Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN
Vakgroep Informatica
Laboratorium voor Programmeerkunde

A Goal-Driven Approach for Documenting and Verifying Design Invariants

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Isabel Michiels

Academiejaar 2006 - 2007

Promotoren: Prof. Dr. Theo D'Hondt en Dr. Dirk Deridder

Nederlandstalige Samenvatting

Software ontwikkelaars verantwoordelijk voor het aanpassen van bestaande software moeten op de hoogte zijn van onderliggende afhankelijkheden in het gedrag van een systeem. Zulke afhankelijkheden kunnen ontstaan bij de initiële ontwikkeling van software, waarbij bepaalde beslissingen moeten worden genomen wat de design van het systeem betreft. Dit komt vooral voor bij technisch en algoritmisch complexe systemen omdat meerdere technologisch hoogstaande componenten in eenzelfde implementatie moeten geïntegreerd worden. Dit geeft aanleiding tot subtiele nuances in de broncode die fout systeemgedrag tot gevolg kunnen hebben. Deze afhankelijkheden, of *design invarianten*, vormen een ernstig probleem voor ontwikkelaars omdat ze meestal enkel impliciet gekend zijn. Bijgevolg kunnen aanpassingen aan de broncode zo het breken van deze invarianten tot gevolg hebben.

In dit proefschrift wordt een oplossing uitgewerkt om de ontwikkeling van technisch en algoritmisch complexe systemen te ondersteunen door design invarianten te gaan documenteren en verifiëren gebruik makende van een lichtgewicht mechanisme. De belangrijkste bijdrage van de voorgestelde aanpak is het gebruik van *temporeel logisch programmeren* als uitvoerbare specificatietaal om design invarianten te kunnen specifiëren. Een belangrijk voordeel van dit declaratief formalisme is het hoog abstractie niveau dat wordt aangeboden. Naast de declaratieve eigenschap van een logisch formalisme wat het gebruik van hoog niveau concepten toelaat, kunnen ook abstracties gemaakt worden over tijdsstructuren door gebruik te maken van tijdsafhankelijke operatoren. Dit resulteert in de documentatie van de design invariant in een gedragsmodel gespecifieerd op een hoog abstractieniveau.

In tweede instantie wordt in de voorgestelde oplossing een causale link voorzien tussen de specificatie van de design invariant en de broncode die een lichtgewicht verificatie toelaat op basis van dynamische analyse. Hierbij wordt gebruik gemaakt van een broncode instrumentatie mechanisme gebaseerd op logisch meta programmeren. Zo kunnen run-time events zorgvuldig geselecteerd worden en meteen op conceptueel niveau gespecifieerd worden in plaats van op broncode niveau. De voorgestelde aanpak is *goal-driven* door het combineren van een selectieve instrumentatie met het

zorgvuldig kiezen van een uitvoeringsscenario relevant voor een bepaalde design invariant, wat de toepasbaarheid op delen van grotere programma's mogelijk maakt.

Om de aanpak te valideren werd het BEHAVE platform gebouwd. Dit platform biedt ondersteuning voor het documenteren en lichtgewicht verifieren van design invarianten in C programma's. Als validatie werd BEHAVE gebruikt ter ondersteuning van de ontwikkeling van de Pico virtuele machine. Pico bleek een ideaal experimenteel platform om onze aanpak te valideren doordat de ontwikkelaar van Pico beschikbaar was. Zo kon enerzijds inzicht verkregen worden in de genomen beslissingen aangaande het ontwerp van Pico en anderzijds konden eventueel gevonden inconsistencies geverifieerd worden. Daarbovenop wordt Pico nog steeds actief gebruikt en ontwikkeld. Gebruik makende van BEHAVE werd er een set van drie design invarianten gedocumenteerd en geverifieerd voor Pico: run-time programma documentatie, garbage collection en staartrecursie optimisatie.

Acknowledgments

I am greatly indebted to my promoter Theo D'Hondt for his unconditional support throughout all these years. As head of the Programming Technology Lab, he succeeded in creating not only a stimulating but also a fun environment to work in. He encouraged me in whatever research topics I wanted to pursue and he always kept believing in me. During the final months of the writing process he was always available for answering the numerous (Pico) questions I had, for proofreading my dissertation text and for replying the many mails I spammed him with. Thanks a lot Theo!

I am especially grateful to my co-promoter Dirk Deridder. About a year ago, he gave me the mental push I needed at a time when I had almost given up on ever finishing this dissertation. He created a motivating work environment by holding weekly meetings and by initiating many fruitful discussions in our office. I am also thankful for his many revisions of my dissertation text. But I would especially like to thank him for helping me to regain my enthusiasm for doing research and for being a great friend!

I thank the 'Instituut voor de Aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen (IWT)' as they funded the ARRIBA project in which a large part of this research was conducted.

Thanks to Kim Gybels for his implementation of the Zombie library as the BEHAVE platform is based on it. Also a big thank you to Coen De Roover for having taken the initiative to jointly write the first BEHAVE paper and for his implementation of the temporal logic meta interpreter.

A special thank you goes to Johan Brichau and Coen De Roover for proofreading parts of this dissertation text.

I thank the members of my jury, Arie van Deursen, Serge Demeyer, Viviane Jonckers and Wolfgang De Meuter for their helpful comments and suggestions on the first version of my dissertation text.

I am also indebted to Brecht Desmet and Coen De Roover for relieving me of my teaching duties during these last few months. Also thanks to Kris Gybels, Kris De Schutter and Thomas Cleenewerck for taking over some of my project work and other tasks during the final stages of writing.

I would like to thank my colleagues (and ex-colleagues) at PROG for providing a challenging and fun environment to work in. So thank you Andy Kellens, Bram Adams, Brecht Desmet, Charlotte Herzeel, Coen De Roover, Dirk Deridder, Dirk Van Deun, Elisa González Boix, Ellen Van Paesschen, Jessie Dedecker, Johan Brichau, Johan Fabry, Jorge Vallejos Vargas, Kris Gybels, Kris De Schutter, Linda Dasseville, Pascal Costanza, Peter Ebraert, Sofie Goderis, Stijn Mostinckx, Stijn Timbermont, Thomas Cleenewerck, Tom Van Cutsem and Wolfgang De Meuter. Also many thanks to our secretaries, Lydie Seghers, Brigitte Beyens and Simonne De Schrijver, not only for helping me out all these years with the (sometimes very tricky) university administration, but also for a friendly chat every now and then.

I am also grateful to Elisa González Boix and Jessie Dedecker for persuading me to go to the gym during the earlier stages of my writing. They convinced me many times (and sometimes I had to convince them) to combine indoor running with chatting about whatever topic we felt like (we were mostly gossiping of course). Not only did it divert my thoughts, but most of all it was fun!

I would like to thank my friends for providing me with a necessary distraction once in a while. And this cannot have been easy as I was not a very enjoyable person during these past months. Especially Elisa had to endure a lot; she always tried to cheer me up, but she often had a hard time doing so. Thanks for being patient with me!

I also want to thank my family for their moral support and for putting things into perspective. They always offered a listening ear on the phone when I was feeling down again. I thank Olga, with whom I could share many experiences, although working in completely different fields. And I especially thank my mum and my grandmother not only for having given me the opportunity to study, but most of all for believing in me during all these years and for letting me do things at my own pace.

Finally, I would like to thank my partner Frank. I admire him for not having fled the house at stressful times. He supported me in every possible way he could, by taking care of the household chores but also by proofreading parts of this dissertation. But most of all I thank him just for being there and for convincing me not to give up at those times I needed it the most.

Isabel Michiels
August 2007

Table of Contents

Nederlandstalige Samenvatting **iii**

Acknowledgments **v**

1 Introduction **1**
- 1.1 Research Context . . . 1
- 1.2 Problem Statement . . . 4
- 1.3 Thesis Approach . . . 5
- 1.4 Thesis Contributions . . . 7
- 1.5 Organisation of the dissertation . . . 8

2 Invariants in Software **11**
- 2.1 What are Invariants? . . . 12
- 2.2 Characterising Software Behaviour . . . 13
- 2.3 Uses of Invariants . . . 15
- 2.4 An Overview of Invariants in Software . . . 16
  - 2.4.1 Embedded Assertions . . . 17
  - 2.4.2 Class/Object Invariants . . . 18
  - 2.4.3 Detecting Dynamic Invariants . . . 21
  - 2.4.4 Law-Governed Architecture . . . 22
  - 2.4.5 Communication Protocols . . . 23
  - 2.4.6 Other Behavioural Invariants in Software . . . 23
- 2.5 Specification Languages for Invariants . . . 24
  - 2.5.1 The Z Notation . . . 24
  - 2.5.2 Temporal Logic . . . 25
  - 2.5.3 Other Behavioural Specification Languages . . . 27
- 2.6 A Particular Type of Invariants: Design Invariants . . . 28
- 2.7 Conclusion . . . 30
- 2.8 Summary . . . 30

3 Program Analysis for Supporting Design Invariants 33
3.1 Tool Support for Design Invariants 34
3.2 What is Program Analysis? 35
3.3 Analysis of Models 36
3.4 Analysis of Code 39
3.4.1 Static vs. Dynamic analysis 40
3.5 Static Analysis 41
3.5.1 Comparison Criteria 42
3.5.2 Data-flow Analysis 43
3.5.3 Evaluation 45
3.6 Dynamic Analysis 45
3.6.1 Comparison Criteria 46
3.6.2 Relevant Application Domains 50
3.6.3 Aspect-Oriented Programming 53
3.6.4 Evaluation 54
3.7 Conclusion 54
3.8 Summary 56

4 A Goal-Driven Approach for the Documentation and Lightweight Verification of Design Invariants 57
4.1 Need for Documenting and Verifying Design Invariants 58
4.1.1 Problem Statement 60
4.1.2 Implicit Design Invariants 61
4.1.3 Detached Design Invariants 62
4.1.4 Non-Oblivious Design Invariants 63
4.1.5 Lack of Partialness 63
4.2 A Lightweight Goal-Driven Verification Approach 64
4.2.1 Verifying Design Invariant Behaviour 65
4.2.2 Specifying Design Invariant Behaviour 66
4.3 Reasoning About Program Behaviour 68
4.3.1 A Partial Model of Program Behaviour 68
4.3.2 An Executable Behavioural Formalism 69
4.4 Obtaining High-Level Events 71
4.4.1 Logic Meta Programming 73
4.4.2 A Means for Event Selection 74
4.5 A Four-Step Recipe 76
4.6 Summary 76

5 BEHAVE: A Lightweight Verification Platform for C 79
5.1 History of BEHAVE 80
5.2 Logic Meta Programming 81
5.2.1 Logic Programming 81
5.2.2 Frequently Used Predicates 85

5.2.3 Smalltalk Open Unification Language (SOUL) 86
5.2.4 Temporal Logic Programming 86
5.2.5 The Representational Mapping 87
5.3 BEHAVE: A Dynamic Analysis Platform 90
5.4 The BEHAVE Architecture 93
5.4.1 An Overview 94
5.4.2 The Reification Module 95
5.4.3 The Instrumentation Module 98
5.4.4 The Reasoning Module 104
5.4.5 The Temporal Logic Meta Interpreter 106
5.5 A Four-step Recipe for Using BEHAVE 108
5.5.1 Step 1: Identifying High-Level Run-Time Events 109
5.5.2 Step 2: Specifying the Behavioural Model 109
5.5.3 Step 3: Application-Specific Instances of High-Level Events . 110
5.5.4 Step 4: Lightweight Consistency Verification 112
5.6 BEHAVE Tool Support 113
5.7 System Overhead 115
5.8 Conclusions 116
5.9 Summary 117

6 Using BEHAVE for Supporting Program Development 119
6.1 The Pico Language Interpreter 120
6.1.1 The Pico Execution Model 120
6.1.2 The Pico Memory Model 123
6.2 Supporting Development in Pico 124
6.3 Case Study 1: Verifying Behavioural Program Documentation 125
6.3.1 Behavioural Program Documentation in Pico 125
6.3.2 Using BEHAVE to Document and Verify the Behaviour of the Pico Interpreter 126
6.4 Case Study 2: Verifying Garbage Collection 133
6.4.1 Using BEHAVE to Document and Verify Garbage Collection in Pico 134
6.5 Case Study 3: Verifying Tail Recursion Optimisation 140
6.5.1 Tail Recursion in Pico 141
6.5.2 Using BEHAVE for Verifying Tail Recursion Optimisation . . 143
6.6 Conclusions 150
6.7 Summary 152

7 Conclusion and Future Work 155
7.1 Summary 155
7.2 Conclusion 157
7.3 Future Work 160
7.3.1 Possible Improvements and Additions for BEHAVE 160

7.3.2 Model-Driven Engineering Support . . . 162
7.3.3 Computer Science Education . . . 163
7.3.4 Garbage Collection as an Aspect . . . 163
7.3.5 Using BEHAVE for Supporting Evolution . . . 165
7.4 Closing Remarks . . . 166

A The Full Representational Mapping 167

B BEHAVE Instrumentation Module 171
B.1 Generation Layer . . . 171
B.2 Aspect Layer . . . 177

C BEHAVE Temporal Logic Meta Interpreter 181
C.1 Documentation . . . 181
C.2 Implementation of the meta interpreter . . . 182
C.2.1 Marking the timeline . . . 182
C.2.2 The Temporal Logic Operators . . . 182
C.2.3 The `prove/2` predicate . . . 185

D BEHAVE for Pico 1.0 187
D.1 Documentation . . . 187
D.2 Implementation of all predicates . . . 188
D.2.1 User-defined `Intercept/3` predicates . . . 188
D.2.2 Reasoning Predicates . . . 189
D.2.3 Behavioural Models . . . 191
D.2.4 Keywords . . . 193

E BEHAVE generated code excerpt 195

Bibliography 201

Index 213

List of Figures

1.1 C source code excerpt of a virtual machine 3

2.1 Different types of invariant constraints (adapted from [Bol04]) 14
2.2 A simple class illustrating the use of an object invariant 18

3.1 Model checking approach 38
3.2 A finite state machine representing a modulo 3 counter 38
3.3 Control-flow graph of an iterative factorial function 43
3.4 Example program with live variables at every program point (taken from [Sch]) 44

4.1 Lightweight verification of a behavioural model against actual program behaviour 66
4.2 An event as a unit of program behaviour 69
4.3 A Conceptual Representation of Mapping Source Code Entities to High-Level Events 72
4.4 The *intercept* mechanism for identifying application-specific instances of high-level events 73
4.5 A logic meta programming setup (taken from [Vol98]) 74
4.6 An example of the *intercept* mechanism using LMP 75
4.7 A four-step recipe for applying the proposed approach 76

5.1 A simple logic database 87
5.2 A logic meta programming setup (taken from [Vol98]) 88
5.3 Tree representation of a C function definition 90
5.4 Using BEHAVE as a dynamic analysis platform 91
5.5 The BEHAVE Architecture 94
5.6 Simple C Stack implementation - Stack.h 95
5.7 Logic representation of the `push` function depicted in figure 5.6 96

5.8 Logic representation examples for the stack example depicted in figure 5.6 . . . 96
5.9 An excerpt of the BEHAVE basic layer . . . 97
5.10 An excerpt of the BEHAVE generation layer . . . 99
5.11 The `generateAux/3` predicate . . . 100
5.12 An example declaration of the `generateAux2/3` predicate for generating an assignment expression . . . 101
5.13 An excerpt of the BEHAVE aspect layer . . . 102
5.14 Using the `captureEvent/4` predicate for instrumenting all stack pop operations for the running example depicted in figure 5.6 . . . 102
5.15 Keywords denoting how associated run-time values . . . 104
5.16 Example predicates of the BEHAVE reasoning layer . . . 105
5.17 The `expressionIn/3` predicate . . . 105
5.18 Marking the timeline in the temporal logic meta interpreter for the events in the execution trace . . . 106
5.19 Implementation excerpt from the temporal logic meta interpreter . . . 107
5.20 A Four-step Recipe for Using BEHAVE . . . 108
5.21 Behavioural Model specifying invariant stack behaviour . . . 110
5.22 Source code and corresponding behavioural documentation of a C stack implementation. . . . 113
5.23 A screen capture of BEHAVE tool support in Smalltalk . . . 114

6.1 The assignment(ASS) continuation - PicoEva.c . . . 121
6.2 The Pico continuation network . . . 122
6.3 Pico execution stacks - Implementation view . . . 123
6.4 Logical Representation of Pico Behavioural Program Documentation . 126
6.5 Source code and corresponding behavioural documentation extracts of the Pico interpreter. . . . 127
6.6 Specification of the behavioural model for active behavioural program documentation in Pico . . . 128
6.7 High-level event specification of a continuation entry and exit . . . 129
6.8 Application-specific instances `continuationEntry` and `continuationExit` . . . 129
6.9 The `cntName` and `cntPointer` keywords . . . 130
6.10 Excerpt of an execution trace logging all continuation entries and exits 131
6.11 A well-chosen Pico execution scenario . . . 132
6.12 Conceptual representation of heap memory . . . 133
6.13 Using the BEHAVE platform for verifying Pico memory management . 135
6.14 Excerpt of an execution trace logging possible garbage collects and temporary variable usage . . . 136
6.15 Specification of the behavioural model for verifying garbage collection in Pico . . . 137
6.16 High-level event specification of garbage collection behaviour . . . 137

6.17 Application-specific instances for the garbage collection design invariant 138
6.18 The `var(V)` and `assVar` keywords 138
6.19 The `COX` continuation from PicoNat.c 140
6.20 Tail-recursive factorial function 141
6.21 Recursive factorial function 141
6.22 The return continuation (`RET`) from PicoEva.c 142
6.23 Non-optimised tail-recursive call structure 142
6.24 Using the BEHAVE platform for identifying tail recursion optimisation 143
6.25 The behavioural model for capturing tail recursion optimisation 144
6.26 The adapted behavioural model 145
6.27 High-level events specification for tail recursion optimisation 145
6.28 Application-specific instances for tail recursion optimisation 146
6.29 The `topCnt` keyword 146
6.30 Pico implementation of the quicksort algorithm 147
6.31 The `installReturnOnContinuationStack/2` predicate 147
6.32 The `BEG` continuation from PicoNat.c 149
6.33 The `BXG` continuation from PicoNat.c 149
6.34 Tail recursion check in the `BND` continuation from PicoEva.c 150
6.35 Optimisation of the `BXG` continuation from PicoNat.c 150

7.1 Pico code excerpt of `_eval_CAL_` continuation 164
7.2 A counter object in Pico 2.0 165

List of Tables

5.1 The representational mapping for reasoning about C programs 89
5.2 BEHAVE system overhead comparison 115

C.1 `Not, Next` and `Previous` Operators 181
C.2 `Sometime` Operators 181
C.3 `Always` Operators 182
C.4 Compound Operators 182

D.1 Pico Reasoning Predicates 187
D.2 Other Pico Reasoning Predicates 188
D.3 Pico 1.0 keywords 188

Chapter 1

Introduction

1.1 Research Context

Current software systems are becoming more complex than ever. Not only the vast amount of functional variation is growing, the underlying complexity of the implementation is increasing as well. This complexity poses several challenges to the programmer who sees a daunting task set forth to create a reliable system.

System complexity is cited as a major reason for the difficulties software engineers encounter when dealing with large software systems [FPB87]. However, not only the size of systems causes complexity problems [Min96b]. A range of software systems exist which need to integrate several technologically and algorithmically challenging system components. Consider the Pico virtual machine as an example. Pico is an interpreter for a fairly simple programming language which integrates several technologically sophisticated features such as support for reflection and meta programming, continuations and first class environments, techniques for supporting automatic memory management, tail-recursion optimisation, etc. Integrating all these components creates a complex software system exhibiting subtle interactions, which makes such a system difficult to maintain and evolve.

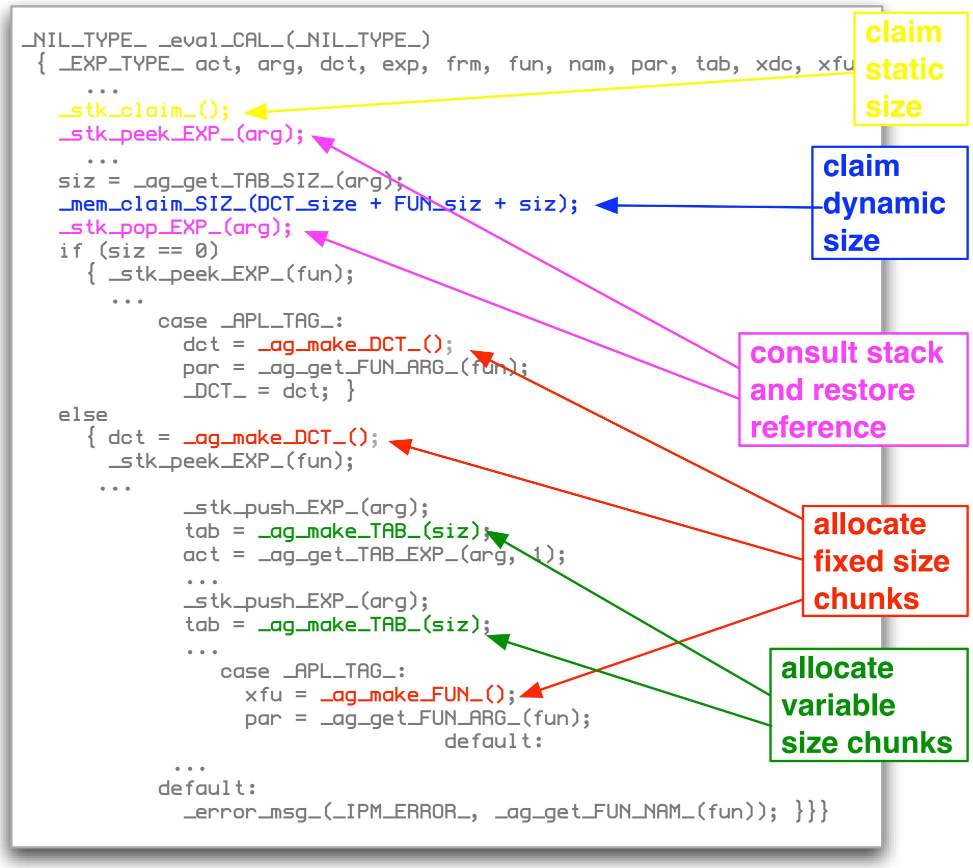
Dealing with software complexity manifests itself in different ways. A major difficulty to overcome is the adaptation of existing software functionality. One possible way to support the changes is to make the *process of change* easier by preparing software systems *structurally* [MBZR03]. This is often realised by advocating a well-designed system with a clean separation of concerns (having low coupling and high cohesion) according to the decomposition mechanism of the programming language at hand. Furthermore, modularising cross-cutting concerns which cannot be captured in a language component can be factored out into an aspect using the aspect-oriented paradigm [KM05a, KM05b]. As a result, when adaptations to the software have to be made, changes addressing a particular concern are more localised and are therefore easier to manage.

Next to aiding the process of change, controlling the *consequences of a change* on other parts of the software is equally important to keep software reliable. Especially the impact on the *behaviour* of a program is difficult to control as adapting certain parts of a program (well-designed or not) might break intended underlying behaviour in other parts. These underlying behavioural constraints, usually referred to as *invariants*, play a crucial role as they can protect a programmer from making changes which the correctly working behaviour of a software program depends on. Therefore, to see if they are satisfied at all times during the evolution of a software implementation, they must be made explicit and machine-verifiable, which is often done in the form of annotations or embedded assertions.

Although assertions have shown to contribute significantly to the reliability of source code [Ros95, YB94], they can only express more local invariants as they are annotated at specific places in the source code. Moreover they are also tightly coupled to the implementation as they are typically specified using the programming language they are embedded in. Such invariants pose constraints on local state or data and are referred to as data invariants. However, other invariants exist which are non-local and which not only constrain the data at a particular execution point but also the order in which statements can be executed. Such invariants pose a real problem as they are much more difficult to express and verify.

Especially systems which integrate technologically advanced components exhibit many subtle dependencies which are non-local and which depend on expert knowledge about the internal operations of a system. This type of invariants are referred to as the *invariants of a system's design*. They are unavoidably introduced during the software design phase, when a developer makes particular design choices which undoubtedly have an impact on other parts of a system. These invariants represent the specific characteristics that limit the future adaptation, flexibility and evolvability of a software implementation [ABE$^+$04]. It is exactly this type of invariant that we study in this dissertation.

To demonstrate the subtleties of such design invariants in the code of a technically complex system, we present an excerpt of C source code of the formerly mentioned Pico virtual machine listed in figure 1.1. This piece of source code forms a critical section for implementing automatic garbage collection in Pico. It represents a continuation function that takes care of evaluating a Pico function application. As shown by the coloured boxes, different concerns interact and cross-cut throughout the code. When a static allocation is needed for storing Pico objects of a fixed size (denoted in red in figure 1.1), a static memory claim can easily be performed at the beginning of the continuation function (the first arrow denoted in yellow). However, the developer faces a complicated dependency problem when a dynamic allocation occurs for storing a Pico object of variable size (e.g. a Pico table). In that case, a memory claim for claiming dynamic size should be included in the code (the arrow denoted in blue in figure 1.1) which severely burdens the developer with a chicken-and-egg problem. On the one hand this memory claim cannot be placed at the very beginning of the continuation as the size of the needed memory has to be known first. And on the other hand, the further down this claim is placed in the source code of the continuation, the more



```
_NIL_TYPE_ _eval_CAL_(_NIL_TYPE_)
 { _EXP_TYPE_ act, arg, dct, exp, frm, fun, nam, par, tab, xdc, xfu
     ...
   _stk_claim_();
   _stk_peek_EXP_(arg);
     ...
   siz = _ag_get_TAB_SIZ_(arg);
   _mem_claim_SIZ_(DCT_size + FUN_siz + siz);
   _stk_pop_EXP_(arg);
   if (siz == 0)
     { _stk_peek_EXP_(fun);
       ...
           case _APL_TAG_:
             dct = _ag_make_DCT_();
             par = _ag_get_FUN_ARG_(fun);
             _DCT_ = dct; }
   else
     { dct = _ag_make_DCT_();
       _stk_peek_EXP_(fun);
      ...
             _stk_push_EXP_(arg);
             tab = _ag_make_TAB_(siz);
             act = _ag_get_TAB_EXP_(arg, 1);
             ...
             _stk_push_EXP_(arg);
             tab = _ag_make_TAB_(siz);
             ...
                case _APL_TAG_:
                  xfu = _ag_make_FUN_();
                  par = _ag_get_FUN_ARG_(fun);
                                  default:
            ...
           default:
             _error_msg_(_IPM_ERROR_, _ag_get_FUN_NAM_(fun)); }}}
```

claim static size
claim dynamic size
consult stack and restore reference
allocate fixed size chunks
allocate variable size chunks

Figure 1.1: C source code excerpt of a virtual machine

temporary variables have been used for which their references have to be saved and restored respectively before and after the performed memory claim (denoted in pink in figure 1.1).

This type of invariants is obviously problematic as they are usually only known in the head of a developer and thus *implicitly* present in the software. Additionally, design invariants are often *cross-cutting* the entire application code as they represent behavioural entities depending on scattered source code. And on top of that they are non-externally verifiable, which means that they cannot be traced by investigating output values. They depend primarily on the order in which particular source code statement (or events) are executed, which requires expert knowledge about the internal workings of a system.

In the context of software systems with several algorithmically complex components, design invariants represent the subtle interplay between different program parts that realise a particular concern often only known to the developer. They severely limit future system development because after any change to the code these design invariants might be violated. We want to make design invariants *explicit* by specifying them in a behavioural specification language.

Given this complex interplay of crosscutting entities, verifying design invariants manually is not desirable. Therefore, next to making design invariants *explicit*, we want to apply a program analysis approach to make them *machine-verifiable*. Existing formal verification approaches such as model checking, but also other static analysis approaches are not suited in this context as they provide no means for focusing the analysis on a behavioural cross-cutting entity. Instead they work on an abstract behavioural model representing an entire program's behaviour, which makes these approaches too exhaustive to promote practical use. Dynamic analysis approaches support a lightweight methodology as they analyse execution traces instead of exhaustively exploring an entire system model. They are efficient and precise, but their results depend on the provided program input. The power of these approaches lies in the fine-tuning of its characteristics, depending on how run-time information is gathered [HLL04] and what means of event selection is used. However, most existing approaches represent behaviour in low-level implementation constructs [GOA05, Roo04, DFW04, DGD05], which can be a disadvantage for promoting practical use as any behavioural representation is therefore tightly linked to the source code.

In view of the characteristics of a design invariant, namely that they are non-externally verifiable and therefore mainly concerned with the order of events, and that they cross-cut an entire system, we envision a lightweight approach which offers a means to focus the analysis on only the behaviour relevant for a particular design invariant.

1.2 Problem Statement

In this dissertation, we want to offer support to a programmer for the development of technically and algorithmically complex software systems by making these design invariants explicit and at the same time machine-verifiable.

Having considered existing analysis approaches such as formal verification, static and dynamic analysis, we concluded that all of them have certain shortcomings to support design invariant verification. In summary, an approach is needed which needs to address the following main problems:

- **Implicit Design Invariants**
  Design invariants impose underlying behavioural constraints on a system, but they are implicitly present in the software. Hence, every time a change is made to the software, the design invariant might be violated, possibly inducing severe program errors.

- **Detached Design Invariants**
  Even if design invariants are made explicit, they are often detached from the source code. As design invariants are system-wide and may cross-cut an entire system, this poses a problem as it is difficult and time-consuming to check them

manually. Moreover, as design invariants might depend on the run-time state of a system, such a check might even be impossible to perform. On top of that, these checks have to be done *every* time a change is being made to the system.

- **Non-Oblivious Design Invariants**
  As a design invariant has to be checked *every* time a change is made to the system, the impact of changes to the design invariant description presents another problem. Specifying the invariant in low-level implementation constructs creates a tight coupling with the application's source code. Such a link with the source code inhibits practical use as the specification needs to be adapted *every* time a change is made to the source code.

- **Lack of Support for Partialness**
  To check a particular underlying behavioural constraint, existing program analysis approaches analyse the complete system behaviour by exhaustively checking the possible program states. Other approaches allow to separately analyse modules of a system as they are decomposed by the programming language at hand. However as design invariants can be cross-cutting entities, this would still entail analysing an entire program's behaviour, which seriously inhibits practical use. Such approaches lack support for partialness as they are not able to focus the analysis on only a particular part of the software's behaviour.

1.3 Thesis Approach

In this dissertation, we outline an approach for the documentation and lightweight verification of behavioural dependencies in source code, named *design invariants*. To address the problems identified in the problem statement, we target the following solutions:

- **Making Design Invariants Explicit, Machine-Verifiable and Oblivious**
  First, design invariants are made explicit by specifying their (un)wanted behaviour in a behavioural model which is at the same time machine-verifiable. This is done by specifying behaviour in a high-level executable behavioural formalism which makes the design invariant model oblivious from low-level implementation constructs.

- **A Lightweight and Goal-driven Approach**
  Second, the consistency of the behavioural model of the design invariant is verified in a lightweight manner against only the actual program behaviour which is relevant to that particular design invariant using dynamic analysis.

Specifying Design Invariant Behaviour

As a first part of our approach, a behavioural specification formalism is needed to make design invariants *explicit*. We propose the use of temporal logic programming as behavioural formalism for specifying design invariants in high-level behavioural models. Such a model is specified as a set of temporal assertions in terms of high-level events recorded in the execution trace. Such a formalism supports the event-based nature of design invariants. It is also declarative which makes it extremely suitable for introducing high-level concepts in the behavioural descriptions. The temporal operators make this formalism extremely suited to model the temporal relationship between the run-time events.

Next to specifying the high-level behavioural model of a design invariant, we also specify the run-time events *over* which these temporal assertions are checked at a high level of abstraction. This allows us to specify the high-level behavioural model of the invariant directly in terms of high-level events instead of low-level programming constructs. High-level run-time events are represented as logic facts representing a partial model of program behaviour, which makes the assertions representing design invariant behaviour machine-verifiable against the high-level run-time events.

Verifying Design Invariant Behaviour

As we want to support the development process of technically and algorithmically challenging software in a practical way, we propose a lightweight verification approach that is causally linked with the source code. This is realised by using *dynamic analysis* where the program under investigation is executed along a well-defined execution scenario. The actual observed behaviour is obtained through executing selectively instrumented source code, which in addition to executing the program, records certain run-time events of interest. Automatic verification thus amounts to checking whether the events recorded in the execution trace exhibit the desired behaviour as defined in the design invariant model. This implies that the verification results are always relative to the user input. Although this only allows us to find invariant violations instead of proving their absence, a well-chosen execution scenario generally offers a convenient way to focus the verification on specific parts of a larger program.

Furthermore, the approach requires a sophisticated aspect-like code instrumentation scheme for obtaining high-level run-time events. To identify constructs in an application's source code that give rise to particular events of interest, we propose the use of a logic meta programming approach for representing the structure of a base language program as a logical representation at the meta level. Constructs of interest are then described as a logic program representing a condition and checked against all logic parse tree nodes. Instrumentation for recording a high-level run-time event is then generated. It is left up to the user to specify *what* high-level program points are of interest (and which associated values) and *how* they should be recorded in the execution trace.

The BEHAVE Platform: A Proof-Of-Concept Implementation

A proof-of-concept platform called BEHAVE is implemented that exhibits the elements of the outlined approach. BEHAVE is a lightweight verification platform for supporting design invariants in C. It is fully implemented in the logic programming language Prolog. Modest tool support is available in Smalltalk to aid in setting up and using the experimental platform. A C parser transforms a C base language program into a logical representation up until the statement level.

The three main components of the platform are:

- *a reification module* containing a logic representation of a C base language program to be analysed,
- *an instrumentation module* which generates C source code from the logical representation while checking pointcut expressions of interest defined by the user of the system, and
- *a temporal logic meta interpreter* which verifies the design invariant models against the actual program behaviour.

To test the validity of our claims, we applied the BEHAVE platform to the Pico virtual machine. As the availability of knowledge about made design decisions and about the internals of a program is crucial for applying our approach, Pico formed the ideal case study. The original developer was available to give us insight in the main design decisions that were undertaken for implementing Pico. This is important since it not only allowed us to question the developer about which design invariants the Pico behaviour should adhere to during program execution, but also to verify possible invariant violations found by BEHAVE.

To support the development process of Pico, we expressed and verified the undesired behaviour of three representative design invariants of some of its main technological components. Active behavioural program documentation is created of the Pico execution model, garbage collection is now supported and tail recursion optimisation can now be verified. Further development of Pico is now supported as these design invariants are made an explicit and verifiable part of future implementations of Pico.

1.4 Thesis Contributions

The main contributions of the research presented within the context of this dissertation can be summarised as follows:

- **Identification of *Design Invariants***
  Design invariants were identified and defined as important underlying behavioural dependencies which severely limit future system development. Their characteristics were discussed and they were found to be non-externally verifiable and possibly cross-cutting an entire application.

- **A Goal-Driven Approach for the Documentation and Lightweight Verification of Design Invariants**
  An executable descriptive formalism is used for specifying design invariants in a high-level behavioural model. A causal link with the source code is supported allowing lightweight verification based on dynamic analysis. The approach is *goal-driven* in the sense that lightweight consistency checking is performed against only a partial model of program behaviour and by choosing a well-defined execution scenario.

- **The BEHAVE Platform**
  Implementation of the BEHAVE platform as a proof-of-concept of our proposed approach for supporting design invariants. BEHAVE is a lightweight verification platform for C.

- **Validating the Use of BEHAVE by Supporting the Development of Technically and Algorithmically Complex Software**
  The BEHAVE platform is validated by supporting three technically sophisticated design invariants in the Pico language interpreter: creating active behavioural program documentation, supporting garbage collection and checking tail recursion optimisation.

1.5 Organisation of the dissertation

Chapter 2: Invariants in Software
Investigates the concept of invariants in software development. It is discussed how invariants are specified and what specification languages exist. A definition is given of a design invariant, i.e. the type of invariant which is targeted in this dissertation.

Chapter 3: Program Analysis for Supporting Design Invariants
Discusses existing program analysis approaches and evaluates their suitability for supporting design invariants.

Chapter 4: A Goal-Driven Approach for the Documentation and Lightweight Verification of Design Invariants
Proposes an approach for the documentation and lightweight verification of design invariants. In essence, design invariants are made explicit by documenting their dynamic behaviour in descriptive behavioural models which can be verified in a lightweight manner throughout an application's lifetime.

Chapter 5: BEHAVE: A Lightweight Verification Platform for C
Illustrates the practical platform which was implemented as a proof-of-concept to validate our approach outlined in chapter 4. It provides a technical discussion of the main components that constitute the platform.

Chapter 6: Using BEHAVE for Supporting Program Development
Illustrates the use of the BEHAVE platform and the validation of our proposed approach by specifying three representative design invariants of the Pico language interpreter: active behavioural program documentation, garbage collection and tail recursion optimisation.

Chapter 7: Conclusion and Future Work
Presents the main conclusions of this dissertation. A summary of the work is presented, the main contributions of this approach are emphasised and we formulate the conclusion. We end with a presentation of the future research directions.

Appendices
Appendix A contains the full representational mapping which is used in the LMP set-up for reasoning about C programs. Appendix B lists the instrumentation module of BEHAVE for performing selective code instrumentation. The documentation and implementation of the temporal logic meta interpreter is presented in appendix C, while appendix D contains a reuse framework for using BEHAVE to reason about the behaviour of Pico 1.0. Finally, appendix E shows a BEHAVE generated code excerpt for one of the performed Pico experiments.

Chapter 2

Invariants in Software

The complexity and size of today's software systems provides many occasions for developers to introduce faulty and erroneous behaviour in an implementation. Any small change which has to be made to software can have a major impact on other parts of a system. Most program development support tries to make the process of change easier by advocating well-structured systems with a clean separation of concerns. But what about the consequences of change on those parts of a system that are not allowed to change when a system evolves? In this chapter we elaborate on *invariants in software*, i.e. crucial behavioural assertions that must hold through every change cycle.

In this chapter we investigate the concept of an invariant in today's software development. On the one hand we elaborate on what is meant by an invariant and the factors that influence the different types of invariants. On the other hand we discuss *how* invariants can be specified and what specification languages exist. We give a definition of a *design invariant*, i.e. a type of invariant which is targeted in this dissertation. Note that we only elaborate on the *notion* of an invariant in this chapter. How these invariants are checked or verified is the subject matter of the next chapter.

We start by pinpointing what invariants really are in section 2.1 by comparing some of the frequently adopted definitions of invariants. We continue our discourse in section 2.2 by discussing in more detail the two fundamental approaches to reason about the behaviour of a system. We then elaborate in section 2.3 on the different uses of invariants, i.e. the different roles they can play in the software development life cycle. Continuing in section 2.4, we survey existing invariants by seeing what type of invariants they can specify and how they are specified. In section 2.5 we have a closer look at existing formal and non-formal specification languages for specifying invariant behaviour. In section 2.6 we define a type of invariant called *design invariants* which will be referred to throughout the remainder of this dissertation and we state the main characteristics of these invariants. We end with a conclusion in section 2.7 and a chapter summary.

2.1 What are Invariants?

Software systems are becoming more and more complex. Although complexity can be attributed to the continuous increase in size of software, it is a fact that complexity is an inherent and irreducible property of software systems [FPB87] .

To deal with this inherent complexity, tool support for changing and evolving a system is becoming crucial. On the one hand, tool support for aiding the evolution of systems tries to ease *the process of making changes* by providing mechanisms for clearly separating concerns of software systems [MBZR03]. When changes have to be made, they are more localised and made faster and in a straightforward way. On the other hand, software analysis tools are built to understand the *impact of a change on the behaviour of a system.* Because making a change to even a well-designed structural system with a clear separation of concerns (with low-coupling and high cohesion) might break the underlying intended behaviour.

Invariants in software form a crucial part of a system's *behaviour.* They do not just represent characteristics of a system at a certain moment of its evolution, but they represent *constraints* which must hold through *every* evolution cycle if the system is working correctly. They are essential to the comprehensibility of a software system in general. However, they pose a problem as they are rarely made explicit and therefore, when changes are made to a system, they can easily be violated. Explicitly stated program invariants can help programmers by documenting certain aspects of program execution and identifying program properties that must be preserved when modifying code. As invariants represent an immutable part of the *behaviour* of a program, the fundamental ways of *specifying* behaviour need to be taken into account.

Different types of invariants exist as they are used in different contexts and in different phases of software development. After providing some definitions of invariants, in section 2.2 the fundamental ways to specify program behaviour are addressed.

Definitions The term 'invariant' is being used extensively in the context of software and also in mathematics. In the software engineering community, different definitions are used. Evans [Eva01] states that 'invariants are properties of software that are always true at particular program points'. According to Ernst et. al. [ECGN99] invariants are 'program properties that must be preserved when modifying code' which is already less restrictive. Broadly speaking, an invariant is formulated as something, i.e. a condition, data value, object or class, that does not change, or *should not*, if the system is working correctly. In the totally different context of network architectures, invariants are referred to as a negative entity which 'describes aspects of a design that limits its changeability [ABE$^+$04]. In general terms (closer to mathematics), an invariant is referred to as something that does not change under a set of transformations.

If we interpret the more mathematical definition in a software context we might say that an invariant is 'something' that does not change during the execution of a program. Although the definitions from Evans and Ernst did not explicitly state an invariant as being a property of the *behaviour* of a program, their examples concur with this interpretation. For example they consider a non-zero value of a variable ($a \neq 0$) inside a function to be an invariant because during execution of that function, the expression $a \neq 0$ should always evaluate to true. As a consequence, to enable reasoning about invariants in software, reasoning about the *behaviour* of software is crucial. A first step for doing so is to see *how* system behaviour is specified, which is discussed in the next section.

2.2 Characterising Software Behaviour

In the domain of formal behavioural specification languages, a lot of attention is given to how the state and operations of a software program should be specified. These specifications are used later on to prove the correctness of a *model* of a program with regard to a specified property. Therefore *how* they represent behaviour determines the kind of behavioural reasoning that can be performed at a later stage.

Two kinds of models dominate the abstract description of software: object models and state transition diagrams [JR00]. These models are based on the following *two fundamental ways to reason about the behaviour of software* [Bol04]:

- One can mainly think of behaviour in terms of **state** that changes (or does not change) during program execution, or
- about executed operations or **events** which take place at a certain time when running a program (which also influences a program's state).

Behavioural specification languages tend to favour either one or the other. A possible influence might be the type of a software system. Software systems can be *control-oriented* or more *data-oriented.* Control-oriented systems are systems where the focus is on event ordering and timing (such as communication systems), whereas for data-oriented systems the focus is on storage (organization) and retrieval (information systems such as a library system). However, most systems exhibit properties of both (such as information systems with real-time requirements). An example of a specification approach that tends more towards state-oriented thinking is Z [Spi89], while CSP [Hoa78] (i.e. a special-purpose specification approach for modelling processes) thinks more in terms of system control.

Bolognesi et.al. [Bol04] introduce a conceptual state-event framework for recognising and balancing the two above-mentioned fundamental behavioural reasoning approaches. Figure 2.1 depicts their conceptual framework to pinpoint the *kind* of reasoning that can occur when combining state-event behavioural representations. The authors distinguish four important types of constraints between event types (a certain operation which takes place at run-time, for example a particular message send) and state fragments (part of the global program state) that represent (part of) the behaviour of a program. They identify a *state invariant* as being a *state-to-state constraint* on the values of any number of state variables during system execution (i.e. the type 1

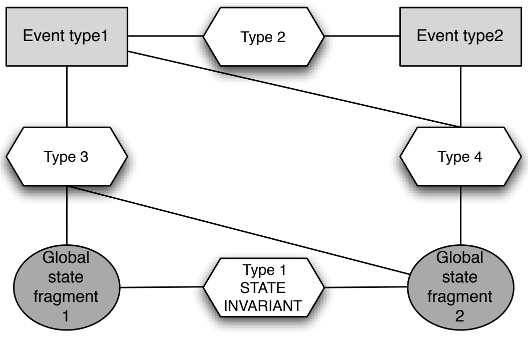


Figure 2.1: Different types of invariant constraints (adapted from [Bol04])

constraint as depicted in figure 2.1). They conclude that although these state invariants might be a convenient starting point to express invariant state behaviour, they provide no information about *what* can actually happen, meaning about the nature and sequence of events.

The pure state constraint (Type 1), also known as a state invariant, together with the *pure event constraint* (Type 2) are the simplest of the constraints. An event can be regarded as an action which takes place at a certain point in time which might influence the state of that program. A pure event constraint thus reasons about an ordering of events which take place at run-time without considering information about a program's state or data. The usefulness of this type of constraint is sometimes doubted since it leaves out any information about a program's state. And in general reasoning about global system state is agreed upon to be of extreme importance.

The mixed constraint types 3 and 4 in figure 2.1 both represent constraints on both fragments of global state and event types. A distinction is made between constraints which consider multiple fragments of program state together with one type of event (Type 3), or multiple types of events while considering only one fragment of a program state. A constraint of type 3 considering multiple fragments of global states might be used for considering parallel execution processes all having their own state. Reasoning about a particular pattern of events along one execution path (representing one global state) is captured by a type 4 constraint. To conclude, this conceptual framework states that, in order to specify and reason about program behaviour, two fundamental approaches are possible. Either behavioural reasoning can be performed primarily based on events which occur at a certain time during execution. Or reasoning can be based on checking the state of program variables. Combining both approaches might lead to the possibility of expressing and reasoning more sophisticated behavioural constraints.

The next section pinpoints the different uses of invariants in software.

2.3 Uses of Invariants

Invariants in software provide very useful information, both for the developer of the software and for different peers working on a software project [Ern00]. Even for software tools such information is important. They are also useful in all aspects of programming and different phases of software development, such as the design phase, during coding, testing, making optimisations, maintenance of the software and in general for supporting evolution.

Invariants also play different *roles* depending on what context they are used in or in what phase of the software life cycle they are employed. In the context of software evolution they play a crucial role as they can protect a programmer from making changes which the correctly working behaviour of a software program depends on. In program development for example, they are part of the formal specification of a system (and its behaviour) which is then consequently refined into a correct program (for example using the Z specification language [Spi89], see section 2.4.6).

The following uses of invariants emphasise how they can optimally be exploited and why developers should care about making invariants an explicit part of software.

Creating reliable programs Invariants have long been considered a useful technique for creating more reliable programs [Gri87]. Program bugs often result from programmers implicitly assuming invariants that are not valid. As an example, consider a function which always returns a non-NULL value [EP06]. Some invariants can also formalise the *contract* of a piece of code, clarifying its intended operations [Mey92b]. Also, thinking about code formally where invariants are an explicit part of a formal specification [Spi89] can result in more disciplined design and hence also implementation. However, also informal use of invariants in the software design phase can help programmers [Gro06].

Program documentation Invariants characterise specific aspects of program behaviour and they provide valuable documentation of a program's algorithms, operations and data structures. As such, they support program understanding, which is a prerequisite for every task during the software development life cycle. Even better is having *active* invariant documentation, as having a causal link between the documentation and the source code provides a way to keep the documentation up-to-date with the source code. Also for educational purposes, documented invariants provide a valuable source of information to students having to master a particular application. After all, the invariants document some of the main elements of the software application's behaviour [EP06].

