form of Blocks) that are specified to be executed before the transition is taken are executed. After this, the control is given back to the method that was called. When this method has finished his execution, the *after:ofClass:withArgs:andSender:* method is called, the composed DAR is retrieved again and is offered the method name for consumption and the actions that should execute after the transition is made are executed. The DAR consumes the input and if it finds itself in a trap state, the *SuperVisor* knows that a constraint was violated. It then uses the trace of the composed *Constraint* as input for the original constraints to discover which original constraint was violated. Once this is known an exception is thrown and a trace to the constraint violation and the name of the violated constraint is printed.

6.7 Message interception

One important aspect of the runtime verification technique introduced in this thesis is the ability to intercept messages. The runtime verification system is built around the principle that it is possible to be notified just before a message is received and just before a return message is sent back. There are several methods to implement such behaviour, for instance filters in *Compose** [2], *around* advice in AspectJ [1] or using a Meta Object Protocol (MOP) [23].

In this thesis, however, we have decided to use MethodWrappers [5], because they deliver an elegant solution to the message interception problem, especially for Smalltalk. Also, MethodWrappers deliver a solution that can be easily simulated in other programming languages and is not only limited to completely reflective languages. For instance, it is possible to simulate MethodWrappers in Java using Proxy's [11] or by using one of the Aspect Oriented approaches mentioned before.

Before going deep in the details of MethodWrappers, it is good to know how messages are delivered to objects and how method lookup is performed in Smalltalk. In this discussion some organisational areas of Smalltalk are discussed as well.

Because everything in Smalltalk is an Object, methods are represented by Objects as well. To be more precise, a method is an instance of the *CompiledMethod* class. Every class has a member that is called *MethodDictionary*. This is a dictionary with method names as keys and with *CompiledMethods* as values. This means that a *CompiledMethod* does not know anything about the class it belongs to, it could even be the case that the same *CompiledMethod* is used in several classes. When a method is called (a message is sent) to an instance of a class, the *MethodDictionary* is used to look up the corresponding *CompiledMethod*. Then the *CompiledMethod* instance is sent the *valueWithReceiver:arguments:* message. The first argument is the object instance to which the original message was sent and the second argument is an array with the arguments. The *CompiledMethod* uses the object instance to be able to refer to the members of the object on which it performs and the array is used to look up potential arguments. This is depicted in Figure 6.5, in which the relations between a class, its *MethodDictionary* and *CompiledMethods* is shown.

A Method Wrapper [5] is a wrapper around a method. When the method gets called, the wrapper is called instead. This wrapper then gets the possibility to perform certain tasks before, after or instead of the original method.

In Smalltalk this can be easily implemented by replacing the *CompiledMethod* instance in a Method Dictionary by a Method Wrapper instance. When a method call is performed, the correct class is looked up by the Virtual Machine (VM) and the *CompiledMethod* object is retrieved. The VM then sends a *valueWithReceiver:arguments:*

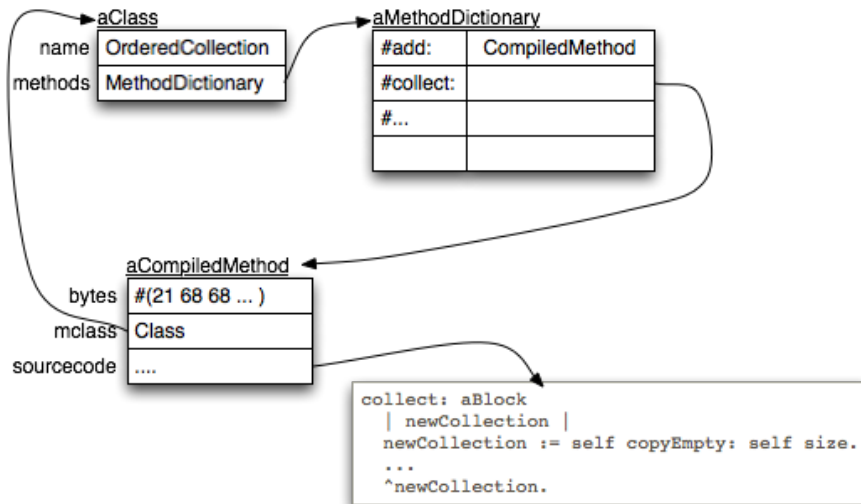


Figure 6.5: Normal method lookup in Smalltalk

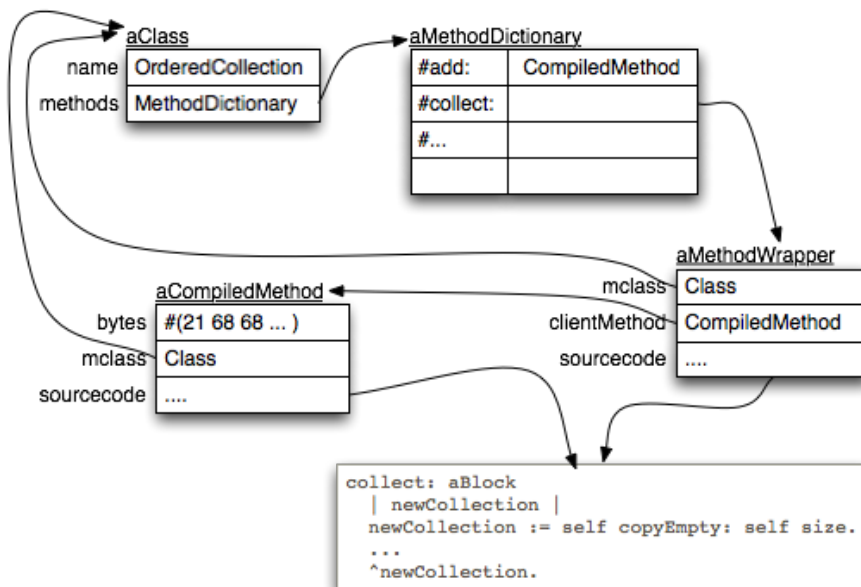


Figure 6.6: Method lookup in Smalltalk using MethodWrappers

message to the object stored in the dictionary.

If a `CompiledMethod` is replaced with a wrapper, the only method that has to be implemented in the wrapper is the *valueWithReceiver:arguments:* method. Within this method the wrapper has the possibility to perform his own tasks before, after or instead of the original method call. By keeping a reference to the original `CompiledMethod` it is still possible to use this method, and even to deinstall the wrapper itself. Method lookup in Smalltalk using `MethodWrappers` is displayed in Figure 6.6, in which is shown that a reference to the original `CompiledMethod` is kept not only for invocation of the original method, but for uninstallation of the `MethodWrapper` as well. When a `MethodWrapper` is uninstalled, the only thing that happens is that the original `CompiledMethod` is put back in the `MethodDictionary` of the class on which the `MethodWrapper` was installed.

6.8 Working out the example

Throughout this thesis we have used several examples of Vibes constraints to show how Vibes can be applied to the Resource Operations Model. In this section we work out the example as used in Chapter 4, where we used two Vibes specifications constraining use of the *write* operations. These two Vibes specifications constrain a *ProtectedResource*, one stating that the *write* operation should always directly be preceded by a *read* operation and the other constraint specifies that somewhere before a *write* operation an *authenticate* operation should be performed. These constraints are given in Figure 6.7.

In this section we will show how these constraints can be used in the implementation of the verification framework. We begin by implementing the constraints themselves, after which we will show how meta information can be attached to classes. After this we show how the code to be verified can be updated using the *SuperVisor* and that the constraints are indeed verified once the code is used. Note that we have chosen to use constraints with actions, to give a complete overview of the implementation.

6.8.1 Creating the constraints

In the implementation we have chosen not to include a translation mechanism to translate visual Vibes constraints to DARs, because we feel that implementing such a translation would fall outside of the scope of this thesis. We therefore use the DARs directly. How a Vibes constraint is translated into a DAR has been outlined in section 5.2.