Checking behavioural assumptions Invariants, which can be checked automatically (which are coupled to the source code), can make sure they are not violated later when the software evolves. `Assert` statements for example can be used for checking program invariants. They can be inserted into the code at the appropriate place to check them at run-time.

Advanced code optimisation Invariants which consist of an ordering of events which take place during the execution of a program can also be used to capture certain behavioural event patterns. An abstract example of such a pattern might be of the form that no event of type X may occur between any two events of type Y. This way, not only program errors can be checked, but also execution patterns which are more optimised than others. In cases where code optimisation is a fundamental design principle of the software at hand, this also creates more reliable programs with respect to the expected optimisation features of the software.

2.4 An Overview of Invariants in Software

In this section we have a closer look at existing invariants in software and we identify what kind of invariant behaviour they can specify and how they specify it. Many different references to invariant behaviour are encountered. We distinguish them further according to the following categories:

- Data/state invariants: assertions, loop invariants, dynamically detected invariants,
- Class/object invariants, type invariants,
- Evolutionary Invariants under Law-Governed Architecture,
- Communication Protocols.

One of the most widely recognised invariants are *data invariants*, i.e. conditions on a program's data that are to be maintained throughout the execution of a program. These kind of invariants are generally recognised and referred to as 'invariants', however in this dissertation we refer to them as *state invariants* to make the distinction clear with other invariant behaviour viewed and reasoned about in software. The varying factor of most well-known approaches that target data invariants is the *scope* these invariants belong to. A *loop invariant* for example represents a local data invariant, which imposes invariant behaviour on the variables used within a program loop such as a `for` or a `while` loop.

Assertions are a *means to* check data invariants dynamically at run-time and depending on where they are placed in the code they restrict the data used in that scope.

Dynamically detected invariants (section 2.4.3) are exactly the same as state invariants, however since they are *detected* using a dynamic analysis approach they are referred to as being *dynamic*. *Class or object invariants* represent a particular kind of data invariants that are used in object-oriented programming languages to put constraints on the encapsulated data fields of objects. Since they are able to restrict the state of an object or hierarchies of objects, they are also referred to as *type invariants*. Evolutionary invariants under a Law-Governed Architecture represent *regularities of software* which have to hold also when software evolves over time. These kinds of invariants can be structural as well as behavioural, however in this context we consider only the behavioural regularities (see section 2.4.4). Communication protocols have to specify a set of rules to constrain the *order* in which communications can take place. These rules are an example of behavioural invariants which are based purely on event specifications.

Finally, other behavioural invariants in software are briefly discussed in section 2.4.6. Each of these invariants are explained in the following section. In section 2.5 we discuss (formal) specification languages which can be used to specify invariant behaviour.

2.4.1 Embedded Assertions

Several programming languages (C, C++, Java, etc...) provide constructs that allow developers to write assertions that are checked during the execution of a program. An assertion is a predicate (which evaluates to true or false) placed directly inside a program to indicate that the assertion must be obeyed at that place during every execution. A programmer typically annotates the source code with an assert statement that tests the predicate about the state of a computation and terminates the program if the predicate evaluates to false. Assertions are generally specified using the comment feature of their programming language, although some languages offer a specific `assert` construct. The following is an example of the assert macro in C:

```
#include <assert.h>

void f(char *p, int n)
{
   ...
assert(p != NULL);
assert(n>0 && n<5);
   ...
}
```

Both assertion statements on lines 5–6 express constraints on the values of the parameters of a certain function f. The variable `p` cannot be a `NULL` pointer and the integer n should have a value of 1 through 4. Such assertion checks are used to test *invariant behaviour* about the state of the program within a certain scope (in the example the scope of the invariant is the function f). Loop invariants can also be checked by putting assertions about loop variables inside an iteration statement.

Embedding assertions in the source code can help a programmer design, develop and reason about the local state of a program. The use of assertion statements also simplifies the later stages of software development such as testing, debugging and maintenance. If an expected assertion fails, the program is aborted and the user is notified of the assertion failure. In general, the assertion expression which failed is returned, together with the filename and line number where the failure occurred. Furthermore, by adding assertions to the source code, certain values that should remain invariant are made explicit and they serve as a documentation artefact as well [Ros95]. As adding assertions to source code can be a time-consuming activity, assertion tools exist that direct the insertion of assertions. One example of such a tool is C-Patrol [YB94]. Assertions are expressed as C expressions in source comments which are then translated by a pre-processor into executable code, while associated directives in the comments identify the assertion's location.

Problems with Embedded Assertions However, considering embedded assertions as specifications for documenting invariant behaviour raises some problems. First, these type of assertions are specified like a program statement in the implementation language and are thus hard-wired into the system. Changing the code means having to certainly adapt assertions at the same time, which provides an overhead for the user of the system. Second, only *local* assertions can be expressed as they are evaluated at particular program points in the execution of a program. One might want to express behavioural properties that should remain invariant during a computation, *independent* of a control point. But then assertions should be placed at all relevant program points, which is clearly not what is wanted.

Another type of embedded assertion languages broaden the scope of associating assertions with procedures or methods instead of program points by using pre- and post-conditions around these entities. These are discussed in the following section 2.4.2.

2.4.2 Class/Object Invariants

In object-oriented programs, the most frequently used invariants are called *object invariants*, which reason about the encapsulated data belonging to a particular object. For class-based object-oriented languages, an object invariant is usually defined on its class. Figure 2.2 demonstrates the use of an object invariant specified on a simple class `T`. The invariant used in the example is formulated to prevent a division-by-zero error within the public method `M` defined on lines 5–8.

```
1  class T {
2    int a,b;
3    invariant 0 ≤  a < b;
4    public T() {a:=0; b:=3;}
5    public method M(...) {
6      int k;
7      k := 100/(b-a);
8      a:= a+3; b := (k+4) * b; }
```

Figure 2.2: A simple class illustrating the use of an object invariant (adapted from [BDF$^+$04])

One of the best known object-oriented development methodologies which incorporate such invariants is Design By Contract(DBC). We first briefly discuss DBC here and then point out some problems encountered with object invariants.

Design By Contract(DBC) is a design and programming methodology developed by Bertrand Meyer for designing object-oriented software applications [Mey92a, Mey92b]. It has its roots in theorem proving as mentioned by Hoare [Hoa69]. It is based on the fact that software engineers should, next to designing and programming software components of a system, also define machine-verifiable specifications (or conditions) these components should adhere to during program execution. It is based on the theory of abstract data types together with the conceptual metaphor of a business contract. The main goal of the DBC approach is to help improve the *reliability* of software systems.

The central idea of the business metaphor reflects itself in how elements of a software system collaborate with each other on the basis of mutual *obligations* and *benefits*. For example, the *supplier*, i.e. the routine or method delivering a particular functionality, has the obligation to deliver something to the *client*, i.e. another subroutine or method, but requires that some constraints are met beforehand. Three kind of constraints are identified in DBC: *pre-conditions*, *post-conditions* and *class invariants*. Pre-conditions specify conditions that must hold right before a method execution and hence they are evaluated at that time. The system state is involved together with the arguments which are passed into the method. In the same spirit, post-conditions are statements that must be true after a method is executed. Both the new and old system state, the method's argument and return value, are involved. While on the one hand preconditions serve as obligations which a client must meet before employing a particular method, post-conditions specify guarantees that a software component makes to its users (if the pre-condition was met, the component can guarantee a positive outcome for its post-conditions).

```
1   put_child(new:NODE) is
2            -- Add new to the children of current node
3       require
4            new /= Void
5       do
6            ...Insertion Algorithm...
7       ensure
8            new.parent = Current;
9            child_count = old child_count + 1
10      end -- put_child
```

To get an idea how these conditions are specified, the above excerpt shows an example of a method belonging to a class representing the node of a tree structure annotated with pre- and post-conditions. It is written in Eiffel, the object-oriented language that DBC was first developed for. The precondition on line 4 states that a newly added node should not be a NULL object. On lines 7–9 the post-condition states that the newly added node's parent must be the current node itself. On line 9, the *old* keyword refers to a value of the program state before execution. Evaluation of such assertions requires saving a copy of *before* values. The assertion on line 9 states that the `child_count` variable should have increased by one after execution.

An *invariant* in DBC is defined as part of a class definition (therefore often referred to as a class invariant) and specifies a condition that must hold every time a method is invoked of an object that is an instance of that class. In practice they are checked every time before and after a method is executed on any of the class' instances.

```
invariant
        left /= Void implies (left.parent = Current);
        right /= Void implies (right.parent = Current)
```

The above example shows an invariant for the same `Node` class to demonstrate the pre- and post-condition specification. It states that if there is a left or a right child node, this always implies that its parent is the current node. Note that contracts are also inherited by classes from their superclass, which is referred to as *subcontracting*. Because of this feature, some also refer to a class invariant as being a *type invariant* as it might constrain a whole class hierarchy.

The basic idea of how DBC is implemented boils down to the notion of embedded assertion statements, i.e. a boolean expression about a particular state of a software system (at a particular point in time) which must evaluate to true during program execution.

Although DBC has initially been developed for the Eiffel language, extensions exist for other OO languages and for procedural programming languages as well. The analogy for a class invariant as scope for formulating constraints is that of a function. Of course, certain features like inheritance are not applicable in a procedural context.

Note that although class invariants formulate constraints relating to state within the scope of an object, that pre- and post-conditions specify inter-component constraints. In a way they also represent invariant behaviour of the communication between two objects.

Problems with Object Invariants As object invariants are checked dynamically at run-time (for example when exiting a public method), it does not always guarantee that an object's data is consistent with the specified object invariant. Although such dynamic checks catch many errors, at certain times the object invariant might be temporarily broken. This depends on *when* the object invariant condition is checked at run-time. One possible way to view an object invariant is simply as a shorthand for a post-condition on every public method. This implies that the invariant should be checked *after* the execution of each public method. It is excluded to also check the invariant *before* the execution of a public method, because the caller of such a public method does not need to be concerned with satisfying an invariant expression related to the internal state of the object. To illustrate this problem, consider again figure 2.2 where on line 3 an invariant is expressed over a simple class `T`. Consider a scenario with a slightly different version by replacing line 8 as follows:

```
    a:= a+3; P(...);b := (k+4) * b; }
```

Assume that method P calls M. Note that, at the time P is called, the invariant is not certain to hold. This problem of re-entrancy might create a division by zero as should have been prevented by the invariant. This problem is mainly caused because of ignoring to check the invariant also *before* the execution of method M (as a pre-condition). However, we do not want to impose the invariant as a pre-condition to possible callers, because this breaks object encapsulation as callers need to be responsible for establishing the consistency of the internal representation of an object belonging to class T.

Solutions are worked out for callers of such public methods to know if the invariant holds without revealing the details of the invariant, because it might depend on internal encapsulated state[LM04, BDF$^+$04]. This is not further discussed here as this is outside the scope of this dissertation.

2.4.3 Detecting Dynamic Invariants

Substantial research has been done about automatically discovering likely invariants in software. They are referred to as *dynamic invariants* because they are detected using a dynamic analysis approach, which means they are invariant with respect to the program executions they were detected by. They also belong to the category of *data invariants*.

DAIKON [ECGN99, EPG$^+$07] is such a tool which dynamically discovers such invariants in software. Examples of what is referenced to as invariants are constant values ($x = a$) or non-zero values ($x \neq 0$), linear relationships like $y = ax+b$, ordering ($x \leq y$) and also conditional invariants or implications such as $a \neq 0 \Rightarrow b \leq a$ are possible.

DIDUCE [HL02] is a similar detection tool that aids in detecting complex programming errors. The type of invariants which are detected are similar, however the tool support which is provided and what is done with the results distinguishes both approaches. Instead of presenting the user with the collection of program invariants found after execution as in DAIKON, DIDUCE continuously checks the behaviour of the program against the invariants hypothesised up until that point and reports the violations detected. Then this invariant is weakened and program execution is resumed. The application of DIDUCE lies mainly in debugging, program testing and software evolution.

Such kinds of invariants provide valuable documentation of a program's data structures and operations. These dynamically discovered invariants can be inserted into a program later as an `assert` statement as mentioned in section 2.4.1.

One recent attempt has been made to detect temporal behavioural properties, belonging to the group of *invariant properties*. These also represent invariant behaviour, however specified mainly as an ordering of events which occur during program execution [YE04]. The approach is based on starting from a pre-defined base of property patterns which occur frequently. This is a first attempt towards detection of behavioural invariants which allow reasoning about a temporal ordering of events. Invariant properties are discussed more into detail in section 2.5.2.

2.4.4 Law-Governed Architecture

Relevant related work is discussed under what is referred to as Law-Governed Architecture (LGA) [Min93, Min96b, Min96a, Min01]. The intent of LGA is to associate with a system a set of *regularities* (or *architectural invariants* [Min00]) which are 'the laws of a system which must hold at all times when a system evolves'. In that sense these regularities (or constraints) are conceptually close to invariants. They impose mainly structural constraints on a system related to its architecture (which is not further considered within this dissertation), but also some behavioural constraints can be enforced on the interactions (or events) between system objects. An example of combining structural and behavioural constraints under LGA is that of a layered architecture: the components of a system need to be grouped into layers combined with constraints on the communication between objects from different layers. Even principles such as *encapsulation* which are usually integrated in a programming language are thought of as regularities.

LGA is based on a language-independent object model of a software development project. We refer to a model of a software *project* instead of only the software as LGA can specify more than constraints on an object-oriented system itself. It also addresses constraints which are applicable on the *process of software development*, thereby representing the developer of a project as an object interacting with the system. LGA can specify constraints on relations between objects referred to as *interactions* or *events*. Possible interactions are for example the creation of a class by a programmer, or the usage relationship between two classes. Constraints can be put on these interactions depending on properties of the objects that play a role in the interaction.

Darwin/E represents a concrete implementation of LGA, where regularities are specified dealing with interactions between components in the Eiffel programming language. The formalism which is used for expressing constraints is the language Prolog and these dynamic event (or interaction) constraints are enforced at run-time. Consider an example rule for constraining the exchange of regular message sends in a layered architecture (taken from [Min96b]):

```
sent(S,M,T) :-
      level(Ls)@S,level(Lt)@T,
      (Ls=Lt | Ls=Lt+1),
      do(deliver(M)@T).
```

The rule declares that a message send event, object S sending the message M to object T, can only occur if object S belongs to the same layer or one layer higher as that of object T (assuming that each object in the layered architecture is given a level number to denote the layer it belongs to).

LGA only allows constraints about events occurring between objects belonging to the object model, like object creation and deletion and invoking methods on objects. Events occurring within a method body, such as an assignment statement or a variable declaration, cannot be considered.

2.4.5 Communication Protocols

Communication protocols in software also have to adhere to invariant behaviour related to safety properties as mentioned further on in section 2.5.2. A protocol is a kind of agreement about the exchange of information in a distributed system. Holzmann defines the five elements of a communication protocol [Hol91]: the *service* to be provided by the protocol, the assumptions about the environment in which the protocol is executed, the vocabulary of messages used to implement the protocol, the encoding or format of each message and the procedure rules guarding the consistency of message exchanges. The procedure rules are the most difficult to design and also to verify. These set of rules (also called the protocol) governs how information is delivered. A complete and consistent set of rules must be designed to arrange the interactions in a distributed system. These procedure rules consist of specified invariant behaviour based on events (in this context usually called *processes*) rather than state that must be guaranteed to hold at all times.

For reasoning about protocol procedure rules, a partial specification is advocated to abstract from other issues of protocols (such as message format for example). Such specifications are defined at an abstract level in terms of processes, channels and variables which form the needed building blocks of distributed systems to reason about communication problems (for example the formal specification language LOTOS provides these abstractions [LG00]). Communication protocols are often checked using a formal approach such as model checking. In the case of protocols, a formal approach is more suited to prove the absence of certain invariant properties (for example deadlocks or improper communication termination).

2.4.6 Other Behavioural Invariants in Software

Other research exists that tackles the problem of expressing behavioural properties in software that could also be used to specify invariant properties. A significant part is associated with dynamic analysis tools (further discussed in section 3.6) which formulate behavioural assertions over program behaviour represented by an execution trace. As tools and techniques to reason about invariant behaviour are discussed in chapter 3, we only summarise relevant concepts which deal with expressing invariant behaviour.

Assertion Checking Auguston et.al. [Aug98] tackle the problem of *generic assertions* , i.e. an assertion which must always hold. Assertions are formulated in the assertion language Forman, a functional language. These assertions are checked *over* an execution trace of a program obtained through dynamic analysis. An execution trace contains occurrences of observable *events* that happen during program execution, augmented with associated run-time values. So in the first place they are event-based, and they are mostly formulated using implementation constructs such as function calls and variable accesses. Because of this lack of abstraction, high-level invariant properties are difficult to express over an execution trace. Depending on the application domain, ranging from program comprehension to finding errors in code, debugging or testing, assertions specify wanted behaviour *over* events occurring in a certain order at run-time. A different approach is using an SQL-like language to check invariant assertions (at run-time) where execution events are represented as tuples in a database [GOA05] (see also section 3.6.2).

2.5 Specification Languages for Invariants

Having surveyed how invariants are dealt with in current software development, we illustrate here two representative examples of formal behavioural specification languages. The Z specification language is a formal specification language which specifies system behaviour by expressing admissible system states for object systems. Z is an example of a formal specification formalism which is state-based. The properties of interest are specified by invariants constraining the objects in a system and by pre- and post-conditions constraining the system operations. Another formal specification mechanism which can be used for specifying invariant properties is temporal logic. Specifications formulated in temporal logic are history-based specifications, i.e. they specify behavioural properties representing an ordering in time. Properties are usually specified as logic assertions about system objects.

The Object Constraint Language (OCL) is also briefly mentioned in section 2.5.3 as an example of a non-formal specification language which expresses regular software invariants at the design level for systems based on object models.

2.5.1 The Z Notation

Z is a formal state-based specification language [Spi89]. Z uses the notation of predicate logic to describe in an abstract way the effect of an operation on the system to enable reasoning about an entire system's behaviour. Z provides some conventions to specify sequential, imperative programs by using *schemas* to represent the state space (a set of states) and operations for abstract data types (ADT) . *State invariants* for abstract data types can be expressed in Z inside these schemas, as can be seen in the following example of a Z schema representing a counter ADT [Spi89]:

$Counter$

$value, \ limit \ : \ mathcal(N)$

$---------$

$value \ \leq \ limit$

This schema represents the state space for a simple counter ADT with a $value$ and a $limit$ representing its state. Both states have a natural number as value. All states of a $Counter$ ADT must obey the invariant relationship $0 \leq value \leq limit$ documented by the declaration and by the predicate part in the schema. The set of initial states is documented by a separate schema and the ADT may only start in one of the initial states. The following schema represents one initial state for the same counter example:

```
─────────────────── InitCounter ───
Counter
---------
value  =  0
limit  =  100
```

The next schema demonstrates how operations on an abstract data type are specified in Z. For an operation, the state of the ADT *before* the operation is modelled by the undashed schema components, while the dashed components model the state *after* the performed operation. In the example schema below, an increment operation is shown which increments the value of the counter by one:

```
─────────────────── Inc ───
Counter
Counter'
---------
value'  =  value  +  1
limit'  =  limit
```

Note that the properties of both $Counter$ and $Counter$' are implicitly part of the property of this schema: the invariant relationship defined for the counter ADT should hold before and after this operation. In the same manner, operation schemas can be specified for defining pre- and post-conditions. The invariants the formal specification language Z supports are targeted towards constraining the state of entities and the relationship between two entities.

The Z notation is a specification formalism which can be used to specify an entire software system based on object models. The admissible system states and operations are specified through integrating invariant expressions (as shown above) in the specification of the system. Invariant specifications cannot be decoupled from the system as a whole; they are embedded in the system specification and through refinement an entire system can be built from that specification.

2.5.2 Temporal Logic

Temporal logic is a form of logic specifically created for reasoning (formally) about statements which involve the notion of order in time. Time is modelled as a sequence of states and depending on the type of temporal logic, one can reason (infinitely) about the future and/or the past. Temporal operators allow one to speak about the sequencing of the states along an execution rather than only about the states individually. Consider the following example of a property for an elevator system which can be expressed with temporal logic: 'the elevator never traverses a floor for which a request is pending without satisfying this request' [BBF+99].

Before we can formulate how an invariant is specified using temporal logic, the three main constituents of temporal logic are first explained here:

- Atomic propositions are used to make statements about the states. These propositions represent elementary statements which have a truth value in a given state. $x + 1 = y$ or 'nice_weather' are examples of propositions,
- Classical logic connectors such as $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation) and $\Rightarrow$ (logical implication) which allow for the construction of complex statements relating simpler sub-formulas.
- Temporal operators allow reasoning about the sequencing of events rather than about the states considered individually. Possible temporal operators are: $\bullet$ (next), $\Box$ (always), $\Diamond$ (sometimes). If P represents a property of the current state, then $\bullet$P says that the next state satisfies P. $\Diamond$P satisfies P at some state without saying which state exactly and $\Box$P denotes that the property P will always hold. Temporal operators can also be nested arbitrarily.

Having these three constituents for temporal logic available, one can state properties of one computation path. This type of temporal logic is called *linear time logics* and is therefore most suitable to formulate requirements about sequential systems. *Branching time logic*, however, can reason about multiple time lines which means that they can also be used to reason about concurrent processes. In that case, we need to reason about a tree structure representing behaviour instead of a single execution path (which means that alternative futures are possible). For that reason, we need two special purpose quantifiers ,A and E, which allow to quantify over the *set* of executions. In this case, a formula $A\phi$ states that all executions starting from the current state satisfy property ϕ, whereas $E\phi$ states that *there exists* an execution satisfying ϕ.

The combination of temporal operators with the quantifiers can yield the following combination: $A\Box P$. This formula states that for all possible execution paths, P must always be true. This combination states that the property P is an *invariant*. In this context, 'invariants are properties that are true continuously'. Invariants in temporal logic can also be used to express *safety properties*. This type of property expresses that, under certain conditions, an event never occurs. A very common example of a safety property is that 'both processes will never be in their critical section simultaneously'. In Linear Temporal Logic (LTL) this is formulated as follows:

$\Box\neg(crit\_sec_1 \wedge crit\_sec_2)$

Temporal logic is frequently being applied in formal verification, and in particular in model checking. It is used to specify and verify requirements of software models, which are specified in essentially a formal finite-state modelling language. The advantage of using temporal logic for specifying invariant behaviour is the level of abstraction in which the behaviour can be formulated: a property directly reveals *what* should remain invariant. Behaviour is expressed *declaratively* at the *domain level* which makes abstraction of the actual implementation constructs. E.g. in the critical section example, the safety property is not expressed using implementation language constructs. Instead the domain concept of a critical section $crit\_sec$ is used. On top of that, the temporal operators provide a *context abstraction* to abstract over time.

2.5.3 Other Behavioural Specification Languages

In this section we have shown up until now two well-known specification formalisms in which system behaviour can be specified. They are representative within this context in that Z specifies behaviour of object models, while temporal logic specifies behaviour interpreted *over* time structures (or transitions in time). A complete different example of a non-formal specification language is presented which expresses invariant behaviour for object models at the design level.

The Object Constraint Language (OCL) is a declarative language used to describe constraint expressions on UML models [Gro06]. UML is a non-formal graphical modelling language for describing object-oriented analyses and designs of software systems. OCL supplements UML at the design level by providing a textual language to formulate expressions which typically specify invariant conditions that must hold for the system being modelled, or queries over objects described in a model. UML modellers can use OCL to specify application-specific constraints in their models as well as queries which are completely programming language independent. OCL allows a special construct for specifying data invariants, as can be deduced from the example below. The scope of the invariant is explicitly denoted here by the 'context' keyword. In the example the context represents a class `Company` for which a limit is imposed on the number of employees. OCL has been developed as a business modelling language

```
context c:Company inv:
self.numberOfEmployees > 50
```

to counter traditional formal specification languages based on mathematics notation. It claims to be formal but still easy to read and write. OCL is meant to be used at the *design level* and is not a programming language. Only *object invariants* can be specified over the UML models.

Other specification paradigms According to Van Lamsweerde [vL00], five different types of formal specification paradigms exist, of which two well-chosen representatives were presented in this section. A third group is characterised as transition-based specifications, which represent languages in which to formally describe state transition diagram models. They are based on capturing the required transitions from state to state. Properties of interest are then described as a set of transition functions in the state machine transition. A specification language based on an FSM notation is both transition-based and state-based. Such specification languages are sometimes considered too operational and so they run the risk of being too close to implementation constructs [JR00] (this is mentioned again in section 3.3). Operational specifications, the fourth paradigm for formal specifications, is not discussed here, however the language LOTOS used for defining the communication rules for protocols (see section 2.4.5) supports this paradigm and also CSP [Hoa78]. A last specification paradigm are functional specifications, but as the main goal here is not to discuss all possible specification languages, we limit ourselves to the here presented formal specification language paradigms.

The next section outlines a definition of *design invariants*, i.e. the kind of invariants which are targeted in the context of this dissertation.

2.6 A Particular Type of Invariants: Design Invariants

In general, current programming languages usually adopt a *module-centered view* of software. They deal in the first place with the internal structure of modules and with the interface of a module and the other parts of a system. But they generally provide few means for making explicit statements about the system implementation as a whole.

This view is reflected in the acknowledged use of *data invariants* as the primary behavioural reasoning approach. Such invariants represent constraints about the program data that should always hold at specific points during program execution (depending on the scope of the invariant). Although these invariants, when used, undoubtedly improve the reliability of the software, these behavioural invariants alone cannot sufficiently document invariant program behaviour. In the field of testing, this view is currently reflected as well: popular testing approaches such as the XUnit framework [Ham04] focus on testing the functional behaviour of small system modules.

In this dissertation we take into consideration a particular kind of *invariants* like the Pico garbage collection invariant as demonstrated in the introductory chapter. Such invariants are not externally verifiable by only considering program data and not localizable according to the decomposition mechanism of the programming language at hand.

We advocate the term *design invariant* to be used within the remainder of this work and we formulate the following definition:

Definition
*A **design invariant** is a behavioural regularity of the design of a program that is not aligned with the structure of the program.*

We added the connotation ‘behavioural’ to make a clear distinction between structural and behavioural regularities. This is mainly to avoid confusion with similar terms used in a more structural context such as Minsky’s regularities under LGA (see section 2.4.4) or other approaches [MK06].

Design invariants exhibit the following main characteristics:

- **Design invariants represent behavioural properties related to the design of a system**
  Design invariants intend to capture system-specific behavioural concerns related to a software's implementation design. This implies the need for a specification formalism in which design invariants are expressed should be descriptive to capture behaviour at the conceptual level rather than low-level implementation constructs.

- **Design invariants cross-cut an entire system**
  Design invariants can be non-local to a particular module of the programming language at hand. The definition of a design invariant reveals that such invariants are not aligned with the structure of the source code. Instead they consist of an ordering of events (or executed statements) that are *triggered* at scattered program points of interest. Combined with associated run-time information about a program's particular state when execution reaches those program points, they form the constituents of a design invariant. To support practical use during program development, the used formalism for specifying a design invariant should be descriptive and partial so that only the behaviour relevant to the design invariant has to be specified.

- **Design invariants are non-externally verifiable**
  Design invariants are non-externally verifiable in the sense that they cannot be verified solely by observing data values. They represent advanced knowledge about the underlying design subtleties of a complex system that resides at the code level. They are mainly based on an ordering of events which occur at a certain time during program execution. Specifying design invariants requires to have a close look at the internal execution structure and necessitates expert system knowledge rather than inspecting the behaviour of variables *on the outside*. An analogy can be drawn with specifying tests in that a test can also either be based on operations or on function. A unit test is purely functional, because the program it *reasons about* is treated as a black-box. A unit test specifies input data and expected output data. Comparing the actual output data with the *expected* output shows the success/failure of a test. Operational (execution structure) tests on the other hand inspect the internals of the software and its operational details (this is referred to as white-box testing). Such testing which relies on operations rather than pure functionality [Bei90] allows the specification of more sophisticated behavioural patterns.

In the next chapter, the underlying mechanisms to check invariants at run-time are addressed. The domain of program analysis is studied, with an emphasis on tools and techniques which are capable of checking the concept of a *design invariant* which were defined in this chapter. There we distil a set of requirements needed for the analysis of design invariants in software.

2.7 Conclusion

In this chapter, invariants in software were identified, their uses were highlighted and they were classified according to what aspects of behaviour they reason about.

This classification was primarily based on their ability to support the two fundamental ways of reasoning about software behaviour. Approaches can be either state-oriented or more event-oriented, thereby not excluding a combination of both. It can be concluded that both approaches complement each other in different ways:

- **State-oriented approaches** support reasoning about invariant behaviour based on the *data* of a program belonging to a particular scope. These types of invariants contribute significantly to the software development process by making software more reliable, by preventing system errors and by documenting mainly the state restrictions of software components. These type of invariants form the largest group, which can be attributed to the module-centered view current programming languages adhere to. They primarily deal with *data* which can be captured *locally* within the boundaries the decomposition mechanism of the programming language enforces.

- **Event-oriented approaches** enable reasoning according to *events which occur at a certain time* during program execution. Information about the state of a program at a certain point in time can also be considered, although the main building blocks of reasoning consist of behavioural events. Although such reasoning can deliver a significant contribution for checking invariant patterns of behaviour, these type of invariants have not yet been widely established. However, event-based invariants could perfectly complement data invariants as they target the execution structure of a system which is non-externally verifiable. Moreover, events can be entities depending on program points of interest which cross-cut an entire software system, which makes them viable to reason about system-wide behavioural program invariants as well.

In section 2.6 we pinpointed the notion of a *design invariant*, i.e. the invariants which are considered throughout the remainder of this dissertation. Design invariants are invariant **properties of the design** of a system. Following the classification of *event-based vs. state-based* behavioural reasoning approaches, *design invariants* form the ideal counterpart of data invariants. As they are the conceptualisation of primarily event-based behaviour, they complement data invariants in that they can express **system-wide behaviour** based on **temporal events**. In addition, they capture **non-externally verifiable behaviour** about the design of a software system.

2.8 Summary

In this chapter, we started with an introductory section on invariants in software. In section 2.1 we elaborated on what invariants really are by having compared some of the frequently used definitions of invariants. We subsequently discussed in more detail the two fundamental approaches for reasoning about the behaviour of a system. Then the different uses of invariants were emphasised in section 2.3. Continuing in section 2.4, we first classified invariants into categories and then an overview was provided of different types of invariants as referred to in current day's software development. In section 2.5 we had a closer look at existing (formal) specification languages for specifying invariant behaviour. Consequently we defined a type of invariant called *design invariant* in section 2.6. We articulated a definition for a design invariant and we pinpointed its main characteristics and the requirements which it must fulfil.

In the next chapter, existing program analysis techniques are surveyed which can check or verify design invariants in source code.

Chapter 3

Program Analysis for Supporting Design Invariants

In the previous chapter, we provided an overview and a classification of invariants in software based on the way their behaviour is specified and on their scope. We have shown that invariants are specified and reasoned about in different phases of the software engineering life cycle. We emphasised that invariants serve multiple purposes ranging from high-level system documentation up to advanced code optimisation opportunities. To round it up, we ended by defining a particular kind of invariants, named 'design invariants', which represent behavioural regularities of the design of a system that *cross-cut* the structure of a program. On top of that, design invariants are *non-externally verifiable* and they represent *design properties* of a system at the code level (as defined in section 2.6).

In this chapter, we survey existing program analysis techniques and we highlight their features that allow checking design invariants for a particular software application. We start in section 3.1 by pinpointing the requirements for an analysis tool so as to be able to verify design invariant specifications against the program under investigation. Section 3.2 elaborates on the different types of analyses and defines important terminology needed throughout the remainder of this dissertation. Next, we distinguish two main analysis approaches. Section 3.3 elaborates on analysis methods which are applied to *models* of a program rather than directly to the source code. These methods are usually referred to as formal verification approaches. The second main group of analysis methods is applied directly on code and is described in section 3.4. As code analysis can either be static or dynamic, the main characteristics of both code analyses are compared. In section 3.5, we present the main building blocks of static analysis which are needed for checking design invariants. The same exercise is done for dynamic analysis in section 3.6 and for both analyses an evaluation is performed within the context of this dissertation. We end with a conclusion in section 3.7.

3.1 Tool Support for Design Invariants

In the previous chapter invariants in software were discussed. Two main categories of invariants were established based on the two fundamental approaches for specifying software behaviour. *Data-based* invariants, which impose a constraint on a program's data, formed the largest group and they were classified according to the scope of the invariant. *Event-based* invariants, which are based on temporal properties, formed the second group, and specify invariant behaviour in terms of events which occur at a certain time during program execution.

In section 2.6 we defined the concept of a *design invariant*, with the following characteristics:

- They represent behavioural regularities related to the *design* of a system,
- They *cross-cut* an entire system, i.e. they are not aligned with the structure of a program,
- They are *non-externally verifiable*.

These characteristics pose some important restrictions on tool support for supporting design invariants.

First of all, as design invariants represent behavioural properties of the *design* of a system, a behavioural specification language is needed which must be able to represent system-specific design invariants at a high level of abstraction. This is crucial as formulating design properties is done at the domain level instead of implementation level. Furthermore, a decoupling of the design invariant description from the actual code is needed to make sure that the invariant specification does not have to be adapted every time a change to the source code is being made. This results in a specification of a design invariant which consists of high-level domain concepts instead of low-level implementation constructs. Such a specification can serve as documentation to make the invariant *explicit* to all peers involved in a software project.

Second, design invariants represent behavioural properties which are not always localised in a particular component according to the decomposition mechanism of the programming language at hand. They represent complex behaviour which is dependent on program statements within different components scattered throughout a system. Checking such system-wide entities in an automated way poses difficulties regarding computational complexity. Every time a particular design invariant needs to be checked, the whole system is involved. Therefore to promote practical use, tool support should consider ways to *narrow down* the process of checking design invariants in the sense that not all behaviour needs to be studied for checking an invariant. This is especially necessary in an incremental software development setting where software evolves as we develop it.

Third, design invariants are primarily *operational*. This implies that they are not mainly targeted towards specifying *data* that remains to be invariant, but rather the combination of executed system operations or *events* that needs to remain invariant. Additionally, this also means that design invariants are non-externally verifiable as was explained in section 2.6 (in contrast to unit testing for example [Ham04]). They represent expert behavioural knowledge about the behavioural *structure* (or control flow) of a particular system. Hence tool support for making them machine-verifiable imposes the *need for a specification language* in which a developer can specify system-specific expert knowledge about the design invariant. That specification needs to be automatically verifiable against the actual system.

3.2 What is Program Analysis?

Before having a closer look at existing program analysis approaches and how they can support documenting and checking design invariants, some terminology needs to be discussed first. As the field of program analysis consists of many different categories of analysis support, we introduce some definitions to make a clear distinction between different groups of tool support.

First of all, what is meant by *program analysis*? As indicated in the previous chapter, we focus solely on reasoning about the *behaviour* of a software system. Hence, even though other interpretations are possible, we advocate the following definition of *program analysis*:

Program Analysis
Program analysis is the extraction of behavioural information from software, where the software is represented as an abstract model or the source code itself [JR00].

Other forms of analysis are also important, such as structural analysis which reason about the syntactical structure of source code[WM06, MK06]. Analysis of human factors, for example the usability of software, is also a form of analysis. However, these lie outside the scope of what is discussed in this dissertation.

Within this defined domain of program analysis, different distinctions between existing techniques can be made. What is usually referred to as *formal verification* groups those techniques that perform analysis on a *model of a system* rather than the system itself. Theorem proving and model checking can be classified under this category (which is discussed in section 3.3). Another group of program analysis is called *static analysis*. Abstract interpretation and static analysis techniques belong to this category. A third group of more informal analysis methods is called *dynamic analysis* and groups techniques such as program monitoring, profiling and software testing.

Formal verification techniques distinguish themselves from static analysis in the sense that they analyse a *model* of the software instead of the software itself. In essence they reason about the *possible* executions of software. This brings us to the following definitions of static and dynamic analysis as they are interpreted in this context.

Static Program Analysis
Static program analysis is the investigation of behavioural properties of a program that is performed without actually executing that program.

To say it differently, static analysis extracts information form the source code about the *possible* executions of software systems. It analyses the program with as goal to obtain information that is valid for *all possible executions*.

Dynamic Program Analysis
Dynamic program analysis is the investigation of the properties of a running program over one or more program executions [Bal99].

In contrast to static program analysis, the results obtained from a dynamic analysis is typically valid only for one (or some) program executions.

In the following section we provide an overview of these different analyses within the context of this dissertation. Their suitability to document and check design invariants is discussed together with the extent to which they can be employed to aid a developer in constructing his software in an incremental development environment.

3.3 Analysis of Models

In the field of program analysis, a particular group of techniques analyses behaviour related to requirements or design on an *abstract model* of the source program. A well-constructed model of the software rather than the software itself can include extra information such as domain knowledge, design information or even environmental assumptions. On top of that, it is generally agreed upon that a carefully constructed model is amenable to more advanced reasoning as a high level of abstraction is used.

In general, formal verification methods fall under this category of analysis techniques. Such methods are mathematically based techniques for the specification, development and verification of reliable software (and hardware) systems. The emphasis of such methods lie on the description of software and the analysis of the possible behaviours a software system can have.

One way these approaches fit into the software development process is to first specify a model of the software in a formal specification language after which the actual source code is developed informally. This has as drawback that it cannot be guaranteed that the source code works correctly according to the specified properties as the specification of the model has been proven correct and not its implementation. Another approach is to use the formal specification to produce a program in a formal way. Certain properties of a system can then be formally proven and a refinement technique may be used to transform the specification into an implementation. A third option is to use a theorem prover to undertake formal machine-checked proofs of certain properties against the formally specified system model.

Theorem Proving

Theorem proving is a technique where both the systems and its desired properties are expressed as formulas in some mathematical logic [CWA^{+}96]. This logic is supported by a *formal system*, which defines a set of axioms and inference rules. Theorem proving is the deductive process of finding a proof for the properties starting from the axioms of the system and by using the inference rules. They can be classified ranging from highly automated provers for general-purpose programs to interactive systems requiring user input for dealing with special-purpose properties. Using interactive theorem provers are very slow mainly because they require a great deal of user input and are only practical to use if the cost of mistakes is extremely high (e.g. for safety-critical applications). As theorem proving is very time-consuming and hard to realise in practice, this may be most appropriate in high-integrity systems involving safety or security and hence they are not applicable within the context of this dissertation.

Next we discuss another formal verification technique called *model checking*, as it might be used for specifying and verifying design invariants. It has proven to be one of the more successful formal verification approaches for verifying requirements and design of a variety of systems. In model checking, properties of a system are specified using temporal logics, which was mentioned already in section 2.5.2 as behavioural specification medium in which invariant properties can be expressed.

Model Checking

The term *model checking* in the context of software refers to checking whether all possible behaviours of a software program are *models* of a property of that software program [CGP02]. We refer to a *model* of the software as it does not directly represent the code of the software itself but a model of its design or requirements. Model checking formally verifies the behaviour of a model of software under study in a static way. Verification is done by exhaustively exploring the possible state spaces of an abstract model against the specification of a property to see if the property holds for that model.

The essential idea behind model checking is depicted in figure 3.1. A model-checking tool takes as input a model of a software program, representing its design or its system requirements, and a specification of a certain property that the final system is expected to satisfy. The model checker then returns YES if the software model satisfies the property specification or returns a counterexample otherwise [Pal04, CWA^{+}96]. Note that providing a counterexample (i.e. an execution path for which the model does not hold according to the property at hand), also called a *proof-by-refutation* is a powerful part of model checking. This way errors can be easily pinpointed in the model as this provides constructive feedback for the developer. The idea of model checking is to make sure that the model of the software satisfies as many properties as possible in order to increase the confidence in the correctness of the model.

Before any checking of properties can begin, the system under study first needs to be modelled. *How* the software is modelled determines also which kind of properties

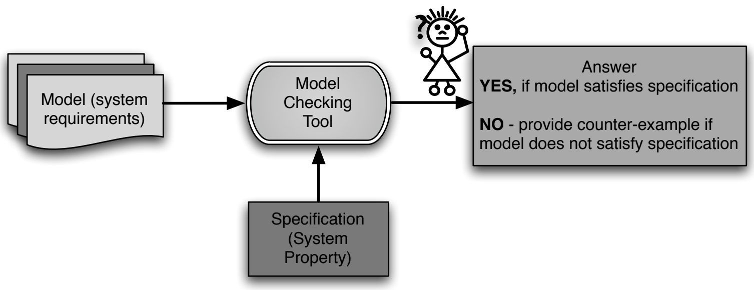

Model (system requirements)
Model Checking Tool
Answer
YES, if model satisfies specification
NO - provide counter-example if model does not satisfy specification
Specification (System Property)

Figure 3.1: Model checking approach

can be checked later. Most model checkers have their own formal language for specifying a model of the software which a tool can understand. However, in general they are all based on some extended version of Finite State Machines (EFSM's). Briefly put, a state machine or automaton is a machine evolving from one *state* to another under the actions of *transitions*. A state machine is often depicted by drawing each state as a circle and each transition as an arrow. Figure 3.2 depicts an example of a finite state machine representing a modulo 3 counter. Only two transitions (increment, decrement) and 3 possible system states (0,1,2) are possible.

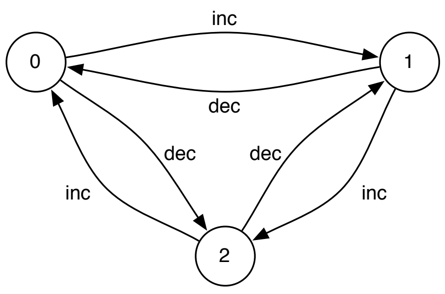


Figure 3.2: A finite state machine representing a modulo 3 counter (example from [BBF+99])

Representing a software system's source code directly as a finite state machine would mean that the global program state (of all variables) is represented as a state, while the transition between states would be a program statement. A specification language based on an FSM notation is both event-based (or transition-based) and state-based, as discussed in section 2.2. Such specification languages are sometimes considered too operational and so they run the risk of being too close to implementation constructs [JR00].

After having specified a model of the software, the properties of the system representing requirements or design properties are usually specified in a temporal logic, as mentioned in section 2.5.2. A sophisticated algorithm (different algorithms exist for different logics) then *checks* the formula for the property against the model of the software.

Model checking in general suffers from the following problems. First, specifying properties in temporal logic can be very difficult. And second, the number of states may possibly be very large. This is commonly referred to as the *state space explosion problem* [Pal04]. Abstraction techniques for simplifying automata exist for overcoming this problem (i.e. abstraction by state merging or by variables), but as correctness of the model with respect to the system properties can then no longer be guaranteed, model checking loses a part of its power as a formal verification technique.

Evaluation The appealing aspect of model checking in this context is the expressiveness of the specification language and the range of properties which can be expressed and verified. Specifying properties *over* an FSM specification allows one to formulate invariant behaviour based on states as well on events (or transitions). This allows specifying also operational, system-wide properties which can be expressed at the design level. And although Linear Temporal Logics (LTL) formulae are far from easy to express as they are referred to as being a main difficulty of model checking [Hol02a], patterns of property specifications might aid in this difficulty [DAC99].

Note that the high abstraction level of the property specification can be attributed to the model of the software *over* which the properties are checked. As the model describes the design or requirements of a system rather than the implementation constructs itself, the properties of this model are also at a high level of abstraction. The big cost thus lies in establishing the behavioural model of the program, which is an enormous investment upfront. This is also very impractical as the model is not directly coupled to the software itself. In an agile setting where software evolves as we develop it, this would require that for every change that is made to the software, a manual update of the model is required and then also the verification process of the properties needs to be done again over the entire model. Current research is aiming more and more to apply model checking to analyse *software* directly instead of a model [CGP02]. However, expressing properties *over* the source code directly instead of a high-level model might lower the abstraction level again.

In general, as model checking is a formal approach, one can prove the correctness of the model with respect to certain properties. This means that it can be proved that a certain invariant property is guaranteed not to occur in the model on which it is checked. Dealing with the computational complexity of this approach in an incremental software development environment is still a topic to be investigated [MS03].

3.4 Analysis of Code

As already mentioned, approaches performing analysis on a model of software have one important disadvantage: the lack of coupling between the model and the actual implementation. Especially in the later stages of development, after software has evolved, a model becomes less useful if it has not been kept up to date. At a certain time, a model can even be discarded at the later stages of maintenance.

In the context of this dissertation, next to making design invariants explicit by specifying them at a high level of abstraction, a next step is to be able to check them *directly* against the source code. As a causal link between the behavioural property specification and the software is needed, analysis of code is considered rather than analysis of a model of software.

3.4.1 Static vs. Dynamic analysis

Considering program analysis, one might consider using either static or dynamic analysis as defined in section 3.2. Applying a dynamic analysis approach usually consists of instrumenting the program to collect information as it runs. This means that the obtained system information is typically valid only for the particular execution (or run). Static analysis approaches on the other hand extract information that holds for all possible program executions. Another way to formulate the difference between static and dynamic analysis is the different meaning of a *run* of a system: static analysis runs a program over an abstract domain (over a description of possible values), while dynamic analysis, also even though it uses a certain abstraction, runs a program with concrete run-time values [Bal].

To demonstrate the difference between static and dynamic analysis, consider an example where the values of some global variables need to be known. A dynamic analysis would instrument the program to simply print out these values at particular points during the program (for example every time they are assigned a value). A static analysis on the other hand would find all program statements that *might* affect the values of the global variables and then analyse those statements to extract information about the values assigned to them.

A major advantage of dynamic analysis is the precision of the analysis information which is obtained, although be it only for the considered execution(s). Practically all static analysis approaches provide approximate information about software behaviour, which can also be a benefit. Sometimes execution paths which can never actually occur when running a program are considered . This degree of imprecision means that such an analysis may not provide information which is accurate enough to be used. This is more problematic in tools in which a user sets up to detect specific errors than in software inspection tools which extract interesting properties about software [JR00].

Dynamic analysis tools are in general also easier to implement as information about one single execution trace is much simpler to obtain than information over all *possible* executions. So dynamic tools are available for a wider range of problems.

Static and dynamic analysis tools complement each other: static analysis tools rather look for simpler classes of errors over the entire source code of a program [JR00]. Dynamic analysis tools can check for deeper semantic errors (but only in those program statements that execute for a given execution scenario).

In the next section static analysis is discussed and more in particular the criteria relevant for the analysis of design invariants. As using static analysis for checking design invariants would impose the use of a *flow-sensitive*, *context-sensitive*, *interprocedural* analysis, these terms are explained in detail.

3.5 Static Analysis

Property checking tools using a static analysis approach have recently gained importance [Raj06]. Contrary to model checkers or theorem provers, these tools do not ensure that the software correctly implements intended functionality. Instead they look for specific kinds of errors by analysing the data and control flow of a program. They analyse all the statements of a program, without regard to how frequently these statements execute in practice.

A result from theoretical computer science is that, for arbitrary programs, some program properties cannot be calculated precisely through static analysis. Consequently, all static analysis tools make approximations. This can lead to false positives that do not correspond to runtime behaviour but that cannot be safely eliminated [ARTZ03].

According to Rajamani [Raj06], tools based on static analysers have innovated primarily in two main areas. The first one is the specification language these tools use. Early tools like Lint (now known as Splint [Spl]) and also more recent tools [AT01, ARTZ03] check for common errors which can be stated directly at the level of the programming language (for example referencing an uninitialised variable). They are generally called *software inspection* tools. These tools are automated code inspectors, exhaustively examining a program for defects. In general, static analysis tools uncover low-level coding defects that, nonetheless, can cause severe errors in program behaviour (such as a program crash). At the most basic level, there are tools that find language-level defects such as integer overflow, division by zero, dereference of a null pointer, and buffer overflow. Another type of static analysis tools allow a user of a tool to specify the unwanted behaviour that is of interest to them. METAL is an example of a specification language used in combination with a static analyser [HCXE02] integrated in a compiler (gcc). The language is based on FSM descriptions while the checking is done with a depth-first search algorithm on the control-flow graph (see section 3.5.2) of a program. This type of analysis is referred to as *metacompilation*. A disadvantage here is that the analysis is more platform-dependent.

The second area of innovation is the exploration of engineering trade-offs. Precision, speed, completeness and soundness represent the most important ones. A *Sound* static analysis aims at producing information that is guaranteed to be valid for all possible program executions. However, to obtain this guarantee, static analyses need to perform complex analyses, which can be time-consuming. Consider again the same example of wanting to know the values of some global variables. An *unsound* analysis would scan through the assignment statements that might affect the global variables, while a sound analysis would also consider indirect assignments that take place via pointers to these global variables. Unsound analyses are often sufficient as, although the analysis information is not always correct, it might provide a starting point for further analysis. Another point in favour is the ease of implementation and efficiency of unsound analyses. In the above example, having to analyse indirect pointer references to global variables (this is called pointer analysis) is a very complex process. Other trade-offs to be made concern the *precision* of the analysis against the *analysis speed*. The more precise the results of a static analysis have to be, the more complicated the analysis gets which results in a time-consuming process. Two influential factors for tools to be more precise (or more complex) are context- and flow-(in)sensitivity, which are discussed next.

3.5.1 Comparison Criteria

Flow-sensitive vs. flow-insensitive analysis *Flow-sensitive analyses* take the execution order of the program statements into account. They normally use some form of iterative data flow analysis (as is shown in the next paragraphs) to produce a potentially different analysis result for each program point. *Flow-insensitive analyses* do not take the execution order of the program statements into account, and are therefore incapable of extracting any property that depends on this order. For example, flow-insensitive pointer analysis algorithms can scale for programs consisting of hundreds of thousands of lines of code. But because the analysis is flow-insensitive, they cannot, for example, determine if a pointer is initialised before it is used or determine that a pointer has different values in different program points. Both of these properties depend on the order in which the statements of the program execute [JR00]. An example of a frequently used static analysis which is flow-insensitive is *type analysis*.

Context-sensitive vs. context-insensitive analysis A static analysis can either be context-sensitive or context-insensitive, depending on how language constructs such as procedures are analysed. A context-sensitive analysis is an interprocedural analysis that takes the calling context into account for example when analysing a function call. With *context* is meant that the values of the parameters are taken into account where they might influence statements in the function body. With such an analysis, a different result is produced for each different analysis context. A context-insensitive analysis on the other hand produces a single result that is used directly in all contexts (i.e. without considering different values for parameters). In the case of a context-sensitive analysis, two directions might be followed. Either a construct is re-analysed for every new context, or a sort of parameterised analysis is produced once which can then be specialised for each analysis construct.

In general, the basis underlying static analysis methods is strongly mathematically founded. A range of different techniques exists which aims specifically on what kind of problems of the software need to be analysed. This explains why a lot of special purpose tools exist: if the underlying mechanisms can be focused on particular problems, for example finding nil-pointer references, as in UNO [Hol02b], the static analysis and the needed algorithm can better be fine-tuned for that particular purpose. Most techniques do have in common that they are based on how data and control flow through a program. We briefly discuss them below.

3.5.2 Data-flow Analysis

Data-flow analysis is a technique for gathering information about the *possible* set of values calculated at program statements in a software program. For performing data-flow analysis, a control flow graph is used for providing the link between the consecutive program statements. Such a graph visualises the possible propagations of calculated values assigned to variables.

Control Flow Graphs A control flow graph is needed to perform flow-sensitive analysis (in combination with data-flow analysis), as the *order* in which the statements are executed is taken into consideration. A control flow graph (CFG) is a directed graph in which the *nodes* correspond to program points (or statements) and the *edges* represent the *possible* flow of control (or program transitions). An example of a CFG is shown in figure 3.3, which depicts the flow of control for executing an iterative factorial function [Sch].

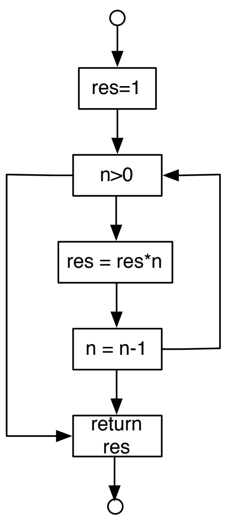


Figure 3.3: Control-flow graph of an iterative factorial function

Data-flow Analysis Classical data-flow analysis makes use of a control flow graph where a certain variable is assigned to each node. The possible values for that variable are grouped in a mathematical structure called a *lattice*. Basically a lattice represents a (finite) partial order [1] between elements which is used to represent constraints between the associated variables belonging to different nodes of the CFG. A collection of constraints over these variables can systematically be extracted for a complete CFG. Different analyses can be based on this data-flow analysis scheme. Consider an approach in which variables in a program need to be analysed to make sure that their values have already been initialised. For this purpose a data-flow analysis might be used to detect the points in a program where variables are *live* or not. A variable is *live* at a program point if its value may be read during the remaining execution of the program. By way of illustration we have included a program fragment in figure 3.4. The column on the right shows for each program point the least solution for live variables. As shown on line 6, the variables x and y can be read from that point on in the program, i.e. they are live variables. In this example, the *lattice* for this analysis is a set containing all possible subsets of the set $\{x, y, z\}$ with binary relation $\subseteq$.

```
1   [entry]               {}
2   var x,y,z;            {}
3   x = input;            {}
4   while (x>1) {         {x}
5     y = x/2;            {x}
6     if (y>3)            {x,y}
7        x=x-y;           {x,y}
8     z = x-4;            {x}
9     if (z>0)            {x,z}
10       x = x/2;         {x,z}
11    z = z-1;  }         {x,z}
12  return x;             {x}
13  [exit]                {}
```

Figure 3.4: Example program with live variables at every program point (taken from [Sch])

Other data-flow techniques with different goals exist that use the same principle as explained above. Calculating the *reaching definitions* for each program point is one example [Sch]. The reaching definition for a particular program point are those assignment statements that may have defined the current values of variables. To perform this analysis, the elements in the lattice will be the possible sets of assignment statements. Note that data flow analysis as it was presented here is only used for *intraprocedural* analysis, i.e. analysis which is done local to the body of a function. When complete programs containing function calls are considered, the analysis is called *interprocedural*. Precise interprocedural data-flow analysis is difficult to realise as the CFG for a whole program can become excessively large and consequently analysis does not scale. This is especially the case for programming languages which allow higher order functions, objects or function pointers, since this intertwines control flow and data-flow.

[1]A finite partial order is a mathematical structure consisting of a finite set with a binary relation defined on it. The relation must satisfy the following conditions: *reflexivity*, *transitivity* and *anti-symmetry*

3.5.3 Evaluation

In this section we briefly touched the subject of static program analysis. The very basic concepts of such analyses which are relevant to study the usefulness for checking design invariants were discussed. In section 3.1 the characteristics of design invariants were matched with the capabilities an analysis tool must have to be able to document and check them. First, as they are system-wide, if static analysis would be used, an interprocedural analysis would be needed as multiple function calls spread over a system need to be considered. The analysis would ideally be context-sensitive to produce results as precise as possible. Second, as design invariants are primarily operational, it was concluded in section 2.7 that they are mainly based on events which occur at a certain time during execution. Hence a flow-sensitive analysis would be needed as well. Flow-sensitive combined with context-sensitive interprocedural analysis is very difficult to realise practically due to the fact that every program statement needs to be taken into account. Limiting the analysis to a particular design invariant would be hard if not impossible. Static analysis can be performed in a modular fashion, but only according to the modularisation constructs provided by the programming language (e.g. functions or abstract data types). Another issue which makes it difficult to use static analysis for checking design invariants is the level of abstraction which is offered by static analysis approaches. Static analysis stands very close to the implementation level as it is based on analysing program statements. Expressing properties formulated at the design level introduces another level of complexity.

In the next section we discuss dynamic analysis and we study the features needed for checking design invariants.

3.6 Dynamic Analysis

As stated in section 3.2, dynamic analysis is the analysis of properties of a running program. In contrast to static analysis, dynamic analysis derives properties that hold for one or more executions. As a consequence they cannot guarantee that a certain property holds for all future executions. Nevertheless they are precise and efficient, whereas static analysis approaches are conservative and sound [Ern03].

For checking behavioural properties which are primarily based on specifying the order of operations rather than state, dynamic analysis is the most straightforward solution. The notion of an *event* is the primary constituent of acquired run-time information. Run-time values can be associated with events which hold precise information about the run-time state at the moment the event occurred. Such an event can be compared with a *part* of an execution path. Using the terminology for describing execution paths for a finite state machine as described in section 3.3, an event consists of an event type or *transition* whereas actual run-time values associated with that type of event represent the *state* of that transition *before* or *after* the transition has taken place. The *before* or *after* state depends on whether the instrumentation code for logging that statement was inserted *before* or *after* the program statement corresponding to the particular transition. This means that an execution trace holding a collection of events can be regarded as a collection of execution paths or even a path through a finite state machine. We come back to this later in this chapter.

In general, tool support performing dynamic program analysis by reasoning about the behaviour of a program consists of two main phases:

- **Collecting run-time information** Usually program instrumentation is set up for recording run-time operations or *events* together with associated run-time values at specific points in the program. Dynamic analyses differ in the level at which the code instrumentation is performed, the means used for selecting events and how behaviour is specified.

- **Analysing the collected run-time information** Depending on how the gathered run-time behaviour is represented, different analysis techniques exist. Support for analysing this behaviour is needed. The analysis can either be performed online while the system is still running or post-mortem after execution terminated.

3.6.1 Comparison Criteria

In this section the varying factors of dynamic analysis are discussed and how this relates to the context of checking design invariants. The following comparison factors of dynamic analysis approaches are handled here:

- how to collect the run-time events,

- the means of selecting program parts of interest,

- the meta model used to represent run-time events (how program behaviour is represented),

- expressiveness of the language used to instantiate the behavioural analysis to a developer's needs,

- the analysis time.

Collecting run-time events

For collecting run-time events of a program under execution, we need to monitor the execution in some way. According to Hamou-Lhadj et. al. [HLL04] three main ways exist to obtain execution information of a program automatically at run-time. The first two techniques are based on instrumentation, which consists of inserting instrumentation code (such as for example a print statement) at particular locations of interest either at the source code or at the machine-code level. A third way they mention is using a debugger for collecting events at certain identified breakpoints in the code.

We consider the instrumentation approaches here as debugging is further considered in section 3.6.2 as a dynamic analysis approach on its own.

A frequently used way to collect run-time events is to instrument the target program under investigation, so that in addition to executing the code normally, it also produces behavioural information of the running program. This is done by identifying those points in the system under investigation that are of interest for further analysis. At that point in the program, extra statements are inserted to reveal the dynamic information of exactly that event. The two most commonly used instrumentation techniques are to modify the source code of a program or to modify a compiled binary representation of the target program. Instrumentation at the source level is performed when target language information is needed, i.e. when reasoning about a higher level of abstraction is necessary. Analysis at the byte-code level has the advantage that the source code is not required to perform the analysis. This is useful in cases where the source code is not available. Another advantage is that no complex parsing of the source code is needed (a list of machine instructions is much easier to parse). A difficulty of binary-level instrumentation is coping with the abstraction level as instrumentation is done at a lower level (at machine instruction level), however the user expects feedback at least at the implementation level. Some approaches even try to combine instrumentation at both levels [Guo06].

Means of Selecting Program Parts of Interest

Not all applications of dynamic analysis need to capture every run-time event which occurs during the execution of a program. It is very useful to restrict the parts of a program from which events are intercepted in order to focus the investigation of the program and to decrease the amount of information that is recorded in an execution trace. The expressiveness of the medium through which developers can select parts of the program to be investigated is therefore another interesting criterion of comparison. Within the domain of program understanding, many tools exist based on *trace exploration* [HLL04]. These tools *need* to aim at extracting any information to comprehend a software system better, so in general little attention is paid to event selection through instrumentation. For example for object-oriented languages usually all method calls and object allocations are logged. Their contributions thus mainly lie in the techniques they use for *coping with* the size of the traces. These tools typically perform off-line analysis of (often very) large execution traces. Some top down exploration approaches try to use knowledge already gathered about a system to filter the execution traces afterwards. In Walker et al [WMFB$^+$98] they use architectural knowledge about a system to visualise only the interactions of those main components at the architectural level.

Many tools exist which all have their own way to select events. Some approaches instrument the entire program according to a fixed set of implementation language constructs [GOA05, Roo04] (for example instrumenting all method calls and object allocations), however some provide a means to instrument individual method calls or object allocations selectively (e.g. by enumeration) [DFW04, DGD05]. Stolz et. al. use an AspectJ-like pointcut language to select events and instrumentation is done at the byte-code level by using boolean combinations of regular expressions [SB05]. Other approaches analyse the behaviour in parallel with program execution performed in a stepwise fashion allowing unimportant evens to be skipped [GDJ02, Duc99]. Other ways of selective instrumentation are through parse tree annotation according to type and value masks [TJ98] or by using a functional expression directly related to programming constructs [Aug98, Aug00, AJU02].

How events are selected seriously influences the scalability (and also the usability) of the dynamic analysis approach. Performing code instrumentation by for example logging all function calls and all variable assignments already might result (even for a simple execution scenario) in an extremely large execution trace. This heavily influences the usability of a tool. Moreover, the behavioural analysis performed on the execution trace needs to be able to deal with a huge amount of information, which might limit possible analysis approaches.

Representing Program Behaviour

The computational process often expects the recorded run-time events to adhere to a well-defined meta model. Such a meta model prescribes the different kinds of events that might occur at run-time and the amount of run-time information that is recorded about each event. The granularity of the meta model not only determines the information accessible about a program's execution, but also has an impact on the space-time cost of the computation, especially in combination with a post-mortem analysis strategy. As an example let us consider dynamic analyses of object-oriented programs. Such analyses usually capture method invocation run-time events. A coarse-grained model would for instance just record the selector of each invoked method while a finer-grained model might also record the identity of the receiving objects and the arguments that were passed along. An extremely fine-grained model could also store deep copies of the involved objects before and after the method invocation. The nature of the meta model itself can vary as well: events can be modelled as instances of classes in an object-oriented meta model [KF04], as procedures in a procedural language [TJ98], as predicates in a logic meta model [RD99, Ric02, RD02, Roo04, DGD05, GDJ02] or even as tuples in a relational meta model [GOA05]. The meta model, its granularity and nature are thus other important criteria, again depending on the application domain for which the analysis is intended. Specifying program behaviour greatly influences the kind of behavioural analysis which can be applied later. The level of abstraction plays an important role. Creating a high-level model of program behaviour instead of specifying a model in low-level implementation constructs provides more opportunities for applying the behavioural analysis at a high level of abstraction. In this context, this would allow to specify a design invariant directly over high-level events.

Support for Analysing Program Behaviour

Some application domains of dynamic analysis require the actual computation over run-time events to be customisable by the user. General program monitoring frameworks are made to offer this customisability. Such monitor notification tools (such as JMonitor [KF04], CCI [TJ98], PMMS [LC92] and many others [PN81]) provide a general framework that can be used to instrument a program, but they provide no further means for analysing the information obtained from the instrumentation. Two components have to be specified by the user to instantiate the monitoring framework. First an event pattern needs to be provided where the user expresses the kind of events he wants to reason about together with the run-time information of that particular event. And second, a program monitor that is attached to that event pattern needs to be defined to specify what code needs to be executed when that event occurs. General program monitors provide no domain-specific computational language specifically tailored to reasoning about intercepted events. This is left up to the user. Such a framework might for instance be instantiated with a user-implemented monitor which logs time stamps to a file whenever it is notified of a run-time event.

Lightweight program verification tools might on the other hand offer a high-level language in which to express assertions over run-time events. The expressiveness of the language with which we analyse the run-time events is another important comparison criterion. One previously mentioned approach by Stolz et. al. expresses such behavioural assertions as temporal logic formulae over AspectJ joinpoints [SB05]. Auguston et al. express behavioural assertions in an expressive functional high-level language called FORMAN which includes quantifiers [Aug98, Aug00, AJU02].

Using a language to express assertions in allows to create specifications of wanted behaviour which are directly executable *over* run-time events. Moreover, in this context, using an expressive high-level language provides a means to document invariant behaviour and have it checked immediately against an execution trace. This way, a form of active documentation is created which couples documentation actively with application's the source code.

Analysis time

Depending on a particular analysis' needs, analysis of run-time information can either be performed on-line during an application's execution or post-mortem. With post-mortem analysis (or off-line analysis), events are written to an execution trace and after program execution ends, this trace is analysed. With performing analysis at run-time, the dynamic information obtained so far is analysed in parallel with further program execution.

Both ways of analysing dynamic information has its advantages and pitfalls. On the one hand, off-line analysis allows one to work on a constant size of memory, but in reality processing large execution traces generally takes up an equal amount of resources than doing analysis on-line. The main advantage of off-line analysis is that the analysis process does not have to compete with the running application for space. Furthermore, off-line analysis approaches can exploit the ability to go through the trace sequentially multiple times and they allow for more sophisticated reasoning. Managing large traces of data does produce a considerable overhead, even taking into account that nowadays storage is cheap. On-line querying might seem simpler for the user as post-processing steps are eliminated and quicker feedback can be given. The program can also be stopped whenever certain behaviour is detected.

Plenty of dynamic analysis tools perform off-line analysis [DFW04, RD99, Ric02]. Debugging tools are usually performed on-line, such as Coca for debugging C programs [Duc99, GOA05], while other debugging systems use a combination of both on-line and off-line analysis [GOA05, Aug98, Aug00, AJU02].

Target language being analysed

Program analysis in general is being performed on systems in all types of programming languages since tools in many different application domains need some kind of information of that system. The kind of analysis which is generally used is not completely independent from the target language being analysed. For example, dynamic analysis of object-oriented systems has been explored quite often these past few years. Static analysis has proven to be less useful for these type of systems due to polymorphism and late binding, which makes dynamic analysis a very welcome addition. Also a procedural language such as C is known to be an unsafe language that is quite vulnerable to memory errors. Because of this, sound static analyses for C are more difficult to produce. It also sometimes causes some static tools to produce many false positives due to difficult analyses of indirect referencing through pointers. In such cases, a dynamic analysis might be a better choice, certainly in the case where correct information is needed, which is easier to obtain at runtime.

When having a closer look at currently used systems, many older and often very large (procedural) systems are still employed that are often still heavily used. These systems are usually referred to as *legacy systems* . These type of systems often need all the help they can get from static analysis as well as dynamic analysis as they are often maintained in an environment where no documentation is present (or not kept up to date) and where the original developers have already left the company. Analyses best suited for procedural languages are thus still heavily needed [MDTZ03].

3.6.2 Relevant Application Domains

A whole range of software engineering tools based on dynamic analysis exist that serve many different application domains. As the intention is to focus here on the application domain of supporting design invariants in software, we zoom in on some of these domains that are close to this dissertation context.

Query-based debuggers Coca [Duc99] performs a dynamic analysis of C programs that is especially suited to debugging and program profiling purposes. The program under investigation is executed in a stepwise fashion by a classical debugger. The computational process runs in parallel with the debugging process (the analysis is performed on-line) to guide the execution of the program. This allows events that are unimportant to the computational process to be skipped. The computation over run-time events is thus performed on-line and no events need to be stored. The event meta model is declarative in nature: a logic fact represents the current run-time event which can be directly mapped to one of the C programming language constructs. The computational language used to express behavioural debugging and profiling queries is also declarative in nature. It is a Prolog variant augmented by predicates to request specific future run-time events. Event selection happens at run-time by binding attributes of the event to concrete values. This allows for dynamic filtering of events. The computational process reasons at run-time about very low-level events to which dynamic filters can be applied. The query language is declarative and highly expressive. Due to the on-line nature of the dynamic analysis, the computational process has only access to the current run-time event. The applicability of this approach is therefore targeted at debugging and profiling purposes: it is impossible to consider alternative matches for a run-time event without advancing the application. This makes it difficult to reason about nested events for example.

Error detection tools A wide range of software engineering tools serve the purpose of detecting certain errors in software, for example profilers, program testing tools and assertion checkers. They all strive to making a program work without unacceptable system errors. Profilers provide numerical summaries of dynamic information of a program, such as the length of time spent executing a certain procedure.

In the work by Goldsmith et al. [GOA05] an online dynamic analysis approach is presented for finding correctness or performance bugs in Java programs at run-time. More generally it can also be used for profiling and debugging. Events are specified as records in a relational table. Two different relations are specified, method invocations and object allocations, with the different fields representing the run-time information of a particular event. For obtaining the events, their PARTICLE compiler generates and inserts instrumentation - at the byte-code level - before each method and after any object allocations. Analysis of Java program behaviour by querying is done using PTQL (their Program Trace Query Language) - a declarative SQL-like language.

Lightweight verification through assertion checking Program verification is usually understood as a heavyweight approach using *model checking* (as described in section 3.3) to verify the correctness of a program with respect to a particular program property.

We label the verification approaches we consider here *lightweight* as they are based on a dynamic analysis approach where only part of program execution is modelled based on a particular execution scenario. Hence the properties are rather *checked* instead of being *proven* to hold for the entire program model, as they are only considered along one execution path. So properties can only be guaranteed to hold for a particular execution path and not proven correct for the entire program execution.

Auguston et al. [Aug98, Aug00, AJU02] present an expressive specification language to describe an application's behavioural aspects and apply dynamic analysis to verify these assertions in a lightweight manner. A program's behaviour is modelled as a graph over - possibly composite - run-time events supported by an inclusion and precedence relation. Atomic events directly related to programming language constructs occur at a specific moment in time, while composite events occupy an interval in time. For each programming language, an event grammar formally specifies the constituents of high-level events and their mutual ordering on the time line. This way, the event grammar allows automatic low-level event selection according to a given assertion over high-level events. Verification is in general performed post-mortem, but can be optimised to an on-line computation when the assertions allow it. Assertions are expressed in the functional, highly expressive language FORMAN which includes quantifiers, boolean operators and aggregate operations over events as well as target language operations over target program variables to express intended behaviour or known types of error conditions. The run-time event's meta model is formally defined by an event grammar specifying inclusion and ordering relations over low-level events directly related to programming language constructs. While the event grammar allows automatic low-level event selection, the run-time information that is recorded about each composite event is still dependent on its low-level constituents.

In the work by Stolz et. al. [SB05] a lightweight run-time verification framework for Java programs is presented. Event specification is done by modelling a program's behaviour as a trace of consecutive states. In this approach, run-time states that one can reason about are limited to those that are addressable through AspectJ pointcuts (see section 3.6.3). By allowing a selection of points in the dynamic control flow of a program, pointcut expressions select a set of states from the run-time trace. To collect events at run-time, Linear Temporal Logic (LTL) formulae with pointcut expressions as propositions are translated into automata. By means of aspect advice code, the automaton is *walked through* during program execution, detecting a violation of the LTL assertion when an error state is entered. Also here are problems with reasoning about matching and nested events due to the run-time analysis.

Test Oracles Software testing is a discipline which includes the process of executing software with the intent of finding errors, which classifies it as a dynamic analysis approach. In software testing, an *oracle* is regarded as *some method* for checking whether the system under test has behaved correctly on a particular execution [BY01]. To state it differently, an oracle is a 'system' which takes a specification of a property (i.e. the test) and verifies whether that specification holds for a particular execution path. Although test oracles are frequently being referred to in literature, automatically applicable oracles are not described. Embedded assertions for example as discussed in section 2.4.1 can also be seen as oracles. Considering assertions as specifications of a certain property (usually specified in the implementation language where they are embedded in), assertion support is a way of using those specifications as test oracles. However, they suffer from significant problems as already noted in section 2.4.1.

Log file analysis as mentioned by Andrews provides another interesting example for looking at test oracles [And98]. A framework is offered for automatically analysing log files and a language (and its implementation) is presented for specifying analyser programs. A log file *machine* is seen as a set of parallel state machines as discussed in section 3.3, each state machine analysing one execution of events. These machines react in sequence to each line of a log file , doing a transition for a corresponding event whenever possible. If at the end of going through the events from the log file the machine is not in a final state, an error is reported. They use the LFAL language to specify a log file analyser which is based syntactically on a representation of state machines. However, the approach does not address how to obtain the log files (i.e. they rely on manual instrumentation). Another example of test oracles is described by Richardson [RAO92]. They are derived manually from system specifications (written in multiple languages) for dealing with reactive systems.

3.6.3 Aspect-Oriented Programming

In any programming language, certain concerns, such as logging or error recovery, are scattered throughout many components of that language. Such concerns cannot be separated cleanly according to the decomposition the programming language at hand provides. For example for an object-oriented language we are talking about concerns that are spread over different classes of a system. Aspect-oriented programming (AOP) [KM05a, KM05b] provides a means to concentrate the necessary code for these crosscutting concerns in one single module (called an aspect) instead of spreading it over the code. AspectJ [Asp], an aspect-oriented extension for Java, accomplishes this advanced separation of concerns via *aspects*, *join points* , *pointcuts* and *advice* . A *join point* is a point in the program's execution, such as a method entry or exit. A *pointcut* is a set of join points. *Advice* is code to be executed either *before*, *after* or *instead* of the code at the joinpoint. Advice code has access to run-time data about the joinpoint that triggered its execution. An *aspect* then consists of pointcuts and advice. As an example consider a logging aspect that contains a pointcut describing all places in the code where some data of interest should be written to a log file. The advice then would contain some `print` statement to perform the actual logging.

Although AOP is a programming methodology and has nothing to do with program analysis, there is common ground in the sense that a logging aspect might be used for selectively instrumenting a program. This approach is then comparable to a general purpose program monitor in the sense that the approach itself does not offer any way to specify or analyse the logged events. The user has to take care of that. The most interesting point of comparison is the language used for defining the pointcuts, i.e.

means of event selection as was specified above in this section. The work described by Gybels et al. for example uses a logic language as pointcut language to avoid tight coupling between the aspect and the program [GB03].

3.6.4 Evaluation

In this section, the concept of dynamic analysis was explained into more detail and important criteria of comparison were discussed. Evaluating a dynamic analysis approach for supporting the specific properties of design invariants leads us to the following conclusions.

- Dynamic analysis is a *lightweight* methodology which is very well-suited to be used in an agile setting. Analysing execution traces instead of *exhaustively* exploring an entire model of software is less computationally expensive. A compromise to make is that the results apply only for the analysed run(s) in question and not for all possible executions.

- The analysis can focus on only those parts of interest for particular invariant behaviour. *Event types* (with associated run-time values) of interest (i.e. specific transitions) can be identified through employing a sophisticated means of event selection. *Program states of interest* can be selected by carefully choosing a suited execution scenario for particular invariant behaviour. Note that this provides a way to focus the analysis to entities that may *cross-cut* an entire system, i.e. entities that belong together because of addressing a behavioural concern, but which are not necessarily captured within one module of the programming language at hand.

- To address the operational characteristic of design invariants, dynamic analysis approaches fit right in as the concept of an event forms the primary building block for representing actual behaviour (as mentioned in section 3.6).

- Using dynamic analysis for checking *design* properties of a system requires the representation of the actual behaviour (i.e. the execution trace) to be at a high level of abstraction. In that case the high-level specification of the design invariant can be directly verified in terms of the execution trace. However, existing dynamic analysis approaches make use of an event grammar to fix the representation of actual behaviour [Aug98] which is usually dependent on low-level events. Advantages of using a fixed set of (low-level) events are the automatic selection of events and less user involvement.

3.7 Conclusion

In this chapter, existing program analysis approaches were surveyed and their suitability for documenting and verifying design invariants in an incremental software development context was evaluated. A formal verification method called *model checking* was first discussed. This type of formal analysis is conducted on a model of a program rather than a program itself. Next, we elaborated on analysis methods directly analysing source code, thereby making an important distinction between static and dynamic program analysis approaches.

The appealing aspect of *model checking* was found to be the high level of abstraction of the specification language and the range of properties which can be expressed and verified. Although model checking is used for different purposes in the sense that it is an *exhaustive* method for guaranteeing correctness with regard to a specific property instead of merely checking that property, it does meet the need for specifying properties at the design level. Moreover, specifying properties *over* an FSM model of a system allows one to formulate invariant behaviour based on states as well as on events (or transitions). This also allows the specification of operational, system-wide properties which can be expressed at the design level. However, the high abstraction level of the property specification language can be attributed to the level of abstraction of the software model *over* which the properties are checked. As the model describes the design of a system rather than the implementation constructs themselves, the properties of this model are also expressed at a high level of abstraction (i.e. at the same level as that of the model). However, establishing such a model is costly and is not affordable in a lightweight agile setting of aiding a software developer in creating reliable software as it is developed. Moreover, a correctly proven model of a system does not imply that the corresponding implementation is correct at the technical level.

Static analysis approaches were found to be less suited for checking design invariants. The nature of such invariants imposes the need for a static analysis approach to at least perform a flow-sensitive interprocedural static analysis as explained in section 3.5. Such an analysis would require the representation of a control-flow graph where each program statement is included. An analysis would be exhaustive, such as for model checking, except that the implementation statements are exhaustively traversed instead of traversing a model of the software. This explains the numerous existence of static analysis tools that only target specific kinds of errors. Analysis can be more focused and be made more scalable so that they also produce fewer false positives. However there is no means for focusing the analysis on a cross-cutting entity such as a design invariant. Another problem is that analysis is directly performed on the source code. Static tools offering a specification language to let a user for example specify invariant behaviour is directly expressed in low-level implementation constructs. This limits the expressiveness and hence the suitability for the purpose of documentation.

Having surveyed dynamic analyses and their criteria of comparison, dynamic analysis is first of all a *lightweight* methodology in the sense that it analyses execution traces instead of exhaustively exploring a behavioural model of a program. Second, such an analysis can focus on only those parts of interest that verify particular invariant behaviour through careful event selection and choosing a particular execution scenario. This supports possibly the cross-cutting property of a design invariant. And third, carefully selecting points of interest by instrumenting them, together with specifying the corresponding behaviour at a high-level of abstraction provides a way of verifying a high-level specification of a design invariant directly against high-level behavioural events. This bridges the gap between low-level implementation constructs and high-level design properties of a software system.

3.8 Summary

In this chapter, we surveyed existing program analysis techniques. We highlighted their features which allow checking *design invariants* for a particular software application. In section 3.1 we pinpointed the characteristics an analysis tool needs to possess in order to verify design invariant specifications against the program under investigation. Section 3.2 outlined the different types of analyses and important terminology needed throughout the remainder of this dissertation was defined. Next, we distinguished between two main analysis approaches. Section 3.3 discussed analysis methods which are applied to *models* of a program rather than code. The second group of analyses which is performed on code was handled in section 3.4. After comparing the main characteristics of static and dynamic code analysis, we presented in section 3.5 the main building blocks of static analysis which are needed for checking design invariants. After doing the same for dynamic analysis in section 3.6, an evaluation was performed for both analyses within the context of this dissertation.

In the next chapter we present a conceptual framework for the documentation and lightweight verification of behavioural design invariants. We first introduce a set of four requirements which should be met by the framework, based on our study of invariants in chapter 2 and of program analysis which was discussed in this chapter. After that, we demonstrate how the conceptual framework tackles these requirements.

Chapter 4

A Goal-Driven Approach for the Documentation and Lightweight Verification of Design Invariants

In the previous chapters we introduced the essential background information that is needed to position and understand the contribution of this dissertation. This chapter describes our approach to documenting and verifying design invariants. It is a lightweight declarative approach combining selective code instrumentation with high-level behavioural analysis.

Section 4.1 emphasises the need for supporting documentation and lightweight verification of design invariants. First the problem context for this dissertation is defined in section 4.1.1. In a nutshell, design invariants pose a problem for program development as they are *implicitly* present in software. Even if they are made explicit, they are *detached* from the source code so that discrepancies in either the invariant documentation or the source code need to be updated manually. Heavyweight program verifiers are able to analyse design invariant behaviour. However as the result of formal verification consists of a correctly proven *system model* (with respect to certain properties), these approaches are not applicable for supporting technically and algorithmically complex software as the true difficulty lies in development at the source code level.

Based on an evaluation of existing program analysis approaches (Chapter 3), we define a set of four main requirements in sections 4.1.2 – 4.1.5. These must be satisfied by our approach to offer support for documenting and verifying design invariants. In these requirements, the need for a descriptive specification language is formulated to make design invariants explicit (Requirement 1). Moreover, a causal link with the actual source code needs to be provided to make the design invariant machine-verifiable (Requirement 2). To support practical use during program development, a tight coupling with the source code should be avoided for not having to change the design invariant specification every time the source code changes (Requirement 3). Furthermore a goal-driven approach is advocated to allow focusing the analysis on specific parts of a larger program (Requirement 4).

We continue the discussion in section 4.2 by presenting a global overview of our proposed approach for supporting design invariants. We couple the main solutions of our approach to the requirements that were identified in section 4.1. Subsequently, the two main phases of our approach are explained in more detail. We first elaborate on the verification phase in section 4.2.1, followed by the specification of behaviour in section 4.2.2, in essence the most distinguishing feature of our approach.

We present in section 4.3 the behavioural formalisms used for analysing program behaviour. First we show how a partial high-level model of program behaviour is represented, followed by the executable behavioural formalism that is advocated in our approach for documenting and verifying design invariants. Section 4.4 then pinpoints how the behaviour is obtained at run-time by adopting a selective source code instrumentation scheme. For that purpose, the logic meta programming paradigm is introduced as a means to reason about the structure of a base language program. As more developer involvement is required in our approach, we introduce in section 4.5 a four-step recipe on how the proposed approach can be optimally exploited.

A prototype implementation supporting the proposed approach is presented in chapter 5. How it can be used to reason about design invariants is extensively demonstrated in chapter 6.

4.1 Need for Documenting and Verifying Design Invariants

The complexity of software systems yields too many opportunities for developers to introduce faulty and erroneous behaviour in an implementation. System complexity is cited as a major reason for the difficulties software engineers encounter when dealing with systems [FPB87]. A range of complex software systems exist which combine different system components that are technically and algorithmically challenging. For such systems, complexity resides at the code level. The combination of all these components gives rise to subtle interactions at the code level that are prone to faulty behaviour.

Dealing with software complexity during program development manifests itself in different ways. One major difficulty to overcome is the adaptation of existing software functionality. Support for easing the process of change by designing systems according to a clean separation of concerns is a first step [HL95]. Consequently, when changes have to be made, they are more localised and hence made faster and in a straightforward way. However, the impact of a change on the overall behaviour of a system cannot be guaranteed. Even changes to a well-designed system with a low coupling and high cohesion might break underlying intended behaviour.

As discussed in section 2.3, invariants form a crucial part of a system's behaviour. They do not only represent characteristics of a system at a certain moment of its existence, but they also represent constraints which must hold throughout every change cycle if the system is working correctly. They are essential to the comprehensibility and reliability of a software system in general [Ern00]. Explicitly stated invariants can help program development by documenting certain aspects of program execution and identifying program properties that must be preserved when modifying code.

In the context of this dissertation, we focus on design invariants in software. In section 2.6 the following definition of a *design invariant* was explained in detail:

Definition
*A **design invariant** is a behavioural regularity of the design of a program that is not aligned with the structure of the program.*

In a first step, we want to support program development by making these type of invariants, which are usually implicit, an *explicit* part of the software. It is important to make people aware of these constraints as they represent a crucial piece of information for being able to reliably adapt the software at the further stages of development [HT00]. In a second step, to support practical use, we want to partially automate program development with respect to these design invariants by providing a method for *automatically verifying* them in a lightweight manner against the actual source code.

Due to their specific characteristics, design invariants are difficult to capture and verify. First, as they represent properties related to the design of a system at the code level, a higher level of abstraction is needed in order to specify them. Second, their often cross-cutting characteristic poses constraints on the verification mechanism which can be used, especially when taking into account technically challenging software. And third, as they are non-externally verifiable, the formalism for specifying their behaviour should be primarily event-based (see section 2.6).

Some limitations need to be taken into account before presenting our approach. The type of software system to which the proposed approach can be applied is any sequential (single-threaded) software system for which a correctly working version of the source code is available:

- *A sequential system*: the proposed behavioural formalism as it is now can only be used to reason about the behaviour of single-threaded systems as time is represented in a linear way by using sequential points in time.

- *A running version of the source code*: a running version of the source code needs to be provided as program instrumentation at the source code level is performed.

- *Programming Language*: the proposed approach can in principle be applied to programs written in any programming language.

In the next section we first define the problem statement for this dissertation. Next, based on the problem statement a set of requirements is distilled which a tool for supporting design invariants needs to satisfy.

4.1.1 Problem Statement

Software developers responsible for adapting and evolving an existing implementation of software must grasp the underlying behavioural dependencies implicitly imposed by making certain design decisions about the software at hand. This is problematic because these so-called behavioural *design invariants* are often not known to developers or other peers involved in a software project. Even if they are known, they are often only available in some *implicit* form. This severely compromises program development as making a change might result in violating unknown behavioural constraints which ultimately lead to unreliable software. Even in a context where such behavioural design invariants exist in an explicit form, this documentation is not linked to the actual program. This would imply that developers themselves have to detect manually those places in the source code which trigger the invariant behaviour, which would be a difficult and time-consuming activity. It might even be more complicated as actual run-time values might play a leading role in capturing a particular invariant.

Considering existing analysis approaches for checking design invariants, formal verification approaches which analyse a model of a system rather than the code itself are difficult to use for technically challenging software. As the main difficulty lies in the technical development of the source code, having a correctly proven system model (with regard to certain properties) does not help at all with the technical realisation of the software at hand. Other static analysis approaches do not provide means to focus the analysis on a particular behavioural design invariant not localised in a component of the programming language at hand.

In summary, an approach is needed for making design invariants explicit by first documenting them in a behavioural formalism and subsequently making that documentation machine-verifiable at the same time. Such an approach should address the following main problems:

- **Implicit Design Invariants**
  Design invariants impose underlying behavioural constraints on a system, but they are implicitly present in the software. Hence, every time a change is made to the software, the design invariant might be violated, possibly inducing severe program errors.

- **Detached Design Invariants**
  Even if design invariants are made explicit, they are often detached from the source code. As design invariants are system-wide and may cross-cut an entire system, this poses a problem as it is difficult and time-consuming to check them manually. Moreover, as design invariants might depend on the run-time state of a system, such a check might even be impossible to perform. On top of that, these checks have to be done *every* time a change is being made to the system.

- **Non-Oblivious Design Invariants**
  As a design invariant has to be checked *every* time a change is made to the

system, the impact of changes to the design invariant description presents another problem. Specifying the invariant in low-level implementation constructs creates a tight coupling with the application's source code. Such a link with the source code inhibits practical use as the specification needs to be adapted *every* time a change is made to the source code.

- **Lack of Support for Partialness**
  To check a particular underlying behavioural constraint, existing program analysis approaches analyse the complete system behaviour by exhaustively checking the possible program states. Other approaches allow to separately analyse modules of a system as they are decomposed by the programming language at hand. However as design invariants can be cross-cutting entities, this would still entail analysing an entire program's behaviour, which seriously inhibits practical use. Such approaches lack support for partialness as they are not able to focus the analysis on only a very particular part of the software's behaviour.

Based on the problem statement, a set of requirements is distilled in the next sections which our approach has to satisfy to support the documentation and lightweight verification of behavioural design invariants.

4.1.2 Implicit Design Invariants

In this dissertation, we want to support the development of technically and algorithmically challenging systems. One of the factors that inhibit a developer from producing reliable software are behavioural design invariants. Such invariants impose underlying behavioural constraints on a system, but they are implicitly present. Hence, every time a change is made to the software, the behaviour of the invariants might be broken, possibly inducing severe program errors.

So, in a first phase, developers must make these invariants *explicit* by documenting them using a behavioural specification mechanism. Due to the particular characteristics of design invariants (section 2.6), some issues have to be taken into account. Simply annotating them in the code as embedded assertions (section 2.4.1) where they are specified directly as source code statements constraining local state is not an option as design invariants are system-wide and often cross-cut an entire system. Hence they are not localizable into a component of the programming language at hand.

As design invariants represent behavioural properties which are system-specific, a developer should therefore be offered a *behavioural specification language* in which system-specific invariant properties can be specified. Since developers themselves have to specify their system invariants using this formalism, specification should not become too tedious. Moreover, as design invariants are non-externally verifiable and depend more on execution order, the specification language should support primarily event-based behavioural specifications (section 2.2).

We summarise the findings through formulating the following requirement:

Requirement 1: Need for a Behavioural Specification Language
As design invariants represent behavioural properties which are ***system-specific****, a developer should be offered a* ***specification language*** *in which design invariants can be documented. The language should primarily support* ***event-based specifications*** *and as the specification should be used for documenting design invariants, the specification language should be* ***descriptive****.*

4.1.3 Detached Design Invariants

Making design invariants explicit is a first step towards supporting a developer in keeping her software as reliable as possible. It makes developers aware of underlying behavioural dependencies present in the software. As design invariants might be scattered all over a program, their behaviour inadvertently might be broken when a change is made to the system.

However, as design invariants are system-wide and may possibly cross-cut an entire system, it is difficult and time-consuming to check them manually. Moreover, as they might depend on the run-time state of a system, such a check might even be impossible to perform. On top of that, these checks have to be done *every* time a change is being made to the system. This imposes the need for a machine-verifiable approach in which the high-level behavioural specification of such a design invariant can be automatically verified against the source code.

Therefore a link needs to be provided between the behavioural design invariant specification and the actual source code. Hence when changes are made to the code for further development or software maintenance, the design invariant can be automatically re-verified. In chapter 3 existing program analysis methods for the verification of invariant properties were studied.

As formal verification approaches aim at proving the correctness of a *system model* with respect to certain properties, they are not suitable for supporting the development of technically and algorithmically complex software. For such systems, the true difficulty lies in development at the source code level. For static analysis approaches the required abstraction level would pose a problem as static analysis approaches stand very close to the actual source code and they provide no means for focusing the analysis on a cross-cutting entity such as a design invariant (section 3.7).

In this context, we therefore formulate a second requirement the proposed approach needs to address:

Requirement 2: Causal link with the Source Program
In addition to making design invariants explicit, ***a causal link*** *with the actual source code needs to be provided to make the invariant documentation* ***machine-verifiable****.*

4.1.4 Non-Oblivious Design Invariants

Providing a causal link to make design invariants explicit and automatically verifiable already supports a developer in keeping her software as reliable as possible. It makes developers aware of underlying behavioural dependencies present in the software. On top of that, coupling the design invariant specifications to the code makes them at the same time machine-verifiable which partially automates the development process.

However, as these checks have to be done *every* time a change is being made to the system, next to automating the verification of the design invariant descriptions, the impact of changes to the design invariant description should also be taken into account. Hence, although a causal link is needed with the source code to enable machine-verifiability, tight coupling should be avoided at all times for not having to change the invariant specification *every* time a change is made to the source code.

Again we summarise the findings in the following requirement:

Requirement 3: Oblivious Design invariants
To support program development, a tight coupling with the source code should be avoided for not having to change the design invariant specification every time the source code changes. So we require the behavioural specification language to be at a high level of abstraction *to create a design invariant specification which is* oblivious *to implementation constructs.*

4.1.5 Lack of Partialness

Technically and algorithmically complex software systems have many complex components to deal with at once. Not only the size of such systems makes them complex, but the intertwining of their components makes analysis even more difficult.

As summarised in section 3.7, another problem of formal verification approaches and static program analysis approaches is their inability to focus the analysis on a particular part of a program. For formal approaches, a complete representation of a system model representing all program states needs to be explored. Although research is focusing on ways to reduce this problem through introducing abstraction techniques (such as state merging or abstraction on the variables), this still remains a problem. As for other static analysis techniques, verifying unwanted design invariant behaviour based on analysing the control flow of programs would require at least a flow-sensitive interprocedural analysis in which every program statement needs to be included. They do allow ways to analyse only particular parts of a program, however this is limited to only those parts which are components of the programming language at hand (for example through intraprocedural analysis). However, this is not applicable to design invariants as they might be cross-cutting an entire system (section 2.6).

What is needed is a lightweight approach which allows the focus of analysis to be narrowed down to the design invariant behaviour we want to analyse. Such an approach needs to be *goal-driven* rather than having to go through a representation of the entire system behaviour, as in practice the behaviour to be analysed addresses a specific *behavioural design concern*.

Therefore we advocate the use of an approach which is based on steering the entire analysis by a specific design invariant rather than verifying it through exploring an entire system's behaviour. This allows us to analyse parts of a larger program as well.

This leads us to the fourth and last requirement:

Requirement 4: A Lightweight Goal-Driven Approach
As design invariants can be system-wide entities which cross-cut an entire application, to support practical use, the analysis used must offer a degree of ***partialness*** *to* ***focus the analysis*** *onto the design invariant in question.*

We now present our approach addressing these requirements.

4.2 A Lightweight Goal-Driven Verification Approach

In the previous section four main requirements were identified which our approach should satisfy. In this section we couple these requirements to the proposed solutions. Our approach is based on specifying a behavioural design invariant in a high-level behavioural model. In essence, our approach for supporting design invariants proposes the following solutions:

- Using a behavioural formalism for specifying a design invariant in a *behavioural model* using *high-level concepts*.
- Providing an approach for making the high-level behavioural model at the same time *machine-verifiable*:
  - A *lightweight verification* approach based on *dynamic analysis*.
  - A *goal-driven* approach by allowing to focus verification on specific parts of a larger program.

To reconcile the stated requirements from section 4.1, we offer a descriptive behavioural formalism in which system-specific design invariants can be expressed. This formalism primarily supports event-based specifications as non-externally verifiable entities need to be specified (Requirement 1). Second, we will make these behavioural models machine-verifiable by providing a mechanism to check whether the wanted behaviour expressed in the behavioural model matches the actual program behaviour (Requirement 2). Third, we introduce high-level concepts in the behavioural descriptions of design invariants. Such high-level concepts are of a higher semantic level than the individual programming constructs offered by a particular implementation language. This way, a behavioural specification serves as a means of documentation so that design invariants of a system can be described at the conceptual level (Requirement 1). On top of that, introducing high-level concepts also avoids a tight coupling between the design invariant model and the source code, which makes the design invariant model oblivious from implementation constructs (Requirement 3). Fourth, we adopt a lightweight verification strategy based on dynamic analysis. When applying such an approach, the program under investigation is executed along a well-defined scenario. Run-time events arising during the execution of the scenario are recorded in an execution trace. This makes the approach *goal-driven* as it allows us to focus the verification on specific parts of a larger program (Requirement 4).