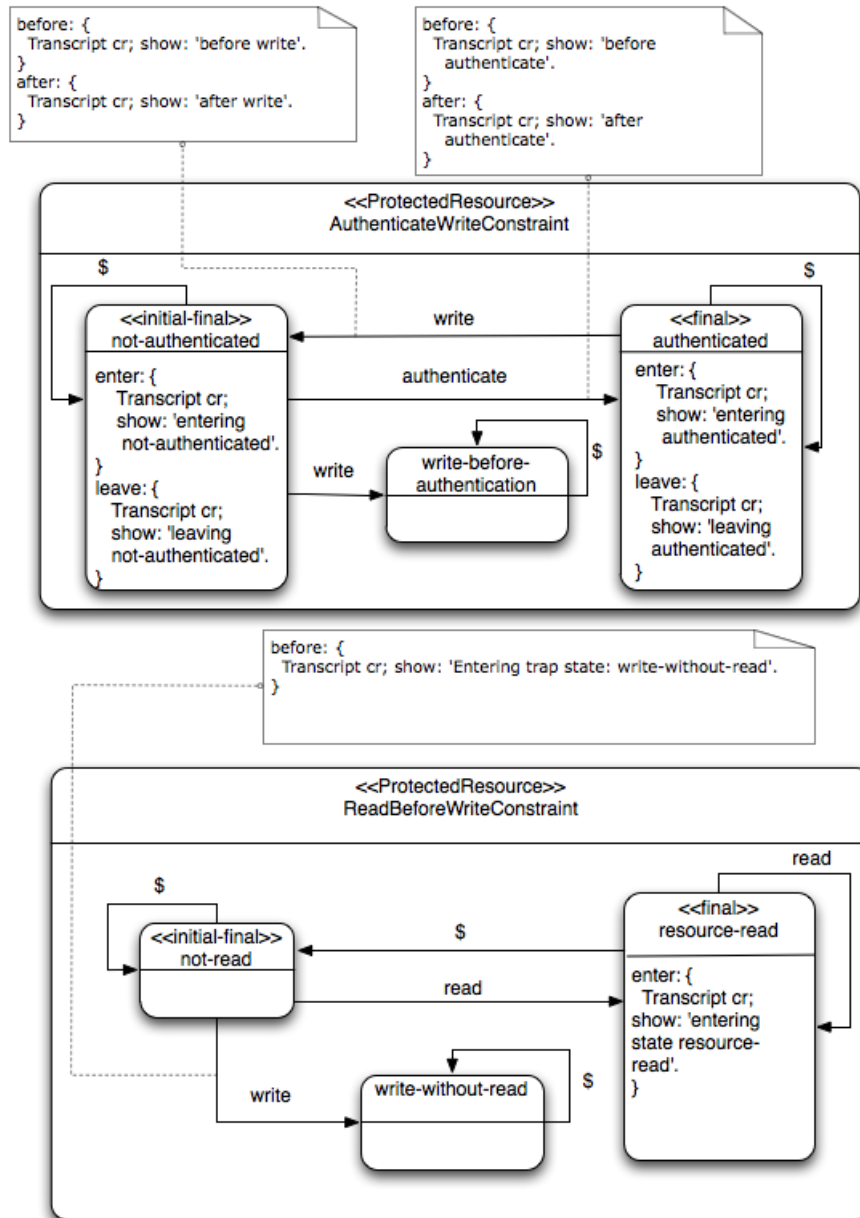


Figure 6.7: Two constraints with actions, both constraining ProtectedResource.

```

1  createReadWrite
2      "This method creates the ReadWriteConstraint"
3      | da notread resourceread writewithoutread
        readwriteconstraint |
4      da <- DAR new.
5
6      notread <- DARState new.
7      notread statename: 'notread'; finalState: true.
8      resourceread <- DARState new.
9      resourceread statename: 'resource-read'; finalState:
        true.
10     resourceread enterBlock: [Transcript cr; show: 'RBW
        Entering resource-read'].
11     writewithoutread <- DARState new.
12     writewithoutread statename: 'write-without-read';
        finalState: false.
13
14     da addTransitionFrom: notread to: notread withLabel:
        '#'.
15     da addTransitionFrom: notread to: resourceread
        withLabel: 'read'.
16     da addTransitionFrom: notread to: writewithoutread
        withLabel: 'write' andBeforeBlock: [Transcript
        cr; show: 'RBW Entering trap state: write-
        without-read'].
17
18     da addTransitionFrom: resourceread to: resourceread
        withLabel: 'read'.
19     da addTransitionFrom: resourceread to: notread
        withLabel: '#'.
20
21     da addTransitionFrom: writewithoutread to:
        writewithoutread withLabel: '#'.
22
23     da initialState: notread.
24     readwriteconstraint <- Constraint new.
25     readwriteconstraint constraintName: '
        ReadBeforeWriteConstraint'; dar: da;
        maxInitials: 3; constraintExpression: '
        ProtectedResource'.
26     ^readwriteconstraint

```

Listing 6.1: Creating the ReadBeforeWrite constraint in the ConstraintHelper class

In Listing 6.1, we have given the implementation of the *createReadWrite* method in the ConstraintHelper class. In this method we create the ReadBeforeWriteConstraint. We begin at line 1 with defining the method itself. Then in line 2 a comment specifying what this method is supposed to do is given, and in line 3 we specify the variables we will use in the rest of the code. Then, in line 4, we create an instance of the DAR class. In line 6, 8 and 11 we create new DARStates, which are given labels and are optionally made final states in lines 7, 9 and 12 respectively. In line 10 we add an action to the enter event of state *resource-read*. In lines 14 to 21 we add transitions to the DAR and we also add a Block to execute before the transition is made in line 16. In line 23 we designate state *not-read* to be the initial state. After this we create a new instance of a Constraint in line 24, and in line 25 we set its name, the DAR that implements the constraint and the scope expression of the constraint.

We also set the *maxInitials* parameter to be 3. This parameter is used to define the maximum amount of time the trace of inputs can encounter the initial state. The trace of inputs is printed whenever a constraint violation is detected, and can be used by the developer for traceability purposes. In line 26 the created constraint is returned.

In Listing 6.2 the implementation of the `AuthenticateBeforeWrite` constraint is given. The implementation of this constraint is similar to the implementation of the `ReadWriteConstraint`, but because it specifies another constraint, the transitions and states are different, and different actions have been added.

```

1  createAuthenticateBeforeWrite
2      "This method creates the AuthenticateBeforeWrite
        constraint"
3      | da notauthenticated authenticated
        writebeforeauthentication constraint |
4      da <- DAR new.
5
6      notauthenticated <- DARState new.
7      notauthenticated statename: 'not-authenticated';
        finalState: true.
8      notauthenticated enterBlock: [Transcript cr; show: '
        AW Entering not-authenticated'].
9      notauthenticated leaveBlock: [Transcript cr; show: '
        AW Leaving not-authenticated'].
10     authenticated <- DARState new.
11     authenticated statename: 'authenticated';
        finalState: true.
12     authenticated enterBlock: [Transcript cr; show: 'AW
        Entering authenticated'].
13     authenticated leaveBlock: [Transcript cr; show: 'AW
        Leaving authenticated'].
14     writebeforeauthentication <- DARState new.
15     writebeforeauthentication statename: 'write-before-
        authentication'; finalState: false.
16
17     da addTransitionFrom: notauthenticated to:
        notauthenticated withLabel: '#'.
18     da addTransitionFrom: notauthenticated to:
        authenticated withLabel: 'authenticate'
        andBeforeBlock: [Transcript cr; show: 'AW Before
        authenticate'] andAfterBlock: [Transcript cr;
        show: 'AW After authenticate'].
19     da addTransitionFrom: notauthenticated to:
        writebeforeauthentication withLabel: 'write'.
20
21     da addTransitionFrom: authenticated to:
        authenticated withLabel: '#'.
22     da addTransitionFrom: authenticated to:
        notauthenticated withLabel: 'write'
        andBeforeBlock: [Transcript cr; show: 'AW Before
        write'] andAfterBlock: [Transcript cr; show: '
        AW After write'].
23
24     da addTransitionFrom: writebeforeauthentication to:
        writebeforeauthentication withLabel: '#'.
25
26     da initialState: notauthenticated.
27
28     constraint <- Constraint new.
29     constraint constraintName: '
        AuthenticateWriteConstraint';
30     dar: da; maxInitials: -1; constraintExpression: '
        ProtectedResource'.
31     ^constraint.

```

Listing 6.2: Creating the AuthenticateBeforeWrite constraint in the ConstraintHelper class

6.8.2 Creating the annotations

Now that we have implemented methods to generate the constraints, it is time to designate a class to be a `ProtectedResource`. For this we will use what in the implementation is called the annotation system. Note that for historic reasons the meta information system introduced in Chapter 5 is called the annotation system in the implementation. In this example we will annotate the *BankAccount* class as a `ProtectedResource`. We have implemented a method that creates the annotations and adds them to the annotation system in Listing 6.3.

```
1  annotateBankAccount
2    "This method annotates the BankAccount class as a
   ProtectedResource"
3    | ca ma |
4    Annotations instance removeAllAnnotations.
5    ca <- ClassAnnotation new.
6    ca annotationName: 'ProtectedResource'.
7    ma <- MethodAnnotation new.
8    ma annotationName: 'read'.
9    ca addMethodAnnotation: ma forMethod: #doSomething.
10   Annotations instance addAnnotation: ca forClass:
      BankAccount.
```

Listing 6.3: Annotating the *BankAccount* class as a `ProtectedResource`

The method begins by removing all existing annotations from the annotations system, by first getting an instance (`Annotations` follows the Singleton pattern [11]) and then calling the *removeAllAnnotations* method in line 4. In line 5 a new *ClassAnnotation* instance is created and in line 6 this annotation is called 'ProtectedResource'. In line 7 we create a new *MethodAnnotation* as well, and in line 8 we call this annotation 'read'. In line 9 we add the method annotation to the class annotation and we say that this is an annotation for the *doSomething* method. From now on, the *doSomething* method is regarded as a *read* operation on the *ProtectedResource*. We then add this annotation to the annotation system for the *BankAccount* class in line 10.

6.8.3 Using the Supervisor to tie it all together

Before we use the *SuperVisor* to verify whether the behaviour of *ProtectedResources* is what we expect, we will first take a look at the *BankAccount* class and the *Bank* class that uses the *BankAccount*. The *BankAccount* class is a very simple class with only three methods: *read*, *write*: and *authenticate*:. The *read* and *write* methods return or set the amount of money in the bank account and the *authenticate* method is used to authenticate the user, the implementation of these methods is given in Listing 6.4. We have added code to display text on the console when the method is called for demonstration purposes.

```

1  read
2      "Return the amount of money in this account"
3      account ifNil: [account <- 100].
4      Transcript cr; show: 'Bankaccount read: ', account
5          asString.
6      ^account.
7
8  write: amount
9      "Set the amount of money in this account"
10     account <- amount.
11     Transcript cr; show: 'Bankaccount written, new value
12         is ', account asString.
13     ^amount.
14
15 authenticate: pin
16     "authenticates the user for this account with the
17         given PIN"
18     (pin = 1234) ifTrue: [Transcript cr; show: '
19         Authentication successful'. ^true ] ifFalse: [^
20         false].

```

Listing 6.4: The *read*, *write*: and *authenticate*: methods of the BankAccount class

The Bank class has two methods, which use the BankAccount. These methods are given in Listing 6.5.

```

1  withdrawAmount: amount from: bankAccount
2      "withdraw the given amount from the given bank account
3      "
4      | currentAmount |
5      currentAmount <- bankAccount read.
6      currentAmount <- bankAccount write: (currentAmount -
7          amount).
8      ^currentAmount.
9
10 authenticateForAccount: bankAccount withPIN: pin
11     "authenticates the user for the given bankaccount with
12         the given PIN"
13     ^bankAccount authenticate: pin.

```

Listing 6.5: The *withdrawAmount:from:* and *authenticateForAccount : withPIN:* methods of the Bank class

The *withdrawAmount:from:* method first reads the current amount from the BankAccount and then updates the amount directly after it using the BankAccount's *write:* method. This method conforms to the *ReadWriteConstraint* given earlier, but does not conform to the *AuthenticateBeforeWrite* constraint, because there is no authentication taking place. What we expect is that after annotating the BankAccount class as a *ProtectedResource*, the violation of the *AuthenticateWrite* constraint will be detected at runtime when we call the *withdrawAmount:from:* method on the BankAccount class.

The *authenticateForAccount:withPIN:* method is used to authenticate for the given BankAccount with the supplied PIN and it delegates the actual authentication to the BankAccount class.

In Listing 6.6 we show how we use the *SuperVisor* class to bind the constraints to

the BankAccount. This is done by adding the annotations to the annotation system, then adding the constraints to the SuperVisor and then telling the SuperVisor that it should instrument all classes that are annotated as resources in the given Smalltalk category. A category in Smalltalk is similar to a package in Java. We have chosen to use the category in the instrumentation method to prevent that the SuperVisor instruments the whole Smalltalk image.

```
1 | superVisor bank bankAccount constraintHelper
  annotationHelper |
2 annotationHelper <- AnnotationHelper new.
3 constraintHelper <- ConstraintHelper new.
4
5 annotationHelper annotateBankAccount.
6
7 superVisor <- SuperVisor instance.
8 superVisor deleteAllConstraints.
9 superVisor addConstraint: constraintHelper
  createAuthenticateBeforeWrite.
10 superVisor addConstraint: constraintHelper
  createReadWrite.
11 superVisor instrumentCategory: 'Banking-Category'.
```

Listing 6.6: Using the SuperVisor to instrument the BankAccount and verify the constraints

We begin by first declaring the instance variables, after which we create new instances of the AnnotationHelper and the ConstraintHelper classes in lines 2 and 3. These two helper classes are used to define the annotation and to create the constraints. The AnnotationHelper implements the *annotateBankAccount* method from Listing 6.3 and the ConstraintHelper class implements the *createReadWrite* and the *createAuthenticateBeforeWrite* methods from Listings 6.1 and 6.2 respectively.