In the remainder of this section, we outline an approach for making design invariants explicit by documenting their dynamic behaviour in descriptive behavioural models which can be verified in a lightweight manner throughout an application's lifetime.

Two important phases can be distinguished:

- In the first phase the design invariant is made explicit by documenting its behaviour in a model. This documentation phase consists of two parts:
  - specifying the behavioural model as behavioural assertions capturing wanted and unwanted program behaviour, and
  - a description of the high-level concepts *over* which these assertions are expressed (this is explained in section 4.2.2).
- In a second phase, the consistency of the model representing the wanted behaviour is then verified against the actual program behaviour represented by the recorded execution trace (see section 4.2.1).

Note that in case of inconsistencies, a third phase might be in order to interpret the obtained verification results. In section 4.3.2 a formalism is presented which lets us specify design invariants in high-level behavioural models. These models are at the same time machine-verifiable as the formalism we use is supported by a programming language which makes the high-level behavioural models executable. We also explain in that section the specification of the actual behaviour as descriptions of the high-level concepts over which the design invariant knowledge is expressed. To better understand the different components of the proposed approach, we first explain in the following section how the consistency of the actual behaviour with the design invariant behaviour is checked in the second phase. After that, the first phase of specifying behaviour is discussed in section 4.2.2.

4.2.1 Verifying Design Invariant Behaviour

As was clarified in section 4.2, the proposed approach should support a verification mechanism that is both *lightweight* and *goal-driven* (Requirement 4). Therefore we adopt a dynamic analysis strategy where the program under investigation is executed along an execution scenario and where run-time events arising during execution are

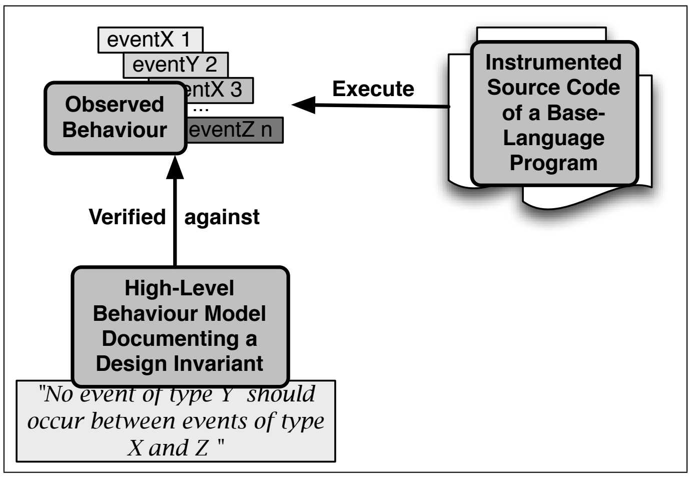

eventX 1
eventY 2
eventX 3
...
eventZ n
Observed Behaviour
Execute
Instrumented Source Code of a Base-Language Program
Verified against
High-Level Behaviour Model Documenting a Design Invariant
"No event of type Y should occur between events of type X and Z "

Figure 4.1: Lightweight verification of a behavioural model against actual program behaviour

recorded in an execution trace. As discussed in section 3.6, a dynamic analysis approach is efficient and precise, but lightweight in the sense that behavioural inconsistencies are found (for a particular program run) rather than that their absence is proven (as in heavyweight verifiers). Moreover, the careful selection of the components of a dynamic analysis approach allow for a goal-driven approach so that only the behaviour relevant to a particular design invariant is considered.

Figure 4.1 shows the lightweight verification set-up of our approach. The actual observed behaviour is obtained through executing instrumented source code, which, in addition to executing the program, records certain run-time events of interest. How the code instrumentation works is explained in section 4.4 in more detail. Automatic verification thus amounts to checking whether the events recorded in the execution trace exhibit the wanted behaviour as defined in the design invariant model. This implies that the verification results are always relative to the user input defined by the execution scenario. Although this only allows us to find invariant violations instead of proving their absence, a well-chosen execution scenario generally offers a convenient way to focus the verification on specific parts of a larger program.

4.2.2 Specifying Design Invariant Behaviour

After having briefly explained the idea behind the lightweight verification phase of our approach, we focus here on the most distinguishing ability of the proposed approach to document particular parts of a program's behaviour in high-level behaviour program models. As was made clear from the stated requirements in section 4.1, a behavioural formalism is needed to specify design invariant behaviour (Requirement 1). As this formalism should be used by developers to write system-specific design invariants, it should stimulate frequent use.

Specifying a Design Invariant in a High-Level Behavioural Model

As can be seen from figure 4.1, in the verification set-up the behavioural model of the design invariant is specified as a set of assertions in terms of the run-time events that are recorded during the application's execution. As was discussed in section 3.6, most verification approaches relying on dynamic analysis demand these assertions to be expressed over a fixed set of low-level run-time events (this is often realised through formulating an event grammar) that are directly related to programming language constructs. Examples of such low-level events are for instance calls to functions or assignments of variables. An assertion might then state that after a call to the function `f`, the value of the variable `var` has been adapted. Expressing program behaviour models using low-level concepts results in models which do not reveal any semantic information. Moreover, making small changes to the source code immediately results in adapting the behavioural model of the design invariant as well. Given the documentation setting and wanting to stimulate practical use by a developer making changes to the software, we want to document design invariants in high-level concepts that are of a high semantic level. For example, in a banking application the concepts of $money\_deposit$, $money\_withdrawal$ and $bank\_account\_status$ are frequently used. A possible assertion might state that after every money deposit to a certain bank account, the status of that account should have increased with the deposited amount. Such an assertion is decoupled from a particular implementation since it does not contain implementation-level constructs. Hence, the assertion itself does not need to be adapted when small changes are made to the software's implementation (Requirement 3).

The executable behavioural formalism in which assertions of wanted design invariant behaviour are specified should at least allow for composing higher level events from low level run-time events. This way these events can be decoupled from the actual source code. Going back to the banking application example, this already allows us to specify the high-level concept $money\_deposit$, referenced from assertions in a high-level behavioural program model, being implemented as a call to the concrete function actually performing the deposit. However, the values of low-level run-time events are fixed too. For example for a low level assignment statement only the assigned variable and its value are available. In case we want to know the value of another variable relevant to a certain bank account we would have to retrieve it from another low-level event. This would make the analysis of behaviour quite expensive, which would conflict with the lightweight requirement (requirement 3). For exactly that reason we opted for recording run-time events during the execution of an application to be also at a high level of abstraction.

Specifying High-Level Run-Time Events

Specifying run-time events also at a high level of abstraction allows us to specify the assertions representing the behaviour of the design invariant directly in terms of high-level entities instead of low-level implementation constructs. This allows us to create a partial model of actual program behaviour (instead of creating an entire model of behaviour) that only contains those events of interest for verifying the design invariant. This complies with the goal-driven requirement (Requirement 4) as specified in section 4.1. As the high-level concepts used in the run-time events represent invariant-specific concepts needed for specifying a behavioural invariant model, we leave it up to a developer for specifying them. To do so, a developer has to specify *when* such a high-level event takes place at run-time and also *how* to obtain any additional run-time information that is associated with such an event (see section 4.4). Another implication is that recording every single low-level run-time event is no longer necessary, which supports the lightweight and goal-driven requirement (Requirement 4). The resulting high-level execution trace consists in general of fewer run-time events which allows our approach to be applied on parts of a larger program as well. The executable behavioural formalism which is used for representing a model of program behaviour and its meta models representing the design invariant behaviour is discussed in more detail in the next section.

4.3 Reasoning About Program Behaviour

In section 4.2.2 our approach was discussed and the need for a high-level behavioural formalism to specify program behaviour was emphasised. This formalism should be highly expressive so that a behavioural specification can serve as documentation to make design invariants *explicit*. It should stimulate frequent use, so specifying design invariants in this formalism should not become impractical. And it should be a machine-executable specification language to be able to check design invariants automatically.

Before we continue the discourse about the behavioural formalism, we first mention how program behaviour is modelled.

4.3.1 A Partial Model of Program Behaviour

Before building a model of program behaviour by using a dynamic analysis approach, some considerations need to be taken into account. The first assumption we make is that the model is discrete, meaning that it consists of a finite set of well-separated elements. Using dynamic analysis, such an element or a *unit* of program behaviour is referred to as an *event*. The event is an abstraction for any detectable action performed during the execution of a program. Actions (or events) evolve in time and the program behaviour represents the temporal relationship between these actions. This implies the necessity of introducing an ordering relation for events.

As we mentioned in the previous section, in our approach we record run-time events during program execution which are, like the behavioural model of the design invariant, also at a higher level of abstraction. The event representation is depicted in figure 4.2. The ordering relation of events occurring at run-time is established by adding

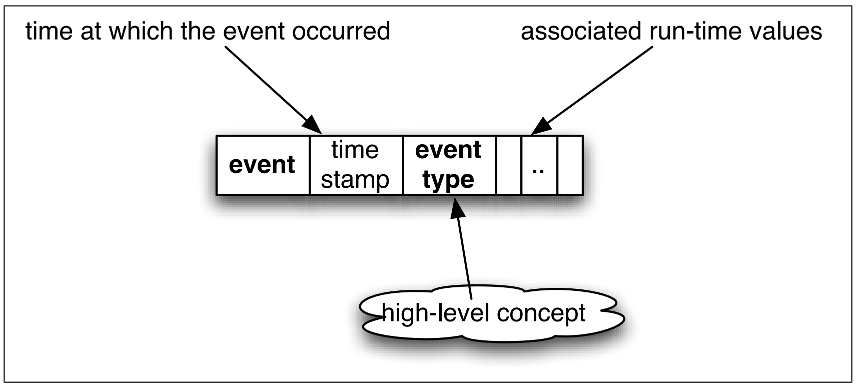


Figure 4.2: An event as a unit of program behaviour

a $timestamp$ to an event representation referring to the time at which the event occurred. The $event\_type$ represents a high-level concept referring to a description of an action performed during execution rather than a low-level implementation construct (in contrast to other dynamic analysis approaches). Going back to the banking example, $money\_withdrawal$ is an example of a high-level event type. In order to specify meaningful program behaviour properties, we enrich events with attributes, which contain static information and run-time values associated with the event. In figure 4.2 a variable number of associated values is depicted as they are freely chosen by a developer.

Program behaviour is then modelled as a finite collection of events, called the execution (or event) trace, which is actually a model of program's behaviour temporal aspect. It is a *partial* model as it only represents behaviour according to a well-chosen execution scenario (i.e. relevant to the design invariant under investigation).

4.3.2 An Executable Behavioural Formalism

The next issue to be addressed is the behavioural formalism for specifying design invariants, representing properties of program behaviour.

As the specification language should be highly expressive for documentation purposes, we advocate the use of a *declarative language* as behavioural formalism. This ensures that the resulting design invariant descriptions convey as much semantical information as possible. Declarative languages are very well suited to creating a model of program behaviour as such languages describe *what* information is needed rather than *how* it is obtained. On top of that, declarative languages support the following benefits in this context [JR00]:

- *Partiality*: A declarative model of a design invariant can easily express only the behaviour relevant to the invariant, and not the entire system model.

- *Incrementality*: A consequence of supporting partiality is incrementality. One can formulate a model as consisting of only essential properties, but later discover through analysis that other properties are needed. These can easily be added at a later stage.

- *Separation of Concerns*: A behavioural model can be organised so that distinct properties of the design (which at the implementation level would possibly intertwine) can be recorded separately. This exactly fits the purpose of design invariants being system wide entities which possible cross-cut an entire system.

On top of being declarative, the behavioural formalism should be machine-executable. In our approach we advocate a machine-executable logic programming language, with Prolog as the most well-known representative, to reason about program behaviour. In such programming languages, a program consists of logic clauses (or facts) representing knowledge about a particular problem at hand.

In the proposed approach, the logic program consists of a logic representation of the source code of a base language program (this will be discussed further in section 4.4.1) and an event trace of the base language program under investigation (as discussed in section 4.3.2). So the representation of an event as depicted in figure 4.2 is represented in a logic clause of the form `event(time_stamp,event_type(...))`.

The behavioural assertions documenting a design invariant which are expressed in terms of the run-time events from the execution trace, are expressed as logic formulas. To determine whether such a formula is a logical consequence, logic programming languages make use of a proof procedure.

While behavioural assertions about the events in the execution trace could be expressed in a regular Prolog-like logic language, we propose the use of an extended Prolog variant which is even more appropriate for reasoning about behaviour. This Prolog extension based on *temporal logic* is extremely suited to model the temporal relations between run-time events.

Temporal Logic Programming The logic formalism underlying these types of languages is called *temporal logic* and was already briefly discussed in section 2.5.2. For the sake of completeness we briefly repeat how temporal logic formulas are constructed. They comprise the classical logic formulas possibly qualified by temporal operators such as $\Box$ (always), $\diamond$ (sometime), $\bullet$ (previous) and $\circ$ (next). The truth value of a logic formula depends on an implicit temporal context: a formula can be true at a certain moment in time, while it might be false at the next moment (see section 2.5.2).

Different time models are supported in temporal logics. For example, time can be *bounded* or *unbounded*. Time is said to be unbounded to the future or to the past if every time is succeeded by a later or earlier time respectively. Either *time points* or *time intervals* can be chosen as primitives to reason over time. Time can also be *discrete*, *dense* or *continuous* depending on whether time is mapped onto the set of integers $\mathbb{Z}$, rational numbers $\mathbb{Q}$ or real numbers $\mathbb{R}$ respectively. *Linear* or *branching* time means that, in the case of considering time points as units of time, time is linear if the set of time points is totally ordered. In the case of a partial ordering of time points, time is said to be branching. For our approach, a discrete bound linear time model will suffice. In this model, informally, the temporal formula $\Box\phi$ is true if ϕ is true at all moments in time. Similarly, we have that $\diamond\phi$ is true when ϕ is true at some moment in time.

Temporal logic programming languages [OM94, Org94] are based on a subset of temporal logic such that programs written in this subset are machine-executable. MTL [Brz95] is a temporal logic programming language based on metric temporal logic. Metric temporal logics incorporate an additional quantitative aspect into the temporal operators. The $\diamond_t$ (sometimes within t time points) operator is for instance available. In this dissertation we propose the use of a variant of MTL to specify design invariants in a behavioural model in terms of high-level events.

While temporal logic formulas have been successfully applied in the program verification domain (see section 3.3), they are sometimes hard to understand. In our approach, regular logic programming can also be used for expressing design invariants as logical assertions. However, temporal operators allow the reasoning of temporal contexts in a very descriptive manner without having to explicitly manipulate the time stamps of events representing points in time. Behavioural program models expressed in regular Prolog are thus less descriptive, but they are still suited to support the specification of models at the conceptual level instead of an implementation level. This is mainly due to the fact that the (partial) model of program behaviour, i.e. the execution trace, is also represented at a high level of abstraction.

Another important advantage of our approach is the use of a temporal logic programming language as specification language instead of plain temporal logic. Formulas in temporal logic are sometimes very difficult to express [Hol02a, DAC99]. When using a temporal logic language, one can combine temporal operators in a higher-order rule so as to reuse them later. This way there is no need to remember their expression in plain temporal logic.

In the next section, we demonstrate how this model of program behaviour is coupled to the actual source code and how the actual behaviour is obtained *over* which the executable behavioural specification is verified.

4.4 Obtaining High-Level Events

In section 4.3.1 we illustrated how units of program behaviour are represented. To provide a coupling with the source code, a mapping is needed to couple an action occurring at run-time time with a description of a high-level event. And not only the type of event or action needs to be coupled to such a high-level description, but also the associated run-time values which provide *run-time context* for a particular type of event. Such a mapping of source code entities to high-level events is realised by using a dynamic analysis approach which adopts a sophisticated way of event selection through selectively instrumenting a base language program.

Figure 4.3 depicts the conceptual mapping from a base language program to high-

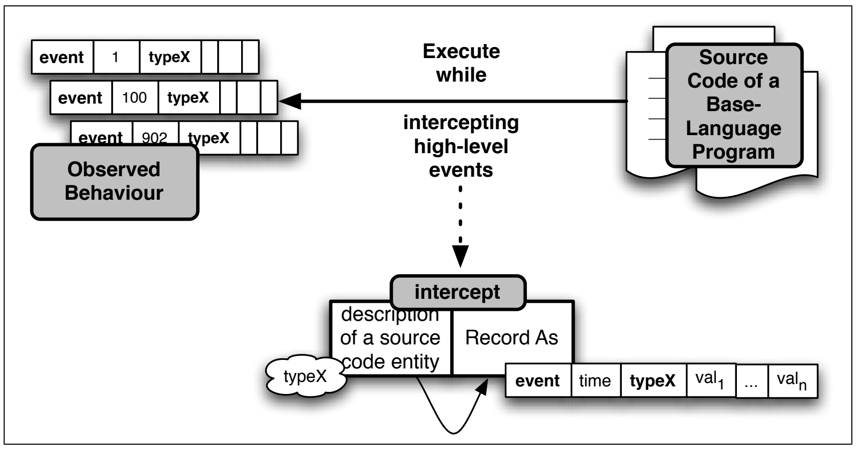

event 1 typeX
event 100 typeX
event 902 typeX
Observed Behaviour
Execute while
intercepting high-level events
Source Code of a Base-Language Program
intercept
description of a source code entity
Record As
typeX
event time typeX val1 ... valn

Figure 4.3: A Conceptual Representation of Mapping Source Code Entities to High-Level Events

level behavioural events. This figure resembles figure 4.1, only now we focus on *how to obtain* the high-level events instead of *how to analyse* them by formulating and verifying behavioural assertions.

The conceptual mapping is achieved by executing a base language program while *intercepting* high-level concepts in the source code and recording them as high-level events. As figure 4.3 demonstrates, intercept constructs are needed which on the one hand describe those application-specific constructs that lead to such a high-level event. On the other hand, a specification of the high-level event is needed denoting *how* they should be recorded and what *static and run-time values* are of interest for such a type of event.

To identify constructs in an application's source code that give rise to a particular high-level event of interest, an application's parse tree should be made available, as depicted in figure 4.4. A description for recognising a source code entity denoted $typeX$ can be regarded as a *condition* which is checked against all parse tree nodes of a base language program. If a concrete parse tree node satisfies the condition, then that particular node should be instrumented to make sure that at run-time, a high-level event is recorded when that particular node is visited. Instrumenting a concrete parse tree node is done either *before* or *after* that particular node. Base language code needs to be inserted which prints a textual representation of a logic clause (as was specified in section 4.3.2) to a file representing the event trace. In figure 4.4, base language code needs to be created for writing `'event(`$time$`,typeX(`$val_1, ..., val_n$`)).'` to a file.

In such a high-level event, associated values must be included as complementary information. In figure 4.4, in the representation of an event to be logged depicted there, the labels $time$, val_1 and val_n are examples of such run-time values. $val_1 ... val_n$ represent other values which can be freely specified by a developer, depending on what is of interest for that particular event. Static information as well as run-time

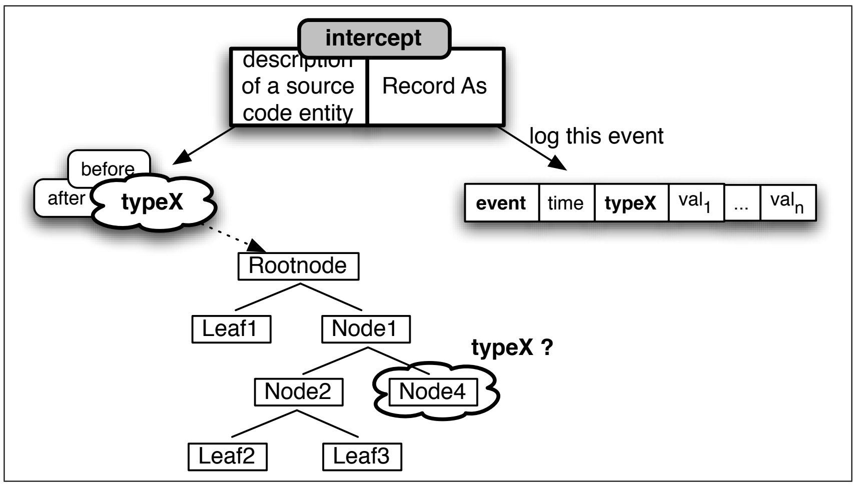


Figure 4.4: The *intercept* mechanism for identifying application-specific instances of high-level events

information might be included. As the mapping maps a particular parse tree node of the base language program to a high-level event specification, static information about that node can also be included. Run-time values are those associated values which evolve over time, such as $time$ denoting a time stamp. These values can only be obtained at run-time so they must be specified by the user in base language source code.

Logic Meta Programming (LMP) is a well suited approach for reasoning about the structure of a base language at a meta level. In our approach we advocate the use of LMP for writing declarative descriptions of source code entities which give rise to a high-level event that is of interest for a particular design invariant. Therefore we briefly explain LMP in the next section.

4.4.1 Logic Meta Programming

The main idea of logic meta programming (LMP) is to use a logic language to reason about and to manipulate the structure of programs written in some base language [Vol98]. The technique of LMP itself is not limited to a particular application domain nor to a particular base language and has been applied in many different situations that require meta programming in general. LMP has already been used to discover design patterns in both Java and Smalltalk programs [Wuy98, FM04], to check, enforce and search for programming patterns in code [MMW02], to check programming conventions [Mic98], to co-evolve design and implementation [Wuy01], to create aspect-oriented crosscuts[GB03], to document and check source-code regularities through intensional views and relations [MK06] and to generate code [Bri05].

The central concept of LMP is a mapping which associates every base-language program with a set of logic declarations which describe the program in sufficient detail

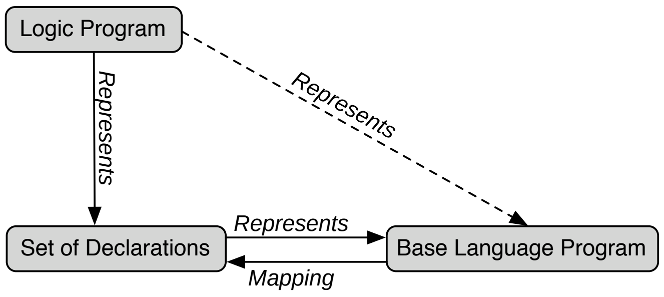


Figure 4.5: A logic meta programming setup (taken from [Vol98])

to be called a *representation* of it. Such a mapping is called *representative* if the base language program can be reconstructed from the set of declarations [Vol98]. In this set-up as depicted in figure 4.5, logic programs can be used to specify base language programs indirectly. A useful mapping describes the syntactic structure of the program in more detail. Representing the syntax of a program at some point is best done in a structured hierarchical form, like a program's parse tree. The mapping also depends on the aspects we are interested in for the particular application we have in mind.

4.4.2 A Means for Event Selection

Applying LMP for reasoning structurally about a base language program allows us to formulate declarative descriptions for identifying particular program parts of interest. Figure 4.6 demonstrates the use of LMP by showing the *intercept* mapping for the previously mentioned high-level concepts in a banking application. The description of the source code entity describes that after a $money\_deposit$ has been done, that we want to create a high-level event of type $after\_money\_deposit$. This event denotes the time (an ordering of the event with respect to other recorded events) and together with the type of event, shows the money which was deposited (the label $amount$) and the bank account status (val). Defining the $money\_deposit$ predicate then amounts to identifying the constructs in the application's source code that give rise to this high-level concept. The run-time values denoted by the labels also have to be specified. As these are values which can only be obtained at run-time, base language code needs to be provided by the user.

Any approach for selectively instrumenting source code constructs of interest is closely related to an aspect-like approach, based on the principles of Aspect Oriented Programming, as was described in section 3.6.3. To recall the used terminology within this context, a *pointcut expression* is a description of a source code construct of interest. In figure 4.4, the *condition* specified as $typeX$ relates to a pointcut expression which quantifies over all parse tree nodes. A *join point* is a concrete parse tree node which fulfils the pointcut description (making a pointcut a collection of joinpoints). *Advice* is code to be executed either *before* or *after* a particular join point (replacing code using *instead* is also possible, but for logging purposes this is not applicable). In this way,

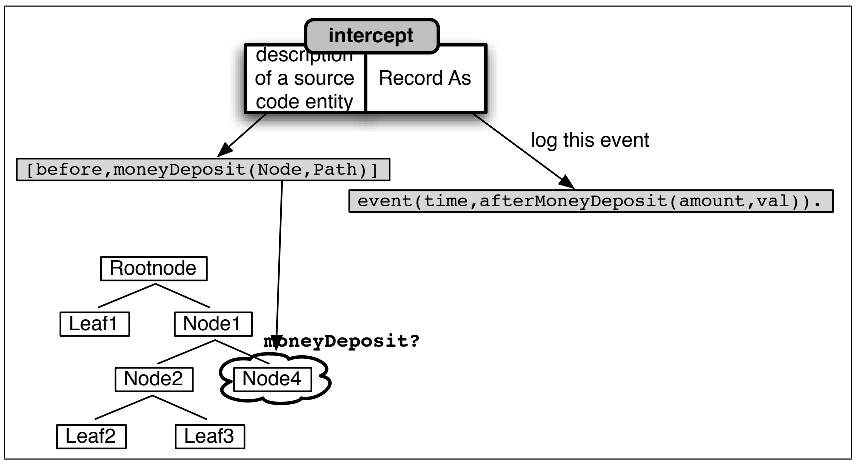

intercept
description of a source code entity
Record As
log this event
[before,moneyDeposit(Node,Path)]
event(time,afterMoneyDeposit(amount,val)).
Rootnode
Leaf1
Node1
moneyDeposit?
Node2
Node4
Leaf2
Leaf3

Figure 4.6: An example of the *intercept* mechanism using LMP

our approach uses a logging aspect in the sense that we only use logging code as advice code (written in the language of the base program) that records the needed high-level events to an execution trace.

In essence, for gathering the run-time events, any aspect approach could be adopted for selectively instrumenting a base language program, as long as high-level events are recorded instead of events based on low-level implementation constructs. However, the used pointcut language and the granularity of the parse tree representation (the chosen representational mapping) might possibly impose limitations on what conditions can be formulated.

Using LMP, we adopt a logic language as pointcut language in which we formulate declarative descriptions about particular source code constructs. This creates the extra advantage that, next to the recorded high-level events, also these pointcut descriptions are at a high level of abstraction. This also makes the specified *intercept* mappings reusable to apply them on other base language programs as well. Using a declarative logic language as pointcut language avoids a tight coupling with the source code [GB03].

In section 3.6, comparison criteria of dynamic analysis approaches were discussed. To summarise the instrumentation approach based on those criteria, in order to obtain run-time events we adopt an approach which:

- performs *instrumentation at the source code level*,
- adopts an *aspect-like approach* for selectively identifying joinpoints to add instrumentation code *before* or *after* those joinpoints,
- specifies behaviour at a *high level of abstraction* (see section 4.3).

4.5 A Four-Step Recipe for Applying the Proposed Approach

The lightweight verification approach presented in this chapter provides an expressive way to create descriptive executable models of design invariants underlying a software's implementation which are at the same time machine-verifiable. A large part of its strength can be attributed to the ability of freely determining the high-level events over which these behavioural assertions are expressed. However, as this requires more developer involvement, we present a four-step recipe to optimally take advantage of this approach. Figure 4.7 depicts these four steps.

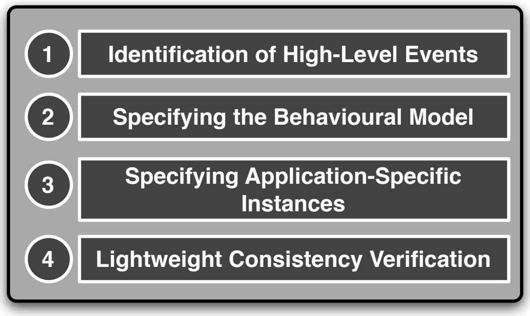

1
Identification of High-Level Events
2
Specifying the Behavioural Model
3
Specifying Application-Specific Instances
4
Lightweight Consistency Verification

Figure 4.7: A four-step recipe for applying the proposed approach

The specification phase of our approach comprises steps 1–3. The first step consists of identifying the high-level events for creating a model of program behaviour (as we explained in section 4.3.1). In a second step, the current understanding of design invariant behaviour is specified in a behavioural model expressing desired and unwanted behaviour. Temporal logic programming is used as behavioural specification medium, as introduced in section 4.3.2. In step 3, the mapping from application-specific source code entities to high-level events, which was clarified in section 4.4, needs to be specified. For the verification phase of our approach, a lightweight consistency check is performed by verifying the wanted or undesired design invariant behaviour against the actual program behaviour.

To demonstrate the practical use of this recipe, we refer to the next chapter where a concrete implementation of this approach is presented with C as base language and where this recipe is applied on a running example.

4.6 Summary

In this chapter we outlined our approach for the documentation and verification of design invariants. It is a lightweight declarative approach combining sophisticated selective code instrumentation with high-level behavioural analysis.

Section 4.1 emphasised the need for supporting design invariants. First the problem context for this dissertation was defined in section 4.1.1. Design invariants pose a problem for developing reliable software as they are implicitly present in software. Even if they are made explicit, they are detached from the source code so that discrepancies in either the invariant documentation or the source code need to be manually updated. Heavyweight program verifiers are able to verify design invariant behaviour, however they verify a model of a system rather than the source code itself, which does not fit the context of technically and algorithmically complex systems. Finally, very few analysis approaches offer a means to focus the analysis on only relevant behaviour to allow analysis on specific parts of a larger program as well.

Based on an evaluation of existing program analysis approaches, we defined in sections 4.1.2 – 4.1.5 a set of 4 main requirements which our approach must satisfy to offer support for documenting and verifying design invariants:

- The need for a descriptive behavioural specification language (Requirement 1),
- The need for a causal link with the source code to make the specification language machine-verifiable (Requirement 2),
- The need for a design invariant specification which is oblivious to the source code to support practical use (Requirement 3),
- A goal-driven and lightweight approach is advocated to allow focusing the analysis on specific parts of a program (Requirement 4).

We continued the discourse in section 4.2 where we presented a global overview of our approach for supporting design invariants. We linked the main solutions of the approach to the requirements that were previously identified in section 4.1. Consequently, the two main phases of our approach were gradually explained. In section 4.2.1 the verification phase of our approach was introduced, followed by the specification of behaviour in section 4.2.2, i.e. the most distinguishing feature of our approach.

In section 4.3 the behavioural formalism used for analysing program behaviour was presented. First, a partial high-level model of program behaviour was represented, followed by the executable behavioural formalism based on temporal logic which is advocated in our approach. Section 4.4 then pinpointed how the behaviour is obtained at run-time by adopting a sophisticated and selective means of instrumenting the source code. For exactly that purpose, the logic meta programming paradigm was introduced as a means to reason about the structure of a base language program. As more developer involvement is requirement in this approach, we propose in section 4.5 a four-step recipe on how our approach can be optimally exploited.

In the next chapter, a prototype implementation of the proposed approach for the programming language C called BEHAVE is presented. How this platform can be used for reasoning about behavioural design invariants is demonstrated extensively in chapter 6.

Chapter 5

BEHAVE: A Lightweight Verification Platform for C

In order to validate the feasibility of our proposed approach, we created a prototype platform named BEHAVE [1] that supports our approach of lightweight verification using high-level behavioural models of design invariants. BEHAVE is implemented in the logic language Prolog, and uses an aspect-like approach for instrumenting programs written in C. Modest tool support is available in Smalltalk to aid in setting up and using the experimental platform. The three main components of the platform are: a reification module containing a logic representation of a C base language program, the instrumentation module which generates selectively instrumented source code and a temporal logic meta interpreter which verifies the design invariant models against the actual program behaviour.

This chapter focuses mainly on the technical basis of our prototype. Its conceptual counterpart was discussed in the previous chapter, whereas the validation of BEHAVE by applying it onto a case study will be handled in the next chapter.

We start by pinpointing the history of BEHAVE so as to understand some of the design decisions made. In section 5.2 we elaborate on the declarative meta programming medium used for implementing our platform and we discuss logic programming in more detail. In section 5.3 we present BEHAVE as a dynamic analysis platform and discuss its main features conform chapter 3 for existing dynamic analysis approaches. An overview of the main constituents of the BEHAVE architecture is shown in detail in section 5.4. In section 5.5 we demonstrate how to optimally exploit the use of the BEHAVE platform by applying the four-step recipe as discussed in chapter 4. A simple stack implementation is used as case study for verifying basic invariant stack behaviour. Section 5.7 elaborates on the system overhead created by the BEHAVE platform. In section 5.6 we demonstrate available tool support. We end with conclusions in section 5.8.

[1]BEHAVE is not an acronym; it does unite several keywords that constitute our approach, although not in the right order: behavioural reasoning, high-level and lightweight automatic verification.

5.1 History of BEHAVE

Before presenting the actual system, we first present some background information about the history of the BEHAVE platform. This will help the reader in putting the system and its design into its proper context. It must be taken into account that this system has been created in the first place as an experimentation medium for implementing our previously outlined approach. Its main purpose is to use it as an environment in which a developer can reason flexibly about behavioural design invariants of programs written in Ansi C. By flexible we mean that on the one hand a decoupling of high-level events from implementation constructs enables reasoning about design invariants in terms of high-level concepts. On the other hand reuse is enabled by abstracting away previously defined concepts from a certain type of application.

The reason why we chose to reason about Ansi C programs as base language programs lies in the context in which this research was performed. The largest part of this research was conducted in the context of the four year ARRIBA project funded by the IWT, Flanders, Belgium [2]. ARRIBA stands for *Architectural Resources for the Restructuring and Integration of Business Applications*. The aim of this research project was to come up with tools and techniques to aid in understanding legacy applications. Because the Belgian companies involved in the project mainly have to deal with code bases containing programs written in imperative languages like C and COBOL, we finally opted to use Ansi C as our language of investigation because of the availability of test cases.

The initial idea for creating a dynamic analysis platform like BEHAVE was based on a concrete question from a software developer who was looking for a way to check certain behavioural invariants in his code that were very hard and time-consuming to verify manually. Involving the declarative meta programming set-up seemed very natural as this approach had already been used successfully in a wide range of application domains. This initial request resulted in a master thesis where the fundamental ideas underlying BEHAVE were implemented, based on the Zombie library [Gyb05].

An additional step was performed in the context of writing a paper about BEHAVE [RMG$^+$06]. Coen De Roover wrote a temporal logic meta interpreter on top of Prolog. The main contribution of this meta interpreter is to provide an extra layer of abstraction for representing actions performed at a certain moment in time during program execution.

To end this section, we would like to inform the reader that this system still is a prototype supporting our lightweight verification approach. In many parts of its implementation there is room for improvement with regard to performance, robustness, etc .... One technical problem we experienced was the implementation of the C language parser to be able to represent a base language program as a set of logical propositions. The difficulty mainly resides in the way C allows macro's to be defined. When

[2]Instituut voor de Aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen - http://www.iwt.be

macro's are immediately expanded in the code before parsing, valid C statements are obtained which can be parsed without problems. However, when you want to reason *about* macro's, you cannot expand them as they have to be represented in the meta model. We chose the latter as we wanted to reason about macro's as source code entities. In this version of BEHAVE we are only able to parse macro calls which appear as function calls (i.e. which look like a valid C expression without expanding them). However, these small deficits do not compromise the strength of our approach.

5.2 Logic Meta Programming

In section 4.4.1 we presented the logic meta programming paradigm (LMP) which is used to reason about the structure of a base language program. In this section we demonstrate the use of LMP for the BEHAVE platform to reason about the structure of C programs. This implies the identification of a representational mapping which reifies the structure of a C program into a logical representation (this is demonstrated in section 5.2.5).

As logic programming is adopted at a meta level to reason about base programs, we first introduce in the following subsections the logic programming paradigm together with its main characteristics. We limit ourselves to presenting some general ideas and a few representative queries. For an in-depth overview we refer the reader to [Fla94].

5.2.1 Logic Programming

The main idea behind logic programming is the use of a subset of logic as a programming language. In the well-known logic language Prolog, the knowledge about a problem is captured in a set of logical axioms in the form of Horn clauses [Fla94]. The execution mechanism of the language, called *SLD-resolution*, is used to *prove* that a *query* or a *goal clause* is a logical consequence of the program. The resolution mechanism which is based on unification and backtracking is a very powerful mechanism that allows for recursion to be used in logic programs. This classifies Prolog as a powerful programming language and distinguishes it from knowledge representation systems.

One of the essential characteristics of logic programming languages is their declarative semantics. In contrast to imperative languages, a logic program denotes *what* a program does rather than *how* it should be executed. The meaning of a given declaration in a logic programming language can be concisely determined from the statement itself. Imperative semantics however, demand more the specification of control flow and is therefore rather complex. For example, the semantics of a simple assignment statement requires at least the examination of local declarations and knowledge of the scoping rules of the language.

Examples and Terminology

For clarity on terminology we describe here how parts of a logic program will be denoted throughout this dissertation. In Prolog, we refer to *logic declarations* rather than program statements and expressions. Any logic program consists of multiple logic declarations and they can be either a *fact* or a *rule*. As an example, consider the following logic database of a small family tree:

```
% A simple family tree example
father(albert,jeffrey).
mother(alice,jeffrey).
father(albert,george).
mother(alice,george).
father(george,cindy).
mother(mary,cindy).
```

The '%' sign denotes a comment line in Prolog. This database contains six logic *facts* which declare that `albert` and `alice` are the father and mother of two children, `jeffrey` and `george`. `george` is the father of one child, `cindy`. The following logic *rules* can be added to the database to retrieve additional information:

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

In Prolog, logic variables like `X` and `Y` are denoted with a capital letter. The `parent` rule is implemented by two logic declarations. They state that `X` is a parent of `Y` if `X` is either the father or mother of `Y`. The last added rule defines the `grandparent` relationship. The following logic query is launched to retrieve all grandparents from the database:

```
?- grandparent(X,Y).
```

This query yields two results `X = albert, Y = cindy` and `X = alice, Y = cindy` which states that both `albert` and `alice` are the grandparents of cindy.

The logic program below presents a more advanced logic rule, i.e. a `member` predicate which defines whether an element is a member in a list. The first declaration denotes a fact which states that the first element in a list is always a member of that list. The second logic declaration denotes a logic rule that states that a random element is part of a list if it is a member of the rest list. The symbol '_' is used to refer to any possible value, i.e. a *not-named variable*.

```
member(First,[First|Rest]).

member(Element,[_|Rest]) :- member(Element, Rest).
```

We also refer to parts of a logic declaration as *logic terms* and *logic clauses*. A term refers to that part of a logic declaration that is manipulated as data, e.g. the

[First|Rest] list used in the above example. A clause denotes an entire logic declaration that is associated with a truth value. The predicate of a clause is uniquely determined by a name and multiplicity. The multiplicity is a number that denotes the number of arguments associated with that predicate. The member/2 predicate above has a name member and a multiplicity 2.

Terms are always contained in clauses and can also be *compound*, i.e. a term can have arguments each of which may again be a term. A compound term is also called a *functor*. A term without arguments is called an atom (for example the name 'george' from the family tree mentioned above is an atom). The nesting of terms into *compound terms* or into a *functor* might be used for providing extra information about a term. In the family tree example denoted above, we might for example use the functors male(X) or female(X) as compound terms inside the predicates mother and father. As such it can be denoted whether a person is either male or female. The following fact of the logic database would then reveal that albert not only is the father of jeffrey, but that he has a *son called jeffrey*, which would be written as follows:

```
father(albert,male(jeffrey)).
```

An essential part of logic programming is that multiple logic declarations (i.e. multiple facts and rules) can define a single predicate. Multiple logic declarations express multiple alternative computations to solve a query. In the example above, the member/2 predicate contains two declarations in the form of a fact and a rule. The fact is applicable when an element is the first element in a list. The rule is applied when it is not. The query described below will first invoke the fact and afterwards the rule. The result returned for Element represents one by one all elements in the list [1,2,3].

```
?- member(Element, [1, 2, 3]).

Element = 1;
Element = 2;
Element = 3;
```

Below we describe another example of the predicate append/3 that appends two lists. The predicate is again described by two logic declarations: a fact and a rule. The logic fact is applicable when the first argument of the predicate is an empty list. It states that, when appending an empty list to any list, that list is the result of the append. When the first argument of the predicate is not the empty list, the rule of the append predicate will be applied instead.

```
append([],List,List).

append([First|Rest],List,[First|RestList]) :-
   append(Rest,List,RestList).
```

The append rule states that any two lists [First|Rest] and List can be appended with as result the list [First|RestList] if and only if the append of Rest and List yields the result RestList.

The example query launched below will invoke the `append` predicate. The result returned is `List = [1,2,3,4,5]`.

```
?- append([1,2,3],[4,5],List).
```

Main Characteristics of Logic Programming

A logic language like Prolog is a dynamically-typed declarative language which offers an abundance of qualities that are used in the context of this dissertation:

- Declarative languages like Prolog focus on *what* needs to be executed instead of *how* we should retrieve something. As a consequence, programs written in a declarative language are easier to understand because they are closer to the semantics of a program. Furthermore, these languages make it possible to reason at a higher level of abstraction, i.e. reasoning is done at the domain level instead of the implementation level.

- The mechanisms underlying logic languages, such as *unification* and *backtracking*, provide a very powerful reasoning engine. The process of *unification* can be described as that of finding appropriate values for variables in order to make two terms the same. As terms can be compound, unification provides a *deep matching* algorithm and can also be seen as trying to make two tree structures the same. *Backtracking* is a powerful search mechanism that enables to find all possible solutions for a particular query.

- Logic programming also allows for the specification of *recursive* logic rules because of these logic mechanisms. This is the main reason why logic programming is much more powerful than knowledge representation mechanisms. Such languages created for retrieving data (e.g. SQL) do not support recursion (an example of the use of recursion will be shown in section 5.4.4).

- Logic programming is multi-way, i.e. one predicate expresses multiple relationships. For example consider the mathematical equation $X + Y = Z$. In a logic programming language the predicate `addition(X,Y,Z)` would be created which can be used to perform three different tasks:

  1. calculating the sum of two numbers,
  2. calculating the difference between two numbers ($Y = Z - X$),
  3. checking whether the relationship $X + Y = Z$ holds for a given X, Y and Z.

- Prolog in particular is an interactive programming environment; a program in Prolog is represented by a set of logic declarations and you can query that program by means of launching queries. For our BEHAVE platform, where we create an execution trace of high-level run-time events in the form of logical facts, you obtain an analysis language for reasoning about the actual behaviour of a program *for free*.

5.2.2 Frequently Used Predicates

Prolog as a programming environment provides a large library of pre-defined standard logic predicates that are used very often when writing logic programs. To get an even better feeling for the logic programming paradigm and to prepare the reader for further investigation of the BEHAVE platform we will provide here some important logical predicates which will be needed later on in examples.

The `findall/3` predicate The `findall` predicate is a *second order predicate* because the second argument is a logical query. After this query is launched, all results of that query are collected in a list which is the last argument of the predicate. The first argument then determines which value is put in the result list for each of the query results. As an example, the following query returns for `List` the value `[1,2,3]`:

```
?- findall(Element,member(Element,[1,2,3]),List).
```

The `forall/2` predicate Another second order predicate is the `forall` predicate. The first argument is also a logical query, whereas the second argument represents a condition. For all alternative bindings found by launching the logical query, they must satisfy the condition. The query returns true or false, however in case the query fails, it does not tell you which of the alternative bindings did not satisfy the second argument condition. The following query returns true:

```
?- forall(member(Element,[1,2,3]),Element > 0).
```

The `foreach/2` predicate A self-defined meta-predicate predicate which we will use in the following example is the foreach predicate. It is defined as follows making use of the Prolog meta predicate `call/2` and of our previously explained `forall/2` predicate:

```
foreach(List,Predicate) :-
    is_list(List),
    forall(member(Item,List),call(Predicate,Item)).
```

The Prolog `call/n` predicate launches the logical query `Predicate(Item)`. Note the `n` multiplicity of this predicate; it can append a variable number of arguments to the predicate. Launching the query `:- call(member(X),[1,2])` is the same as launching `:- member(X,[1,2])`. The `foreach/2` predicate appends every element in a list one by one to the arguments of the predicate and launches that new predicate as a query. We could use the predicate as follows, to check whether all numbers in a list are integers:

```
?- foreach([1,2,5.32],integer).
```

This query returns *false* because `integer(5.32)` will return *false*.

5.2.3 Smalltalk Open Unification Language (SOUL)

The different application areas mentioned in section 4.4.1 in which LMP has been successfully applied did not use Prolog but rather a Prolog derivative called SOUL [sou]. SOUL is a logic meta-language implemented on top of Smalltalk that reasons about Smalltalk programs. It was initially developed by Roel Wuyts in the context of his PhD dissertation [Wuy01]. Although SOUL has a slightly different syntax, it is entirely based on Prolog and can execute regular Prolog programs. SOUL is however much more powerful than a conventional Prolog derivative because SOUL can also contain, next to ordinary Prolog programs, Smalltalk expressions which realises a tight integration with its base language Smalltalk. The BEHAVE platform was first implemented in SOUL. This is still visible in that the Ansi C parser is implemented in Smalltalk using the SmaCC parser generator. However, to really benefit from the powerful symbiosis between SOUL and Smalltalk, SOUL should be used to reason about Smalltalk as a base language. Since for our BEHAVE platform we don't reason about Smalltalk programs, we opted to use Prolog since we only would use the Prolog features present in SOUL. To demonstrate briefly the slightly different syntax of both logic languages, you can find below the SOUL logic declarations that implement the `member/2` predicate as specified above in Prolog:

```
member(?first,<?first|?rest>).

member(?element,<?|?rest) if
   member(?element,?rest).
```

A variable is denoted with a question mark `?variable` as opposed to the first letter being a capital letter in Prolog (`Variable`). A list is written as `<1,2>` instead of the square brackets and the head and body separator for a logic rule `:-` is replaced by an `'if'` in SOUL.

5.2.4 Temporal Logic Programming

In chapter 4 we introduced the concept of temporal logic programming as executable behavioural formalism and we discussed temporal logic and the operators such a logic uses to reason about time structures (see section 4.3.2).