In line 5 we use the AnnotationHelper to annotate the BankAccount class. In line 7 we obtain a reference to the instance of the SuperVisor. We need to obtain a reference and not instantiate it with the *new* method, because the SuperVisor is a Singleton class [11]. In line 8 we delete all constraints that could still be registered with the SuperVisor, and in line 9 and 10 we add the two constraints to the SuperVisor using the ConstraintHelper instance. We then call the *instrumentCategory:* method of the SuperVisor instance, which will apply all constraints on the classes in the given category that are annotated as resources.

```
1 | bank bankAccount |
2
3 bankAccount <- BankAccount new.
4 bank <- Bank new.
5 bank withdrawAmount: 20 from: bankAccount.
```

Listing 6.7: Using the (instrumented) BankAccount class

In Listing 6.7 we see how the BankAccount and Bank classes are used. From line 3 we use the BankAccount class as we would normally do. We create instances of the BankAccount and the Bank classes and call the *withdrawAmount:from:* method on it in line 5. Executing the original code is still the same, but because the SuperVisor instrumented the class, every time a method is called on the BankAccount class, the SuperVisor is signalled, which in turn signals the corresponding constraints.

```

1      complementing...aligning...combining...minimizing...
      complementing...
2      Consistent? true
3      Combining 'AuthenticateWriteConstraint' with '
      ReadBeforeWriteConstraint'.
4      complementing...aligning...combining...minimizing...
      complementing...
5
6      AW Entering not-authenticated
7      AW Leaving not-authenticated
8      Bankaccount read: 100
9      AW Entering not-authenticated
10     RBW Entering resource-read
11     AW Leaving not-authenticated
12     Bankaccount written, new value is 80
13     WARNING: Reached a trap state, this could indicate a
      constraint violation
14     AuthenticateWriteConstraint has been violated
15     Trace to violation of constraint '
      AuthenticateWriteConstraint-
      ReadBeforeWriteConstraint':read write
16     Uninstalled instrumentations

```

Listing 6.8: Output when Listing 6.6 is executed

When we execute the code of Listing 6.6, we get as output the first four lines of Listing 6.8. In line 1 and 2 the output indicates that the constraints are combined to test if the constraints are consistent, because both constraints are defined on the same resource. In line 3 and 4, the constraints are combined to form one constraint, which is used in the rest of the verification. These first 2 steps (checking consistency and combining) could be combined into one operation in future implementations.

Lines 6 to 16 are output generated by the execution of Listing 6.7. Lines 6 and 7 consist of output that was declared as actions in the constraints. The output with the 'AW' prefixes has been specified in the AuthenticateWrite constraints and the output with the 'RBW' prefixes has been specified in the actions of the ReadBeforeWrite constraint.

In Line 8 we see the output from the *read* method of the BankAccount class, and in lines 9 to 11 we see output specified as actions in the constraints. In line 12 we see the output that was specified in the *write* method of the BankAccount class. As we expected, line 13 warns that a possible constraint violation was detected and line 14 states that this violation was detected in the AuthenticateWrite constraint, after which the trace to the violation is given in line 15. This line also indicates that the constraints have been combined into one constraint, because the constraint name consists of a concatenation of the two original constraint names.

The final line indicates that all instrumentations are automatically uninstalled when a violation is detected to leave the original code in a consistent state that does not depend on the SuperVisor.

We have seen that the system is able to detect constraint conflicts, but now we have to illustrate that it does not generate false negatives, i.e. detect conflicts when they should not be there. To this end we add the code from Listing 6.9 between lines 4 and 5 of Listing 6.7, so before money is withdrawn:

```
1    bank authenticateForAccount: bankAccount withPIN: 1234.
```

Listing 6.9: Adding authentication

When this line is added, the output changes to what is shown in Listing 6.10. What can be seen is that the actions are still performed, but no warning is generated anymore and no constraint violation is detected. When compared to the output of Listing 6.8, it can also be noted that the instrumentations are not uninstalled. This is done on purpose, because the SuperVisor cannot know for sure that the user is finished testing. Therefore it chooses to keep the instrumentations in place, until either a constraint violation is detected, or the user uninstalls them explicitly with the *uninstallInstrumentations* method on an instance of the SuperVisor class.

```
1    complementing...aligning...combining...minimizing...
      complementing...
2    Consistent? true
3    Combining 'AuthenticateWriteConstraint' with '
      ReadBeforeWriteConstraint'.
4    complementing...aligning...combining...minimizing...
      complementing...
5
6    AW Entering not-authenticated
7    AW Leaving not-authenticated
8    AW Before authenticate
9    Authentication successful
10   AW After authenticate
11   AW Entering authenticated
12   AW Leaving authenticated
13   Bankaccount read: 100
14   AW Entering authenticated
15   RBW Entering resource-read
16   AW Leaving authenticated
17   AW Before write
18   Bankaccount written, new value is 80
19   AW After write
20   AW Entering not-authenticated
```

Listing 6.10: Output when authentication is added to Listing 6.6

6.8.4 Ordering the actions

We have seen that the actions that were specified in the constraints have been executed, but we did not specify an order in which they should be executed. When no such order is specified, the order is chosen by the SuperVisor. We could however specify an order explicitly by adding the following line of code between lines 10 and 11 of Listing 6.6:

```
1    superVisor addOrderingConstraint: (OrderingConstraint
      newConstraintFromDeclaration: '
      ReadBeforeWriteConstraint <
      AuthenticateWriteConstraint').
```

Listing 6.11: Using OrderingConstraints

The output of execution changes slightly, in that the action associated with entering the *authenticated* state of the *AuthenticateBeforeWrite* constraint is now executed before the action of entering state *resource-read* of the *AuthenticateWrite* constraint, as shown in lines 14 and 15 in Listing 6.12, which differ from lines 14 and 15 in Listing 6.10.

```

1      complementing...aligning...combining...minimizing...
        complementing...
2      Consistent? true
3      Combining 'ReadBeforeWriteConstraint' with '
        AuthenticateWriteConstraint'.
4      complementing...aligning...combining...minimizing...
        complementing...
5
6      AW Entering not-authenticated
7      AW Leaving not-authenticated
8      AW Before authenticate
9      Authentication successful
10     AW After authenticate
11     AW Entering authenticated
12     AW Leaving authenticated
13     Bankaccount read: 100
14     RBW Entering resource-read
15     AW Entering authenticated
16     AW Leaving authenticated
17     AW Before write
18     Bankaccount written, new value is 80
19     AW After write
20     AW Entering not-authenticated

```

Listing 6.12: Output when ordering constraints and authentication are added to Listing 6.6

Another indication that the ordering of actions has changed can be found in line 3 of Listing 6.12. This line states that the *ReadBeforeWrite* constraint is combined with the *AuthenticateWrite* constraint. In the previous listing the combination was made the other way around.

6.9 Conclusions

This chapter has shown how a system has been implemented that can be used to detect behavioural patterns at runtime and that can be used to test whether a system conforms to certain constraints. First the implementation of Deterministic Abstract Recognisers (DARs) was discussed, after which the implementation of the decoupled annotation system was explained which is an implementation of the meta information system of Chapter 5. The overview of the implementation continued with the discussion on how ordering constraints are implemented. This is done by generating an order in which the constraints are combined with each other.

After this we continued with the *SuperVisor*, which is the implementation of the central bookkeeping facility that is responsible for keeping track of encountered usage patterns in the code annotated as constrained resources. We have also explained that in order to notify the *SuperVisor*, we do not use standard Aspect Oriented Programming [10] approaches, because these are not available for Smalltalk. Instead we have chosen for the *MethodWrapper* approach [5], which can be translated to *Proxy* [11] based approaches or regular AOP approaches such as *AspectJ* [1] or *Compose** [2].

We have also seen that the implementation can be used to test whether code that

is annotated as a resource, complies with the Vibes constraints which have been used throughout this thesis. It was shown that the implementation is capable of detecting constraint violations and that constraints can be used to add behaviour using actions. We have also seen that in the implementation it is possible to set the order in which actions are executed.

Chapter 7

Conclusions

The goal of this thesis was threefold: Overcome the poor evolvability of regular expressions, find a way to use Resource Operations Models to specify behavioural joinpoints and find a solution with which programs in dynamically typed languages can be verified using Resource Operations Models. These problems have been addressed in this thesis. The findings of this thesis are summarised in this chapter, and lessons that have been learned are summed up as well. This chapter also features a section on future work and concludes with a personal evaluation.

7.1 Overcoming poor evolvability

Regular expressions are often used to perform pattern detection, especially because they can be easily used to specify the patterns to detect. However, when the system that is generating the input to the constraint changes, regular expressions may not always be the best fit for the problem. Because regular expressions have a fixed alphabet of symbols, the expression has to be updated whenever the pattern generating system introduces new symbols to the alphabet. This constraint is the reason that regular expressions can sometimes introduce extra maintenance work with regard to the pattern detection systems. This is why in this thesis alternatives to regular expressions have been investigated. We especially focussed on methods that are at least as expressive as regular languages, but are more resilient to changing alphabets. In this light we first evaluated Graph Transformation Systems (GTS).

The applicability of graph transformation systems depends on the meta model that is being used when constraint specifications are considered. When the meta model is not general enough it can be hard to create constraint specifications that apply to different variations of the same constraint and great care should be taken when developing such a meta model.

With respect to the context of this thesis, graph transformation systems have an unwanted property: finding a match to a rule is NP completed in the size of the rule, meaning that a solution cannot be found in polynomial time. Because we stated in the Problem Statement that alternatives to regular expressions should be scaleable and should not incur performance penalties when constraint specifications grow in complexity, the worst case behaviour of graph transformation systems leads us to not consider graph transformation systems as an alternative for regular expressions in the context of this thesis.

The second alternative to regular expressions to specify and detect behavioural patterns, is the Vibes specification language, introduced in [14]. The Vibes language extends the concepts of labeled transition systems by introducing a context sensitive

wildcard. This context sensitive wildcard can be used as a label for a transition in the specification, and it represents all outgoing transitions that have not been specified to leave the source state. Using the context sensitive wildcard it is not only possible to model what is important for the specification, but also how the specification should react to non-explicit events. This means that the complete alphabet that is used with a specification does not have to be specified.