In this section we demonstrate the *use* of the temporal operators for our temporal logic meta interpreter based on MTL. Consider the simple logic database in figure 5.1. The database holds five Prolog facts and one rule of a `salary` predicate. They represent the salary of two people, `john` and *bert*. Their salary has changed over the years, so a time stamp denotes the *time context* of what salary they have received at what moment

```
salary(-1, john, 500).
salary(0, john, 1000).
salary(1, john, 1500).
salary(2, john, 2000).
salary(3, john, 3000).
salary(bert, X) :- salary(john, X).
```

Figure 5.1: A simple logic database

in time. The present is denoted by '0'. Using temporal logic programming (based on MTL), the temporal operators are used to *abstract over* the time stamp denoted here as first argument of the `salary` predicate. They denote in an abstract way the *order* of the predicates without using the actual time stamp. For example, launching the following query yields `S=1500` :

```
?- next(salary(john,S)).        %% ∘salary(john,S).
```

The expression denoted in comments represents the same query but specified with the temporal operator notation. This query searches for the salary of `john` at the *next* point in time from now. So the salary of `johan` at time '1' is returned. The next query looks for the salary of `john` at *sometime in the future within 2 time steps* from now.

```
?- sometime(2,salary(john,S)).  %% ◇²salary(john,S).
```

As the time step is a positive number, it reveals that sometime in the *future* is meant. A negative number can be used to refer to a time interval in the past until now. Three results `S=1000; S=1500; S=2000` are returned which denote the salaries `john` had in the time interval from 0(now) to 2.

With respect to our reuse requirement identified in chapter 3, temporal operators allow abstraction of temporal contexts in a very descriptive manner without having to explicitly manipulate integers representing points in time. By combining temporal operators into reusable higher-order logic rules, often recurring temporal patterns can be expressed without having to remember their idiomatic expression in plain temporal logic. Examples are various occurrences and ordering patterns which were identified as appearing most often in specifications for verification systems [DAC99].

5.2.5 The Representational Mapping

The central concept of logic meta programming is a mapping which associates every base-language program with a set of logic declarations which describe the program in sufficient detail to be called a *representation* of it. Such a mapping is called *representative* if the base language program can be reconstructed from the set of declarations [Vol98]. In this setup as depicted in figure 5.2, logic programs can be used to specify base language programs indirectly. A useful mapping describes the syntactic structure of the program in more detail. Representing the syntax of a program at some point is best done in a structured hierarchical form, like for example a program's parse tree. The mapping also depends on the aspects we are interested in for the particular application we have in mind.

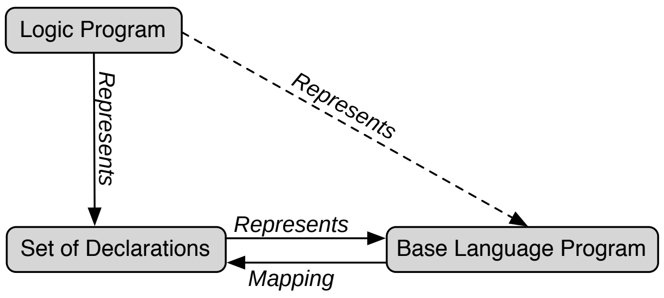


Figure 5.2: A logic meta programming setup (taken from [Vol98])

In this dissertation we use LMP in our setup for building a lightweight verification platform. Contrary to the other LMP applications as mentioned in section 4.4.1, our platform is based on dynamic analysis also to reason about the *behaviour* of a system. Logic meta programming is used as a means to describe program points of interest for selectively instrumenting a program. Moreover, we chose as our base language the procedural language C.

Representing C Programs

Using LMP to reason about and to manipulate C base programs requires a representational mapping from a procedural programming language to the logic meta level representation. This mapping determines the parts of a program that are represented as logic facts. We refer to this process as *reifying* the structure of the base language into separate logic declarations.

The choice of the mapping in a logic meta programming system forms a crucial part of the system's design. It defines what aspects of the base language are made explicit and can be manipulated and reasoned about at the meta level. Choosing the mapping also depends on the level of granularity we want to instrument the code at. Similar to other instrumentation approaches, the internal representation of a base language program is represented as a parse tree of the program.

Table 5.1 describes the representational mapping of a subset of C program elements onto their internal representations. It is based on a C grammar defined for the Ansi C standard [Deg]. The main difference is the extra first argument for a declaration and a function definition (i.e. the file name to which they belong). This mapping directly *reifies* function definitions, function declarations, macro definitions and macro declarations into separate logic facts:

- A *function definition* is represented by a logic fact that declares the file it belongs to, the return type of the function, the function's name, the list of parameters and

C Construct	Logic Representation
Function definition	`functionDefinition(FileName,Ret,Nam,Pars,Body).`
Declaration	`declaration(FileName,DeclSpecifs,InitDecLst).`
Macro definition	`macroDefinition(FileName,Name,Value).`
Function macro	`functionMacro(FileName,Name,Parameters,Body).`
C file	`baseFile(FileName).`
Header file	`includedFile(FileName).`
C project path	`projectPath(PathString).`
User include	`userInclude(FileName,IncludedFileName).`
System include	`systemInclude(FileName,SystemFileName).`

Table 5.1: The representational mapping for reasoning about C programs

the body of the function. Note that the body of a function consists again of a logic fact (or a functor) which contains a parse tree representation up to the statement level.

- A *declaration* is represented by a logic fact declaring the file it belongs to, the declared type and the declared function or variable. Note that a C declaration can contain many declarations of the same type, declared with an initial value or not. So the third argument contains an initialise declarator list.

- *Macro definitions and functions* are represented by a logic fact which again holds the filename and the macro's name and value (in the case of a macro function the parameters are included as well). Note that as we will not reason about the internals of macro definitions, we represent the macro definition's value (or body) as a string.

The five logic facts in the lower part of table 5.1 represent meta information about different files in a C project under investigation. The header files and base files that constitute a project are represented by separate logic facts. Also the path of the project is represented as a separate logic fact. Finally, the `include` statements of a C file also have a separate representation.

Although table 5.1 represents the representational mapping into separate logic facts, the deeper parse tree structure is also reified into functors representing different parse tree nodes. For example, figure 5.3 depicts a tree representation of a function definition as specified on row 1 in table 5.1. The dark-grey boxes denote tree leafs (representing a term), while the white boxes represent a functor [3]. As can be seen from figure 5.3, the body of a function is represented by a `compoundStatement` functor which holds a list of declarations and a list of program statements. In this declaration list, temporary variables to be used within the body of the function are declared. The other lists

[3]Declaration specifiers are coloured in light-grey. They almost represent a tree leaf, except when the type specifier is either a `struct`, `union` or `enum` specifier. In that case one extra functor is needed to represent this type of specifier.

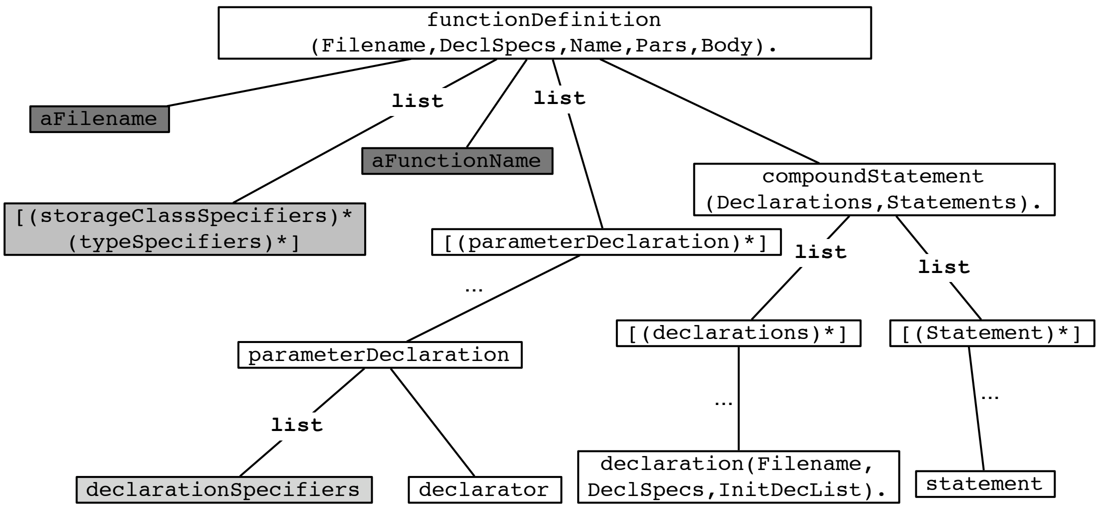


Figure 5.3: Tree representation of a C function definition

holds all statements of the function's body. Each statement on its own is represented by another functor depending on the type of statement that is used.

The return type of a function is captured by a *declaration specifier*. Such a specifier holds a list of both (possibly multiple) storage class specifiers and type specifiers. An example of a storage class specifier is `static`. An example of a function definition and a declaration will be shown in section 5.4.2. The full representational mapping is listed in appendix A.

The next section introduces BEHAVE as a dynamic analysis platform which implements the approach outlined in chapter 4.

5.3 BEHAVE: A Dynamic Analysis Platform

BEHAVE is a dynamic analysis platform that supports the documentation and lightweight verification of design invariants in C according to the proposed goal-driven approach explained in the previous chapter.

Figure 5.4 depicts the *process* of using BEHAVE as a dynamic analysis platform. The core implementation of BEHAVE is depicted in the large database of figure 5.4 comprising 5 main logic layers. The temporal logic meta interpreter is depicted separately as it is a standalone platform component providing an extra layer of abstraction for analysing program behaviour. The implementation of these five layers is discussed in section 5.4.

In section 3.6.1, we discussed the main distinctive features of existing dynamic analysis approaches: the means of selecting program parts of interest (1), specification of program behaviour (2), collecting run-time events (3) and analysing program behaviour (4).

In this section, we describe BEHAVE with respect to these features:

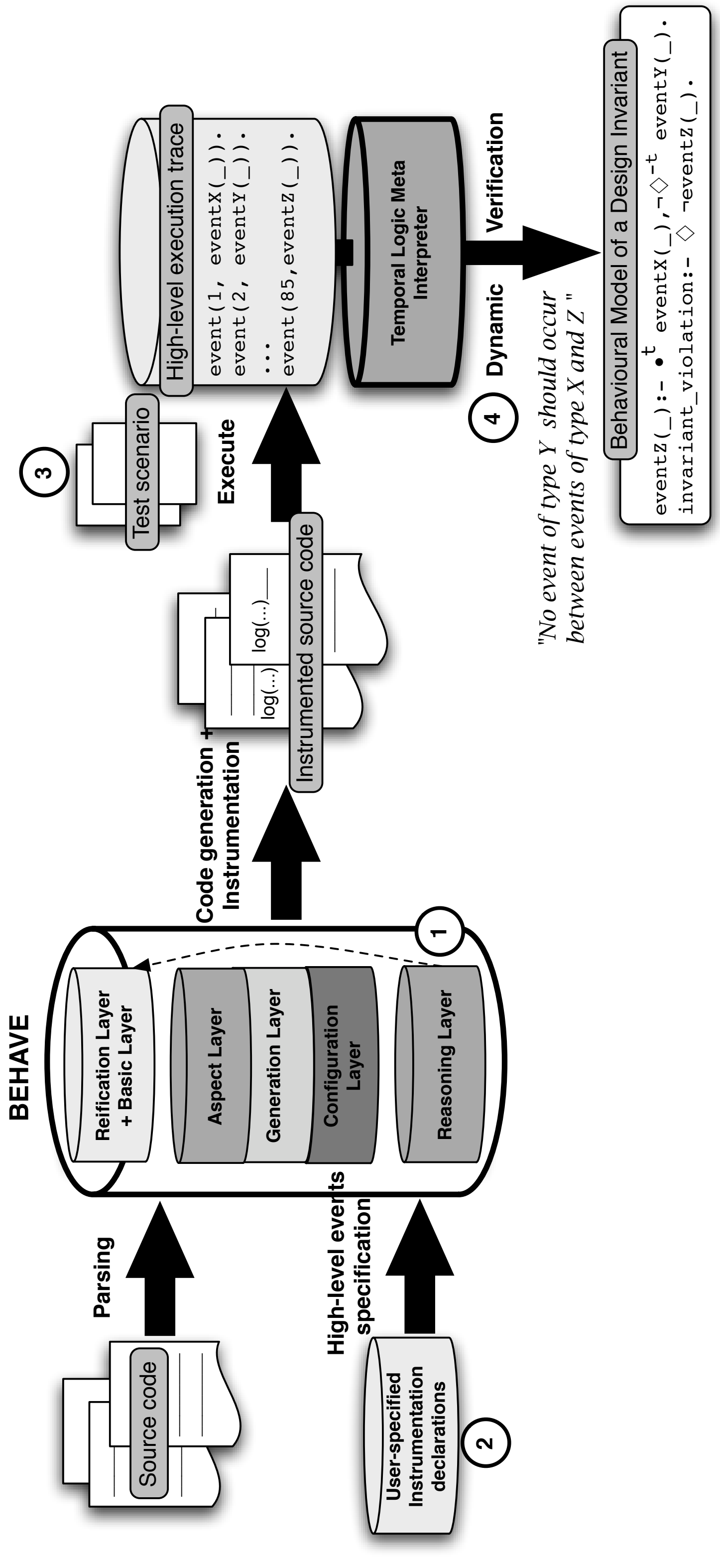

Source code
Parsing
2
User-specified Instrumentation declarations
High-level events specification
BEHAVE
Reification Layer + Basic Layer
Aspect Layer
Generation Layer
Configuration Layer
Reasoning Layer
1
Code generation + Instrumentation
Instrumented source code
log(...)
log(...)
Execute
3
Test scenario
High-level execution trace
event(1, eventX(_)).
event(2, eventY(_)).
...
event(85,eventZ(_)).
Temporal Logic Meta Interpreter
4
Dynamic
Verification
"No event of type Y should occur between events of type X and Z"
Behavioural Model of a Design Invariant
eventZ(_):- ●^t eventX(_),¬◇^-t eventY(_).
invariant_violation:- ◇ ¬eventZ(_).

Figure 5.4: Using BEHAVE as a dynamic analysis platform

1. To create an instrumented base language program, program parts of interest are first selected by describing them in a logic rule in terms of a logic representation of the source code (which is defined by the representational mapping as explained in section 5.2.5).

2. Program behaviour is specified by letting the user specify a mapping to map those program parts of interest (as described in step 1) to a logic fact describing a run-time event.

3. Run-time information is collected in an execution trace by executing the generated instrumented base language program according to a particular execution scenario.

4. Analysing program behaviour consists of specifying a behavioural model (as temporal assertions) representing unwanted behaviour of a particular design invariant and checking the consistency of that model against the collected run-time events. Analysis is performed off-line.

Each of these phases are denoted by their corresponding number in figure 5.4. They are explained below in more detail.

1. Selecting Program Parts of Interest As mentioned in section 4.4.1, BEHAVE adopts a logic meta programming approach for representing the structure of C programs. Therefore, as shown in figure 5.4, the source code of the application under investigation is parsed and reified in a logic representation (in the reification layer) according to the representational mapping defined in section 5.2.5. The reification layer thus holds a logic meta model of a base language program. Program parts of interest are then described by a logic declaration in terms of the logic facts from the meta model (these declarations are stored in the reasoning layer, denoted by 1 in figure 5.4).

Note that these logic declarations describing particular program points of interest are used as *constraints* or *pointcut expressions* (see terminology as defined in section 3.6.3) that are unified (or not) against logic parse tree nodes (as represented in the logic meta model). Note the advantage of using a logic language as pointcut language: the pointcut description can be expressed at a high level of abstraction which avoids a tight coupling with low-level implementation constructs [GB03].

2. Specifying Program Behaviour For specifying program behaviour, we let a user of the platform specify the descriptions of run-time events (and associated values) that are of interest for reasoning about a particular behavioural design invariant. Consequently, a mapping is specified which maps the defined program parts of interest (as explained in the previous step defined above) to the logic descriptions of run-time events (see section 4.4.2). This is depicted by 2 in figure 5.4.

A user can specify the constituents of these run-time events at a conceptual level instead of low-level implementation constructs. Adopting such an approach lets one specify a model of wanted behaviour directly in terms of *high-level* events, thereby making the model oblivious from implementation constructs. This supports program development as making small changes to the source code does not influence the behavioural model directly.

3. Collecting run-time events Next, an instrumented program is generated. BEHAVE generates C source code from the logic representation captured in the reification layer, while trying to unify every element from that logic representation with the pointcut descriptions defined in the mapping (as explained in steps 1 and 2). In case unification succeeds, next to generating source code for that particular element, instrumentation code is generated as well.

To gather the run-time information about the program under investigation, the generated instrumented program is executed along a well-defined execution scenario (see number 3 on figure 5.4). As a result, an execution trace is created which contains the high-level events as described in the mapping from the previous step (only now with concrete run-time values). The recorded execution trace presents a partial behavioural meta model representing only the behaviour which was specified in the mapping by the user, and only that behaviour triggered by the chosen execution scenario.

4. Support for Analysing Program Behaviour As discussed in the previous step, a partial behavioural logic meta model is created represented by the recorded execution trace containing high-level events. For analysing this program behaviour, undesired design invariant behaviour is specified by the user in a high-level behavioural model in terms of these high-level events. As explained in section 4.3.2, we use temporal logic programming to specify this model. We could also use plain logic programming, however temporal logic programming lets us use temporal operators which allow extra abstractions over time structures.

Figure 5.4 depicts the lightweight verification of a behavioural model representing a particular behavioural design invariant (depicted by 4). Lightweight verification is done by checking the consistency of the behavioural model representing undesired behaviour against the high-level execution trace.

In section 5.4 we explain more into detail what logical declarations each of these layers contain and how the declarations collaborate in achieving selective program instrumentation.

5.4 The BEHAVE Architecture

The BEHAVE platform is entirely implemented in Prolog, and more particularly in SWI-prolog [Wie07] . In the previous section we have shown how to instantiate the proposed technologies from chapter 4 for building a dynamic analysis tool. In this section we present the different components of the behave architecture and we zoom in on the technicalities of the different logic layers.

5.4.1 An Overview

Figure 5.5 shows the overall architecture of the BEHAVE platform. For a particular program under investigation, the user of the BEHAVE platform needs to provide the following information:

- An executable version of the C source code, together with a makefile and a configure file to be able to run the instrumented code afterwards. A suitable execution scenario also needs to be chosen, which depends on the kind of behavioural design invariant that needs to be verified,
- The user of the platform has to specify the description of the high-level events. On top of that, a behavioural model of the design invariant under investigation needs to be specified in terms of these high-level events.

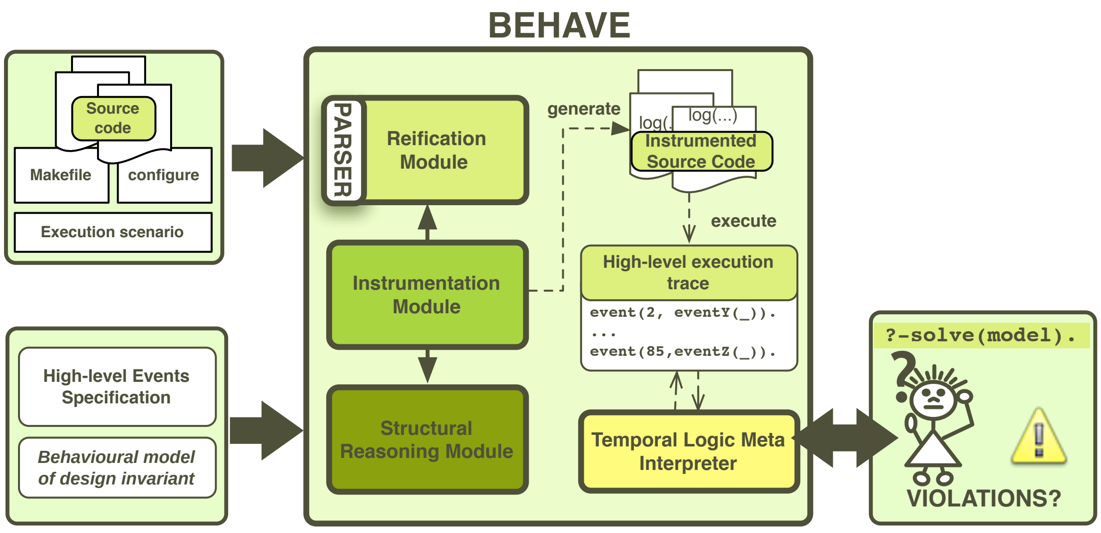


Figure 5.5: The BEHAVE Architecture

As can be seen from figure 5.5, the BEHAVE architecture consists of four main components. The first component is the reification module which, after parsing the code, contains a logic meta model of the program under investigation. The second component, the instrumentation module, takes care of re-generating the source code from the logical representation while inserting instrumentation code at specific program points defined by the user. The third component contains more advanced structural reasoning declarations, both application-specific and declarations applicable to any C program. The fourth component is a meta interpreter built on top of Prolog which provides a highly expressive way to analyse the acquired program behaviour according to a particular behavioural design invariant.

In the remainder of this section we zoom in on each of the modules that constitute the BEHAVE platform so as to gain some insight in how the core of the BEHAVE system works.

A running example: A simple stack implementation To make the logical representations of the different layers as clear as possible, we use a simple stack implementation depicted in figure 5.6 as a C base language program under investigation throughout the remainder of this chapter.

The stack implementation represents a C project consisting of two files `Stack.c` and `Stack.h`[4]. Considering the source code in figure 5.6 the running example contains four declarations, a function macro and two function definitions.

```
1  int *stack;
2  int top;
3  static void init(int);
4  static void push(int);

5  #define pop() stack[--top];

6  static void init(int size){
7     top = 0;
8     stack = malloc(size * sizeof(int));
9  }

10 static void push(int element){
11    stack[top++]=element;
12 }
```

Figure 5.6: Simple C Stack implementation - Stack.h

5.4.2 The Reification Module

As previously explained in our proposed approach in section 4.4.1, BEHAVE adopts a logic meta programming approach. In this setup, the structure of the base language is reified into a set of different logic facts according to the representational mapping defined in section 5.2.5 and depicted in table 5.1. These logic facts are part of the *reification layer* of BEHAVE. This layer, together with the *basic layer*, forms the reification module.

5.4.2.1 The Reification Layer

To get a feel for the logic representation of a C base language program, we present some logic facts by parsing the C stack example (from figure 5.6) according to the representational mapping. Figure 5.7 shows the logic representation of the definition of the `push` function as defined on lines 10–12 in figure 5.6.

A logic fact representing a C function definition consists of five arguments that completely represent a function definition in Ansi C. The first argument represents the name of the file which the function is being defined in. The second argument denotes

[4]We only included the header file here as the c file contains only the `main()` that executes some of the stack's functions and we do not need to refer to it in the course of this chapter.

```
functionDefinition(
    "stack.h",
    [static,void],
    push,
    [parameterDeclaration([int],declarator(optional,identifier(element)))],
    compoundStatement([],[assignment(arrayAccess(identifier(stack),
                                                 postfixIncrement(identifier(top))),
                                     assignmentOperator,
                                     identifier(element))])).
```

Figure 5.7: Logic representation of the `push` function depicted in figure 5.6

a list which captures the function's return type. This term contains a list because for some definitions in C a *storage class specifier* and/or a *type qualifier* [5] might be inserted before the return type. The name of the defined function is captured by the third argument. A list of parameter declarations is held by the fourth argument, which captures one declaration of type `int` and a declarator that has as name the identifier `element`. The optional argument in the declarator functor is used to hold a pointer.

```
1 declaration("stack.h",
2             [int],
3             [initDeclarator(declarator(pointer([],optional),identifier(stack)),
4                             optional)]).

5 baseFile("stack.c").

6 projectPath("/Users/imichiel/Desktop/BEHAVE/Stack-example/").

7 functionMacro("stack.h","pop",[],"stack[--top];").
```

Figure 5.8: Logic representation examples for the stack example depicted in figure 5.6

The logic fact denoted in figure 5.8 on lines 1–4 represents the declaration of the global stack variable as a pointer to an integer (as depicted in figure 5.6). The pointer in the declaration is being represented as a pointer functor. The empty list in this functor can hold also a type qualifier such as `'const'`, the `optional` argument can again hold a pointer. The optional argument of the initDeclarator can contain an initializer which gives an initial value to a declared variable. The `baseFile` and `projectPath` facts in figure 5.8 are trivial. The `functionMacro` logic fact on line 7 needs some explanation though. Note that we do not parse the body of the macro. We keep it in the logic fact as a string because in C it does not always contain a valid C expression or statement as macro's contain C source code that gets expanded into the code at compile time. We could also have chosen to expand the used macro's and then parse the source code. However we chose this approach as we want to reason about the use of macro's as well,

[5]A storage class specifier in C can be one of the following keywords: `typedef`, `extern`, `static`, `auto` or `register`. A type qualifier can be `const` or `volatile`.

although it causes some problems in the current version of BEHAVE.

5.4.2.2 The Basic Layer

As can be seen on figure 5.4, the reification layer, together with the basic layer, holds those logic declarations that represent and are able to access the complete base language program. The basic layer holds those logic rules that can access the deeper parse tree structure of the information contained in the logic facts of the reification layer. For each functor representing a certain C construct (i.e. a node in the parse tree), the basic layer provides a logic rule for accessing each argument the functor holds. This representation and also the names of the functors is based on a general Ansi C grammar.

Below you can see an example of the *accessor* rules on lines 1–10 to retrieve all five arguments of a function definition. The `Filename` and `Name` of the function definition are terms (or leaves in the parse tree representation), while the other arguments are again functors (or parse tree nodes).

```
1  functionDefinitionHasFilename(F,Filename) :-
2     equals(F,functionDefinition(Filename,Return,Name,Pars,Body)).

3  functionDefinitionHasReturn(F,Return) :-
4     equals(F,functionDefinition(Filename,Return,Name,Parameters,Body)).

5  functionDefinitionHasName(F,Name) :-
6     equals(F,functionDefinition(Filename,Return,Name,Pars,Body)).

7  functionDefinitionHasParameters(F,Pars) :-
8     equals(F,functionDefinition(Filename,Return,Name,Pars,Body)).

9  functionDefinitionHasBody(F,Body) :-
10    equals(F,functionDefinition(Filename,Return,Name,Pars,Body)).

11 isFunctionDefinition(F) :-
12    nonvar(F),
13    equals(F,functionDefinition(Filename,Return,Name,Parameters,Body)).

14 isFunctionDefinition(F) :-
15         var(F),
16         equals(F,functionDefinition(Filename,Return,Name,Parameters,Body)),
17         F.
```

Figure 5.9: An excerpt of the BEHAVE basic layer

The last two logic rules implement the `isFunctionDefinition/2` predicate. The first rule gets executed if the variable `F` is already bound to a certain value (this is checked by the `nonvar/1, var/1` standard Prolog predicates on lines 12 and 15 respectively). The first declaration on lines 11–13 serves as a *type check* to see whether the variable `F` indeed holds a representation of a function definition. The second rule on lines 14–17 is triggered in case the variable `F` is not yet bound to a value. In this case, the rule will first bind a `functionDefinition` fact with only variables as its five arguments (line 16). Second, it will match those variables with all function definitions of the C program under investigation from the reification layer (line 17).

5.4.3 The Instrumentation Module

The instrumentation module contains three logic layers: the *generation layer*, the *aspect layer* and the *configuration layer*. This module is the central component of BEHAVE as it consults the rules of other components and it produces the instrumented code needed to obtain the run-time information of the program under investigation. We will elaborate here on the first two layers only. The configuration layer solely contains logic facts to set the path for generating the source code files. Appendix B can be consulted for viewing all the predicates of this layer.

5.4.3.1 The Generation Layer

This layer is responsible for regenerating the C source code from the logical representation. In figure 5.10 you can see some of the main `generate` predicates that constitute the generation layer of BEHAVE. These generation rules *walk through* the logical representation of the C base program in a *top down* fashion to reconstruct the C source code of the program under study. The complete generation of the source code from its logical representation is realised by launching the following query:

```
?- generateProject(_).
```

This query triggers the logic rule specified on lines 1–3 in figure 5.10. This predicate first collects all the names of the files that are part of the C project under investigation and will then generate a file for each of them (`findall/3` and `foreach/2` were explained before in section 5.2.2).

The `zopen/2, zclose/1` and `zwrite/1` predicates. As in almost every programming language, Prolog offers some predefined predicates to handle standard operations on files. Standard operations are the `open/close` and `write` of a file. To use the `zopen, zclose` and `zwrite` predicates in Prolog, self-defined predicates that *translate* them to their Prolog counterparts `open, close` and `write` were created. The reason for doing so is to provide an extra layer of abstraction to be able to run BEHAVE in both Prolog and SOUL (see section 5.2.3) [6]. To demonstrate how we translate these rules to Prolog, we show here one of the above rules:

```
zopen(File,Result) :-
   open(File,write,Result).
```

[6]Putting the 'z' in front refers to the Zombie library, i.e. the basis for BEHAVE, as explained in section 5.1.

The predicate that is used in the body of the `zopen/2` rule is the standard Prolog predicate `open/3` to open a file. The variable `Result` is then bound to a writable stream.

Continuing with the `generateFile/1` predicate on lines 4–14 in figure 5.10, a path needs to be established first where the newly created file will be saved. On line 5 the `generatePath/1` logic fact holds the path where the generated code files will be put [7] and on line 6 the complete path with the filename is created and bound to the `Fullpath` variable. Between the opening and closing of the file on lines 7 and 14 respectively, the whole content of the file is again generated with some extra features for instrumenting the source code. The `includeBehave` predicate adds the line '`#include "Behave.h"`' into each generated file. This header file contains a C function (i.e. a `log` function) which will be used for executing the newly added parts for instrumenting the source code. Then all subparts of a file, such as the included files, the C macro's, declarations and function definitions are generated one-by-one.

```
 1  generateProject(Project) :-
 2     findall(Filename,isProjectFilename(Filename),Filenames),
 3     foreach(Filenames,generateFile).

 4  generateFile(Filename) :-
 5     generatePath(Path),
 6     buildPath(Path,Filename,Fullpath),
 7     zopen(Fullpath,Stream),
 8     includeBehave(Stream),
 9     generateAllIncludes(Filename,Stream),
10     generateAllMacros(Filename,Stream),
11     generateAllDeclarations(Filename,Stream),
12     generateAllFunctionDefinitions(Filename,Stream),
13     znewline(Stream),
14     zclose(Stream).

15  generateAllFunctionDefinitions(Filename,Stream) :-
16          zwritenl("/* Function definitions */",Stream),
17          forall(functionDefinitionHasFilename(Function,Filename),
18                 generateFunctionDefinition(Function,Stream)),
19          znewline(Stream).

20  generateFunctionDefinition(Function,Stream) :-
21      generateFunctionSignature(Function,Stream),
22      functionDefinitionHasBody(Function,Body),
23          generateAux(Body,[Body,Function],Stream),
24          znewline(Stream),
25          znewline(Stream).
```

Figure 5.10: An excerpt of the BEHAVE generation layer

Generating include statements in a file, defined macros and function and variable declarations is done in a straightforward manner since they are represented as separate logic facts in the reification layer. As these parts will not be considered for code instrumentation for obvious reasons, generating these facts only requires a syntax transformation of their logical fact representation into a textual C source code

[7] A user of the system can specify this path through a BEHAVE user interface created in Smalltalk.

construct. Generation of the function definition files is somehow less simpler. The `generateAllFunctionDefinitions/2` predicate defined on lines 15–20 first writes a string to the stream for documentation purposes (line 15) and then repeatedly generates all function definitions found in that particular file, which leads us to the `generate-FunctionDefinition` predicate defined on lines 20-25. It first generates the function signature on line 21 (putting together the function's return type with its name and parameter list [8]). After retrieving the body of the function on line 22, the auxiliary predicate `generateAux/3` is launched. This predicate represents the generation of a certain C construct (a C statement or expression) captured by the `Body` variable. For the second argument, a list is formed with as current first element the C construct and the last element the function definition itself. This particular list represents the *path* or entire *parse tree* to which a C construct belongs to. Obviously the third parameter holds the stream on which we write the generated code.

The generateAux/3 predicate This auxiliary predicate shown in figure 5.11 implements the auxiliary predicate which links the predicates from the generation layer to those of the aspect layer. Next to generating code for a certain C construct held by the `Construct` variable, this first declaration on lines 1–6 of the predicate checks if the user enabled the aspect system (line 2) followed by possibly generating *before* or *after* aspects. If a user of the BEHAVE platform *indicated* (how exactly this can be done will be explained in detail in section 5.5) having interest in watching the behaviour of this particular kind of C construct held in the `Construct` variable, these aspect-generation predicates will generate logging behaviour as well. In case no interest has been shown in instrumenting this particular C construct, the first declaration of the `generateAux/3` predicate will fail and hence the second predicate on lines 7–8 of figure 5.11 will be triggered.

```
1  generateAux(Construct,Path,Stream) :-
2          aspectsEnabled,
3          generateBeforeAspects(Construct,Path,Stream),
4          generateAux2(Construct,Path,Stream),
5          generateAfterAspects(Construct,Path,Stream).

6  generateAux(Construct,Path,Stream) :-
7          generateAux2(Construct,Path,Stream).
```

Figure 5.11: The `generateAux/3` predicate

This second predicate relies on another auxiliary predicate `generateAux2/3` that is implemented by multiple declarations, one for each type of C program element (statements, expressions, operators) which might appear in the body of a C function. Note that if a program point of interest is encountered through a *before* or *after* aspect that this auxiliary predicate is also called. Indeed, in addition to generating specific code

[8]The `generateFunctionSignature/2` predicate can be viewed in the appendix

instrumentation before or after a program point of interest, one must generate the code for that particular C construct as well (this will become clear by investigating the aspect layer predicates in section 5.4.3.2).

```
generateAux2(AssignmentExp,Path,Stream) :-
        assignmentHasLeftExpression(AssignmentExp,LeftExp),
        generateAux(LeftExpr,[LeftExp|Path],Stream),
        assignmentHasOperator(AssignmentExp,Operator),
        generateAux(Operator,[Operator|Path],Stream),
        assignmentHasRightExpression(AssignmentExp,RightExp),
        generateAux(RightExp,[RightExp|Path],Stream).
```

Figure 5.12: An example declaration of the `generateAux2/3` predicate for generating an assignment expression

Figure 5.12 presents one of the declarations of the `generateAux2/3` predicate for generating an assignment expression. Such an expression is typically of the form `leftExpression AssgOperator rightExpression`. For each of these three assignment expression components, the `generateAux/3` predicate specified in figure 5.11 is triggered. Note the close collaboration between the auxiliary predicates of both figures 5.11 and 5.12: for every C expression the `generateAux` predicate checks for aspects; the `generateAux2` predicate goes one level deeper into the parse tree structure calling the `generateAux` predicate again on each of the lower level parse tree nodes encountered, etc .... Note that checking at every parse tree node for possible program points which need to be instrumented, the granularity level for code instrumentation is as fine-grained as can be. We refer the reader to appendix B.1 for consulting the predicates of the generation layer.

Note again here the advantage of using a declarative language as representation medium for representing a base language: the syntax of the C assignment expression has been abstracted away which makes reasoning at a higher level of abstraction possible, away from implementation details. This even makes easy reuse of these logical declarations possible when we would apply our proposed approach on another procedural language. In the case of the assignment expression, all that possibly needs to change is the rule specifying the assignment operator. Next, we will elaborate on the aspect layer of BEHAVE which forms the core mechanism for providing code instrumentation next to generating regular C source code.

5.4.3.2 The Aspect Layer

As shown above when we presented the generation layer, the mechanism for instrumenting the source code is intertwined with the code generation process. Together with generating the source code for function definitions in C, the logical parse tree structure is traversed. During that process, BEHAVE checks every parse tree node for program points of interest as indicated by the user.

Below we elaborate on one part of the aspect layer, i.e. the *before* aspects. For the full implementation of the *after* aspects we refer to appendix B.2.

```
 1  generateBeforeAspects(Construct,Path,Stream) :-
 2     forall(isBeforeAspect(Before),
 3                generateBeforeAspect(Before,Construct,Path,Stream)).

 4  isBeforeAspect(Before) :-
 5     captureEvent(Construct,Path,When,RecordAs),
 6     equals(Before,
 7            before(Construct,Path,ResultingCode,When,
 8                          transformWhat(Construct,Path,RecordAs,ResultingCode))).

 9  generateBeforeAspect(Before,Construct,Path,Stream) :-
10     equals(Before,before(Construct,Path,ResultingCode,Condition,Code)),
11     Condition,
12     Code,
13     generateAux2(ResultingCode,[ResultingCode|Path],Stream).

14  generateBeforeAspect(Before,Construct,Path,Stream).
```

Figure 5.13: An excerpt of the BEHAVE aspect layer

The first predicate on lines 1–3 provides the link with the predicates from the generation layer as this rule is called from the `generateAux/3` predicate. It declaratively states the obvious that it will generate a before aspect (line 3) for all encountered *before aspects* (line 2) specified by the user of the system. The `isBeforeAspect/1` predicate defined on lines 4–8 indeed provides the link with the user-specified program points of interest. This is done on line 5 by the `captureEvent/4` predicate [9]. For every type of run-time event a user wants to inspect at a later time, she can specify a logic fact. It must hold a description of the parse tree node of interest, how he wants to capture it and which run-time information he wants to see. Let us consider the stack implementation program again from figure 5.6. Suppose we want to view all stack pop operations right before they are actually executed. Then we would need the `captureEvent/4` fact depicted in figure 5.14. This fact specifies if a specific C construct `Construct` with parse

```
captureEvent(Construct,Path,
             stackPopOperation(Construct,Path),
             event(time,pop(topOfStack,sizeOfStack))).
```

Figure 5.14: Using the `captureEvent/4` predicate for instrumenting all stack pop operations for the running example depicted in figure 5.6

[9]The predicate which the user of the system will need to specify is the `intercept/3` predicate. It serves as an interface which gets redirected to the `captureEvent/4` predicate demonstrated here. We will show the use of the `intercept/3` predicate in section 5.5 where we explain in detail how to use the platform.

tree path `Path` is a pop operation on a stack (we will explain the `stackPopOperation` condition in section 5.4.4). Immediately before the pop operation occurs, we want to insert instrumentation code that, upon program execution, will write the event specification on line 3 to an execution trace file. Next to revealing the type of the event (pop), we also capture the time the event occurred, what is on top of stack and the stack's size. Note that *time*, *topOfStack* and *sizeOfStack* are not logic variables here: we call these *keywords* which represent the run-time values associated with a particular event. They will be explained in the next paragraph.

Continuing the explanation of the predicate `isBeforeAspect/1` specified in figure 5.13 on lines 4–7, we unify the `Before` variable to a *before/5 functor* (line 6). In addition to the C construct and its parse tree path it holds a variable `ResultingCode` (which will be bound at a later time), the condition that expresses `When` a construct should be instrumented (in the example this is the `stackPopOperation/2` predicate) and as last argument holding a logic predicate `transformWhat/4`. Recalling the first `generateBeforeAspects/3` predicate in figure 5.13, for all found bindings of the `Before` variable to the `before functor` we launch the `generateBeforeAspect/4` predicate.

This predicate (defined in figure 5.13 on lines 8–13) extracts the before functor's arguments and launches the `Condition` argument. Recalling the example wanting to capture all pop operations on the stack, this condition checks whether `Construct` (a parse tree node representing a C construct) we are traversing right now fulfils the `stackPopOperation(Construct,Path)` condition. If this evaluates to true, the `Code` variable bound to the `transformWhat/4` predicate is launched. Again referring to the stack example, this predicate takes the `event(time,pop(topOfStack,sizeOfStack))` specification and transforms it into the base language code needed to produce such a specification at run-time. It decomposes the event specification into predicate names (event and pop), commas and *keywords* (time and fcnName). For every type of component it will create the C base language code needed to produce all these different components. The resulting base language code captured in a `behaveCodeList/1` functor is then unified with the `ResultingCode` variable. As a last action, the `generateAux2` predicate is launched on line 12. The resulting code is passed on to the declaration that takes care of writing this base language code onto the output stream [10]. The second `generateBeforeAspect` declaration on line 14 is needed to not let this predicate fail in `generateBeforeAspects` on lines 1–3. The complete implementation for the `transformWhat/4` predicate can be consulted in appendix B, but we won't discuss it here into further detail.

[10]We previously explained (in figures 5.11 and 5.12) the close collaboration between the `generateAux/3` predicate and the `generateAux2/3` predicate. The latter is implemented by having a declaration for each type of construct of the base language. Likewise there exists a declaration `generateAux2(BehaveCodeList,Path,Stream)` for writing a list of generated code to the output stream.

Keywords We introduced the concept of a *keyword* to be able to reason not only about occurred events but also about run-time values associated with that event. A keyword represents a particular kind of value associated with a run-time event which provides extra information about that event. Keywords must be specified in the base language to be inserted into the source code. Consider the keywords from the running example:

```
keyword(Construct,Path,time,"behaveLog(\"%i\",TIME++);").
keyword(Construct,Path,topOfStack,"behaveLog(\"%i\",stack[top-1]);").
keyword(Construct,Path,sizeOfStack,"behaveLog(\"%i\",top);").
```

Figure 5.15: Keywords denoting how associated run-time values

The keywords specified here can be retrieved at run-time by executing the code specified here as a fourth argument of the `keyword` predicate. Through the `transform-What/4` predicate, this string is written at the right places in the generated file. The C `behaveLog` function is defined as a macro in the file `Behave.h`. The reader can consult appendix E to get a better idea of what the generated code looks like. However, the code shown there is taken from one of the case studies which are presented later in chapter 6.

5.4.4 The Reasoning Module

The reasoning layer captures those predicates that reason about the *structure* of base language programs. Both application-specific predicates as well as predicates generally applicable to any C program reside in this layer. In figure 5.16 two representative predicates of this layer are shown. The first application-specific example is the `stackPopOperation` condition we specified above to investigate all pop operations on the stack. From the code from the stack running example depicted in figure 5.6 we see on line 5 that the `pop` is implemented as a macro. Figure 5.16 shows on lines 1–2 the corresponding reasoning predicate `stackPopOperation`.

The second example is a rule applicable to all C base language programs. It describes a very straightforward predicate to capture the entry of a function.

The `functionEntry/2` predicate checks whether a certain C construct `Construct` in the parse tree path `Path` is the first statement in a function body. If this is the case, the parse tree path of such a statement would contain four nodes (This can be easily deduced from 5.3: to reach the statement level, four parse tree nodes need to be traversed). The fourth (and last) node holds the function definition (lines 4–5), while the third node holds the function body (line 6). And `Construct` should be the first statement in the body (lines 7–8).

In figure 5.17 we depict the `expressionIn/3` predicate which finds any kind of subexpressions of any C construct. This predicate is used to search for expressions of interest in the parse tree representation of a certain expression (as will be demonstrated

```
stackPopOperation(Construct,Path) :-
    macroCallHasName(Construct, 'pop').

functionEntry(Construct,Path) :-
        listAt(4,Path,Function),
        isFunctionDefinition(Function),
        listAt(3,Path,Body),
        compoundStatementHasStatements(Body,Statements),
        first(Statements,Construct).
```

Figure 5.16: Example predicates of the BEHAVE reasoning layer

in the next chapter). Next to finding a certain expression, the predicate also keeps track of the parse tree path to reach `Expression` from the `Construct` construct.

```
1  expressionIn(Construct,Expression,Path) :-
2     expressionInAux(Construct,Expression,[Construct],Path).

3  expressionInAux(Expression,Expression,Path, Result) :-
4     not(is_list(Expression)),
5     equals(Path,Result).

6  expressionInAux(List,Expression,Path,Result) :-
7     is_list(List),
8     member(Element,List),
9     expressionInAux(Element,Expression,[Element|Path],Result).

10 expressionInAux(Functor,Expression,Path,Result) :-
11    not(is_list(Functor)),
12    compound(Functor),
13    Functor =.. List,
14    list_tail(List,Tail),
15    expressionInAux(Tail,Expression,[Element|Path],Result).
```

Figure 5.17: The `expressionIn/3` predicate

The recursive implementation of this predicate accentuates the power of using a logic programming language: first an auxiliary predicate `expressionInAux/4` is set up (line 2) to hold one extra argument for creating the path variable and then three distinctive declarations are defined. Two of the declarations recursively call themselves. Either `Construct` holds a list of C constructs in which case we recursively call the auxiliary predicate on its elements (lines 6–9). Or `Construct` holds again a compound expression, which necessitates a recursive call on all arguments (lines 10–15). The declaration defined on lines 3–5 finds the expression and stops the recursion. Note the use of the '`=..`' Prolog predicate. It transforms a predicate of the form `foo(arg1,arg2)` into the list `[foo,arg1,arg2]`.

5.4.5 The Temporal Logic Meta Interpreter

In section 5.2.4 we have motivated our choice to use a temporal logic programming language as a medium for analysing program behaviour. Because temporal logic programming languages provide additional logic operators for expressing temporal relationships, they are particularly welcome when reasoning about events representing run-time behaviour.

In section 5.1 we mentioned the implementation of the temporal logic meta interpreter which forms an important component of the BEHAVE platform. A common feature of most well-known temporal logic programming languages such as Templog or Chronolog [Org94] is that they use as proof procedure a temporal version of a resolution-based procedure. The temporal logic meta interpreter used in BEHAVE however is based on MTL (see section 5.2.4) and it is shown that MTL can be considered as an instance of the constraint logic programming (CLP) scheme over a suitable algebra [Brz95]. This is reflected in the meta interpreter's Prolog implementation as it employs the `clp/bounds` library in Prolog.

In section 5.2.4 we mentioned that in order to overcome the limitations of MTL with regard to the use of temporal operators in the clause heads and bodies (because of the unboundedness of the underlying logic) that in the meta interpreter time will be bounded to the time stamps from the execution trace. In figure 5.18 the beginning of time is set to 0 (the `bot` predicate), while the end of time is bound to the time stamp of the last event recorded in the execution trace (the `eot` predicate on lines 2–4).

```
bot(0).

eot(Time) :-
        findall(T, event(T, _), Ts),
        max_list(Ts, Time).
```

Figure 5.18: Marking the timeline in the temporal logic meta interpreter for the events in the execution trace

The temporal logic meta interpreter translates both an MTL program and an MTL goal that needs to be proven (launching a query) into a corresponding classic Prolog program which contains constraint predicates over time stamps. In order for the meta interpreter to be able to consult the high-level events contained in the execution trace, they must be of the form (this can also be deduced from the `eot` predicate above) `event(T,_)`. In its implementation, the meta interpreter attaches the variable `T` to a predicate that needs to be proven. This clearly demonstrates the context abstraction (or time abstraction) that is obtained by using this meta interpreter: the use of the temporal operators lets a user abstract over time while the explicit use of a time stamp in logic programming is hidden in its implementation.

The temporal logic meta interpreter provides the following operators: $\Box$ (always), $\diamond$ (sometimes), $\bullet$ (previous) and $\circ$ (next). Besides these, also operators from classical logic are used, such as $\neg$, $\wedge$ and $\vee$. Below we show an excerpt of the implementation of the meta interpreter to give an idea of how it is implemented using the Prolog `bounds` library:

```
1 solve(A) :-
2    prove(A, 0).

3 prove(next(A), T) :- !,
4    NT #= T + 1,
5    prove(A, NT).

6 prove(previous(C, A), T) :- !,
7    C #> 0,
8    NT #= T - C,
9    prove(A, NT).
```

Figure 5.19: Implementation excerpt from the temporal logic meta interpreter

The `solve` predicate represents the *interface* of the meta interpreter. Suppose the `model` predicate is defined which represents a temporal logic formula. Trying to prove the model is done by launching the query `?- solve(model).` In the implementation of the meta interpreter, a `prove` predicate is implemented which has a declaration for proving each kind of temporal operator. The declaration on lines 3–5 of figure 5.19 proves the clause `A` at the next point in time, which means that the clause `A` should be proven at time point `NT`. And this time point is constrained to have the value `T+1`. The `#=/2` predicate is a constraint predicate from the `bounds` Prolog library. Likewise, the `prove` declaration on lines 6–9 uses the `#>` and `#=` constraint predicates to prove clause `A` at `C` previous time steps in the past. Note the use of the Prolog *cut* operator in both `prove` declarations: the operator is used here to make sure that no alternative declarations for the `prove` predicate are tried.

The `prove` predicate itself is mapped onto the generated high-level execution trace by the following declaration:

```
1 prove(A, T) :-
2    A =.. [Predicate | Arguments],
3    append([Predicate, T], Arguments, Extended),
4    Term =.. Extended,
5    clause(Term, B),
6    writeln(Term),
7    prove(B, T).
```

Clause `A` is found in the repository where clauses have one extra argument (i.e. a time stamp). This can be deduced from lines 2–5 where for the clause `A` the variable T is added as first argument to its argument list. For example, at the level of the temporal meta interpreter, a `pop` event is of the form `event(push(Top,Size))`. However, at the level of Prolog and in the execution trace, the same event is denoted e.g. `event(5,pop(Top,Size))`. If such a clause is found, then it is first written to a stream and its body is proved. Other declarations of this predicate exist, but we do not explain them here. For the complete specification of the temporal logic meta interpreter, the reader is referred to appendix C.

By using these operators, we can describe temporal relations between events contained in the execution trace in an even more expressive manner as opposed to when you use traditional logic programming. However, although temporal logic formulas have been successfully applied in the program verification domain, they are sometimes hard to understand and difficult to formulate [Hol02a]. Therefore users not familiar with temporal logic are still free to specify their models using plain Prolog rules.

5.5 A Four-step Recipe for Using BEHAVE

Having provided more insight into the internals of the BEHAVE platform and in particular the different logic layers, we emphasise in this section on *how* a developer can *use* the platform for documenting and verifying a behavioural design invariant in a lightweight manner.

Using the BEHAVE dynamic analysis platform for reasoning about particular behaviour of a system, a user freely determines the high-level events over which they want to express a behavioural model of a design invariant. Although such large degrees of freedom require more developer involvement, it lets us specify the actual behaviour of a system in terms of high-level concepts rather than low-level implementation constructs.

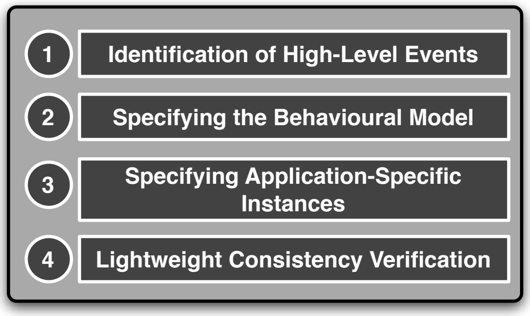

1
Identification of High-Level Events
2
Specifying the Behavioural Model
3
Specifying Application-Specific Instances
4
Lightweight Consistency Verification

Figure 5.20: A Four-step Recipe for Using BEHAVE

To optimally exploit the BEHAVE platform during an application's life-cycle, a developer can therefore adhere to the four-step recipe depicted in figure 5.20. This recipe was already very briefly introduced in the previous chapter in section 4.5. The first step comprises the identification of the *needed constituents* of the high-level events in order to reason about the specific behavioural design invariant we have in mind. On top of that, in step 2 we specify our current understanding of the behavioural design invariant by specifying desired or unwanted behaviour in a behavioural model. In step 3 we formulate the application-specific instances of the high-level events to link the needed domain concepts to the actual implementation. This goes hand-in-hand with specifying the associated run-time values. In step 4 we perform the actual verification of the behavioural model of the design invariant against the actual behaviour, i.e. the high-level execution trace.

In subsequent sections, we detail each of the recipe steps identified in figure 5.20 using again the stack implementation from figure 5.6. We use BEHAVE to document and verify basic invariant stack behaviour [11]. An overview figure of the BEHAVE set up is shown in figure 5.22.

5.5.1 Step 1: Identifying High-Level Run-Time Events

The first step in the recipe to verify invariant behaviour is to *identify the needed constituents of the high-level events* over which the behavioural model will be expressed. When thinking about the behaviour of a stack data structure, the operations *pop*, *push* and *initialise* immediately spring to mind. Independent from a concrete implementation, the *stack size* and the *element on top* are other important stack-related concepts. However, the concrete values of these concepts will vary over time as the stack is being used. So we will consider these as additional run-time information associated with each of the high-level events representing stack operations.

Considering again the concrete stack implementation depicted in figure 5.6, we could also have described the behavioural model directly in terms of calls to the function `push` or occurrences of the macro expansion `pop`. We have however argued in chapter 3 that behavioural models consisting of assertions over low-level run-time events do not match the requirement of being oblivious from source code constructs as a clean separation between application-specific and conceptual instances is then not realised. This would impede program development as making a change to the source code would imply having to change the behavioural model accordingly.

5.5.2 Step 2: Specifying the Behavioural Model

The next step in the recipe consists of specifying the invariant behaviour in a model. We would like to specify and later verify the following most basic invariant behaviour of a typical stack data structure:

INVARIANT: Basic stack behaviour
The size of a stack increases by one whenever a new element is pushed on it; it shrinks by one whenever an existing element is removed through the pop operation and a stack should always be properly initialised before it is being used.

Given MTL as a specification language as we explained in section 5.2.4, we expressed our understanding of basic stack behaviour in the model depicted in figure 5.21

[11]This stack example was also used as an introductory example in De Roover et al [RMG$^+$06]

```
1  push(S) :- event(push(_,S)).
2  pop(S) :- event(pop(_,S)).

3  stackInitialised(S) :- event(init).
4  stackUsed(S) :- push(S).
5  stackUsed(S) :- pop(S).
6  stackOperation(S) :- stackUsed(S).
7  stackOperation(S) :- stackInitialised(S).

8  behaviouralModel :-
9    until(stackInitialised, ¬stackUsed),
10   □(when(push(S) ∧ •stackOperation(S1), S is S1 + 1)),
11   □(when(pop(S) ∧ •stackOperation(S1), S is S1 - 1)).
```

Figure 5.21: Behavioural Model specifying invariant stack behaviour

The model itself can be found on lines 8–11. The second line of the extract (line 9) states that until the stack is initialised, it may not be used. At line 10 the model states that it must always be the case that whenever a push operation left the stack in a state with size S, any previous stack operation should have left the stack in a state with size $S - 1$. Note that this description of desired stack behaviour is truly oblivious from any implementation-level construct. This makes the model easily reusable for evolved versions of this program, but also for completely different stack implementations.

Lines 3–7 define the logic predicates used within these assertions. Lines 6 and 7 define that a `stackOperation` can either be the initialization or manipulation of the stack. The `push` and `pop` operations are considered stack manipulators, which is expressed by the `stackUsed` predicate in lines 4 and 5.

The first two lines of figure 5.21 link the predicates used in the behavioural model to the high-level events observed during the execution of the program. We can see that the high-level `push` and `pop` events in the execution trace record more information about the state of the stack than is actually needed by this model specification. The first recorded value is ignored as we are only interested in the second value which records the size of the stack. By altering the definition of the `push`, `pop` and `init` predicates, the behavioural model specification can be reused even when different run-time values are associated with the high-level events in the execution trace. In the next step of the recipe, a user of the platform has to specify how to observe these high-level events during the execution of the program.

5.5.3 Step 3: Application-Specific Instances of High-Level Events

At this point in the four-step recipe, we have identified the constituents of the high-level run-time events typically associated with a stack data structure. We have also specified a model of its behaviour over these events expressed as a temporal logic program. The high-level run-time events *push*, *pop* and *initialize* will be the constituents of the execution traces against which we will verify the program's behavioural model shown in figure 5.21. An example of such an execution trace is shown below, which shows stack behaviour beginning with a stack initialisation followed by three pushes and a pop operation.

```
event(0,init).
event(1.push(10,1)).
event(2,push(20,2)).
event(3,push(30,3)).
event(4,pop(20,2)).
```

Since the recorded execution trace consists of user-defined high-level run-time events, developers have to specify *which* events should be intercepted during an application's execution and also *how* each event is recorded. Such a specification consists of a set of `intercept(When, What, RecordAs)` declarations:

```
1 intercept(after,stackPopOperation,
2           event(time,pop(stackTop,stackSize))).

3 intercept(after,stackPushOperation,
4               event(time,push(stackTop,stackSize))).

5 intercept(before,stackInitOperation,
6           event(time,init)).
```

On the first line of the specification, we declare that all occurrences of a high-level *pop* run-time event have to be intercepted. We also declare that these events must be recorded in the execution trace as facts of the form `event(time, pop(stackTop, stackSize))`. `stackTop` and `stackSize` are the run-time values associated with the *pop* event. Occurrences of the *push* and *initialise* events are logged in a similar way.

The rules in the behavioural model and *which* high-level events to intercept is information that can be shared by different applications. For each specific program, we only need to specify *how* to intercept the high-level events. Applied to the running example, this for instance amounts to identifying the constructs in the application's source code that give rise to the high-level *pop* event. From the code depicted in figure 5.6, it is clear that the pop operation is implemented by code resulting from an expansion of the `pop` C function macro on line 5. We can express this knowledge in the `stackPopOperation` rule which we have previously discussed as an example predicate of the reasoning layer (we will repeat it here on lines 1–2 for the sake of being complete). Likewise we define `stackPushOperation` and `stackInitOperation` on lines 3–6 which are both implemented as function calls.

Recall that BEHAVE makes an entire application's parse tree available.
The `stackPopOperation(Construct,Path)` predicate is checked at each parse tree node through the `Construct` variable, while the `Path` variable represents the path from the tree's root that leads to that node. Although the identification rules for the running

```
stackPopOperation(Construct,Path) :-
    macroCallHasName(Construct,'pop').

stackPushOperation(Construct,Path) :-
    functionCallHasName(Construct,'push').

stackInitOperation(Construct,Path) :-
    functionCallHasName(Construct,'init').
```

example here only need to access attributes from the parse tree nodes themselves, Prolog's full declarative reasoning power will be needed for the case studies in chapter 6 (for example through the use of the `expressionIn` predicate specified before in figure 5.17).

Retrieving the run-time information associated with each high-level event is done through the specification of the keywords `stackTop` and `stackSize` as we have previously shown at the end of section 5.4.3.2. These run-time values will have to be obtained by the execution of application-specific source code.

5.5.4 Step 4: Lightweight Consistency Verification

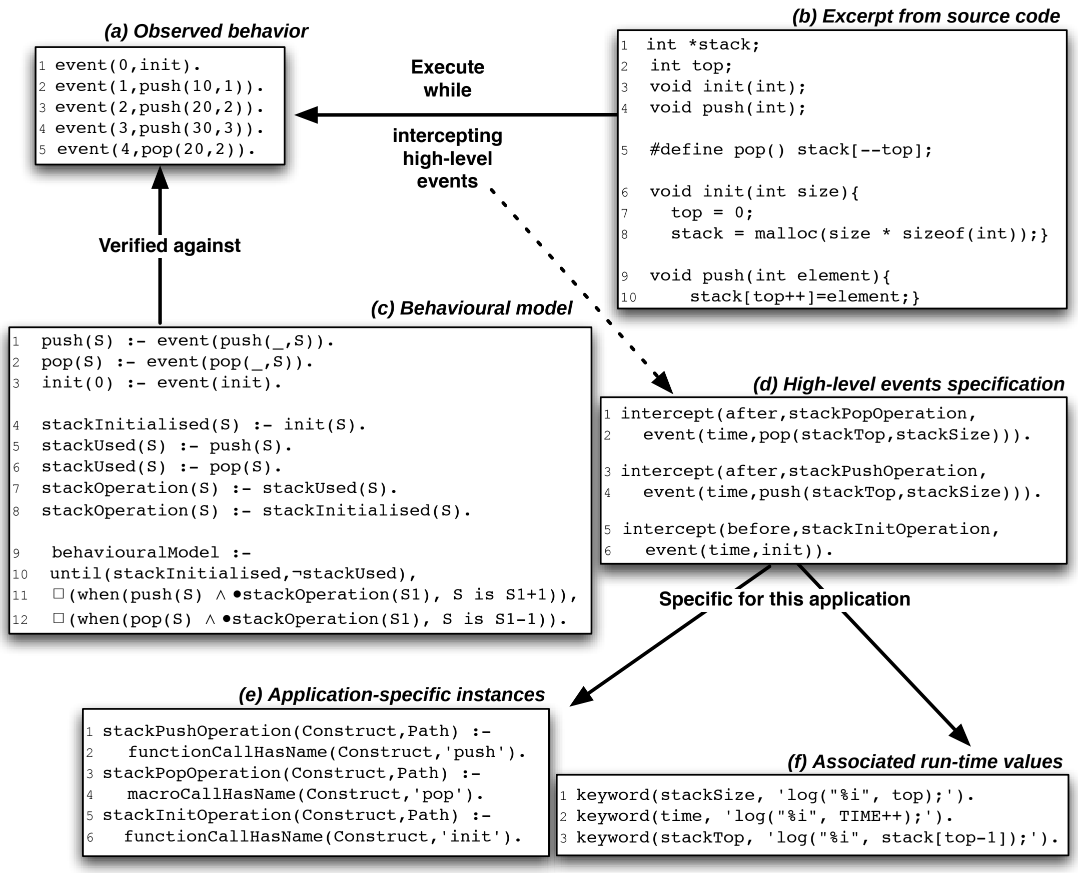
In the last step, a platform user can verify the consistency of a program's actual behaviour with its desirable behaviour specified in the model. This is done by launching logic queries against the recorded high-level execution trace. BEHAVE instruments the source code of the application under investigation in order to record all occurrences of the high-level run-time events specified in the behavioural program model. In order to intercept occurrences of the high-level *pop* event, the platform relies on the `stackPopOperation` logic rule to identify those source code constructs which give rise to the *pop* event. The platform also relies on the definition of the `stackTop` and `stackSize` keywords to obtain the run-time values associated with this event.

To verify the behavioural model specified in figure 5.21, the logic query

```
?- behaviouralModel.
```

has to be launched. In case of a verification failure, the temporal logic interpreter prints the last event that was used in an attempt to prove the query. This information can be used to either adapt the application to the model or the model to the application. The generated execution traces consist of high-level events which makes manual inspection in case of verification failures feasible. As discussed in section 5.3, the generated execution traces contain in general fewer events since not every low-level event needs to be recorded.

Having demonstrated all the constituents for using BEHAVE to verify basic stack behaviour, we provide an overview in figure 5.22.


(a) Observed behavior
Execute while intercepting high-level events
(b) Excerpt from source code
Verified against
(c) Behavioural model
(d) High-level events specification
Specific for this application
(e) Application-specific instances
(f) Associated run-time values

(a) Observed behavior

```
event(0,init).
event(1,push(10,1)).
event(2,push(20,2)).
event(3,push(30,3)).
event(4,pop(20,2)).
```

(b) Excerpt from source code

```
int *stack;
int top;
void init(int);
void push(int);

#define pop() stack[--top];

void init(int size){
  top = 0;
  stack = malloc(size * sizeof(int));}

void push(int element){
    stack[top++]=element;}
```

(c) Behavioural model

```
push(S) :- event(push(_,S)).
pop(S) :- event(pop(_,S)).
init(0) :- event(init).

stackInitialised(S) :- init(S).
stackUsed(S) :- push(S).
stackUsed(S) :- pop(S).
stackOperation(S) :- stackUsed(S).
stackOperation(S) :- stackInitialised(S).

 behaviouralModel :-
until(stackInitialised,¬stackUsed),
 □(when(push(S) ∧ •stackOperation(S1), S is S1+1)),
 □(when(pop(S) ∧ •stackOperation(S1), S is S1-1)).
```

(d) High-level events specification

```
intercept(after,stackPopOperation,
  event(time,pop(stackTop,stackSize))).

intercept(after,stackPushOperation,
  event(time,push(stackTop,stackSize))).

intercept(before,stackInitOperation,
  event(time,init)).
```

(e) Application-specific instances

```
stackPushOperation(Construct,Path) :-
   functionCallHasName(Construct,'push').
stackPopOperation(Construct,Path) :-
   macroCallHasName(Construct,'pop').
stackInitOperation(Construct,Path) :-
   functionCallHasName(Construct,'init').
```

(f) Associated run-time values

```
keyword(stackSize, 'log("%i", top);').
keyword(time, 'log("%i", TIME++);').
keyword(stackTop, 'log("%i", stack[top-1]);').
```

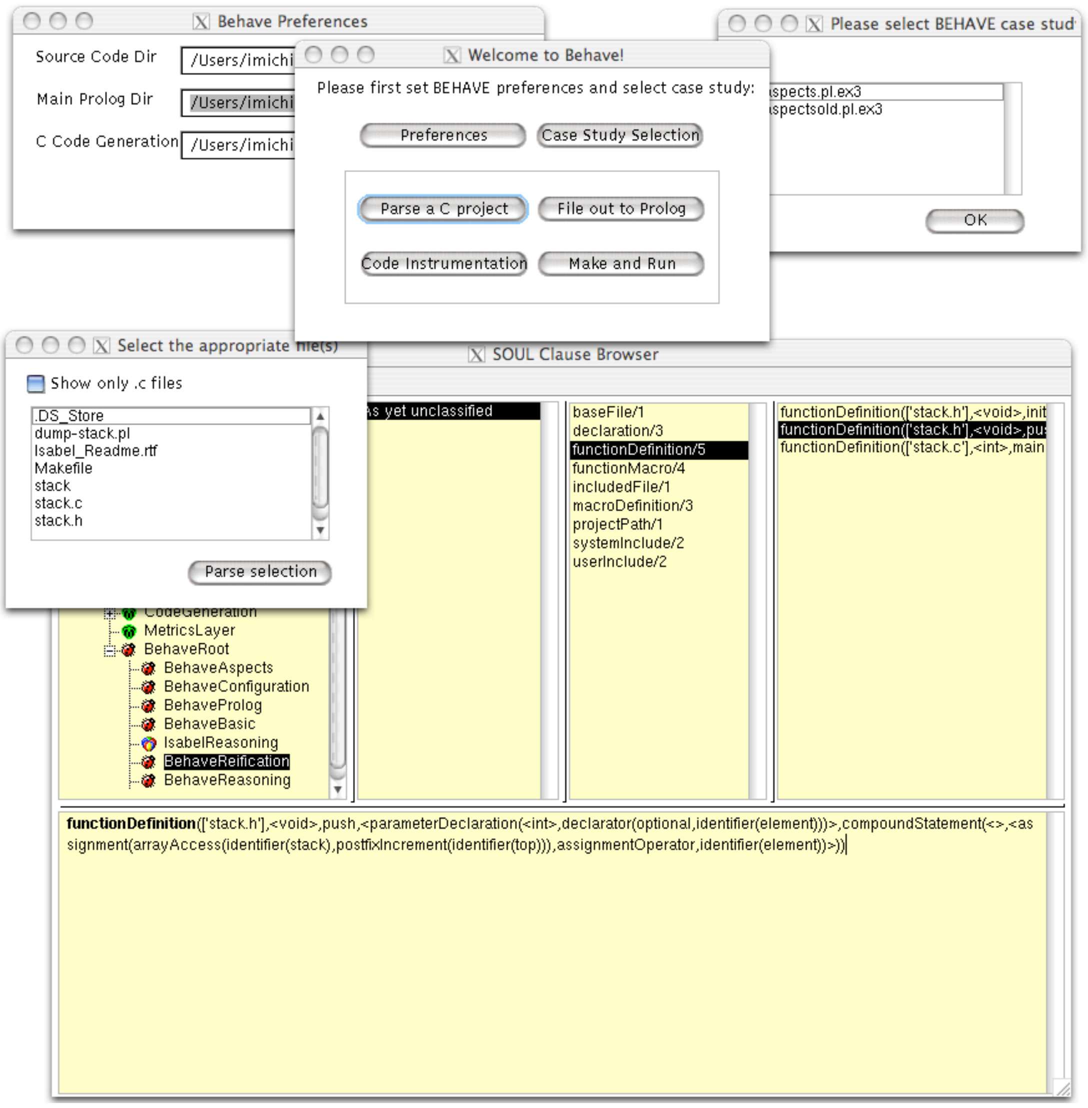
Figure 5.22: Source code and corresponding behavioural documentation of a C stack implementation.

5.6 BEHAVE Tool Support

As mentioned in section 5.1, the BEHAVE platform was originally part of the Visualworks Smalltalk environment [cin]. It was implemented in a Prolog variant SOUL (see section 5.2.3) which has a tight symbiotic relation with its implementation language Smalltalk. In addition to representing regular Prolog programs, SOUL allows Smalltalk expressions to be used in logic terms. A fully similar implementation of BEHAVE still exists in SOUL, however since we do not use SOUL's reflective facilities, we decided to port logic declarations to Prolog, partly also because of performance issues [12].

However, BEHAVE is still being developed and used on top of the VisualWorks environment. The C language parser is written in Smalltalk using the SmaCC parser generator [BR] and SOUL tool support, such as the SOUL clause browser, is useful when adding and browsing logic declarations of the BEHAVE platform. Figure 5.23

[12]Since Prolog is a fully supported programming language, it offers better lookup indexing schemes on the arguments of logic declarations.



Source Code Dir /Users/imichi
Main Prolog Dir /Users/imichi
C Code Generation /Users/imichi

Please first set BEHAVE preferences and select case study:

ispects.pl.ex3
ispectsold.pl.ex3

.DS_Store
dump-stack.pl
Isabel_Readme.rtf
Makefile
stack
stack.c
stack.h

CodeGeneration
MetricsLayer
BehaveRoot
BehaveAspects
BehaveConfiguration
BehaveProlog
BehaveBasic
IsabelReasoning
BehaveReification
BehaveReasoning

s yet unclassified

baseFile/1
declaration/3
functionDefinition/5
functionMacro/4
includedFile/1
macroDefinition/3
projectPath/1
systemInclude/2
userInclude/2

functionDefinition(['stack.h'],<void>,init
functionDefinition(['stack.h'],<void>,pu
functionDefinition(['stack.c'],<int>,main

functionDefinition(['stack.h'],<void>,push,<parameterDeclaration(<int>,declarator(optional,identifier(element)))>,compoundStatement(<>,<assignment(arrayAccess(identifier(stack),postfixIncrement(identifier(top))),assignmentOperator,identifier(element))>))

Figure 5.23: A screen capture of BEHAVE tool support in Smalltalk

shows the tool support available within Smalltalk. The yellow browser window in the figure shows the SOUL clause browser. On the left, the BEHAVE logic layers are shown and the reification layer is selected. The logic fact representing the `push` function definition is selected and its representation can be viewed in the lower part of the browser.

The BEHAVE interface consists of the four smaller windows at the top of figure 5.23. Directories containing the source code location and the destination for the generated and instrumented code can be set through a preferences pane. The behavioural invariant under study and the source code files to be parsed can be selected through a pop-up menu.

5.7 System Overhead

As with any code instrumentation system, the BEHAVE platform introduces some system overhead. In BEHAVE the emphasis of computation lies on the *pre-processing part* of the source code instrumentation. The source code is first being parsed and then reified into a logical parse tree representation. For performing the code instrumentation, the logical parse tree structure is traversed. For each parse tree node, sophisticated high-level checks are performed in parallel with regenerating the source code for that particular node. These sophisticated checks consist of launching a logical query which checks whether that parse tree node *matches* a high-level logical description which the user defined and wants to see instrumented.

Although the pre-processing phase produces some computational overhead, it results in a high-level and selective execution trace. This is in contrast to instrumentation approaches which use a fixed event grammar for example for logging all function entries/exits and/or for logging all assignment statements.

intercepted predicates	**included keywords**	**generate instrumented code**	**# of events in execution trace**
`continuationEntry` `continuationExit`	`cntName`	0.91 sec	9975 events
	+ `cntStack` `cntPointer`	1.05 sec	
`functionEntry` `functionExit`	`fcnName`	0.75 sec	16918 events

Table 5.2: Comparing the time to generate the instrumented source code and execution trace sizes for instrumenting all Pico function calls vs. instrumenting only continuation calls

A consequence of moving a part of the computational overhead to the pre-processing phase is that the behavioural analysis performed on the execution trace *after* instrumentation is simplified significantly. Because of the more compact execution trace, a more high-level behavioural analysis approach can be applied. As the amount of logged actual behaviour is reduced into only design invariant specific high-level behaviour, the reasoning can be focused on only the behaviour relevant for that particular design invariant. Table 5.2 presents a simple comparison between instrumenting all continuations and all function calls in Pico. The Pico input program for producing these results is depicted in figure 6.30 [13]. The Pico program represents the application of a quicksort algorithm on a table of 10 randomly chosen elements. The first row in the table denotes the data when only the continuation entries and exits are instrumented together with the (statically determined) name of the continuation. The second

[13]These experiments were conducted for Pico 1.0 which totals about 8000 lines of C source code. The machine used was a Macbook Pro 2,2 2.33Ghz, 2Gb internal memory.

row shows the same experiment, but two extra keywords are added (representing run-time values), to show the extra time it takes to instrument the source code. The last row depicts the same data when all function entries and exits are logged with only the name of the function as (statically determined) associated value. This demonstrates the impact the sophisticated code instrumentation has on the amount of generated events. While the pre-processing phase of instrumenting the source code takes slightly more time, the difference in execution trace size is significant (also taking into account that Pico uses much more macros than function calls for performance reasons). Although the behavioural reasoning language would allow us to also compose high-level events from low-level run-time events and decouple these events from concrete source code, the process of doing so would become computationally more expensive.

5.8 Conclusions

In this chapter we presented the BEHAVE platform for documenting and verifying behavioural design invariants to support program development. Such design invariants are documented in models which are expressed as high-level temporal assertions in terms of high-level run-time events freely chosen by the user.

In this section we emphasise the most important conclusions made while presenting the platform:

- A user of the platform can specify the constituents of the high-level events at a conceptual level instead of low-level implementation constructs. Adopting such an approach lets us specify a model of (un)wanted behaviour directly in terms of high-level events. This makes the behavioural model oblivious to the source code and makes it reusable with respect to evolved versions of the program under investigation,

- BEHAVE adopts an aspect-like approach for selectively instrumenting the source code (as seen in section 5.4.3.2). As a logic language is used as pointcut language for selecting those program parts of interest, this allows the pointcut expression to be oblivious as well from source code constructs; hence they can also be reused for other adapted implementations,

- An important advantage of BEHAVE is the use of temporal logic programming as specification language instead of plain temporal logic. Formulas in temporal logic are sometimes difficult to express [Hol02a, DAC99], and when using a temporal logic *language*, one can combine temporal operators in a higher-order rule so as to reuse them later. This way there is no need to remember their expression in plain temporal logic,

- Using the BEHAVE platform a user can specify what run-time events should be recorded using a selective aspect-like approach. Consequently the lightweight verification is computationally less expensive because high-level execution traces in general comprise fewer run-time events (see section 5.7).

- Due to the partiality a *declarative* specification formalism offers (as discussed in section 4.3.2), only the behaviour relevant for a particular design invariant is described in the design invariant model. Combined with a well-chosen execution scenario and a selective code instrumentation mechanism, this allows for a lightweight goal-driven verification of parts of a larger program.

We would like to conclude with a note that the invariant stack behaviour we have specified and verified in section 5.5 was only intended as an example for demonstrative purposes as this is a more local form of behavioural design invariant (see chapter 2). It does demonstrate however that the BEHAVE platform can be easily applied for checking these local behavioural invariants as well. However, a great deal more can be achieved, which will become clear in the next chapter where we use BEHAVE to document and verify in a lightweight manner three cross-cutting and non-externally verifiable behavioural design invariants of the Pico language interpreter.

Also the uniformity of the platform's implementation offers clear advantages. As both the structural meta model and the (partial) behavioural meta model are represented as logic facts, dynamic and structural analysis can be easily combined. However, this is not further explored within this dissertation (see future work in section 7.3).

5.9 Summary

In this chapter we presented a prototype platform named BEHAVE that supports the proposed lightweight goal-driven verification approach presented in chapter 4. BEHAVE is implemented in the logic language Prolog and uses an aspect-like approach for instrumenting programs written in C. The main components of the platform are the *reification module* which holds a logic representation of a C base language programs under investigation, the *instrumentation module* which combines a generation layer with an aspect layer to produce selectively instrumented code and the *temporal logic meta interpreter* to reason about high-level run-time events.

We started by pinpointing the history of BEHAVE. In section 5.2 we introduced the logic meta programming set-up used for representing base language programs and we elaborated on logic programming in general and its characteristics. As temporal logic programming is used for analysing program behaviour, we explained the use of this medium as well. In section 5.3 we presented BEHAVE as a dynamic analysis platform, while an overview of the main constituents of the BEHAVE architecture was given in section 5.4. In section 5.5 we outlined a four-step recipe on how to optimally *use* the BEHAVE platform. Each of those steps were gradually introduced by applying the four-step recipe for verifying basic invariant stack behaviour. In section 5.7 we briefly discussed the system overhead the platform produces after demonstrating the

BEHAVE available tool support in Smalltalk (section 5.6). We rounded off with some conclusions.

Chapter 6
Using BEHAVE for Supporting Program Development

In this chapter we demonstrate some sophisticated examples of how the BEHAVE platform can be used for supporting program development by documenting and verifying design invariants. In the previous chapter we introduced our platform BEHAVE and we put forward a four-step recipe to optimally exploit the use of this platform. In this chapter we apply this four-step plan for documenting and verifying three design invariants of a fairly compact but technologically and algorithmically challenging case study to demonstrate the full power of BEHAVE.

An important criterion for selecting a case study was the availability of knowledge about design decisions and about the internals of the program. Even though this kind of information can sometimes be found in documentation, it is better to interview the original developer directly. Another requirement was that different versions of the source code would be available as to study how *invariant* these design invariants really are with respect to program evolution.

Exactly for those reasons we chose to use the Pico virtual machine as our experimental basis. Pico is an interpreted programming language developed at the Vrije Universiteit Brussel and is mainly used for teaching purposes [DM, DM00]. The Pico interpreter incorporates automatic garbage collection, allows higher order functions, supports meta programming and reflection and uses optimised tail-calls for implementing iterative processes. Pico was initially created to teach the basics of computing to first year science students other than computer science. It is still heavily used as an extensive learning tool for teaching programming concepts. However, currently Pico is mainly used in the second year of computer science to teach principles of interpreted languages and memory management. But Pico has also been adopted for doing research-oriented experiments on prototype-based inheritance, code mobility and distribution [MDD04].

As for our needed requirements, Pico formed the ideal case study for applying our BEHAVE platform. First, multiple versions of the Pico interpreter are available which enables us to study important design invariants with respect to program evolution. Second, the original developer was available to give us insight in the main design decisions that were undertaken when implementing Pico. This is important since it not only allowed us to question the developer about which design invariants the Pico behaviour should adhere to during program execution, but also to verify possible invariant violations found by BEHAVE.

Moreover since Pico is being used as a teaching vehicle for teaching concepts of interpreted languages, we were inspired by the course's final attainment level topics to pinpoint Pico's important design invariants. Indeed, one of the sub-goals of this dissertation is to use the BEHAVE platform in a teaching environment. For more information, we refer to our future work in section 7.3.3 where we elaborate on this.

We start the chapter with a detailed explanation of the Pico language interpreter as it is the case study we use throughout this chapter. We zoom in on the Pico memory model as well as the execution model as they are prerequisites for explaining the Pico design invariants we document and verify. In the next sections we demonstrate the use of the BEHAVE platform by verifying behavioural Pico documentation, erroneous garbage collection behaviour and tail recursion optimisation. Finally we end with a conclusion.

6.1 The Pico Language Interpreter

The implementation of Pico was strongly inspired by the book from Abelson and Sussman [AS84]. Its goal was to establish a technological framework for uniting all notions and concepts relevant for exposing students to the matter of building a compact language processor. Pico is a fairly compact and portable virtual machine with a very intuitive syntax. The basic Pico implementation (Pico 1.0) consists of an 8000 lines Ansi C framework that incorporates a fully self-contained computation and storage model. It is documented by a 500 line meta-circular implementation which closely resembles the C version. Pico is a simple, dynamically-typed language with automatic memory management similar to Scheme. It is properly tail-recursive and it is based on first-class functions which are implemented as closures because of static scoping. As Pico is being used in an educational and in a research context, its implementation has heavily evolved over the past years. This resulted, next to occasional small refinements of Pico 1.0, into serious architectural changes which led to different Pico versions being in use today. The Pico virtual machine has a number of advanced qualities which we first explain in more detail as they will be needed to study the case studies in the following sections.

6.1.1 The Pico Execution Model

Pico is an interpreter which relies on the concept of a *continuation* to represent the different sub-computations needed to perform a certain computational task, such as evaluating a Pico expression. A Pico continuation has a slightly different meaning than

```
1  /*-------------------------------------------------------------*/
2  /*  ASS                                                         */
3  /*     expr-stack: [... ... ... DCT VAL] -> [... ... ... ... VAL] */
4  /*     cont-stack: [... ... ... ... ASS] -> [... ... ... ... ...] */
5  /*-------------------------------------------------------------*/
6  static _NIL_TYPE_ ASS(_NIL_TYPE_)
7  { _EXP_TYPE_ dct, val;
8    _stk_pop_EXP_(val);
9    _stk_peek_EXP_(dct);
10   _ag_set_DCT_VAL_(dct, val);
11   _ag_set_DCT_DCT_(dct, _DCT_);
12   _DCT_ = dct;
13   _stk_poke_EXP_(val);
14   _stk_zap_CNT_(); }
```

Figure 6.1: The assignment(ASS) continuation - PicoEva.c

what is usually meant by a continuation. In literature, a continuation is usually referred to as a representation of the *entire* execution state of a program at a certain point (i.e. the *entire* call stack). In Pico, a continuation represents a part of program execution (i.e. a *part* or *unit* of what is put on the call stack). During execution, these continuations are placed on what we call the *continuation stack*, which represents the entire future of the computation at hand. Continuations are implemented as pointers to C functions which take no arguments nor return a value. This is demonstrated in figure 6.1 where the source code of the ASS continuation is shown. On line 6, the signature of the function denotes both as return type and parameter type the self-defined _NIL_TYPE_ which is specified as the C native void type [1]. A continuation may invoke other continuations by placing them on the continuation stack. Arguments are not passed as parameters (as can be seen from the function signature in figure 6.1 on line 6); instead they are passed by placing them on a stack called the *expression stack*. The implementation view of both stacks is depicted in figure 6.3. The heart of the Pico interpreter is a loop that continuously executes the continuation located on top of the continuation stack.

Figure 6.2 shows a schema of the possible sequences in which continuations in Pico are executed to evaluate a certain type of expression [2]. To *read* the continuation network, you start in the middle rectangle labelled EXP that symbolises the evaluation of any expression. Depending on the type of expression that needs to be evaluated, a different sequence of continuations needs to be executed. We will not explain the complete continuation network here as this is out of the scope of this dissertation. We will however consider one of the branches to help us explain the ASS continuation from figure 6.1. When we evaluate a *variable definition* in Pico, the DEF continuation is put on the continuation stack (this is marked in figure 6.2 by the red rectangle). This continuation creates a new data slot to store the new variable-value binding in the Pico global

[1] typedef _NIL_TYPE_ void;

[2] This continuation schema is an excerpt from the syllabus used in the second year Bachelor in Computer Science at the Vrije Universiteit Brussel. For more information, please visit http://prog.vub.ac.be/~tjdhondt/ICP2/HTM.dir/introduction.htm

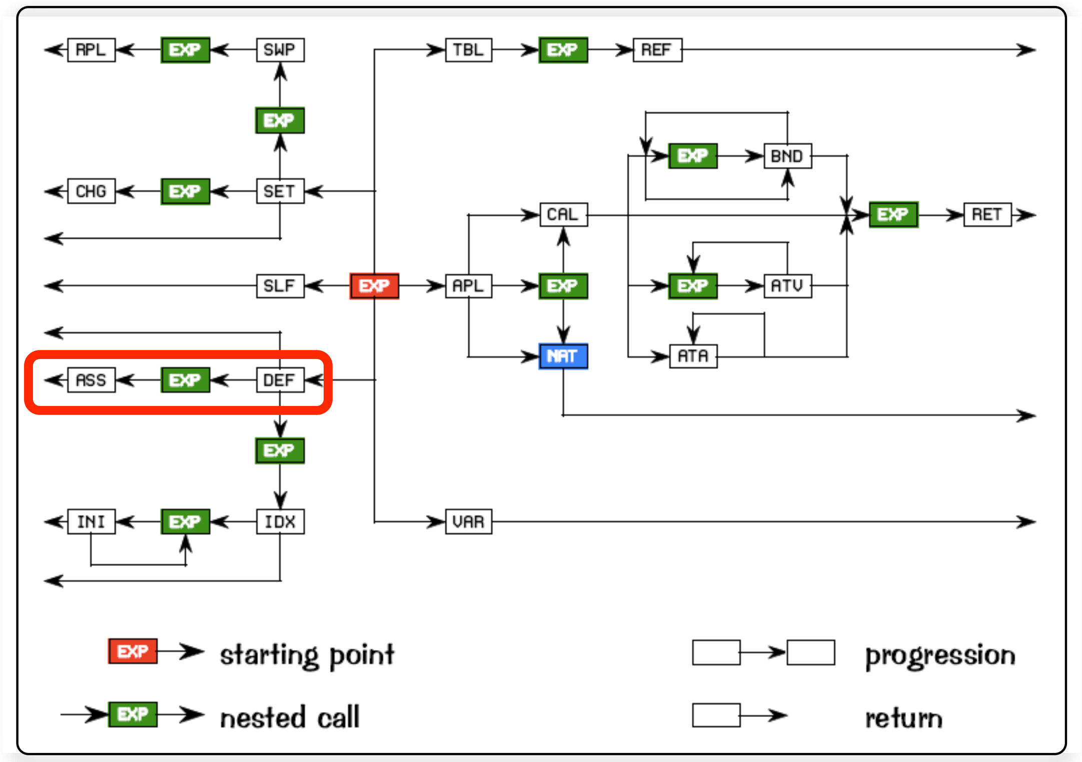


Figure 6.2: The Pico continuation network

environment and already stores the name of the variable in it. This slot is then placed on the expression stack to *pass it on* to the next continuation. The EXP continuation is then pushed on the continuation stack followed by the ASS continuation, which we can deduce from figure 6.2. The former evaluates the expression to be assigned to the variable while the latter performs the actual variable-value assignment and stores the slot in the Pico global environment.

The global environment or global *dictionary* represents the language environment in which variables and functions are stored. It is implemented as a linked list of variable-value pairs (or dictionary slots). Let's have a closer look at the source code of the ASS continuation in figure 6.1. The first five lines denoted here form the documentation of this continuation and represent the state of both stacks *before* and *after* the ASS continuation is executed. The expected elements of the continuation and expression stack are written down between square brackets and separated by spaces. The top of the stacks are located on the right side. The '...' represent possible elements on the stack that are of no importance to the assignment continuation as they are left untouched during its execution. The expected configuration of the stack *before* the execution of the continuation is located to the left of each arrow, while its configuration *after* the execution is located to the right.

To get a feeling of how program execution works in Pico, we consider the following example of evaluating the variable assignment `x:4+7`. The `ASS` continuation is executed right after the expression `'4+7'` was evaluated by the `EXP` continuation. On line 8, the temporary variable `val` is holding the value of that evaluation, which was taken off the expression stack. The `dct` temporary variable on line 9 is holding a new variable-value pair (a new dictionary slot) in which the variable name (`x` in our example assignment) has already been stored by the `DEF` continuation as explained before. On line 10 the resulting value of this evaluation is added to the dictionary value slot and the global dictionary (represented by `_DCT_`) is set as enclosing dictionary of the new `dct` (line 11). Finally, the global dictionary is then set to point to the newly created one on line 12 (which for our example results in the addition of the binding of x to 11 to the global dictionary). On line 8 the value is put again on the expression stack after removing first the `dct` that was still stored on the expression stack [3]. The final statement on line 9 removes the `ASS` continuation from the continuation stack, allowing the next continuation to be executed.

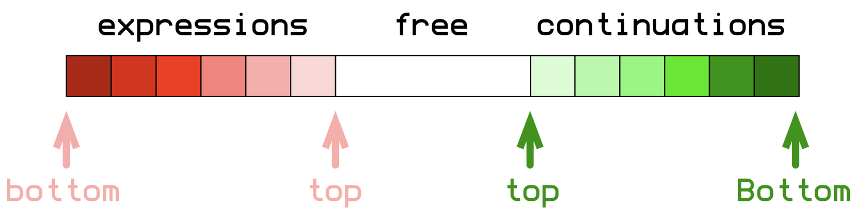


Figure 6.3: Pico execution stacks - Implementation view

6.1.2 The Pico Memory Model

Pico memory consists of a heap where both the expression stack and the continuation stack are stored. Conceptually we distinguish between two separate stacks for storing the continuations and the other one for storing the Pico objects that are passed around between the continuations. In practice however these stacks are stored in heap memory in one array containing both stacks that grow towards each other, see figure 6.3.

The heap memory of Pico is managed by an automatic garbage collector. The garbage collection algorithm can be triggered every time a chunk of memory is requested for storing another object. If no sufficient memory is available, the garbage collector will traverse the Pico environment not only to delete unused objects, but also to move all objects still in use to the beginning of heap memory to defragment it. As a result, memory chunks can be moved to a completely different address in the Pico heap. This change of location happens transparently because the garbage collector also

[3]To represent the frequently performed stack operations in Pico on both the expression and the continuation stack, five operators are used. Next to *push* and *pop*, *peek*, *zap* and *poke* are also used. The *peek* operation just looks at the top of stack, a *zap* operation throws away the top of stack without looking at what is on there and a *poke* is a combination of a *zap* followed by a *push* operation.

updates any references to that chunk. However, this requires that all references to that chunk are also stored on the heap, which is not always the case. In many parts of the Pico implementation, references to chunks of memory on the Pico heap need to be stored in a temporary variable inside a C function. As a result caution should be taken with the use of temporary variables. Since the garbage collector may cause these references to become invalid, we will elaborate on the design concerns of the Pico garbage collector in section 6.4.

6.2 Supporting Development in Pico

In this chapter, we validate the BEHAVE platform by supporting development in Pico through documenting and verifying design invariants. After consulting the original developer of Pico about the main design decisions made during its development, a representative set of three design invariants was identified: active behavioural program documentation, garbage collection and tail recursion optimisation.

The first design invariant we consider in the next section is active behavioural program documentation. It is not a true design invariant as it checks the consistency between actual and documented program behaviour, but it presented us with an initial opportunity to test the BEHAVE platform. Furthermore, checking this particular behaviour demonstrates the need for having precise run-time information available, which justifies the choice of using a dynamic analysis approach (as discussed in chapter 3). The goal of this initial experiment is to check the accurateness of the currently available program documentation and update it accordingly if any inconsistencies are encountered.

Garbage collection is the second design invariant that we investigate. This case study provides an excellent example of a behavioural regularity which crosscuts the source code of an application (as demonstrated in the introductory example in chapter 1). On top of that, garbage collection represents a design concern which is non-externally verifiable. Erroneous garbage collection behaviour results in unpredictable and irregular system behaviour which only in some cases might lead to a severe system crash. The goal here is to identify and pinpoint exactly those scattered source code fragments that might lead to this particular kind of erroneous behaviour.

The third design invariant we document and verify is tail recursion behaviour. Again here we demonstrate the need for precise run-time information for capturing true tail recursion behaviour. As guaranteeing true tail recursion behaviour might be of utmost importance for a virtual machine, we intend to verify with BEHAVE whether this is the case in Pico.

6.3 Case Study 1: Verifying Behavioural Program Documentation

In this section we present the results of an initial experiment performed using the BEHAVE platform [4]. In this experiment, we verify the actual behaviour of the Pico interpreter against available documentation. Although verifying behavioural documentation does not represent a true design invariant, the run-time documentation of the interpreter presented a unique opportunity to initially test our approach. First of all the original developer documented the behaviour thoroughly in a non machine-verifiable format. Second, many changes have been made to the interpreter over time. As this interpreter, together with its behavioural documentation, is nowadays used to introduce computer science students to the foundations of interpretation, it is important to have reliable documentation conveying its dynamics in a concise but descriptive manner. As it is uncertain whether the available documentation still accurately reflects the current behaviour of the interpreter, we intend to identify possible discrepancies between them with as goal to update the behavioural documentation accordingly.

6.3.1 Behavioural Program Documentation in Pico

The internals of the Pico interpreter are documented in a very consistent manner by the developer. A well-defined sequence of continuation and expression stack manipulations determine the operational semantics of each Pico expression. So the interpreter's documentation conveys how these stacks evolve during the evaluation of a program. For each continuation, the documentation describes what the continuation stack and expression stack are expected to look like before and after the execution of the continuation. Consider the documentation of the ASS continuation again (the documentation and implementation was shown and discussed above in section 6.1.1). Its behaviour is documented in terms of expression and continuation stack states *before* and *after* the execution of the continuation:

```
/*-------------------------------------------------------------*/
/*  ASS                                                        */
/*    expr-stack: [... ... ... DCT VAL] -> [... ... ... ... VAL]  */
/*    cont-stack: [... ... ... ... ASS] -> [... ... ... ... ...]  */
/*-------------------------------------------------------------*/
```

This human-readable schema sufficiently documents the semantics of the assignment expression as implemented by the ASS continuation. At a certain point during the evaluation of a program, the Pico driver loop removes the assignment continuation (ASS) from the continuation stack and executes it. The ASS continuation in turn expects a variable environment (represented by the dictionary DCT) and the value that is to be assigned VAL) to be on the expression stack. They are both needed to store a

[4]This experiment was published in 2006 in the proceedings of the International Conference on Program Understanding [RMG$^+$06]

variable-value binding in the current environment. At the end of its execution, the ASS continuation pushes the assigned value VAL on the expression stack, as Pico assignment expressions evaluate to their right-hand side.

6.3.2 Using BEHAVE to Document and Verify the Behaviour of the Pico Interpreter

Over the course of some years, several modifications to the Pico source code have been made. We therefore wanted to verify whether the actual dynamics of the continuation stack still matched the documented behaviour. For this experiment, we first had to formalise the existing documentation and then we used BEHAVE to verify whether it was still loyal to the actual behaviour of the interpreter. The general process overview is depicted in figure 6.5.