We have demonstrated that constraints specified with Vibes are more robust than those specified with regular expressions, because a complete alphabet does not have to be defined. As a result, Vibes constraint specifications do not need to be changed whenever the operations on a resource change. Compared to regular expressions this is a great advantage. All newly introduced methods or symbols are caught with the context sensitive wildcard in Vibes, whereas in a regular expression all transitions for this new symbol have to be created.

Because Vibes specifications are almost the same as deterministic finite automata, they are not computationally complex, they can be implemented using an adapted form of state machines. If no transition exists for a given input symbol, the wildcard transition is followed. This means that a constraint specification can be implemented at relatively low costs and checking constraints is nothing more than taking transitions in the state machine. Because the state machine knows its current state, it only has to check all outgoing transitions from that state and follow the correct one. With respect to performance, Vibes specifications are more attractive as an alternative to regular expressions than graph transformation systems for the specification of resource operation constraints.

7.2 Behavioural joinpoints

Inspired by Aspect Oriented Programming concepts, we recognise that the state of a Vibes specification represents a more abstract state of the constrained resource as well. This abstract state information can be used as a form of pointcut expression and thus can be used to insert new functionality. Therefore the concepts of the original work on Vibes are used, in which the Vibes specifications are used to introduce new functionality to the system under verification.

In [21], Kiczales notes that current joinpoint models in Aspect Oriented Programming are not capable of capturing all events in the execution of a program and in [6] Cazzola *et al.* call for a joinpoint model that is based on the behaviour of the program. Therefore we have investigated how the Resource Operations Model can be used to define a behavioural joinpoint model.

In this thesis we recognise that the state of a Vibes specification represents a more abstract state of the constrained resource. This abstract state information can be used as a form of pointcut expression and thus can be used to insert new functionality. Therefore the concepts of the original work on Vibes are used, in which the Vibes specifications are used to introduce new functionality to the system under verification.

In contrast with the original work, four different events in the processing of a constraint have been identified, each of which can be used as a point in the execution of the system to insert new functionalities. These four events are leaving a state, beginning a transition, ending a transition and entering a state. We have also shown that leaving a state is essentially the same as beginning a transition and that ending a transition is similar to entering a state. Because of this we note that these four events could be reduced to two, but we intentionally choose to keep the four different events, because this reduces the required effort during creation of specifications.

7.3 Verifying dynamically typed systems

The third and final part of this thesis consists of applying the theories of the first two parts to dynamically typed languages. Performing pattern detection on dynamically typed languages poses a problem, because these languages lack explicit type information in the Abstract Syntax Tree (AST) used during compilation. This type information is used to determine whether a variable can be regarded as a resource. Because dynamically typed languages lack type information in the AST, it is very hard to determine either the type of a variable or the kind of resource as which the variable should be regarded without resorting to type inference mechanisms that complicate the verification process. Because type information is only available at runtime, we turned to performing behavioural pattern detection at runtime. As an added benefit the flow of control is *exactly* known at runtime, which results in knowing the exact type of a variable.

To perform behavioural pattern detection in dynamically typed languages using Resource Operation Models, three problems have to be overcome:

1. Find a method to keep track of how much of the pattern has been detected.
2. Designate what pieces of code should be regarded as a resource
3. Coupling patterns to constraints on resources

7.3.1 Keeping track of patterns

The first one consists of finding a method to keep track at runtime of how much of the pattern has been detected. To do this, Deterministic Abstract Recognisers (DARs) can be used. A DAR is comparable to an LTS, but with the addition of an extra symbol representing every symbol not in the alphabet. Vibes specifications can be translated into DARs by an algorithm laid out in [14]. Although regular expressions can be used to record the state of the pattern detection, they are dismissed for the same reasons they are not used for the specification of the patterns.

7.3.2 Designating resources

The second problem consists of how to designate what pieces of code should be regarded as a Resource. A solution for this can be found in a meta information system. However, where traditional methods to attach meta information to pieces of code (annotations) rely on the meta information to be specified in the source code, we choose to not take such an approach, because this would introduce an explicit coupling between the source code and the resource operations model. This does not seem to be a correct approach, because it should be possible to use the verification techniques even in situations where the source code of the system is not available. Therefore a decoupled meta information system is introduced, with which it is possible to designate classes as resources. Meta information can also be added to methods, which can be used to designate a method to be an operation on a resource. This meta information system can be used to designate what part of a system should be regarded as a resource and what should be regarded as operations on these resources.

7.3.3 Coupling patterns to constraints

The last part of introducing a runtime resource operations model constraint verification system consists of coupling DARs to the instances of Resources. For this a central

bookkeeping facility is introduced that couples DARs to Resource instances and which is informed whenever an operation is performed on the resource. This information is then used to advance the DAR instance of the Resource.

Providing information to the central bookkeeping facility poses a problem in itself, because in order to achieve this, some form of aspect oriented programming has to be used. The problem however, is in creating the correct pointcut expressions to use to superimpose the behaviour of automatically informing the central bookkeeping facility.

We have shown that these pointcut expressions can be created automatically, because all information is available. The Vibes constraints have a scope expression which specifies which Resources they constrain and the annotation system knows which pieces of code are designated as resources. This information can thus be combined to automatically create the correct pointcut expressions.

7.4 Implementation

In this thesis we have also provided an implementation in Smalltalk of the techniques proposed in this document. In the implementation the concept of DARs is implemented, as well as a meta information system. The final part, the central bookkeeping facility is implemented as well. Instead of using aspect oriented programming to inform the central bookkeeping facility that a method is invoked, we choose to use the concept of MethodWrappers [5], because the inner workings of Smalltalk lend themselves better for this concept, and because they mimic the behaviour of *around* advice in AspectJ. Another reason to use MethodWrappers is that they can be automatically generated so that the problem of scattered code is solved.

7.5 Lessons learned

During this thesis we have encountered some problems, which in hindsight could have been dealt with differently. One of these problems was the conceived NP completeness of Graph Transformation Systems, which during the research resulted in abandoning this approach. Only when the focus had shifted to the Vibes specification language, we discovered that solutions exist to counteract the performance problems, such as for instance incremental pattern matching. This specific solution should have been investigated more thoroughly, because we feel this might be a solution to the performance problems.

During the research of Graph Transformation Systems (GTSs) we used the Groove system [28][3] to evaluate GTSs. We feel that instead of only investigating Groove, other systems should have been investigated as well, because these systems might have different approaches.

With respect to the implementation of the system, we have mixed feelings with respect to the implementation language Smalltalk. Smalltalk proved to be well suited to create prototypes very fast, but because of its little usage nowadays we also feel that an implementation in another dynamically typed language would have been more useful. Although we were able to demonstrate that the proposed methods could be implemented and used using Smalltalk, an implementation in another language would have paved the way for actual case studies in real-life projects.

7.6 Future work

Although this thesis has introduced solutions to three different problems, we feel that more research should be carried out. Especially the usability and the real-life

application of the proposed system should be investigated further. Therefore we feel that the following should be carried out in the future to complement this work:

- To be able to further investigate the effectiveness and usability of the system, the system should be implemented in a more commonly used programming language than Smalltalk. Because Ruby, Python and Javascript are commonly used nowadays, we propose to implement the system in one of these languages.
- Implement a system which automatically translates Vibes specifications into Deterministic Abstract Recognisers. This would greatly improve the usability of the system, because the DARs do not have to be given programatically.
- Investigate methods to specify the precedence of constraints regarding actions. At the moment the order in which actions are executed can only be specified programatically. It would be good to investigate methods with which this could be done more intuitively, such as for instance in the Vibes constraints themselves.
- Perform case studies on actual evolving systems to investigate the real-life effectiveness of Vibes on the evolvability of the constraint specifications.

7.7 Personal evaluation

A master's thesis is not only about the results of the research performed in the context of it, but also a last step in the education of the student writing the thesis. Because of this, the process of coming to the results is as important as the results themselves. During this process, the student acquires specific skills and learns important lessons, and it is important to reflect on these. Therefore this chapter recapitulates what lessons were learned during the process of performing the research and writing the thesis.

General audience

It can often be hard to explain the details of the research to a general audience. This is because once someone is dedicating much of his time to a certain problem, he becomes very intimate with this problem and he spends much time thinking about the problem. This can result in a situation in which only the problems at hand are perceived as problems and that necessary background information is not perceived as important anymore. This can result in the perception in which the necessary background information is thought to be basic knowledge everybody has. When this change in perception occurs, it can be very hard to talk about encountered problems with people not intimate with the situation.

The process of writing this thesis was a fairly long one, and because of this I have often found myself in the situation of not being able to explain problems to other people. Even problems encountered during the research could prove to be very hard to explain in the thesis, because of a slightly changed perception of the basic knowledge people possess. This sometimes made it very hard to explain the added value of this thesis as well, because it was sometimes hard to explain the addressed problems in the thesis. Once I became aware of this changed perception, I was able to change it back, and thus able to better explain the problems to a general audience.

Once the notion of basic knowledge is returned it is important not to change the balance into the exact opposite. During the writing of this thesis I have often been focussing too much on the problem, but not enough on the solution. Because I became too much aware of the basic knowledge of a general audience, an imbalance

between problem and solution was introduced. Although early readers were perfectly able to understand the problems now, they were not able to understand the solutions to these problems. Once I was made aware of this, I was able to move on and restore the balance of explaining the problem and laying out the solutions to these problems. During these explanations, examples proved to be very useful. Examples tend to make topics more concrete and help to convey the problem or solution to the reader. Therefore during this thesis a running example has been used, which helped throughout the text, from providing a background, via problems and solutions to explaining the usage of the implementation. These lessons made it possible to explain the problems, the solutions to these and the added value of this research to a general audience.

Documenting progress

During the research it became very clear that it was important to document progress and findings. This documentation proved to be of great importance during the rest of the exercise, in which the documentation was often used to help remembering what was done and why certain choices had been made. The form in which the progress is documented proved to be less important than the documentation itself, but during this progress Wiki's have proven to be very helpful. Although plain text files can be used, at a later point in time it can often be hard to retrieve information one is looking for. Wiki's help with that problem, in that all documentation is contained in one, central place and documents can be linked to each other, making navigation through the documents less difficult.

Review

Review of the text by others helped a lot in shaping the text of this thesis. Because it can be very hard to read text you have written yourself as a newcomer with only a basic understanding of the problem, it is hard to miss opportunities to give extra explanation when needed, or to be less verbose in other occasions. Someone who reviews the text with an objective mind can help a great deal with identifying these opportunities by asking questions or placing remarks when something is not clear. This not only helped during the process of writing the thesis, but also during the initial research in which various discussions helped focusing on the real problems.

Work and study

One thing that has become clear during the writing of this thesis is that it is often hard to combine a job with a study. Often these two seem to be mutual exclusive, because the two always seem to be competing for time. Sometimes it looks like there is more to learn during the job than during writing a thesis, and other times working seemed to be less important than finalising the thesis. Especially when one has a job with more responsibilities, in which people have to be coordinated and meetings have to take place at times originally planned for working on the thesis this can become a real problem that can lead to delays and deviations from the original planning. The problem is dual, in that one knows that finishing the master's education will most probably increase the chances for future jobs, but that a good resume also helps with this. This can result in a situation in which it is very hard to find a middle ground, and in which it is easier to make a choice for only one of the two.

7.8 Summary

We have seen that Vibes can be used as an alternative to regular expressions and that they are less susceptible to change. Further we have seen that constraints on resource operations models can be used to introduce new behaviour and that it is very well possible to combine these findings to perform behavioural verification of dynamically typed languages.

We have seen what lessons have been learned during this investigation, both on an academic and a personal level, and we have evaluated what we feel should be investigated in the future to complement this work.

To conclude this thesis we will once again give the problems from the Problem Statement in Chapter 2, and will summarise how these problems are addressed in this thesis.

- How can we overcome the poor evolvability of regular expressions, which are used to specify the Constraints? The problem of poor evolvability has been solved by using Vibes specifications as an alternative to regular expressions. Because Vibes specifications have a context sensitive wildcard, the complete alphabet does not have to be known and therefore Vibes specifications do not suffer from poor evolvability.
- How can the Resource Operations Model be used to specify behavioural join-points? In this thesis we have introduced a method with which constraints can be used to insert new functionality to existing systems, by using the notion that the state of a constraint represents a behavioural state of the system under verification. Functionality can be added to the system by using actions in the constraint specifications.
- How can the verification system for Resource Operations Models be used for dynamically typed languages? To use the Resource Operations Models to verify dynamically typed languages, a runtime system was proposed with which it is possible to perform runtime verification of a system. To illustrate that the proposed techniques and solutions could be used, an implementation is given, which is capable of runtime verification.

References

- [1] Aspectj website. URL: <http://www.eclipse.org/aspectj/>.
- [2] Compose* website. URL: <http://composestar.sf.net>.
- [3] Groove website. URL: <http://groove.sourceforge.net/groove-index.html>.
- [4] A.J. Bernstein. Analysis of programs for parallel processing. In *IEEE Trans. on Electronic Computers*, pages 757–762, 1966.
- [5] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the rescue. *Lecture Notes in Computer Science*, 1445:396–??, 1998.
- [6] Walter Cazzola, Jean-Marc Jézéquel, and Awais Rashid. Semantic join point models: Motivations, notions and requirements. In *SPLAT 2006 (Software Engineering Properties of Languages and Aspect Technologies)*, March 2006.
- [7] A Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, (58):345–363, 1936.
- [8] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for smalltalk. *Computer Languages, Systems & Structures*, 32(2-3):125–139, 2006.
- [9] Pascal Durr, Tom Staijen, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about semantic conflicts between aspects. In *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, 2005.
- [10] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.
- [11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1994.
- [12] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.
- [13] J Gosling. *Java Language Specification, Second Edition, The*. 2000.
- [14] G. Gülesir. Evolvable behavior specifications using context-sensitive wildcards. 2008.
- [15] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, Vancouver, Canada*, pages 85–95, New York, NY, USA, 2007. ACM Press.

- [16] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In Robert Hirschfeld, Andreas Polze, and Ryszard Kowalczyk, editors, *NODE/GSEM*, volume 88 of *LNI*, pages 17–32. GI, 2006.
- [17] JR Holmevik. Compiling simula: a historical study of technological genesis. *IEEE annals of the history of computing*, 16(4), 1994.
- [18] Jonathan Jacky. *Way of Z*. Cambridge University Press, New York, NY, USA, 1996.
- [19] Ralph E. Johnson. Type-checking smalltalk. *SIGPLAN Not.*, 21(11):315–321, 1986.
- [20] Alan C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, 1993.
- [21] Gregor Kiczales. The fun has just begun. AOSD’03 Keynote Address, available from <http://www.cs.ubc.ca/~gregor>, 2003.
- [22] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
- [23] Gregor Kiczales, Jim d. Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.
- [24] P Maes. Concepts and experiments in computational reflection. In *In Proc. OOP-SLA’87*, pages 147–155, 1987.
- [25] István Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing aspects at shared join points. In Andreas Polze Robert Hirschfeld, Ryszard Kowalczyk and Mathias Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).
- [26] István Nagy, Lodewijk Bergmans, Wilke Havinga, and Mehmet Aksit. Utilizing design information in aspect-oriented programming. In Robert Hirschfeld, Ryszard Kowalczyk, Andreas Polze, and Mathias Weske, editors, *NODE/GSEM*, volume 69 of *LNI*, pages 39–60. GI, 2005.
- [27] Java Community Process. *JSR 175: A Metadata Facility for the Java Programming Language*. 2004. URL: <http://www.jcp.org/en/jsr/detail?id=175>.
- [28] Arend Rensink. GROOVE: A graph transformation tool set for the simulation and analysis of graph grammars. Available at <http://www.cs.utwente.nl/~groove>, 2003.
- [29] Martin Rinard, Alexandru Salcianu, and Suhabe Bugarra. A classification system and analysis for aspect-oriented programs. *SIGSOFT Softw. Eng. Notes*, 29(6):147–158, 2004.
- [30] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Comput. Lang. Syst. Struct.*, 34(2-3):46–65, 2008.
- [31] T. A. Sudkamp, editor. *Languages and machines : an introduction to the theory of computer science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
- [32] E. Tanter, J. Noye, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification, 2003.
- [33] Scott Wiltamuth and Anders Hejlsberg. *C# Language Specification*. 2008. URL: [http://msdn.microsoft.com/en-us/library/aa645596\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa645596(VS.71).aspx).

Appendix A

CheckDesign

CheckDesign is the part of the Vibes solution that makes sure that multiple design specifications given in Vibes are consistent with each other. This addresses the problem that specifications in a natural language are often hard to automatically analyze and find faults.

If a procedure is constrained by multiple specifications, it could be the case that one constraint invalidates another. Consider for instance the case that within procedure f , procedure g should always be directly preceded by a call to h , but another specifications states that within procedure f , a call to e should occur somewhere before a call to g . This should be possible, because one trace that is compliant with both constraints is to first call e , then h and then g within function f . For these two constraints it seems to be pretty simple to check whether they are consistent, however, to do this in an automated fashion is somewhat more difficult.

To be able to detect such conflicts between constraint specifications, a special algorithm is constructed to see if two constraints are consistent with each other. First of all it is important to detect the constraint specifications that constrain the same procedure. Only specifications constraining the same procedure need to be checked for consistency, constraints defined for different procedures will not violate each other.

Before we can start explaining this algorithm, we first have to explain some of the underlying theorems of the Vibes specification language. A Vibes constraint is translated in a so called Deterministic Abstract Recogniser:

Definition A.0.1 A *Deterministic Abstract Recogniser (DAR)* D is given by:

$$D = \langle Q, \Sigma_a, \delta, q_0, F, \Xi, \eta \rangle$$

where:

- $Q = \Omega \cup \{q_t\}$ is a finite set of states, with Ω the set of user-defined states, q_t the default trap state and $q_t \notin \Omega$
- $\Sigma_a = \Sigma_b \cup \{\#\}$ is the abstract input alphabet, where Σ_b is a finite set of symbols such that $\# \notin \Sigma_b$. Σ_b is called the base input alphabet. $\#$ is a reserved symbol that stands for all symbols not defined in the base alphabet.
- $\delta : Q \times \Sigma_a \rightarrow Q$ is a total function that is called the transition function. $\forall a \in \Sigma_a (\delta(q_t, a) = q_t)$, which defines the behavior of the default trap state.
- $q_0 \in \Omega$ is the initial state
- $F \subseteq \Omega$ is the set of final states

- Ξ is a regular expression defining the scope of D . Ξ is called the scope expression and matches a set of strings.
- η is the name of D . The name of D is a Vibes identifier.

In short: a Deterministic Abstract Recogniser (DAR) is a deterministic finite state machine with an extra symbol $\#$ that matches for any symbol not in the base alphabet.

Definition A.0.2 The **base alphabet** of a DAR with alphabet Σ is:

$$\Sigma_{\text{BASE}} = \Sigma \setminus \{\#\}$$

The algorithm to check the consistency of multiple constraints was put forward in the work of Güleşir [14] and the basis is formed by composing two constraints by aligning their alphabets before the two DARs of the specifications are executed in parallel, resulting in a new automaton simulating the parallel behaviour of the two automata. In the following we will create such a automaton for the two constraints depicted in Figures A.1 and A.2. The former constraint specifies that within function f , before a call to g is made, a call to e has occurred and the latter constraint specifies that immediately prior to a call to g a call to h has to be made.

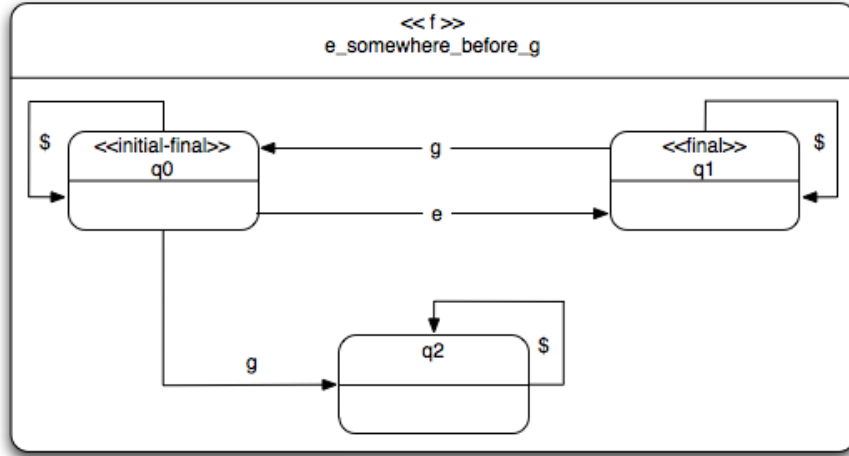


Figure A.1: Constraint on method f stating that somewhere before a call to g , a call to e should occur.

The crux of the algorithm lies in the fact that if multiple DARs are combined in what is called a cluster, at least one string of symbols is accepted if the conjunction of the languages of the DARs is nonempty: $L(DAR_{\text{CLUSTER}}) = L(DAR_1) \cap \dots \cap L(DAR_n) \neq \emptyset$. In order to verify this, the construction of the cluster DAR is based on DeMorgan's law:

$$L(M'_1) \cap \dots \cap L(M'_n) = \overline{\overline{L(M'_1)} \cup \dots \cup \overline{L(M'_n)}}$$

The first step of the algorithm consists of creating the Deterministic Abstract Recognisers (DARs) that correspond with the constraints. This means that all $\$$ transitions are expanded, such that the $\#$ transition can be introduced. Expanding means that for each $\$$ transition new transitions are introduced for the symbols from the base alphabet that are not specified for the source node of the $\$$ transition and that have the same target node as the $\$$ transition. At the same time a transition with a $\#$ label is added with the same source and destination node as the original $\$$ transition.