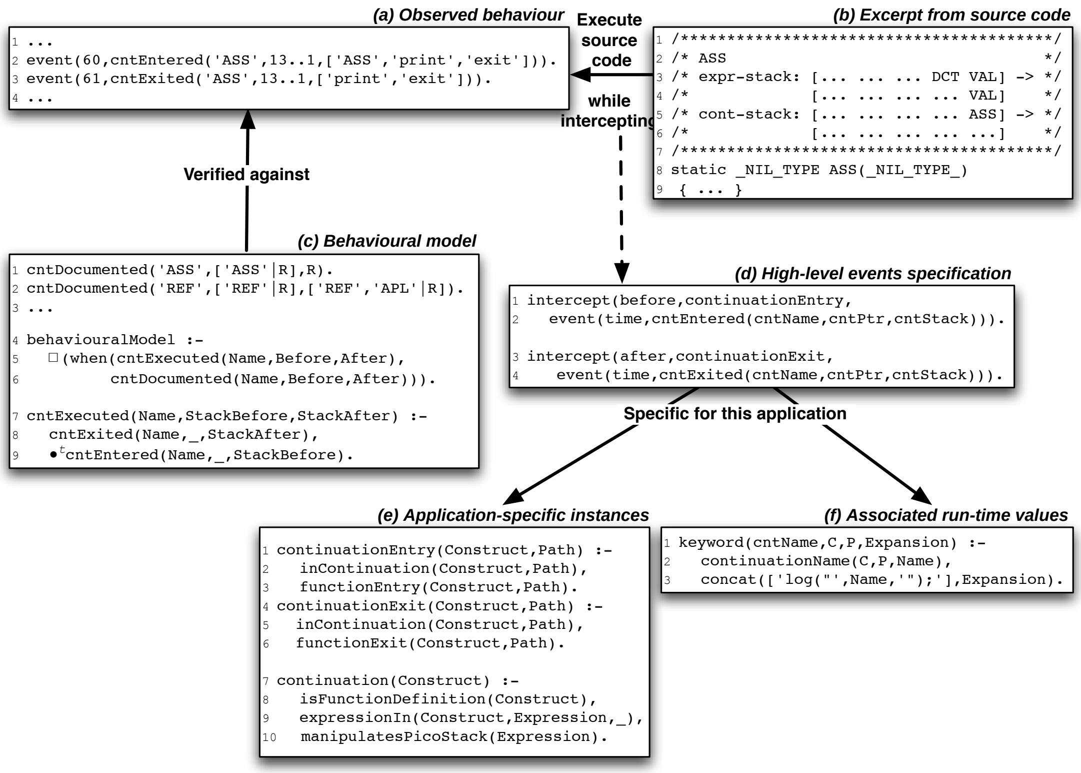
```
                              Before            After
expDocumented('ASS', ['VAL', 'DCT'|R], ['VAL'|R]).
cntDocumented('ASS', ['ASS'|R], R).
                       Before    After
```

Figure 6.4: Logical Representation of Pico Behavioural Program Documentation

For formalising the existing documentation, we first transformed it into a format readable by our platform. The documentation for the ASS continuation as shown above is denoted by the logical facts defined in figure 6.4. The *before* and *after* stack representations denoted in figure 6.4 are captured in Prolog lists with the first elements in the lists representing the top of the stacks. The names of the continuations are quoted because otherwise they would be regarded as Prolog logic variables as they start with a capital letter. The documentation both for the expression stack and the continuation stack is specified in figure 6.4. However in this experiment we describe and verify the state of the continuation stack only. The experiment can be applied analogously to the expression stack as well. The second logical fact denoted above thus specifies for the ASS continuation the state of the continuation stack *before* and *after* its execution. In Prolog, the partial list `['ASS'|R]` matches any list starting with the element `'ASS'` while the rest of the list is bound to the variable R. We use this feature to represent the source code documentation. Now we have this documentation available in a format readable by our platform, we will use BEHAVE to verify it against the actual program behaviour.

Step1: Identifying High-level Run-time Events

Following the four-step recipe outlined in the previous chapter in section 5.5, we first have to identify the high-level events in terms of which we will document the behaviour. Following the original documentation, we chose to model the execution *entry*


(a) Observed behaviour
Execute source code while intercepting
(b) Excerpt from source code
Verified against
(c) Behavioural model
(d) High-level events specification
Specific for this application
(e) Application-specific instances
(f) Associated run-time values

(a) Observed behaviour

```
...
event(60,cntEntered('ASS',13..1,['ASS','print','exit'])).
event(61,cntExited('ASS',13..1,['print','exit'])).
...
```

(b) Excerpt from source code

```
/*****************************************/
/* ASS                                   */
/* expr-stack: [... ... ... DCT VAL] -> */
/*             [... ... ... ... VAL]    */
/* cont-stack: [... ... ... ... ASS] -> */
/*             [... ... ... ... ...]    */
/*****************************************/
static _NIL_TYPE ASS(_NIL_TYPE_)
 { ... }
```

(c) Behavioural model

```
cntDocumented('ASS',['ASS'|R],R).
cntDocumented('REF',['REF'|R],['REF','APL'|R]).
...

behaviouralModel :-
  □(when(cntExecuted(Name,Before,After),
         cntDocumented(Name,Before,After))).

cntExecuted(Name,StackBefore,StackAfter) :-
  cntExited(Name,_,StackAfter),
  ●^tcntEntered(Name,_,StackBefore).
```

(d) High-level events specification

```
intercept(before,continuationEntry,
  event(time,cntEntered(cntName,cntPtr,cntStack))).

intercept(after,continuationExit,
   event(time,cntExited(cntName,cntPtr,cntStack))).
```

(e) Application-specific instances

```
continuationEntry(Construct,Path) :-
  inContinuation(Construct,Path),
  functionEntry(Construct,Path).
continuationExit(Construct,Path) :-
 inContinuation(Construct,Path),
 functionExit(Construct,Path).

continuation(Construct) :-
  isFunctionDefinition(Construct),
  expressionIn(Construct,Expression,_),
  manipulatesPicoStack(Expression).
```

(f) Associated run-time values

```
keyword(cntName,C,P,Expansion) :-
  continuationName(C,P,Name),
  concat(['log("',Name,'");'],Expansion).
```

Figure 6.5: Source code and corresponding behavioural documentation extracts of the Pico interpreter.

and *exit* of a continuation as a high-level event. The run-time values which we will associate with this event are the configuration of the continuation stack before and after the execution of the continuation. The type of events which we will identify to model this behaviour are `cntEntered` and `cntExited`. Figure 6.5(a) shows instances of these events exactly as they will be captured later in the execution trace by executing the instrumented source code. For both types of events, the name of the continuation (denoted by `cntName`), the continuation pointer (`cntPointer`) and the state of the continuation stack (cntStack) represent the associated (run-time) values. These values are specified by *keywords* (as defined in section 5.4.3.2) which are explained in more detail in step 3.

Step2: Specifying the Behavioural Model

We can now specify our behavioural model as assertions in terms of the high-level events we just identified. As the model specification abstract in figure 6.6 summarises, we use logic facts of the form `cntDocumented(CntName,StackBefore,StackAfter)` to document the configuration of the continuation stack before and after the continuation's execution. The first line of the model therefore specifies that at the start of the `ASS` continuation's execution, the `ASS` assignment continuation should be on top of the stack. After its execution, the continuation has to be popped (or zapped) from the stack.

The assertions in our behavioural model need to state the following invariant:

INVARIANT: Active behavioural documentation
Whenever a continuation is executed, the configuration of the continuation stack before and after its execution should match the ones documented in the model.

The actual behaviour model capturing this design invariant is denoted on lines 4–6 of figure 6.6. It holds a temporal relation between the `cntExecuted` and `cnt-Documented` predicates. This relation declares that *every time* when a continuation has been executed with the `Before` and `After` variables holding the continuation stack before and after its execution, there should be a documented continuation holding those same continuation stack configurations. It is important to note that the `Before` and `After` variables used in both predicates on lines 5 and 6 are represented by the **same** variable in order to enforce that the observed complete stack configurations (line 5) agree with their partial specifications (line 6). As we represented these stacks as concrete Prolog lists and partial Prolog lists respectively, we are relying on Prolog's built-in unification algorithm to perform the actual matching.

```
1  cntDocumented('ASS',['ASS'|R],R).
2  cntDocumented('REF',['REF'|R],['REF','APL'|R]).
3  ...

4  behaviouralModel :-
5  □(when(cntExecuted(Name,Before,After),
6         cntDocumented(Name,Before,After))).

7  cntExecuted(Name,StackBefore,StackAfter) :-
8     cntExited(Name,_,StackAfter),
9     ●^tcntEntered(Name,_,StackBefore).
```

Figure 6.6: Specification of the behavioural model for active behavioural program documentation in Pico

The final three lines of the extract in figure 6.6 link the predicates used in the behavioural model to the high-level events that will be observed during the execution of the Pico interpreter. This time, instead of just omitting unwanted information from the recorded events, we are using the temporal operator $\bullet^t$ to express that a continuation has been completely executed once it is exited and was entered t time points ago in the past.

Step3: Application-Specific Instances of High-Level Events

The third step in our recipe consists of a precise specification of the high-level events used in the behavioural program model. The model from the previous section completely relies on two high-level run-time events: the start and the ending of a continuation's execution. More importantly, we are interested in the configuration of the continuation stack at those moments in time. We therefore associate these values with the `cntEntered` and `cntExited` run-time events. The high-level event specifications shown in figure 6.7 declare that these run-time events have to be recorded as facts of the form `cntEntered(cntName,cntPtr,cntStack)` and `cntExited(cntName,cntPtr, cntStack)`.

```
intercept(before,continuationEntry,
   event(time,cntEntered(cntName,cntPtr,cntStack))).

intercept(after,continuationExit,
   event(time,cntExited(cntName,cntPtr,cntStack))).
```

Figure 6.7: High-level event specification of a continuation entry and exit

The definitions of the `continuationEntry` and `continuationExit` predicates, which the high-level event specifications rely on, are depicted in figure 6.8. They are responsible for statically locating those constructs in the Pico interpreter's source code that represent the entry and exit points of a continuation. The `continuationEntry` rule states that a `Construct` is the entry point of a continuation if it is part of a continuation and if it corresponds to an entry point of a function as well. As explained in chapter 5, the `Path` variable represents the path from the program's parse tree root that leads to the parse tree node bound to the `Construct` variable. It is used by the `inContinuation` clause denoted on lines 7–9 of figure 6.8 to check whether the programming language construct is part of a continuation. The `functionEntry` clause on line 3 checks whether `Construct` is the first C statement in a function body (this predicate was explained in section 5.4.4). The `continuation/1` predicate as spec-

```
1  continuationEntry(Construct,Path) :-
2     inContinuation(Path,Construct),
3     functionEntry(Construct,Path).

4  continuationExit(Construct,Path) :-
5     inContinuation(Path,Construct),
6     functionExit(Construct,Path).

7  inContinuation(Path,Construct) :-
8     last(Path,Construct),
9     continuation(Construct).

10 continuation(Construct) :-
11    isFunctionDefinition(Construct),
12    expressionIn(Construct,Expression,_),
13    manipulatesPicoStack(Expression).
```

Figure 6.8: Application-specific instances `continuationEntry` and `continuation-Exit`

ified on lines 10–13 captures the definition of a Pico continuation. Since we know that continuations invoke other continuations and pass around arguments by respectively manipulating the continuation and expression stack, we identify continuations based on the semantical definition shown in figure 6.8 on lines 10–13. The predicate states that a C code construct is a continuation if first of all it is a function and at least one expression in the body of that function manipulates either the continuation or the expression stack. Such expressions are identified in the Pico source code by the `manipulatesPicoStack(Expression)` clause (see appendix D).

Besides the rules described up until now, there are three important keywords our meta model specification relies on: the `cntName`, `cntPtr` and `cntStack` keywords. As stated before in chapter 5, keywords are used to declare *how* the information associated with each high-level run-time event can be retrieved from an application's run-time state. The `cntStack` keyword is responsible for capturing the run-time configuration of the continuation stack. It holds a piece of C code, which is tightly bound to Pico's internals, to walk over the continuation stack. We do not explain this keyword here; we refer the reader to appendix D. Figure 6.9 shows the keywords `cntName`

```
1 keyword(cntName,C,P,Expansion) :-
2    continuationName(C,P,Name),
3    concat(["behaveLog("',Name,'");"],Expansion).

4 keyword(cntPointer,Construct,Path,Result) :-
5    continuationName(Construct,Path,Name),
6    concat(["behaveLog(\"%i\",",Name,");"],Result).
```

Figure 6.9: The `cntName` and `cntPointer` keywords

and `cntPtr`. They differ from the keywords we have seen in the BEHAVE introductory example in the previous chapter. Instead of simply declaring the code it expands to as a logic fact, these keywords are actually logic rules which are allowed to query the application's parse tree to obtain information that is to be incorporated in the expanded source code. The `cntName` keyword shown on lines 1–3 in figure 6.9 obtains the name of a continuation from a program's parse tree since it is impossible to obtain a static function's name at run-time given Ansi C's limited reflective capabilities. The `cntPointer` keyword (lines 4–6) obtains the name of the continuation, but instead of logging that name, the pointer for that continuation is logged at run-time. Note that, because of the limited reflective capabilities of C, the same problem occurs for obtaining the state of the continuation stack through the `cntStack` keyword. In practice, we obtain a representation of the continuation stack that contains the *pointers* of the continuations instead of the continuation names as is needed for the lightweight verification against the documented stack representations. However, as we obtain both the name of each continuation and its pointer through the `cntName` and `cntPointer` keywords respectively, a logic rule is easily formulated to transform these pointers into their corresponding names.

Step4: Lightweight Consistency Verification

So as to verify whether the Pico interpreter indeed behaves as indicated by its documentation, we used the Pico interpreter to evaluate the Pico program as depicted in figure 6.11 containing a representative set of Pico expression types. This well-chosen execution scenario represents a Pico implementation of coroutines from Modula-2. This way, we obtained high-level execution traces containing information about the dynamics of the continuation stack. An example of such an execution trace is listed in figure 6.10. Note that for simplicity reasons we depicted the *transformed* list representation of the continuation stack (i.e. holding the names of the continuations instead of their pointer) as was explained right above. We verified whether these traces con-

```
...
event(60,cntEntered('ASS',13..1,['ASS','print','exit'])).
event(61,cntExited('ASS',13..1,['print','exit'])).
...
```

Figure 6.10: Excerpt of an execution trace logging all continuation entries and exits

form to our model specification by launching the logic query `?- behaviouralModel`. By doing so, several interesting conflicts were found between Pico's documented behaviour and the behaviour that was observed. One of them was located in the documentation of the `REA` continuation which is executed when an expression is read. The documentation indicated that this continuation just takes off the top of the continuation stack during its execution, while in reality the `EXT` and `EXP` continuations were placed on top of the stack. Other discrepancies were found in the documentation of three other continuations: `EXp`, `INV` and `FCT`. In the `EXp` continuation, `EXP` was used instead of `EXp` [5]. In the documentation of both the `INV` and `FCT` continuation, their auxiliary continuations were wrongly depicted.

In addition, several naming inconsistencies were detected. The `_eval_exp_` continuation was for instance abbreviated in the documentation as `EXP`, but another continuation already had this name. Such inconsistencies are very likely to confuse programmers and is especially troublesome for didactic purposes, such as students studying the documentation. Upon interpretation of our verification results, we were able to adapt the machine-verifiable behavioural documentation to the actual program behaviour. Since this documentation is at least as descriptive as the original documentation in the source code comments, the machine-verifiable documentation has now been adopted as the official documentation of the interpreter. This allows the user to keep the documentation and source code in sync as their consistency can easily be verified after future modifications to the interpreter.

[5]In Pico, a naming convention is used where all main continuations are denoted with three uppercase letters (for example MAI), while auxiliary continuations which are used inside a main continuation are denoted with one lowercase letter (MAi in the example).

```
{ newcoroutine(body()):
    { my_continuation: void;
      handler(value):
        if(is_continuation(value),
           my_continuation:= value,
           call({ hold: my_continuation;
                      value(continuation);
                      continue(hold, handler) }));
      call({ my_continuation:= continuation;
             handler(handler);
             body() }) };
  transfer(p, q):
    q(p);
  p: q: void;
  p:= newcoroutine(while(true,
                           { display("ping", eoln);
                             transfer(p,q) }));
  q:= newcoroutine(while(true,
                           { display("pong", eoln);
                             transfer(q,p) })) ;
  transfer(p, p) }
```

Figure 6.11: A well-chosen execution scenario: A Pico implementation of coroutines from Modula-2

Evaluation

The experiment demonstrated that the introduction of high-level events in the interpreter's behavioural program model resulted in machine-verifiable documentation that was as declarative as the original. At the same time, the introduction of carefully selected high-level events in the execution traces resulted in relatively compact traces allowing for a more lightweight verification.

For this experiment, the interpretation of the verification results resulted in adapting the behavioural documentation. However, also the actual behaviour might be wrong and then the source code would have to be adapted. This approach shows us inconsistencies between actual and wanted behaviour; it is left up to the user of the platform to interpret what is causing this inconsistency.

Finally, we would like to conclude this evaluation with a side note. Although the behavioural model of the Pico interpreter as presented here only expresses desirable behaviour, we mostly use our verification platform to detect instances of undesirable behaviour, which will become clear in the next case studies. To specify unwanted behaviour, one must express the dynamic conditions leading to a crash or to a possible code optimisation. In most cases this provides the user with direct access to those places needed for correcting the discovered unwanted behaviour.

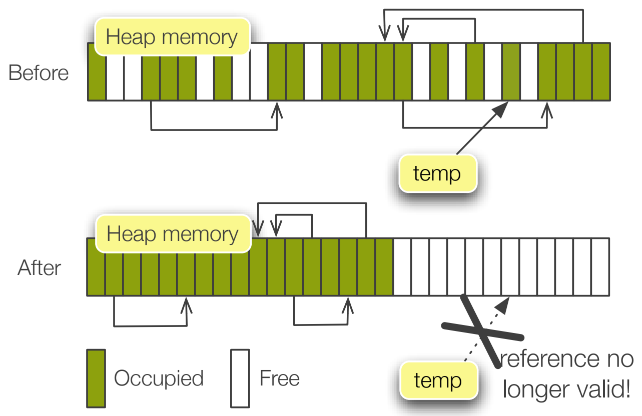


Figure 6.12: Conceptual representation of heap memory before/after a garbage collect

6.4 Case Study 2: Verifying Garbage Collection

In this experiment we document a particular behaviour of the Pico memory model dealing with automatic garbage collection [6]. In section 6.1.2 we explained the Pico memory model. The problem with garbage collection behaviour is summarised in the conceptual representation depicted in figure 6.12. The picture shows a particular state of heap memory before and after a garbage collection occurred. In the *before* schema a limited amount of consecutive free slots (the white slots) are available and when memory would be requested to store for example a chunk of size 3, the garbage collection algorithm would be triggered [7]. The state of that same heap after the garbage collection algorithm is triggered is depicted in the *after* schema. The algorithm moves all used memory to the bottom of the heap in order to free and compact unused memory. As a result the addresses of chunks of memory might change. This poses no problems for Pico objects whose references are also stored on the heap. For temporary C variables however, as such references are *not* stored as Pico variables, a garbage collect might result in temporary variables that are referencing invalid locations (as shown in the *after* schema in figure 6.12). Hence whenever the possibility of a garbage collection exists, the temporary variables should *always* be restored before using them.

Obviously, we want to capture such behaviour and verify if the execution of Pico exhibits such behaviour. This means that we need to detect situations where a garbage collect occurs in between the assignment and use of a temporary variable.

Note that functional testing is insufficient here because such a test only fails if a garbage collect actually defragments memory. Garbage collection is a concrete and

[6]This experiment has been previously published in 2006 in the proceedings of the International Conference on Software Engineering and Knowledge Engineering [MRB$^+$06]

[7]For storing a chunk of size 3 in heap memory, 4 consecutive free slots have to be available because a first extra slot needs to store the size of the chunk. For more in depth information about the Pico memory model we refer to the syllabus of the Interpretation II course: http://prog.vub.ac.be/˜tjdhondt/ICP2/HTM.dir/notes.htm

difficult problem which is very time-intensive and for which standard debugging techniques do not work. Depending on the actual memory consumption and organisation, such tests thus might fail or not, although the erroneous behaviour of a *possible* garbage collect is always present. Therefore, we need to test the *possible* occurrences of a garbage collect in between the assignments of and the references to a temporary variable inside a C function. To this extent, we specified a model that captures this particular kind of behavioural design concern and we will verify this model against the actual program behaviour using the BEHAVE platform.

6.4.1 Using BEHAVE to Document and Verify Garbage Collection in Pico

Figure 6.13 provides a general overview of the experiment. As explained in section 6.1.1, the Pico implementation consists of many continuations which all represent a particular part of program execution. As temporary variables might be used incorrectly within these continuations, we have to watch their actual behaviour within the continuations.

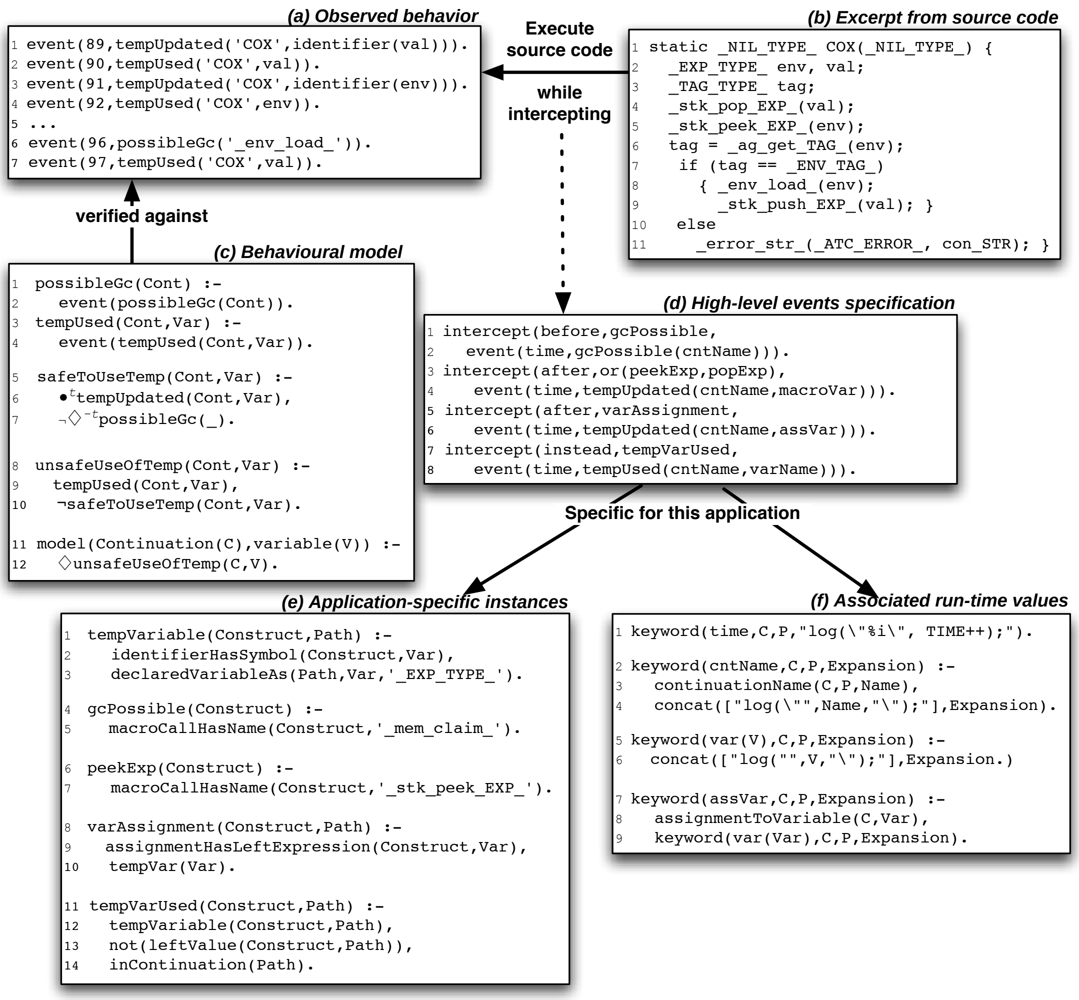
Figure 6.13(b) shows an excerpt from the Pico source code. The execution of that code results in the creation of an execution-trace (shown in figure 6.13(a)) that only contains the observed behaviour in terms of high-level events. These high-level events are used to verify the behavioural model of the invariant specified in figure 6.13(c). Finally, figure 6.13(d), 6.13(e) and 6.13(f) specify how the high-level events need to be recorded during the execution of the program. We further explain the details of each of these parts throughout the remainder of the section.

Step1: Identifying High-Level Run-Time Events

The first step of our recipe comprises the identification of the high-level run-time events needed to verify the garbage collection invariant. Having analysed the problem description, we need to specify the following high-level run-time events:

- When might a possible garbage collect occur (`possibleGC`),
- When is a temporary variable used (`tempUsed`),
- When is a temporary variable being assigned a value (update or initialisation) (`tempUpdated`).

Indeed, we want to detect occurrences of possible garbage collection events *in between* the assignment and use of temporary variables holding a reference to the Pico memory. In figure 6.14 some instances of these events are shown as they will appear later in the execution trace. Besides a time stamp, the needed associated values are the name of the continuation and the name of the temporary variable.



(a) Observed behavior

```
event(89,tempUpdated('COX',identifier(val))).
event(90,tempUsed('COX',val)).
event(91,tempUpdated('COX',identifier(env))).
event(92,tempUsed('COX',env)).
...
event(96,possibleGc('_env_load_')).
event(97,tempUsed('COX',val)).
```

(b) Excerpt from source code

```
static _NIL_TYPE_ COX(_NIL_TYPE_) {
  _EXP_TYPE_ env, val;
  _TAG_TYPE_ tag;
  _stk_pop_EXP_(val);
  _stk_peek_EXP_(env);
  tag = _ag_get_TAG_(env);
   if (tag == _ENV_TAG_)
     { _env_load_(env);
       _stk_push_EXP_(val); }
   else
     _error_str_(_ATC_ERROR_, con_STR); }
```

(c) Behavioural model

```
possibleGc(Cont) :-
   event(possibleGc(Cont)).
tempUsed(Cont,Var) :-
   event(tempUsed(Cont,Var)).

safeToUseTemp(Cont,Var) :-
   ●ᵗtempUpdated(Cont,Var),
   ¬◇⁻ᵗpossibleGc(_).

unsafeUseOfTemp(Cont,Var) :-
  tempUsed(Cont,Var),
  ¬safeToUseTemp(Cont,Var).

model(Continuation(C),variable(V)) :-
   ◇unsafeUseOfTemp(C,V).
```

(d) High-level events specification

```
intercept(before,gcPossible,
   event(time,gcPossible(cntName))).
intercept(after,or(peekExp,popExp),
    event(time,tempUpdated(cntName,macroVar))).
intercept(after,varAssignment,
    event(time,tempUpdated(cntName,assVar))).
intercept(instead,tempVarUsed,
    event(time,tempUsed(cntName,varName))).
```

(e) Application-specific instances

```
tempVariable(Construct,Path) :-
   identifierHasSymbol(Construct,Var),
   declaredVariableAs(Path,Var,'_EXP_TYPE_').

gcPossible(Construct) :-
   macroCallHasName(Construct,'_mem_claim_').

peekExp(Construct) :-
   macroCallHasName(Construct,'_stk_peek_EXP_').

varAssignment(Construct,Path) :-
   assignmentHasLeftExpression(Construct,Var),
   tempVar(Var).

tempVarUsed(Construct,Path) :-
   tempVariable(Construct,Path),
   not(leftValue(Construct,Path)),
   inContinuation(Path).
```

(f) Associated run-time values

```
keyword(time,C,P,"log(\"%i\", TIME++);").

keyword(cntName,C,P,Expansion) :-
   continuationName(C,P,Name),
   concat(["log(\"",Name,"\");"],Expansion).

keyword(var(V),C,P,Expansion) :-
  concat(["log("",V,"\");"],Expansion.)

keyword(assVar,C,P,Expansion) :-
   assignmentToVariable(C,Var),
   keyword(var(Var),C,P,Expansion).
```

Figure 6.13: Using the BEHAVE platform for verifying Pico memory management

```
event(89,tempUpdated('COX',val)).
event(90,tempUsed('COX',val)).
event(91,tempUpdated('COX',env)).
event(92,tempUsed('COX',env)).
...
event(96,possibleGc('_env_load_')).
event(97,tempUsed('COX',val)).
```

Figure 6.14: Excerpt of an execution trace logging possible garbage collects and temporary variable usage

Step2: Specifying the Behavioural Model

We can now specify an invariant model using temporal logic clauses that describes the desired as well as the undesired behaviour of the program with respect to these events. In section 6.1, we described the Pico memory model and we explained that a possible garbage collect event should not occur between the assignment (which is either an update or an initialisation) and the use of a temporary variable.

INVARIANT: Garbage Collection
After a possible garbage collection occurred, you should not use temporary variables, unless they have been updated.

Expressed as unwanted behaviour, we will look for all uses of temporary variables that do not obey this last statement. The model that specifies this behaviour is shown in figure 6.15. Lines 1-4 introduce additional abstractions in terms of the events in the execution trace, i.e. `possibleGC` and `tempUsed` [8]. On lines 5–10 the concepts of unsafe and safe uses of temporary variables are defined as the logic assertions `unsafeUseOfTemp` and `safeToUseTemp` respectively. This last assertion (defined on lines 5–7) states that it is safe to use a temporary variable `Var` within a continuation `Cnt` if within `t` time steps in the past from now, the variable `Var` has been updated (or given an initial value if it did not have a value yet) and if within that time frame no possible garbage collection could have occurred. The `unsafeUseOfTemp(Cnt,Var)` assertion on lines 8–10 captures the *unsafe* use of a temporary variable that states the complement. The actual model of undesirable behaviour can be seen on lines 11–12 and expresses the wish of finding a variable `V` within a continuation `C` that is used in an unsafe way, as it is defined on lines 8–10.

Step3: Specifying Application-Specific Instances

In this step of the recipe we describe how we can specify *which* events have to be intercepted at run-time and *how* they should be recorded. This consists of specifying

[8]Note that we also need an identical abstraction for `tempUpdated(Cnt,Var)`. The declaration implementing this abstraction can be consulted in appendix D.

```
possibleGc(Cont) :-
   event(possibleGc(Cont)).
tempUsed(Cont,Var) :-
   event(tempUsed(Cont,Var)).

safeToUseTemp(Cont,Var) :-
    ●^t tempUpdated(Cont,Var),
    ¬◇^-t possibleGc(_).
unsafeUseOfTemp(Cont,Var) :-
   tempUsed(Cont,Var),
   ¬safeToUseTemp(Cont,Var).

behaviouralModel(Continuation(C),variable(V)) :-
     ◇unsafeUseOfTemp(C,V).
```

Figure 6.15: Specification of the behavioural model for verifying garbage collection in Pico

what source-code constructs raise these high-level events and how associated run-time values need to be extracted in order to be recorded with the event. The `intercept` dec-

```
1 intercept(before,gcPossible,
2    event(time,gcPossible(cntName))).
3 intercept(after,or(peekedExp,poppedExp),
4    event(time,tempUpdated(cntName,macroVar))).
5 intercept(after,varAssignment,
6    event(time,tempUpdated(cntName,assVar))).
7 intercept(instead,tempVarUsed,
8    event(time,tempUsed(cntName,varName))).
```

Figure 6.16: High-level event specification of garbage collection behaviour

larations denoted in figure 6.16 describe the high-level events that need to be recorded. All declarations are of the form `intercept(When,What,RecordAs)`. Consider lines 1–2 where we declare that the high-level event of a possible garbage collect is recorded in the execution trace as `event(time,possibleGc(cntName))`. More precisely, this event will be recorded right `before` the execution of the source code construct that is identified by the `gcPossible` assertion. This assertion is defined by the logic clause on lines 4–5 in figure 6.17. The `gcPossible` declaration specifies that the high-level event of a possible garbage collect is triggered by the execution of the source-code construct `Construct` if it is a C macro call that is named `_mem_claim_`[9].

Furthermore, as usual we also log the `time` at which the event occurred to be able to reason about the order of events and the name of the continuation in which the possible garbage collect event occurs (through the keyword `cntName`).

[9]There are three other macro calls in the Pico implementation besides _mem_claim_ that can trigger the garbage collection property: `_stk_claim_`, `_mem_claim_SIZ_` and `_mem_claim_STR_`. We did not include all of them here because their assertions are very similar to the `_mem_claim_` assertion; they are included in appendix D

On lines 3–6, the same event (`event(time,tempUpdated(cntName,varName))`) is described by two different logic declarations. This is because the high-level event of a temporary variable being (re-) assigned a value can manifest itself in the source code in various ways. More specifically, this event is triggered by the execution of the source code construct that is specified by the `peekExp`, the `popExp` or the `varAssignment` assertions. Another difference between the two clauses has to do with the different run-time values that are needed. These are represented by the keywords (`macroVar` and `assVar`. They all represent variable names, but depending on the type of C construct they should be retrieved in a different way. On lines 1–3 of figure 6.17, a temporary

```
1  tempVariable(Construct,Path) :-
2     identifierHasSymbol(Construct,Var),
3     declaredVariableAs(Path,Var,'_EXP_TYPE_').

4  gcPossible(Construct) :-
5     macroCallHasName(Construct,'_mem_claim_').

6  peekExp(Construct) :-
7     macroCallHasName(Construct,'_stk_peek_EXP_').

8  varAssignment(Construct,Path) :-
9     assignmentHasLeftExpression(Construct,Var),
10    tempVar(Var).

11 tempVarUsed(Construct,Path) :-
12    tempVariable(Construct,Path),
13    not(leftValue(Construct,Path)),
14    inContinuation(Path).
```

Figure 6.17: Application-specific instances for the garbage collection design invariant

variable is defined by the `tempVariable/2` predicate. In essence, some source code construct is a temporary variable if it is an identifier with name `Var` and if it has been declared as being of type `_EXP_TYPE_`. The `tempVarUsed` rule on lines 11–14 states that a temporary variable (denoted by the variable `Construct`) has been used if first of all it is a temporary variable (line 12), if it is used within a continuation (line 14) and if it is not part of the left side of a variable assignment (line 13), as in that case it is updated rather than used. The needed associated values of the high-level events mainly have to

```
keyword(var(V),C,P,Expansion) :-
   concat(["log("",V,"\");"],Expansion.)

keyword(assVar,C,P,Expansion) :-
   assignmentToVariable(C,Var),
   keyword(var(Var),C,P,Expansion).
```

Figure 6.18: The `var(V)` and `assVar` keywords

be obtained by the execution of application-specific source code. From the `intercept` predicates previously defined, the name of the continuation denoted by the keyword `cntName` and the `time` keyword appear in every high-level event specification. As we have previously explained and used these keywords in our previous case study in section 6.3, we reuse them here in the high-level specifications necessary for verifying garbage collection behaviour.

The two keyword declarations specified in figure 6.18 are new: they are used in combination to be able to capture the name of a variable which is the left-hand part of an assignment statement. Note that we need two identical rules to the rule denoted on lines 3–5 for the keywords `macroVar` and `varName`; we refer to appendix D for their specification.

Step4: Lightweight Consistency Verification

After instrumenting the Pico source code, it gets executed according to a well-chosen execution scenario [10] which creates the execution trace containing the high-level run-time events that occurred during execution. Consequently the following logic query is launched:

```
?- behaviouralModel(Function,Variable).
```

A failure of this query would mean that no variables can be found that are used in an unsafe way, which is exactly what is wanted. If the query does not fail, it will return those continuations together with the names of temporary variables that are used in an unsafe way inside that continuation. Three occurrences of unsafe usage of variables were found in two different continuation functions. Let us consider one of the two results: `continuation('COX'), variable(val)`. This result means that the temporary variable `val` in the `COX` continuation is used in an unsafe way.

If a garbage collect event occurred that triggers a heap defragmentation during the execution of the `COX` continuation, the system would exhibit serious erroneous behaviour or even trigger a system crash. The C code fragment representing this function is depicted in figure 6.19. On line 9 the temporary variable `val` is used and apparently the statement on line 8, a call to the function `_env_load_` might trigger a garbage collect (as can be seen on the previously shown execution trace in figure 6.14 on line 6).

Evaluation

This case study is a prime example of what we have referred to before as a *cross-cutting behavioural invariant* of a system. Garbage collection as a design invariant cannot simply be locally annotated in the code (like embedded assertions as discussed in section 2.4.1) as it depends on behaviour that is spread around the entire system.

[10]As mentioned for our first case study, any particular scenario that lets Pico evaluate a representative member of different expressions so that most of the continuations are executed will do. Consequently we used the same input scenario as for the previous case study. This Pico input program is depicted in figure 6.11

```
static _NIL_TYPE_ COX(_NIL_TYPE_) {
   _EXP_TYPE_ env, val;
   _TAG_TYPE_ tag;
   _stk_pop_EXP_(val);
   _stk_peek_EXP_(env);
   tag = _ag_get_TAG_(env);
   if (tag == _ENV_TAG_)
      { _env_load_(env);
        _stk_push_EXP_(val); }
   else
     _error_str_(_ATC_ERROR_, con_STR); }
```

Figure 6.19: The `COX` continuation from PicoNat.c

And although the system developer knew this concern all too well, taking into account this particular behaviour during development, without any automated support, proved to be unfeasible. On top of that, trying to track this kind of behaviour manually after finalising the implementation proved to be extremely time-consuming. To emphasise the difficulty of the garbage collection design invariant, we refer to future work (section 7.3.4) where we propose to treat garbage collection as an aspect.

Using BEHAVE for documenting and verifying a second design invariant of Pico proved to be already more practical and faster. Because of concept reuse, one can more easily identify and use the constituents of the high-level events. *Reuse of application-specific instances* was possible (the `continuation` predicate defining the concept of a continuation) and *reuse of keywords* for retrieving the associated values. For supporting other design invariants in Pico, a user can consult appendix D as it documents all reusable predicates for reasoning about the behaviour of Pico.

6.5 Case Study 3: Verifying Tail Recursion Optimisation

In this section a third Pico design invariant is studied: the principle of tail recursion optimisation. In this experiment we intend to verify whether the Pico virtual machine exhibits real tail recursion behaviour, i.e. whether stack usage is optimised for a tail recursive function. Recursion in general is a method of defining functions in which the function to be defined is applied within its own function body. Tail recursion is a special case of recursion that can be easily transformed into an iteration. As an example, consider the Pico implementation of the tail-recursive version of the factorial function depicted in figure 6.20. The last expression in the body of the `iterate` function (line 5) consists of a recursive call. To make a distinction between recursion and tail recursion, let us consider the regular recursive version of the same factorial function depicted in figure 6.21 yielding the same results as the tail-recursive version. The main difference between both versions lies in the lines of code where the function recursively calls

```
1 factorial(n):{
2    iterate(n,acc):
3       if(n=0,
4          acc,
5          iterate(n-1,acc*n));
6    iterate(n,1)};
```

Figure 6.20: Tail-recursive factorial function

itself. In the recursive version the result of the call to factorial should afterwards be multiplied by n (line 4 in figure 6.21), while in the tail-recursive version (line 5 in figure 6.20) the result of evaluating the function body *reduces* to evaluating the recursive call. Therefore tail recursion can be regarded as an optimisation of recursion: in the context of an interpreter implementation there is no need to save and restore the environment around a tail-recursive call: the result of calling the function reduces to the result of the tail-recursive call.

```
1 factorial(n):
2     if(n=0,
3        1,
4        n*factorial(n-1));
```

Figure 6.21: Recursive factorial function

6.5.1 Tail Recursion in Pico

In section 6.1.1, we elaborated on the Pico execution model and we explained that it consists of a collection of continuations which represent a particular part of the Pico evaluation process. Continuations are placed on the continuation stack whenever their execution is needed.

In Pico, the return continuation `RET` represents that part of program behaviour that restores the environment (in Pico an environment is referred to as a *dictionary*) that was saved on the expression stack. A return continuation is placed on the continuation stack whenever an environment needs to be restored after a certain evaluation has taken place for which another environment was needed. Consider for example when a function application needs to be evaluated, the body of the function needs to be evaluated in the dictionary representing the environment at function creation time augmented with the bindings of the formal parameters to the actual parameters. But after creating that dictionary as the environment to evaluate the function body in, the former environment needs to be restored again. Figure 6.22 denotes the source code of the return continuation `RET`. This function takes the previously saved dictionary from the expression stack and puts the stored value -representing the result of evaluating the last Pico expression-

```
/*-------------------------------------------------------------*/
/*  RET                                                        */
/*     expr-stack: [... ... ... DCT VAL] -> [... ... ... ... VAL] */
/*     cont-stack: [... ... ... ... RET] -> [... ... ... ... ...] */
/*-------------------------------------------------------------*/
static _NIL_TYPE_ RET(_NIL_TYPE_)
 { _EXP_TYPE_ val;
   _stk_pop_EXP_(val);
   _stk_peek_EXP_(_DCT_);
   _stk_poke_EXP_(val);
   _stk_zap_CNT_();
   _ESCAPE_; }
```

Figure 6.22: The return continuation (`RET`) from PicoEva.c

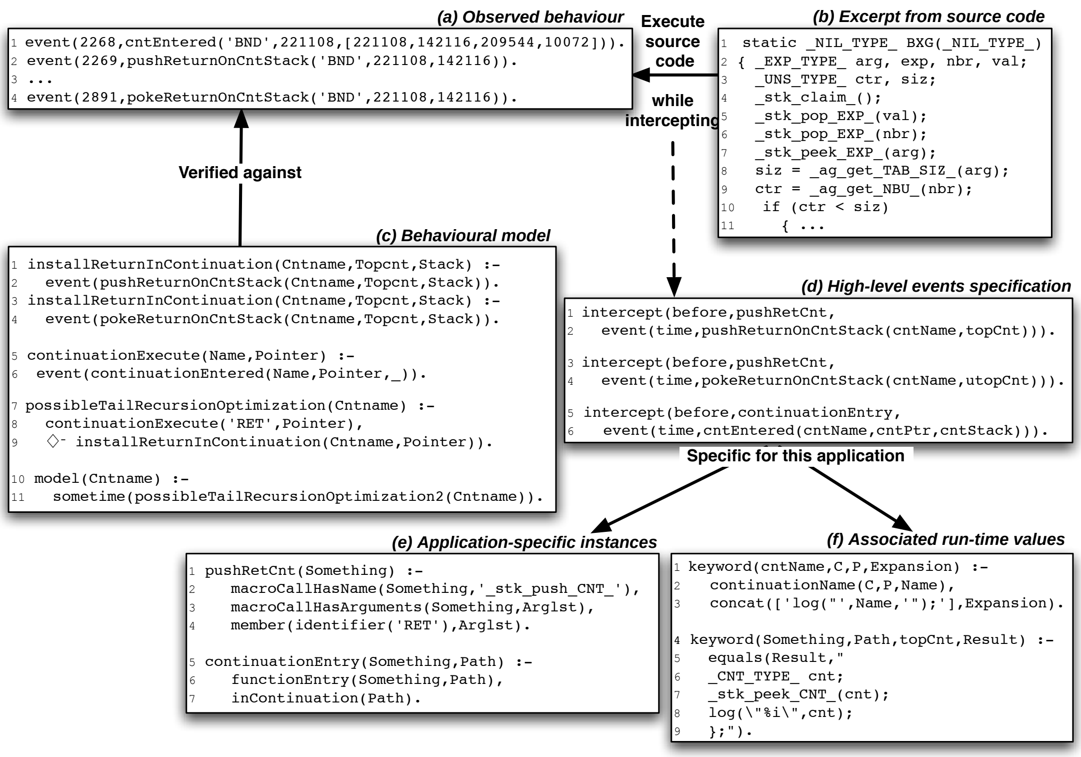
on top of the expression stack. When a function is tail-recursive, all what needs to be done after obtaining the result of a recursive call is returning that result. Nothing needs to be evaluated anymore, so the dictionary is no longer needed. Without optimising tail recursion, many unnecessary returns would have to be executed, as demonstrated in figure 6.23 by calculating `factorial(4)`. This call structure is absolutely essen-

```
factorial(4)
iterate(4,1)
  iterate(3,1*4)
    iterate(2,4*3)
       iterate(1,12*2)
         iterate(0,24)
       restore(env) & keep 24 on stack
    restore(env) & keep 24 on stack
  restore(env) & keep 24 on stack
restore(env) & keep 24 on stack
=> return(24)
```

Figure 6.23: Non-optimised tail-recursive call structure

tial for a recursive function since at every restore step the environment is needed for evaluating the multiplication by n.

In this experiment, we document tail recursion behaviour in order to be able to identify possible places in the code where tail recursion is not yet optimised. Note that this experiment clearly demonstrates the need for behavioural information as identifying tail recursion optimisation amounts to investigating the continuation stack's run-time state at very specific moments during program execution. Formulated more precisely, we need to investigate *which continuation is on top of the stack* at the time a `RET` continuation is placed on the stack. In case the continuation on top of the stack is a *RET* continuation as well, then tail recursion is not optimised.



(a) Observed behaviour

```
event(2268,cntEntered('BND',221108,[221108,142116,209544,10072])).
event(2269,pushReturnOnCntStack('BND',221108,142116)).
...
event(2891,pokeReturnOnCntStack('BND',221108,142116)).
```

(b) Excerpt from source code

```
 static _NIL_TYPE_ BXG(_NIL_TYPE_)
{ _EXP_TYPE_ arg, exp, nbr, val;
  _UNS_TYPE_ ctr, siz;
  _stk_claim_();
  _stk_pop_EXP_(val);
  _stk_pop_EXP_(nbr);
  _stk_peek_EXP_(arg);
  siz = _ag_get_TAB_SIZ_(arg);
  ctr = _ag_get_NBU_(nbr);
   if (ctr < siz)
     { ...
```

(c) Behavioural model

```
installReturnInContinuation(Cntname,Topcnt,Stack) :-
  event(pushReturnOnCntStack(Cntname,Topcnt,Stack)).
installReturnInContinuation(Cntname,Topcnt,Stack) :-
  event(pokeReturnOnCntStack(Cntname,Topcnt,Stack)).

continuationExecute(Name,Pointer) :-
 event(continuationEntered(Name,Pointer,_)).

possibleTailRecursionOptimization(Cntname) :-
  continuationExecute('RET',Pointer),
  ◇- installReturnInContinuation(Cntname,Pointer)).

model(Cntname) :-
  sometime(possibleTailRecursionOptimization2(Cntname)).
```

(d) High-level events specification

```
intercept(before,pushRetCnt,
  event(time,pushReturnOnCntStack(cntName,topCnt))).

intercept(before,pushRetCnt,
  event(time,pokeReturnOnCntStack(cntName,utopCnt))).

intercept(before,continuationEntry,
  event(time,cntEntered(cntName,cntPtr,cntStack))).
```

(e) Application-specific instances

```
pushRetCnt(Something) :-
   macroCallHasName(Something,'_stk_push_CNT_'),
   macroCallHasArguments(Something,Arglst),
   member(identifier('RET'),Arglst).

continuationEntry(Something,Path) :-
   functionEntry(Something,Path),
   inContinuation(Path).
```

(f) Associated run-time values

```
keyword(cntName,C,P,Expansion) :-
  continuationName(C,P,Name),
  concat(['log("',Name,'");'],Expansion).

keyword(Something,Path,topCnt,Result) :-
  equals(Result,"
  _CNT_TYPE_ cnt;
  _stk_peek_CNT_(cnt);
  log(\"%i\",cnt);
  };").
```

Figure 6.24: Using the BEHAVE platform for identifying tail recursion optimisation

6.5.2 Using BEHAVE for Verifying Tail Recursion Optimisation

We again use our BEHAVE platform by applying the four-step recipe as demonstrated in section 5.5. An overview of the complete BEHAVE set-up for tail recursion optimisation is depicted in figure 6.24. We explain all needed components one by one in the following subsections.