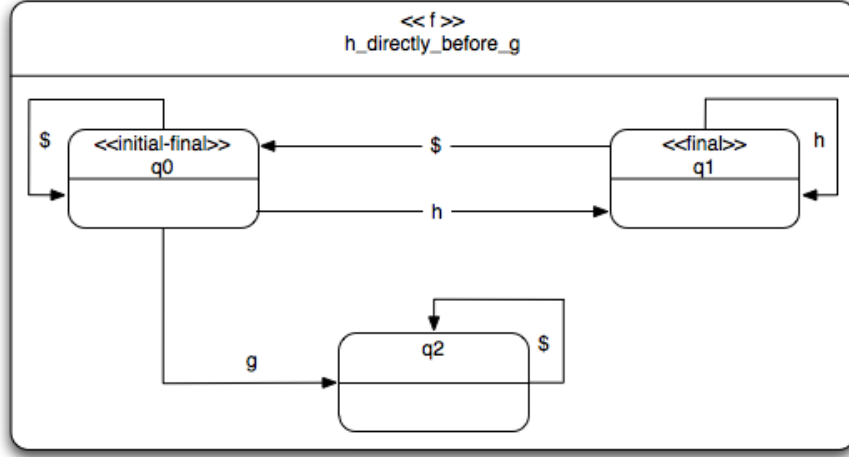


Figure A.2: Constraint on method f stating that immediately before a call to g , a call to h should occur.

Remember that the $\$$ transition is a context sensitive transition, whose meaning depends on other transitions with the same source state, and that the $\#$ label is used for all inputs not part of the alphabet of the DAR. The DARs of the two constraints are depicted in Figure A.3.

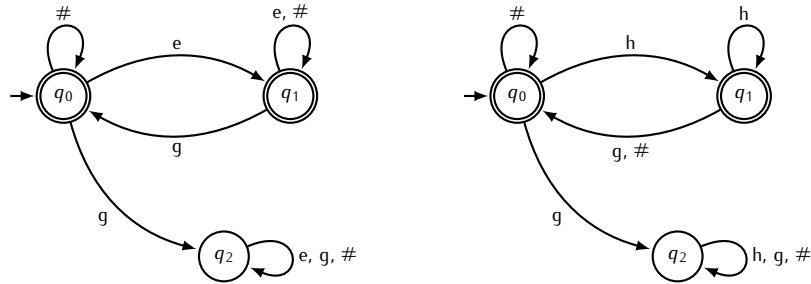


Figure A.3: Deterministic Abstract Recognisers of the $e_somewhere_before_g$ and the $h_directly_before_g$ constraints.

The second step of the algorithm is to align both DARs. This means that the base alphabets of the two DARs are made equal, by performing the following steps, considering we are dealing with two constraints C_1 and C_2 , with alphabets Σ_1 and Σ_2 respectively.

First the so called *cluster* alphabet is constructed, which is the union of the base alphabets of C_1 and C_2 . Remember that the base alphabet of a constraint C was $\Sigma_{BASE} = \Sigma \setminus \{\#\}$, resulting in the union of Σ_1 and Σ_2 without the $\#$ symbol, $(\Sigma_1 \cup \Sigma_2) \setminus \{\#\}$

Then, for each DAR D_i , for each symbol $s \in (\Sigma_1 \cup \Sigma_2) \setminus \{\#\}$ not in the base alphabet of D_i , for each state q_s identify the target state q_t of the $\#$ transition, then add a new transition $q_s \xrightarrow{s} q_t$.

This results in the aligned DARs given in Figure A.4.

After aligning the DARs, it is time to create the complement of the DARs, this

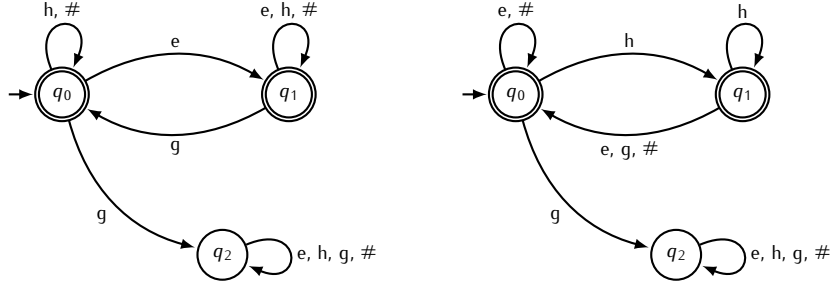


Figure A.4: Aligned DARS corresponding to the `e_somewhere_before_g` and the `h_directly_before_g` constraints.

is because of the use of DeMorgan's law, which states that we need to create the complement of the conjunction of the complements and if the set of accepted strings is nonempty, we know that there exists at least one sequence of symbols that is accepted by the combined DAR, meaning that the constraints are consistent and non-violating.

The creation of the complement of a DAR is very simple, the only thing to do is marking accepting states from the aligned DARS as non-accepting and marking non-accepting states as accepting. This is exactly what has been done in the DARS of Figure A.5.

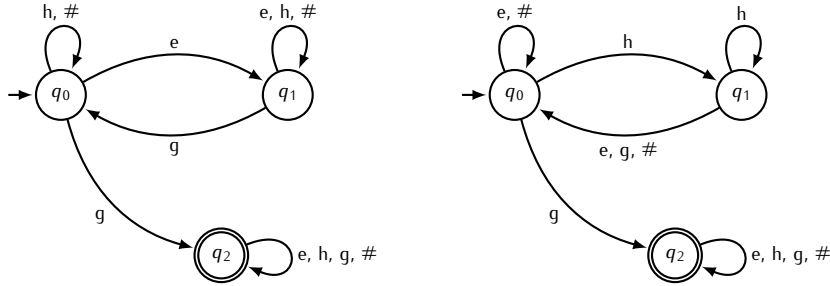


Figure A.5: Complement of the aligned DARS corresponding to the `e_somewhere_before_g` and the `h_directly_before_g` constraints.

Now that we have created the complement of the aligned DARS of the constraints, it is time to unify them, using what Güleşir calls Non-Deterministic Abstract Recognisers with ϵ -transitions, $\text{NAR-}\epsilon$. The difference between a DAR and a $\text{NAR-}\epsilon$ is comparable to the difference between DFAs and $\text{NFA-}\lambda$, they are the same, but a non-deterministic symbol is added to the alphabet. In the case of the $\text{NAR-}\epsilon$ this is the ϵ symbol.

To create the $\text{NAR-}\epsilon$, a new automaton is created, starting with a new state $q_{0_{\text{NAR}}}$. Then we add to this new automaton the two complemented, aligned DARS and we add ϵ -transitions from $q_{0_{\text{NAR}}}$ to the original start states of the two aligned, complemented DARS. We remove the initial markers from the two DARS and mark the new state $q_{0_{\text{NAR}}}$ as initial.

This step applied to our example results in the $\text{NAR-}\epsilon$ of Figure A.6.

The following step is comparable to creating a DFA from a $\text{NFA-}\lambda$, it consists of creating a Deterministic Abstract Recogniser from the Non-Deterministic Abstract Recogniser with ϵ -transitions. This construction consists of creating new states that are unions of states the automaton can be in at one time, and is explained in Algorithm 6.6.3 of [31]. The application of this algorithm to our running example is given in Figure A.7. Note that the labels have been renamed here, but we will give the original names

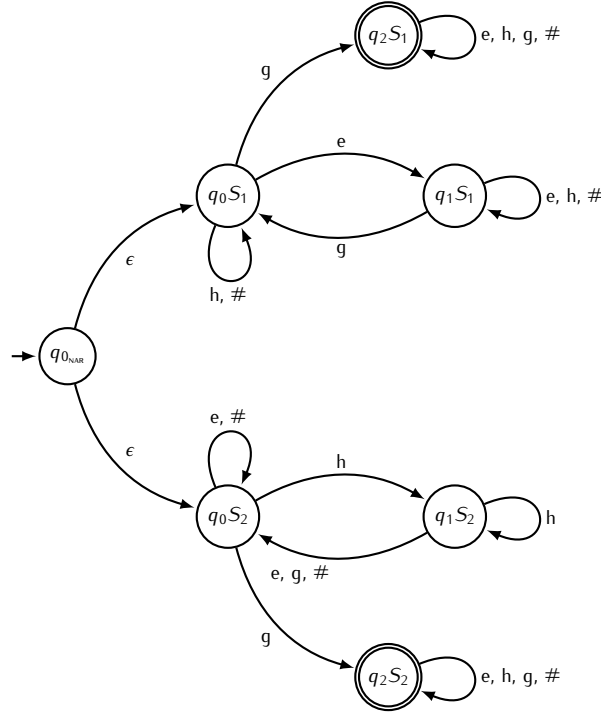


Figure A.6: NAR- ϵ of the combination of the `e_somewhere_before_g` and the `h_directly_before_g` constraints.

of the states for clarity here:

- $q'_0 = \{q_{0_NAR}, q_0S_1, q_0S_2\}$
- $q'_1 = \{q_0S_1, q_1S_2\}$
- $q'_2 = \{q_1S_1, q_0S_2\}$
- $q'_3 = \{q_1S_1, q_1S_2\}$
- $q'_4 = \{q_0S_1, q_2S_2\}$
- $q'_5 = \{q_1S_1, q_2S_2\}$
- $q'_6 = \{q_2S_1, q_2S_2\}$
- $q'_7 = \{q_2S_1, q_0S_2\}$
- $q'_8 = \{q_2S_1, q_1S_2\}$

Now we have a deterministic version of the NFA- ϵ , but the number of states is excessive compared to the simplicity of the two original constraints. This is why we try to minimise the DAR by first finding pairs of distinguishable pairs. This is done following the procedure outlined in Algorithm 6.7.2 of [31] and results in the minimised DAR of Figure A.8.

The final step of the algorithm exists of creating the complement of the minimised DAR. In the work of Güleşir two constraints are consistent if the resulting DAR has a reachable accepting state, because this means that there is at least one sequence of symbols that is accepted by the combined DAR. If there are no accepting states, or no accepting states are reachable from the start state, it means that there exists

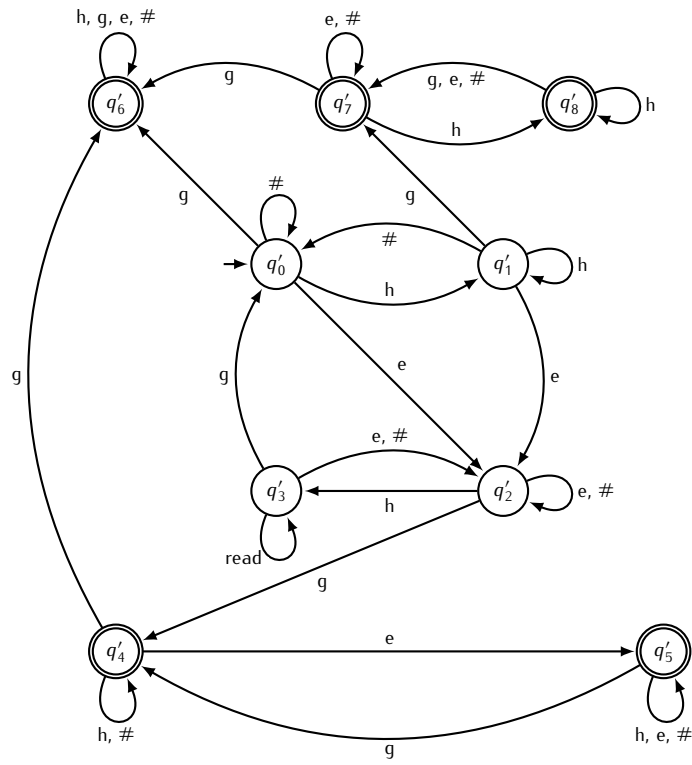


Figure A.7: DAR equivalent to the NAR- ϵ with non-determinism removed.

no sequence of symbols that is accepted by the combined DAR and that the two constraints are inconsistent. The complement of our example is given in Figure A.9 and as can be seen the two constraints are consistent.

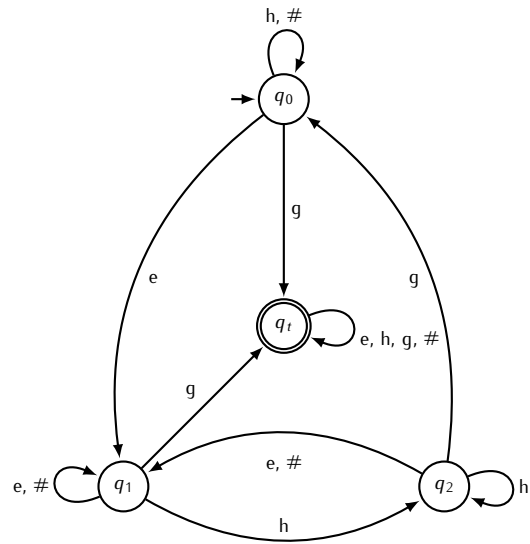


Figure A.8: Minimised DAR equivalent to the DAR of Figure A.7.

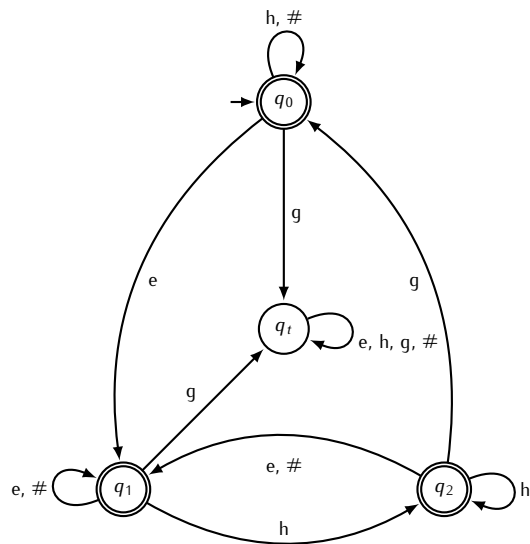


Figure A.9: Complement of the minimised DAR of Figure A.8.

Appendix B

API

This appendix gives the API of the implementation in Smalltalk. The methods of each important concept of the implementation are given and an explanation about their workings is given as well.

B.1 DAR

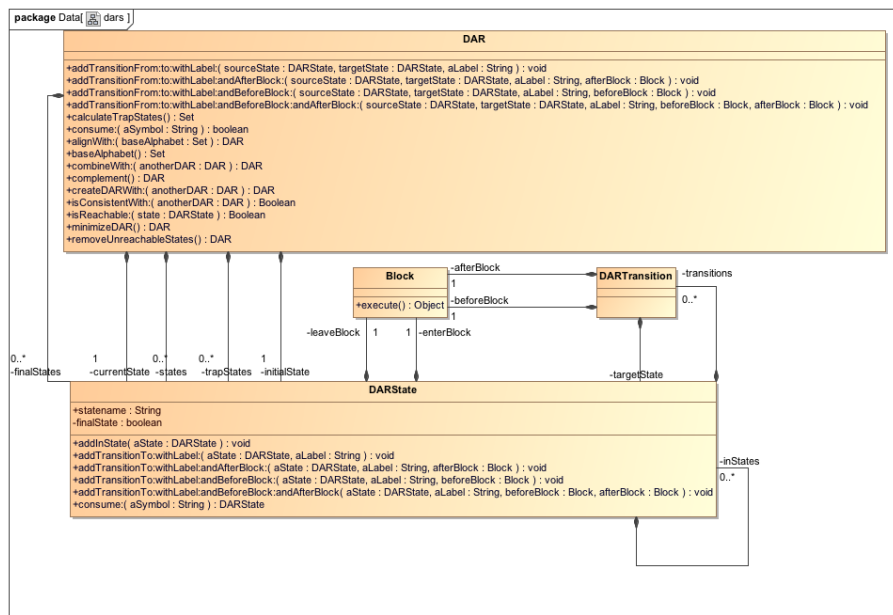


Figure B.1: Class diagram of the DAR implementation

At the heart of the verification of a constraint lies its corresponding Deterministic Abstract Recogniser (DAR). A DAR is implemented using a class called *DAR*, which is actually a collection of *DARState*'s with methods to easily add states and transitions. Its methods are explained in Table B.1. The *DAR* class has several fields to structure the DAR more, such as an *initialState* field and a collection of trap states and a field which refers to the current state. The *DAR* class also features several methods that aid in the composition of multiple DARs.

Table B.1: Method overview of the DAR class

DAR	
Method name	Description
addTransitionFrom:to:withLabel:	This method adds a transition from a source node to a target node and labels it with the label given as the last argument
addTransitionFrom:to:withLabel:andAfterBlock:	This method adds a transition from a source node to a target node, labels it and adds a block to execute right after the transition is done
addTransitionFrom:to:withLabel:andBeforeBlock:	This method adds a transition from a source node to a target node, labels it and adds a block to execute right before this transition is performed
addTransitionFrom:to:withLabel:andBeforeBlock:andAfterBlock:	This method adds a transition from a source node to a target node, labels it and adds blocks to execute right before and after a transition is performed
calculateTrapStates	This method is used to calculate the states from which it is not possible anymore to reach a final state
consume:	This method offers an operation name as a symbol to the state machine. This class delegates this to the current <i>DARState</i>
alignWith:	This method aligns the current DAR with the provided base-alphabet. This means that for each symbol not in the base-alphabet of the DAR, a new transition is added with the same source and target node as the #-transition and with that given symbol as a label.
baseAlphabet	This method returns the base-alphabet of this DAR ($\Sigma_{\text{BASE}} = \Sigma \setminus \{\#\}$).
combineWith:	This method combines the current DAR with the supplied one. This is done by first complementing both DAR, aligning their base alphabets, create the NAR- ϵ , removing unreachable states, minimising and complementing again.
complement	This method returns the complement of the current DAR. The complement DAR has the same states and transitions as the original DAR, but accepting states have become non-accepting and vice-versa.
createDARWith:	This method creates a Nondeterministic Abstract Recogniser with ϵ transitions (NAR- ϵ) by combining the current DAR with the supplied one and extracts a Deterministic Abstract Recogniser (DAR) from it. This is used in the composition of two DARs.
isConsistentWith:	This method takes another DAR as argument and checks if it is consistent with the current DAR.

Continued on Next Page...

Table B.1 – Continued

Method name	Description
isReachable:	This method checks if a given state can be reached from the initial state.
minimizeDAR	This method minimises the current DAR by identifying distinguishable states and creating partitions with indistinguishable states, resulting in a new equivalent DAR with a minimal amount of states.
removeUnreachableStates	This method removes any state that is not reachable from the initial state of the current DAR.

The second class of interest is the *DARState*. This class is the representation of a state and each state holds a mapping of symbols to *DARStates* which is used to represent the transition table of that state. Each state also has a set of states from which it is possible to reach this state, called the *inState* collection. This collection is used in the calculation of the trap states. Each state has its own name and a boolean field indicating whether this state is a final state. The *DARState* class also has fields that refer to the blocks of code to execute when the state is entered or left. Note, however, that these code blocks are actually executed before or after a transition is made in the DAR. These fields are here for ease of specification, so one is not required to add the code to every outgoing or incoming transition. This is automatically done when the actual DAR for a given resource is generated. An overview of the methods of the *DARState* class is given in Table B.2.

Table B.2: Method overview of the *DARState* class

DARState	
Method name	Description
addInState:	This method adds a state to the set of states from which it is possible to reach this state. This set of states is used to calculate so called trap states
addTransitionTo:withLabel:	This method adds a transition from this node to a target node and gives it the given label
addTransitionTo:withLabel: and- andAfterBlock	This method adds a transition from this node to a target node, labels it and adds a block to execute right after the transition is done
addTransitionTo:withLabel: and- BeforeBlock:	This method adds a transition from this node to a target node, labels it and adds a block to execute right before this transition is performed
addTransitionTo:withLabel: and- BeforeBlock:andAfterBlock:	This method adds a transition from this node to a target node, labels it and adds blocks to execute right before and after a transition is performed
consume:	This method returns the <i>DARState</i> that is the target node of the transition with the given label

Because transitions can have actions defined for them, transitions have been reified in the *Transition* class. This class is nothing more than a placeholder for the blocks of code that are to be executed before or after a transition is made. This is illustrated in Figure B.1 by the aggregation of the *beforeBlock* and *afterBlock* which are both of type *Block*. The *Block* class itself is a reification of a piece of code and only has one method, the *execute* method, which executes the code represented by this class. This class has been added here for clarity; in the implementation this is handled by Smalltalk's native *Block* class. A *Block* can have zero or more arguments, and in this implementation this is used to optionally supply a *Block* with an execution context. This execution context is implemented in the *ExecutionContext* class, which delivers reification data to the *Block*. This reification data exists of a reference to the Resource instance, to the method that was called and its arguments.

B.2 Annotations

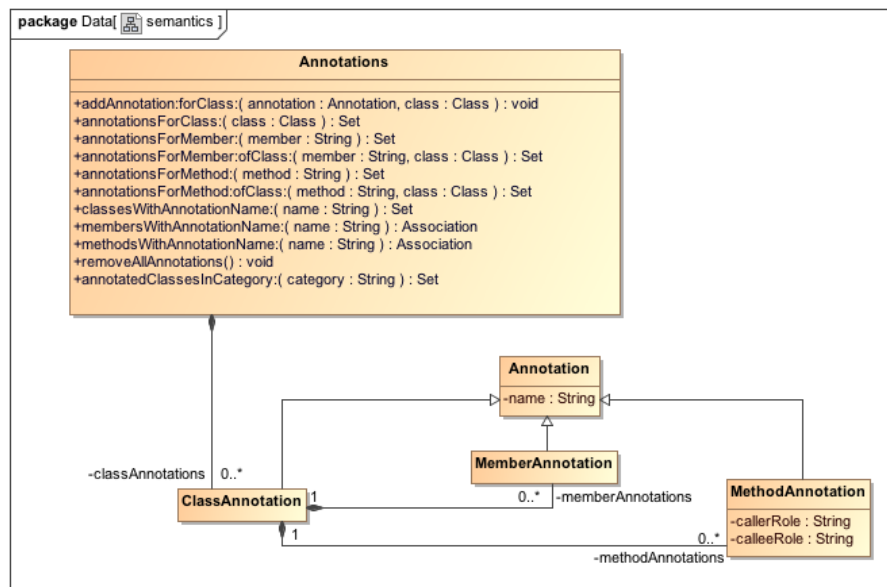


Figure B.2: Class diagram of the Annotation model

Each class can be annotated using the *Annotations* class. All annotations extend the *Annotation* class, which holds an annotation-name. The *Annotations* class holds a dictionary mapping a *Class* to a *ClassAnnotation*. A *ClassAnnotation* has a dictionary of method names to *MethodAnnotations* and a dictionary of member names to *MemberAnnotations*. A *MethodAnnotation* defines the caller and the callee role for the interaction. Note however that these roles are not used in the implementation.

The *Annotations* class also has query methods to quickly lookup annotations belonging to a class, method or member, or to find classes, methods or members with a given annotation. These methods are explained in Table B.3.

Table B.3: Method overview of the Annotations class

Annotations	
Method name	Description
addAnnotation:forClass	This method adds a <i>ClassAnnotation</i> to the annotation table for the given class
annotationsForClass:	This query method is used to find all annotations defined for the given class
annotationsForMember:	This query method finds all annotations defined for all fields with the given name
annotationsForMember:ofClass:	This query method is used to find all annotations defined for the given member in the given class
annotationsForMethod:	This method is used to find all annotations defined for all methods with the given name
annotationsForMethod:ofClass:	This query method is used to find all annotations belonging to the given method of the given class
classesWithAnnotationName:	This query method returns a set of classes that all have the given annotation name
membersWithAnnotationName:	This query method returns a mapping of classes to collections of members that have the given annotation name
methodsWithAnnotationName:	This query method returns a mapping of classes to collections of methods that have the given annotation name
removeAllAnnotations	This method removes all annotations that have been defined
annotatedClassesInCategory:	This method returns all classes in a given category for which annotations have been defined

B.3 Constraints

There are two types of constraints in the implementation: Vibes Constraints and Ordering Constraints. The first type represent the constraints as they are used to specify the expected behaviour of a resource. These constraints have a reference to the DAR that is used to represent the constraint, and have a scope expression and a name. They also have two properties *trace* and *maxInitials*. The *trace* property holds a trace of all operations that have been encountered during the execution when the constraint was verified. When a constraint violation is detected, this property can be used to get a trace to what combination of operations on the resource led to the violation. The *maxInitials* property represents the number of times the initial state of the constraint can occur in the trace. This property is available to reduce the length of the trace, because whenever the initial state of the constraint is reached, the verification process can be interpreted as being restarted. Initially this property is set to -1, which means that the trace will only grow and can encounter an unlimited number of initial states.

Ordering Constraints are used to specify in which order multiple Vibes constraints should be combined. The order of combination also determines in which order the actions of the constraints are performed, and this is where the Ordering Constraints will be used most. An Ordering Constraint can be declared using the class method

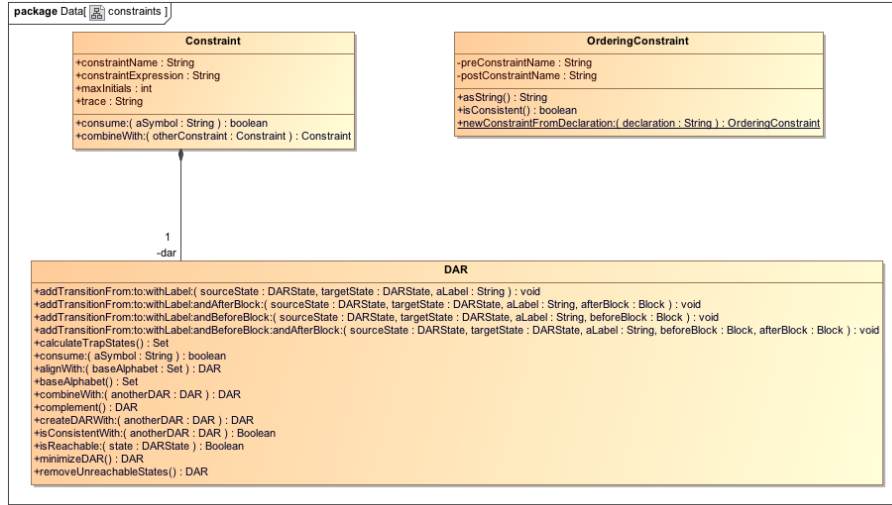


Figure B.3: Class diagram of the constraints implementation

`newConstraintFromDeclaration;` in which the $C_1 < C_2$ operator can be used, as for instance in Listing B.1:

```
1  OrderingConstraint newConstraintFromDeclaration: '
    ReadBeforeWriteConstraint <
    AuthenticateWriteConstraint'.
```

Listing B.1: Using OrderingConstraints

This example specifies that the `ReadBeforeWrite` constraint actions should always be performed before those of the `AuthenticateWrite` constraint. The `OrderingConstraint` class has three fields, the *preConstraint*, the *postConstraint* and the *declaration*.

B.4 SuperVisor

The *SuperVisor* class (Figure B.4) lies at the heart of the constraint verification mechanisms. The *SuperVisor* class is a Singleton class [11], which basically means that at all times there is only one instance of this class. This instance keeps track of all constraints, both Vibes constraints and ordering constraints. Once the constraints are added to the *SuperVisor* instance, and the *instrumentCategory:* method is called, the *SuperVisor* collects all annotated classes in the given category and installs hooks on all methods of these classes. These hooks are used before or after a method in an instrumented class is called and the hooks call either the *before:ofClass:withArgs:andSender:* or the *after:ofClass:withArgs:andSender:* method of the instance of the *SuperVisor* class.

When the *before:ofClass:withArgs:andSender:* method is first called for a given instance of a resource, all constraints that have been defined for that resource are composed into one DAR. This DAR is then stored for that particular instance of the resource. Every consecutive time this method or the *after:ofClass:withArgs:andSender:* is called, this composed DAR is retrieved and is offered the method name for consumption. The DARs all consumes the input and if it finds itself in a trap state an exception is thrown and a trace to the constraint violation is printed. On overview of

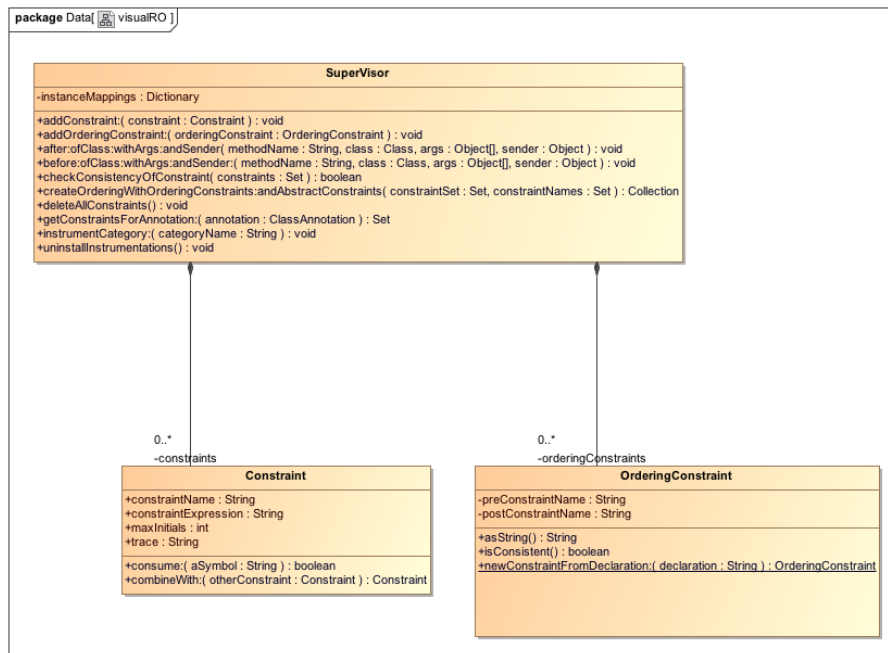


Figure B.4: Class diagram of the SuperVisor implementation

the methods of the *SuperVisor* class is given in Table B.4.

Table B.4: Method overview of the SuperVisor class

Annotations	
Method name	Description
addConstraint:	This method adds a <i>Vibes Constraint</i> to the collection of constraints
addOrderingConstraint:	This method adds an <i>OrderingConstraint</i> to the collection of ordering constraints
after.ofClass.withArgs.andSender:	This method is automatically called by each instrumented method after it has finished its execution. This is implemented by means of installing <i>MethodWrappers</i>
before.ofClass.withArgs.andSender:	This method is automatically called by each instrumented method before it starts execution. This is implemented by means of installing <i>MethodWrappers</i>
checkConsistencyOfConstraints:	This method is used to check if the constraints in the given set are consistent with each other (i.e. they can be combined with each other)
createOrderingWithOrdering Constraints: andAbstractConstraints:	This method takes the ordering constraints and the constraint names that apply for a given resource as arguments and uses this information to create an ordering in which the abstract constraints should be composed to comply with the given ordering constraints.
deleteAllConstraints	This method removes all defined constraints

Continued on Next Page...

Table B.4 – Continued

Method name	Description
getConstraintForAnnotation:	This method returns a set of constraints that have been defined for the given annotation
instrumentCategory:	This method instruments all methods in annotated classes in the given category, making sure that before and after execution of these methods the <i>SuperVisor</i> instance is called
uninstallInstrumentations	This method removes all instrumentations that have been installed before