Step1: Identifying High-Level Run-time Events

As a first step we identify the concepts needed to reason about tail recursion behaviour. For detecting needed tail recursion optimisation, consecutive calls of the return continuation have to be detected. This is achieved by capturing run-time events which place a return continuation on the continuation stack (`placeReturnOnContinuationStack`). Next to identifying this high-level event, the associated run-time values play an important role as well. We capture at run-time what is on top of the continuation stack right before placing the return continuation on the stack.

Step2: Specifying the Behavioural Model

In this experiment we document and verify in a lightweight manner the following design invariant:

INVARIANT: Tail Recursion Optimisation
When a return continuation is placed on the continuation stack, there should not be another return continuation on top of the continuation stack

Using the concepts we defined in section 6.5.2, this model seems very straightforward in that we only need to watch when a return continuation is placed on the continuation stack and at that time check the top of the stack. The model for this design invariant is specified in figure 6.25. The `installReturnOnContinuationStack`

```
1 installReturnOnContinuationStack(Cntname,Topcnt) :-
2    event(pushReturnOnCntStack(Cntname,Topcnt)).
3 installReturnOnContinuationStack(Cntname,Topcnt) :-
4    event(pokeReturnOnCntStack(Cntname,Topcnt)).

5 possibleTailRecursionOptimisation(Cntname) :-
6    sometime(and(installReturnOnContinuationStack(Cntname,TopOfStack),
7                 TopOfStack='RET')).
```

Figure 6.25: The behavioural model for capturing tail recursion optimisation

declarations on lines 1–4 define an extra abstraction over both stack operations events `push` and `poke` that can place a continuation on the continuation stack. The behavioural model describing the design invariant on lines 5–7 declares that in the body of a continuation with name `Cntname`, at some point during program execution, a `RET` continuation is placed on the continuation stack and that at that time *there already is* a `'RET'` continuation on top of the stack. Note that we again specify the unwanted behaviour such that execution can identify those places where we might be able to optimise the unwanted behaviour. If at some point tail recursion is not optimised, we would get as a result the name of a continuation `Cntname` in which tail recursion needs to be optimised.

However, an implementation detail forces us to slightly adapt the behavioural model. Because of the limited reflective capabilities of Ansi C, it is impossible to directly retrieve the name of a static function at run-time when you have only the pointer to the function available (this was previously explained in section 6.3.2). So on line 7 in figure 6.25, the `TopOfStack` variable can never be equal to `'RET'` as its holds a pointer instead of a string. Instead we use the model shown in figure 6.26 by using an additional high-level run-time event `continuationExecute`. This predicate is identical to the `continuationEntered` predicate from the first case study. We provide an additional name abstraction to make our intent of looking for a function to be executed more clear. This model specifies the need for tail recursion optimisation if, during program execution the `RET` continuation is executed (this retrieves the pointer of the `RET` continuation) and sometime before that (sometime in the past), a `RET` continuation is placed on top of the stack. The link between both entities is provided by having the

```
1 possibleTailRecursionOptimisation(Cntname) :-
2    continuationExecute('RET',Pointer),
3    sometime(-,installReturnOnContinuationStack(Cntname,Pointer)).
```

Figure 6.26: The adapted behavioural model

same variable name `Pointer` in the rules on lines 2 and 3 so they have to be unified with the same value. We base ourselves on the fact that a continuation placed on the continuation stack is awaiting execution at a later point in time.

Step3: Specifying Application-Specific Instances

In step 3 we describe *which* events have to be intercepted at run-time and *how* they should be recorded. This entails both the specification of *what* source-code constructs raise these high-level events and *how* associated run-time values need to be extracted. In the intercept predicates specified in figure 6.27, the application-specific instances of the high-level events are mapped to their descriptions. The first two intercept rules on

```
1 intercept(before,
2           pushRetCnt,
3           event(time,pushReturnOnCntStack(cntName,topCnt))).
4 intercept(before,
5           pokeRetCnt,
6           event(time,pokeReturnOnCntStack(cntName,utopCnt))).
7 intercept(before,
8           continuationEntry,
9           event(time,cntEntered(cntName,cntPointer,cntStack))).
```

Figure 6.27: High-level events specification for tail recursion optimisation

lines 1–6 capture the action of placing a `RET` continuation on the continuation stack. In Pico this can be done either by *pushing* something onto the stack or using the *poke* operation [11]. We need to distinguish between these two operations because of the different run-time values associated with them. In case of the *poke* operation, we need to look at what is right below the top of stack because there is still another continuation on top that is of no interest to us (i.e. that will be removed by the *poke* operation). On lines 3 and 6 we use the keywords `topCnt` and `utopCnt` to retrieve the top of stack, or what is right under the top of the continuation stack.

The application-specific instance, `pushRetCnt` shown in figure 6.28 on lines 1-4, refers to a C construct which is a macro call `_stk_push_CNT_` in the Pico code with argument the identifier `'RET'`. The `pokeRetCnt` predicate is identical (see appendix D).

We introduced two new keywords, `topCnt` and `utopCnt`, to obtain as run-time values the continuation on top of the continuation stack or, in the case of a *poke* operation,

[11] A *poke* operation basically replaces the top of the stack with another continuation

```
pushRetCnt(Construct) :-
        macroCallHasName(Construct,'_stk_push_CNT_'),
        macroCallHasArguments(Construct,Arglist),
        member(identifier('RET'),Arglist).

continuationEntry(Construct,Path) :-
        functionEntry(Construct,Path),
        inContinuation(Path).
```

Figure 6.28: Application-specific instances for tail recursion optimisation

the continuation right below the top. The `topCnt` keyword is shown in figure 6.29. As

```
keyword(topCnt,Construct,Path,Result) :-
   equals(Result,
   ";{/* cntStackpeektop Pico 1.0 */
        _CNT_TYPE_ cnt;
        _stk_peek_CNT_(cnt);
        behaveLog(\"%i\",cnt);
        };").
```

Figure 6.29: The `topCnt` keyword

the `utopCnt` is very similar, we did not include it here (see appendix D). The other run-time values that were used in the model, like `cntName`, `cntPointer`, were reused from other case studies. Appendix E contains a generated code excerpt of instrumenting Pico according to the `intercept` predicates defined for this experiment.

Step4: Lightweight Consistency Verification

As discussed in the previous chapter, performing a dynamic analysis of a system requires the execution of the instrumented system and hence also input data is needed to trigger particular behaviour of the problem at hand. In the previous two case studies it was sufficient to let Pico evaluate a collection of different expressions in such a way that it covered a broad range of continuations (the well-chosen execution scenario used for this purpose was depicted in figure 6.11). In this case study, we are concerned with very specific behaviour, so we must provide a sufficiently complex tail-recursive function. We chose as execution scenario to apply the quicksort algorithm as depicted in figure 6.30. For this particular concern, the application of the quicksort algorithm represents a well-chosen execution scenario as this function possesses both a recursive and a tail-recursive call (lines 15 and 16 respectively). Hence we executed Pico by letting the quicksort algorithm sort a table of 10 randomly generated numbers (see lines 17–20). By launching the query

```
?- sometime(possibleTailRecursionOptimization(Cntname)).
```

```
{QuickSort(V,Low,High):{
 Left : Low;
 Right : High;
 Pivot : V[(Left+Right)//2];
 until(Left > Right,
       {while(V[Left] < Pivot, Left := Left+1);
        while(V[Right] > Pivot, Right := Right-1);
        if(not(Left > Right),
          {Save : V[Left];
           V[Left] := V[Right];
           V[Right] := Save;
           Left := Left+1;
           Right := Right-1},
           false)});
        if(Low < Right, QuickSort(V, Low, Right), false);
        if(High > Left, QuickSort(V, Left, High), false)};
V[10]:0;Low:1;High:size(V);
x:1;y:1;
while(not(x>size(V)),{y:= (y+4253) \\ 1237;V[x]:=y;x:=x+1});
QuickSort(V,Low,High);
```

Figure 6.30: Pico implementation of the quicksort algorithm

we want to find out if sometime during program execution tail recursion activity is taking place without it being optimised, i.e. behaving like normal recursion. If anything is found, the name of the continuation where the non-optimised tail recursion takes place is returned.

The query did not return any results, meaning that tail recursion looks fully optimised as it should be. Having browsed through the source code examining those parts where a `RET` continuation is placed on the stack confirmed our results: a check was added to each of those places to see if the top of stack contains already a `RET` continuation. If so, a second `RET` was not placed on the stack.

Evaluation

Although checking the model did not return any results, an important inefficiency was discovered that unknowingly led to non-optimised tail recursion. The behavioural model as specified in figure 6.26 consists of one main run-time event shown in figure 6.31. This predicate provides an abstraction for either a `push` or a `poke` operation

```
installReturnOnContinuationStack(CntName,Topcnt) :-
  event(pushReturnOnCntStack(CntName,Topcnt)).

installReturnOnContinuationStack(CntName,Utopcnt) :-
  event(pokeReturnOnCntStack(CntName,Utopcnt)).
                                    installReturnInContinuuation abstraction
```

Figure 6.31: The `installReturnOnContinuationStack/2` predicate

of the `RET` continuation on the continuation stack. We tested this predicate separately:

```
?- sometime(installReturnInContinuation(CntName,Top)).
```

12 results were found that correspond to 12 pushes of the `RET` on the continuation stack, and indeed, in none of these cases the `RET` continuation was found on top of the stack. However, for *every* result found, the same `BXG` continuation was found on top of the stack with `Top = 142260`. The code below presents one of those results:

```
event(380, pushReturnOnCntStack(BND, 221120, 142260))

CntName = 'BND'
Top = 142260 ;    %% BXG continuation
```

We reused the `cntStack` keyword that was previously defined for the first case study, to be able to view the state of the *entire* continuation stack at run-time. Having added the extra keyword to the above predicate to accommodate an extra variable, the meta interpreter returned the following result [12]:

```
event(4722,pushReturnOnCntStack(BND,'BXG',
                                [BXG,RET,BXG,RET,BXG,RET,BXG,_print_EXP_,exit_loop]))

Cntname = 'BND'
Topcnt = 'BXG'
CntStack = [BXG,RET,BXG,RET,BXG,RET,BXG,_print_EXP_,exit_loop];
```

This result demonstrates a strange alternating behaviour of the `RET` and the `BXG` continuation. The `BXG` continuation represents a part of the execution of a Pico `begin` statement. In Pico, such a statement is either denoted as `begin(expr1,...,exprn)` or as {`expr1;...;exprn`} and defines a construct to group a sequence of expressions to be evaluated. In the quicksort implementation of figure 6.30, a `begin` statement is used on lines 6–16.

The `BEG` and `BXG` continuations are two closely collaborating continuations evaluating such a Pico `begin` statement (both continuations are listed in figures 6.32 and 6.33 respectively). `BEG` represents the entry of evaluation by checking if there is at least one expression (see line 7 in figure 6.32) in the begin statement. If so, a loop is set up to evaluate *all* expressions by pushing a number on the expression stack (line 11 of figure 6.32) and `BXG` on the continuation stack (line 14). `BXG` thus represents a continuation that is executed as many times as there are expressions to evaluate inside the `begin` statement. The alternating execution of `BXG` and `RET` refers to a call to `BXG` when the last expression of a `begin` statement was just evaluated. If not, the code on lines 9–14 of figure 6.33 code would be executed and then the `EVL` continuation would appear on the continuation stack, which is not the case. This leads us to the important conclusion that, when we evaluate a tail-recursive function in Pico, and the recursive function call is the last expression of a `begin` statement (like the situation of our chosen execution scenario `quicksort.pco`), that tail recursion does not work as it should, i.e. it is not optimised! The check added in the source code of the `BND` continuation [13] function (line 6 in figure 6.34) has no effect for intercepting tail recursion

[12]The stack result depicted here is an adapted version for the sake of clarity; the actual stack representation contains pointers instead of the names of the continuation functions. This was previously explained in section 6.3.2.

[13]The `BND` continuation (see the code excerpt in figure 6.34) takes care of that part of evaluation of binding the formal parameters to the actual parameters. A loop also needs to be used here depending on

```
static _NIL_TYPE_ BEG(_NIL_TYPE_)
 { _EXP_TYPE_ arg, exp; _UNS_TYPE_ siz;
   _stk_claim_();
   _stk_peek_EXP_(arg);
   siz = _ag_get_TAB_SIZ_(arg);
   if (siz == 0)
     { _stk_poke_EXP_(_VOID_);
       _stk_zap_CNT_();
       return; }
   _stk_push_EXP_(_ONE_);
   exp = _ag_get_TAB_EXP_(arg, 1);
   _stk_push_EXP_(exp);
   _stk_poke_CNT_(BXG);
   _stk_push_CNT_(EVL); }
```

Figure 6.32: The `BEG` continuation from PicoNat.c

```
static _NIL_TYPE_ BXG(_NIL_TYPE_)
 { _EXP_TYPE_ arg, exp, nbr, val; _UNS_TYPE_ ctr, siz;
   _stk_claim_();
   _stk_pop_EXP_(val);
   _stk_pop_EXP_(nbr);
   _stk_peek_EXP_(arg);
   siz = _ag_get_TAB_SIZ_(arg);
   ctr = _ag_get_NBU_(nbr);
   if (ctr < siz)
     { _stk_push_EXP_(_ag_succ_NBR_(nbr));
       exp = _ag_get_TAB_EXP_(arg, ctr+1);
       _stk_push_EXP_(exp);
       _stk_push_CNT_(EVL);
       return; }
   _stk_poke_EXP_(val);
   _stk_zap_CNT_();
```

Figure 6.33: The `BXG` continuation from PicoNat.c

here, because the `BXG` continuation is *blocking* the check. The problem is caused by the non-optimised code of both the `BEG` and the `BXG` continuation. In the case where a Pico `begin` statement contains only one expression, there is no need to set up a loop and push the `BXG` continuation on the continuation stack. Similarly, and more importantly, to avoid having to *go back* to the `BXG` continuation in case of the last expression, we have to eliminate the last call of `BXG`. We will rewrite the code of the `BXG` continuation and replace it by the source code shown in figure 6.35. We replace the `if (ctr < siz)` check with the check on line 4 to make sure that a loop is set up only if there is more than one expression left to evaluate. If there is only one expression left, we

how many variables need to be bound. In case of the last binding (check on line 4), all bindings need to be added to the enclosing environment and in that new environment the body of a function needs to be evaluated. After that evaluation, the former environment needs to be restored again, which explains storing the dictionary on the expression stack (line 8) and placing a `RET` on the continuation stack (line 10).

```
static _NIL_TYPE_ BND(_NIL_TYPE_)
 { _EXP_TYPE_ act, arg, dct, exp, fun, frm, nam, nbr, par, val, xdc;
   ...
   if (ctr == siz)
     { ...
       if (cnt != RET)
         { _stk_peek_EXP_(exp);
           _stk_poke_EXP_(_DCT_);
           _stk_push_EXP_(exp);
           _stk_push_CNT_(RET); }
       _stk_push_CNT_(EXP);
       _DCT_ = dct; }
```

Figure 6.34: Tail recursion check in the BND continuation from PicoEva.c

```
 1  static _NIL_TYPE_ BXG(_NIL_TYPE_)
 2   {...
 3    exp = _ag_get_TAB_EXP_(arg, ctr+1);
 4    if (ctr+1 < siz)   %% If there is more than one statement left to evaluate
 5      { _stk_push_EXP_(_ag_succ_NBR_(nbr));
 6        _stk_push_EXP_(exp);
 7        _stk_push_CNT_(EVL);
 8        return; }
 9    _stk_zap_EXP_();
10    _stk_push_EXP_(exp);
11    _stk_poke_CNT_(EVL); }
```

Figure 6.35: Optimisation of the BXG continuation from PicoNat.c

can *reduce* the result of evaluating the begin statement to the result of evaluating the last expression of that begin statement (lines 9-11 of figure 6.35), which is exactly what is needed for eliminating the blocking of the tail recursion check. We remove the collection of expressions (i.e. the arguments of the begin statement) contained in the begin statement (on line 9 of figure 6.35 they are removed from the expression stack) as they are no longer needed. Consequently, the last expression of the begin statement is pushed on the expression stack and we have it evaluated by replacing BXG by EVL on the continuation stack (the *poke* operation on line 11). A similar optimisation is performed in the BEG continuation (this code excerpt is not shown here).

After performing the BEG and BXG optimisations, the design invariant capturing Pico's behavioural program documentation (see section 6.3) was used to automatically update the documentation of both the BEG and BXG continuation.

6.6 Conclusions

We have come across some important observations while conducting these case studies:

- Using the BEHAVE platform on a particular case study to document and verify several design invariants creates an easy to use plug-and-play environment for reasoning about the behaviour of a program. A small investment needs to be made at first to define the keywords to retrieve the associated run-time values and the rules for expressing the application-specific instances. But once they are defined, you can start *reusing the concepts* easily for intercepting other parts of the dynamics of your program.

- Because the user herself can specify only those high-level run-time events needed for reasoning about a particular design invariant, a high-level compact execution trace is created. As a consequence, the trace contains concise and to-the-point behavioural domain knowledge of the design invariant at hand which already reveals a great deal of information on its own. In addition to this, analysing program behaviour becomes computationally less-expensive as such a trace in general comprises fewer events.

- Using BEHAVE by plugging in defined concepts and by running simple queries on the high-level run-time events does not only verify the behaviour of a design invariant, it can also *reveal a great deal about a design invariant's domain* around it. In the case of verifying the behavioural documentation (the first design invariant that was verified), some naming inconsistencies were discovered which are equally important to remove as adapting non up-to-date documentation because they are very likely to confuse users of the system too. As for the tail recursion case, we discovered a non-optimised BEG continuation by simply watching the run-time values for specific tail recursive behaviour.

- Although this lightweight verification approach cannot prove the correctness of a program with respect to the invariants because it is based on dynamic analysis, it can take advantage of well-chosen execution scenario's that are relevant to particular invariant behaviour. Although invariants are cross-cutting an entire system, they do address one specific behavioural concern that is often triggered by a particular execution scenario. Hence this might create an extra opportunity to focus analysis on parts of a larger program.

In addition to these observations, each of them validated the following important claims we made in our previous chapters:

- The simple stack example from the previous chapter demonstrated that BEHAVE can also be easily used for the lightweight verification of local invariants such as the ones discussed in chapter 2. An example is the verification of invariant stack behaviour.

- Verifying garbage collection as a design invariant clearly expresses the need to be able to reason about behavioural cross-cutting concerns and not only about local behavioural invariants. The garbage collection model is behaviourally cross-cutting in the sense that multiple points of program execution need to be intercepted. Depending on the order of events and on the associated run-time values a certain pattern adheres to the model or not.

- In comparison to functional testing frameworks such as unit testing [Ham04], our approach offers the possibility to test an invariant model that exhibits *non-externally verifiable* behaviour. The model we specified for the lightweight verification of the garbage collection invariant can lead us to situations that *might* result in erroneous behaviour or even a system crash. So errors do not have to actually occur in order to be detected. An additional consequence of this is that it provides a way to obtain a high degree of test coverage although only one execution trace is examined.

- Studying tail recursion as a design invariant really validates our claims for needing behavioural information to reason about invariants in software. Tail recursion is a pure dynamic concept which can only be intercepted when executing your program.

Having documented and verified three representative design invariants of a technically and algorithmically complex software system, further development of this system is partially automated with respect to these invariants. They are made explicit in a high-level behavioural specification which is loosely coupled with the source code. This makes them oblivious to source code adaptations but verifiable in a lightweight manner throughout an application's lifetime.

6.7 Summary

In this chapter we validated the use of BEHAVE by applying it to a fairly small but a technically and algorithmically complex case study, namely the Pico language interpreter. Pico presented an excellent opportunity to validate our platform: the original developer was still around so he could provide us with the expert knowledge we needed to be able to perform these case studies. In the first section we elaborated on our Pico case study to introduce the most important concepts needed as these design invariants capture expert knowledge about a system. Therefore explanation about the pico execution and memory model was first provided .

We continued our chapter by applying the four-step recipe to three Pico behavioural design invariants. The first invariant supported the creation of *active behavioural documentation* about the Pico execution model. Existing documentation provided by the developer was first transformed into a machine readable format and then verified against the actual program behaviour. For the second case study, garbage collection in Pico was discussed in section 6.4 and its behaviour was also documented and verified in a lightweight manner. In section 6.4 we specified a behavioural model for identifying correct tail recursion behaviour, an important design invariant of an optimised language interpreter. For each of those case studies we identified some problem points which could be used for improving Pico in one way or another. We ended with a conclusion by pointing out our most important observations made while applying BEHAVE and by summarising how these particular Pico design invariants validated our previously made claims about our proposed approach.

Chapter 7
Conclusion and Future Work

In this dissertation, we proposed a goal-driven approach for the documentation and lightweight verification of design invariants. We have demonstrated that our approach supports the development of technologically and algorithmically complex software systems.

In this chapter we present the conclusions of this dissertation. We begin by summarising the work presented, after which we emphasise the main contributions of this work. We end with a presentation of future research directions.

7.1 Summary

Software developers responsible for adapting and evolving an existing implementation of software must grasp the underlying behavioural dependencies implicitly imposed by making certain design decisions about the software at hand. This is problematic because these so-called behavioural *design invariants* are often not known to developers involved in a software project. Even if they are known, they are often only available *implicitly* in the software. This severely impedes program development as making a change might result in violating unknown behavioural constraints which ultimately leads to unreliable software.

Therefore, a developer should be offered a behavioural specification language in which system-specific design invariants can be specified and hence be made *explicit*. Such a specification formalism should support event-based specifications which take the order of events into account due to the non-externally verifiable behaviour design invariants might exhibit. And, as the specification should also be used for documentation purposes, the specification language should preferably be descriptive.

As such behavioural regularities might be *cross-cutting* an entire system, manually detecting those places in the source code which trigger the invariant behaviour is unfeasible. It can even become an impossible task as 'guessing' associated run-time values is not always possible. This implies the *need for a causal link* between the specification and the source code to make the specifications machine-verifiable. However, to support practical use during program development, a *tight coupling* between the design invariant specification and the source code *should be avoided* so as not having to adapt the specification every time a change to the code is being made.

Considering existing program analysis approaches and their suitability for checking design invariants, heavyweight formal verifiers such as model checking are difficult to use in this context. Formal methods analyse system models and prove the correctness of a model with respect to particular properties. However, for technically and algorithmically complex software, the true challenge lies at the code level, and a correctly proven system model does not contribute to its technical realisation. Another difficulty to overcome stems from the possibly cross-cutting behaviour of a design invariant. The applied analysis should allow to focus on parts of a larger program to support practical use. Static analysis approaches are therefore less suited as they provide no means to focus the analysis on a particular behavioural design concern not directly localised in a component of the programming language at hand. And most existing dynamic analysis approaches represent behaviour as fixed low-level implementation constructs, which makes analysing a high-level description computationally more expensive and hence impractical.

In this dissertation we proposed an approach for supporting program development of technically and algorithmically challenging systems by making design invariants *explicit* and at the same time *machine-verifiable*. The most distinctive feature of our proposed approach is the use of temporal logic programming as executable behavioural formalism for making design invariants explicit. One of the main contributions of this formalism is the high level of abstraction it offers. Next to the declarative feature offered by a logic language which allows the use of high-level concepts, abstractions over time structures are offered by the use of temporal operators. This provides a design invariant model which is at a high level of abstraction.

Moreover, a causal link between the design invariant specification and the source code is provided allowing a lightweight verification. This is realised by using a dynamic analysis approach which adopts an aspect-like code instrumentation mechanism by means of logic meta programming for obtaining high-level run-time events. By choosing a well-defined execution scenario, only those high-level events of interest are recorded which are relevant to the design invariant under investigation. This makes the approach goal-driven and hence applicable to parts of a larger program.

To validate and verify the practical feasibility of our proposed approach, we constructed the BEHAVE platform. BEHAVE is a lightweight verification platform which makes design invariants explicit for base language programs written in Ansi C. BEHAVE implements all the needed requirements as formulated in section 4.1. To optimally exploit the use of the platform, we have identified a four-step recipe and demonstrated its use by applying it on a running example.

The practical application of BEHAVE was illustrated by supporting the development of the Pico virtual machine, an interpreter for a fairly simple but technologically sophisticated programming language. Pico presented us with an excellent case study as the developer was available to inform us about the practical design decisions made when Pico was initially developed. We supported a representative set of three important design invariants of Pico: active behavioural program documentation, garbage collection and tail recursion optimisation. Further development in Pico is now supported as these design invariants are made an explicit and verifiable part for future adaptations of Pico.

7.2 Conclusion

In this dissertation we proposed and implemented a goal-driven approach for supporting the development process of technically and algorithmically sophisticated software by supporting design invariants in a lightweight manner. Design invariants represent underlying behavioural dependencies at the code level implicitly imposed by making certain design decisions about the software at hand. Our approach applies temporal logic programming as a behavioural formalism in which these design invariants are first made *explicit*. A causal link between the design invariant specification and the source code is supported allowing a *lightweight verification*. Our approach is based on dynamic analysis supporting selective code instrumentation by means of logic meta programming. Combining the selective instrumentation with a well-chosen execution scenario, only those high-level events that are relevant to the design invariant under investigation are recorded. Lightweight verification then amounts to consistency checking of the temporal assertions (which comprise the undesired design invariant behaviour) against the high-level execution trace.

In the remainder of this section we briefly summarise the main contributions of this dissertation.

Identification of Design Invariants

Within the context of this dissertation we studied invariants in software and how invariants are specified. Two main categories of invariants exist depending on the two fundamental ways behaviour is specified: *state-based* invariants and *event-based* invariants. The former are specified as constraints on a program's *state* or *data*, while the latter constrain non-externally verifiable *operations or events* in software. We have identified a type of invariants called *design invariants* which are primarily event-based and they take the order of the events at run-time into account. They represent underlying behavioural dependencies implicitly imposed by making certain design decisions about the software at hand:

> ***Definition***
> *A* ***design invariant*** *is a behavioural regularity of the design of a program that is not aligned with the structure of the program.*

They represent *the invariants of a design*, which are the specific characteristics that limit its future adaptation, flexibility and evolvability [ABE$^+$04]. These type of system-specific invariants are considered problematic as they are usually *implicitly* present in software, they might be *cross-cutting* an entire application and they are non-externally verifiable. In the context of technically challenging systems with several algorithmically complex components, this may lead to underlying behavioural dependencies which severely limit future system development.

A Goal-Driven Approach for the Documentation and Lightweight Verification of Design Invariants

We proposed a goal-driven approach for the documentation and lightweight verification of design invariants for supporting program development of technically and algorithmically complex software. A highly expressive and executable formalism, temporal logic programming, is used for making design invariants explicit by documenting them in a high-level behavioural model. A lightweight verification approach based on dynamic analysis is adopted for making the behavioural model at the same time machine-verifiable against an execution trace. For instrumenting a program, a declarative aspect-like code instrumentation approach is used to identify those high-level concepts in the source code that give rise to high-level events of interest to a particular design invariant. Consequently, the event trace is then created by running the instrumented program according to a well-defined execution scenario, which makes the approach goal-driven as well.

The main contributions of our approach are the use of temporal logic programming as an executable behavioural formalism for making design invariants *explicit* and *machine-verifiable*. Moreover, we used a logic language as selective code instrumentation language (or pointcut language) to identify high-level source code entities which give rise to high-level run-time events. Expressing temporal assertions representing undesired design invariant behaviour directly in terms of high-level events makes the program analysis computationally less expensive and it results in a high-level design invariant specification which is *oblivious* to source code constructs.

The BEHAVE Platform

We created a prototype platform named BEHAVE to validate the feasibility of our proposed approach. BEHAVE supports the documentation and lightweight verification of design invariants in C by specifying them in high-level behavioural models which are at the same time executable.

BEHAVE is entirely implemented in the logic language Prolog, and uses a declarative aspect-like approach for instrumenting Ansi C programs. A C parser and modest tool support is available to aid in setting up and using the experimental platform. The three main components of the platform are: a reification module containing a logic representation of a C base language program, the instrumentation module which generates selectively instrumented source code and a temporal logic meta interpreter which verifies the design invariant models against the actual program behaviour.

To optimally exploit the use of the BEHAVE platform, we have identified a four-step recipe and demonstrated its use by applying it on a running example. In a first step, users have to identify the high-level events with their associated run-time values that are to be used in the design invariant model. Second, they have to specify the undesired behaviour of a particular design invariant in a high-level behavioural model. As a third step, the application-specific instances need to be defined which give rise to the high-level events. The fourth and last step comprises the verification phase of our approach where developers can have the consistency of the programs documented behaviour and its actual behaviour verified.

Validating the Use of BEHAVE by Supporting Program Development

The BEHAVE platform was validated by supporting program development of the Pico virtual machine (Pico 1.0). Pico is an interpreter of a fairly simple but technologically sophisticated programming language which incorporates automatic garbage collection, allows the use of higher order functions, supports meta programming and reflection, uses optimised tail-recursion for implementing iterative processes, etc. As the proposed approach necessitates the availability of knowledge about design decisions made and about the internals of the program from the program under study, Pico formed an excellent case as the original developer was available to give us insight in the main design decisions that were undertaken.

We supported program development in Pico by studying three technically complex design concerns which form a crucial part of the technical design of Pico. The first concern addressed the lightweight verification of behavioural program documentation of the Pico evaluation engine. As Pico expressions are evaluated by a (variation of a) continuation-based-style interpreter, checking the expected dynamics of the execution model is crucial. The second concern our approach supported was automatic garbage collection. Special care should be taken about restoring temporary references after the garbage collection algorithm is triggered, so we documented this particular behaviour in a high-level model, thereby enabling its lightweight verification. Optimised tail-recursion behaviour forms the third design concern which our approach supported. To be able to use tail-recursion to implement iterative processes, stack usage needs to be optimised, so we have set up a design invariant model to identify possible locations where optimisations has not been done. For all three Pico design concerns, inconsistencies confirmed by the developer were found which were attributed to subtle errors and non-optimised continuations in the Pico source code. For example, the garbage collection inconsistencies encountered were the main cause of occasional system crashes which could not at all be identified externally.

We concluded that our proposed approach contributes significantly to supporting program development by documenting its major design concerns and by having a causal link with the source code.

7.3 Future Work

This section lists a number of directions for future research as well as a number of possible improvements for the current implementation of the platform.

7.3.1 Possible Improvements and Additions for BEHAVE

Dynamic Code Instrumentation The current version of BEHAVE performs code instrumentation which is purely statical. The program points of interest (or, to use AOP terminology, pointcut expressions) are described declaratively and an entire application's parse tree is traversed (while re-generating the code) searching for nodes which unify with those particular pointcut descriptions. On top of this, the instrumentation code inserted at points in the program, contains only code to print strings to an external file (representing the execution trace).

However, the code which is inserted in the C source program could also contain (next to code to print strings) code which performs 'dynamic tests' on run-time values. This way, we could for example filter the instrumentation on run-time values. This would require the expansion of the aspect layer of BEHAVE where the *before* and *after* predicates could be expanded with for example inserting `if` tests on keywords. This would result in an even more compact execution trace. However, it remains to be investigated how this would be integrated in the platform and which applications would benefit of this extra filtering of events.

Combining Structural and Behavioural Reasoning Meta programming systems used for program analysis perform reasoning by querying or matching a particular representation of the program under investigation. Such a representation is defined by a meta model and this model is often specifically fine-tuned for the particular purpose it must fulfil. In the proposed approach, we use two meta models of the base language program under investigation. On the one hand we employ a *structural meta model* established by the logic meta programming (LMP) set-up as discussed in section 4.3.1 for selecting program points of interest. On the other hand a partial behavioural meta model is used to reason about representative parts of a program's behaviour (see section 4.3.1). Although both meta models are represented as logic facts in Prolog, they are used independently of each other. The structural meta model is used to *obtain* the needed run-time behaviour, while the partial behavioural meta model is used to *analyse* the obtained program behaviour.

As both meta models are uniformly represented in Prolog as logic facts, they could be readily combined to reason about the structure as well as the behaviour of a base language program. Possible general applications are pattern detection techniques [MMW02] which could benefit from both structural as well as behavioural information. Design invariants (as defined in chapter 2) could also have structural underlying dependencies in addition to behavioural constraints.

Another use in the context of the proposed approach could be to use the structural meta model to *support the analysis of* the reported inconsistencies of the behavioural design invariant model. These inconsistencies found by matching the design invariant description against the actual program behaviour can be less straightforward to immediately trace them back to the source code. Launching structural queries might help in more easily localising certain run-time values, especially for larger programs.

On-line analysis for BEHAVE Our proposed approach (as demonstrated in the BEHAVE prototype implementation) has currently been designed with a post-mortem (or off-line) behavioural analysis strategy in mind. A post-mortem analysis strategy consists of instrumenting a program under investigation followed by executing a well-defined execution scenario that lets the created execution trace focus on that behaviour relevant for the design invariant in question. So, the analysis of the recorded run-time behaviour starts *after* having finished program execution.

Debuggers for example offer tool support for aiding software development by performing on-line analysis. The main advantage is that it might be simpler for the user as post-processing steps are eliminated. Quicker feedback can be given as well and the program can be stopped immediately when an inconsistency is detected. For performing online analysis (or ad-hoc analysis), a behavioural reasoning process would be needed which runs as a co-routine alongside the application. The execution of the application would have to be interleaved with the evaluation of a logic program, which has to be able to suspend and resume the execution of the application. When a particular execution event is requested by a declarative query, the program execution should continue until this event is encountered. Control should then be returned to the reasoning process where the current event is then analysed and a new request for the following event is being made.

A consequence of on-line analysis is that we do not have a complete execution history at our disposal when a query is analysed. The applicability of this on-line approach is therefore limited to querying for simple rules. Since the execution of a program is advanced whenever we backtrack upon a choice for a run-time event, we cannot access past events in this way and considering alternatives for an event is not possible [Roo04]. It remains to be investigated how our approach could benefit from such an analysis.

Better Support for Macro's in Ansi C As briefly mentioned in section 5.1, we identified some practical limitations of the current implementation of the BEHAVE platform. One major obstacle was found to be the parsing of macros in C. A macro is a source code construct which normally gets expanded first by the compiler before the source code gets parsed. The resulting expanded program is a well-formed program which can be parsed according to a C grammar. However, as the program of investigation was a language interpreter where a lot of attention is paid to performance optimisations, macros are used extensively and they form a crucial part of the execution model. That is why we opted not to expand the macros to be able to reason about them as high-level concepts.

The way we solved this in the current version of BEHAVE is to treat macro calls as function calls and reify them into a logic predicate `macroCall/2` without parsing the macro body but storing it in its logical representation as a string. This works well as long as only simple macros are defined. However, in C, a macro can contain *part* of a C statement, which is joined with the other part of the statement only *after* expanding it in the source code. This explains why the body of a macro is not parsed: it does not always contain a valid C expression. An additional problem caused by not being able to parse the macro body are type declarations. These declarations are not parsed as the macro body is not parsed and hence the type is not known in the remainder of the program.

Another solution should be found for parsing C code without losing information about macro calls. One possible solution could be to expand macro calls anyway, but to use a mechanism to *mark* the expanded code in some way to denote its origin. However, it remains to be researched how feasible such an approach would be and in what way such *macro call markings* can be integrated into a logical representation of the source code.

7.3.2 Model-Driven Engineering Support

Considering the proposed approach and in particular the four-step recipe explained and applied throughout the last three chapters, it is only at the very end of the recipe where the application-specific instances need to be formulated in terms of programming language constructs. The behavioural model, the identification of high-level events and even the description of the intercept mappings where the source code constructs are linked to their respective high-level events description are all base-language independent.

One possible application could be more sophisticated test-driven development. Test-driven development (TDD) is an existing software development technique which is advocated in the agile community. Such testing is based on repeatedly first writing a test case and then implementing only the code necessary to pass the test. This kind of testing is associated with Extreme Programming, an agile way of developing software. Proponents emphasise that TDD is a way of designing software, not merely a testing method. However, almost all testing done in this context (e.g. XUnit framework [Ham04]) are functional tests which test small standalone local components. This is mainly the case because as for agile approaches, testing is part of the development life cycle, so as testing is done more often, to truly support program development, it needs to be automated. On top of that, unit tests are written in the programming language in which the base language program to be tested is written.

BEHAVE can be used on a higher level for writing sophisticated tests for more cross-cutting behavioural entities which can be written before they are implemented. Since the behavioural models are base language independent, they can even be applied later onto different implementations.

7.3.3 Computer Science Education

Having specified and checked design invariants for the Pico language interpreter in chapter 6, we were inspired by the final attainment level of the second year course *Interpretation of Computer Programs II* [1]. The aim of that course is to conceive and construct step-by-step a virtual machine for the Pico programming language as discussed in section 6.1. The virtual machine contains various essential and technically challenging components such as an automatic memory manager, a stack machine, a parser, etc. as thoroughly demonstrated in chapter 6. All the design invariants that we modelled for the Pico interpreter represent those concepts that students should certainly have grasped at the end of the second year course.

Taking into account the base-language independence of the run-time events and the behavioural model of the design invariants as discussed in section 7.3.2, one can imagine a setting where a teacher would provide a student with descriptions of high-level events and a behavioural model which reasons over these events. Even the mapping description can be provided. All of this information would leave the student with a description of a certain behavioural concept which she should grasp. It is then left up to the student to link the conceptual entities to those implementation-level constructs that give rise to such entities. This experiment presents at the same time an opportunity to evaluate the communicative aspect (or documentation aspect) of the behavioural models empirically. A student would have to read and understand the declarative descriptions of conceptual entities. An example use could be to learn novice students new algorithms, where the algorithms are described conceptually and the student should link the concepts to the actual implementation.

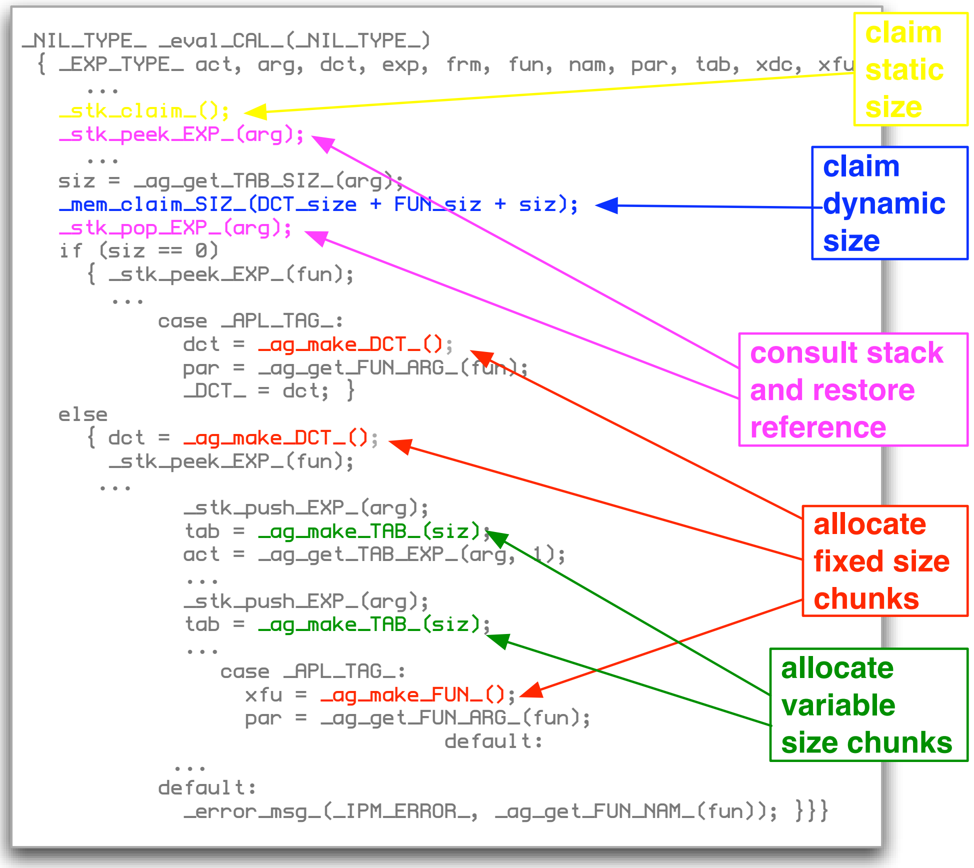
In today's software engineering education very little attention is given to the dynamic interactions in a program. The current trend in educational methods is an *objects-first* approach and preferably even *design-first*, where most attention is paid to structural modelling at the design level in terms of objects [BMBL05, BMF04, MBB03, MBB02, MBFP00]. Although object interactions are also considered, this is more done in a global way and not so much at the code level. Although this 'thinking in terms of objects' philosophy from the early beginnings is crucial for novices, it does not teach a student how algorithms work and in general how more technical components interact at the code level.

7.3.4 Garbage Collection as an Aspect

In section 6.4 we presented a case study in which we documented a particular kind of behaviour of the Pico memory model dealing with automatic garbage collection.

[1]The contents of the course can be consulted at http://prog.vub.ac.be/˜tjdhondt/ICP2/HTM.dir/ introduction.htm

Figure 7.1 repeats the code excerpt of the `_eval_CAL_` continuation (it was previously shown in chapter 1 as figure 1.1) as this piece of code forms a critical section for garbage collection. This continuation evaluates Pico function applications (in the Pico continuation network from figure 6.2 this continuation is denoted as `CAL`). The essence of the garbage collection design invariant lies in the combination of the static and dynamic memory claims denoted in yellow and blue respectively and restoring the references of the temporary variables by consulting the expression stack (denoted in pink). Claiming memory for an object of static size can be done at the very beginning of a continuation function, as shown on figure 7.1 by the `_stk_claim_()` macro (denoted in yellow). Claiming memory of dynamic size is difficult: it should be done *as early as possible* so as to minimise the number of references which need to be restored after the claim. An interesting path to explore in the near future would be, next to



```
_NIL_TYPE_ _eval_CAL_(_NIL_TYPE_)
 { _EXP_TYPE_ act, arg, dct, exp, frm, fun, nam, par, tab, xdc, xfu
     ...
   _stk_claim_();
   _stk_peek_EXP_(arg);
     ...
   siz = _ag_get_TAB_SIZ_(arg);
   _mem_claim_SIZ_(DCT_size + FUN_siz + siz);
   _stk_pop_EXP_(arg);
   if (siz == 0)
     { _stk_peek_EXP_(fun);
       ...
           case _APL_TAG_:
             dct = _ag_make_DCT_();
             par = _ag_get_FUN_ARG_(fun);
             _DCT_ = dct; }
   else
     { dct = _ag_make_DCT_();
       _stk_peek_EXP_(fun);
      ...
             _stk_push_EXP_(arg);
             tab = _ag_make_TAB_(siz);
             act = _ag_get_TAB_EXP_(arg, 1);
             ...
             _stk_push_EXP_(arg);
             tab = _ag_make_TAB_(siz);
             ...
                case _APL_TAG_:
                  xfu = _ag_make_FUN_();
                  par = _ag_get_FUN_ARG_(fun);
                                 default:
           ...
           default:
             _error_msg_(_IPM_ERROR_, _ag_get_FUN_NAM_(fun)); }}}
```

claim static size
claim dynamic size
consult stack and restore reference
allocate fixed size chunks
allocate variable size chunks

Figure 7.1: Pico code excerpt of `_eval_CAL_` continuation

documenting and verifying this typical garbage collection behaviour, to *generate* the necessary garbage collection statements. As became clear from the case study, garbage collection is cross-cutting an entire application, so why not treat it as an aspect? For garbage collection in Pico, this would entail that the memory claims would have to be inserted as *advice*. The pointcut expressions, which require temporal constraints, would be the identification of the chunk allocations (see figure 7.1 where the fixed size chunk allocations are denoted in red and the variable sized chunks denoted in green) as those statement allow to compute the allocation sizes. However, it remains to be investigated how feasible this would be in practice.

7.3.5 Using BEHAVE for Supporting Evolution

So far we have validated the use of the BEHAVE platform by using it to aid the development process. We have done so by documenting and verifying three design invariants for the Pico language interpreter, each representing a behavioural, technically challenging concern of the language interpreter. We have encountered unwanted behaviour in all three concerns and the results allowed us to steer the development process for those particular concerns.

As mentioned in section 6.1, Pico formed the ideal case study for testing the BEHAVE platform as the original developer was available to give us insight in the main design decisions that were taken when implementing Pico. This is important as design invariants capture expert knowledge of technically complex software. We also mentioned the availability of multiple versions of the Pico interpreter. A possible future work would be to apply the same experiments (i.e. the same behavioural models of the three design invariants) on another version of the Pico language interpreter. Pico 2.0

```
 1 { ctr(init):
 2     { n: init;
 3       incr():: n:= n+1;
 4       decr():: n:= n-1;
 5       clone() };
 6   c: ctr(0);
 7   c.incr();
 8   c.incr();
 9   c.incr();
10   c.decr() }
```

Figure 7.2: A counter object in Pico 2.0

is an extended version of Pico 1.0 where specific features were added to move towards an object-oriented version of Pico (a prototype-based version). Figure 7.2 depicts a counter object which can be evaluated by Pico 2.0. For representing objects, first-class environments were introduced as a way to represent the state of an object (in figure 7.2 on line 5, the `clone()` function returns a first-class environment). Also to enable object re-entrancy, the concept of a constant declaration was added (object cloning requires a deep copy of an object's instance variables, but a shallow copy of its methods), which can be seen on lines 3 and 4 (the double colon notation). The dot notation for enabling a message passing mechanism also had to be added. Furthermore, quasiquoting was added, multi-dimensional tables as well as introspection facilities (via additional native functions).

Verifying the same design invariant models for this version of Pico would allow us to investigate the reusability of the behavioural models and see if adaptations have to be made only at the level of the application-specific instances. Although Pico 2.0 incorporated several important functional extensions, the expressed design invariants should have remained the same.

Pico 3.0 presents another version of Pico, but with a completely different architecture. Instead of using continuations as units of program execution, a frame-based style of programming is adopted. In this case, the behavioural models of the design invariants will probably have to be changed as well, as the conceptual building blocks of the models are no longer employed. However, we believe that still a part of the Pico rules could be reused.

7.4 Closing Remarks

In this dissertation we proposed and implemented a goal-driven approach for partially automating program development of technically and algorithmically complex software. We achieved this by documenting design invariants and making them verifiable in a lightweight manner.

To support practical use, our approach advocates oblivious design invariant specifications and a lightweight goal-driven verification. The design invariant obliviousness results in not having to adapt the documentation every time the source code changes. The lightweight goal-driven approach allows a developer to focus the analysis on specific parts of a program. Hence, the approach is applicable to larger systems as well.

The main challenge of technically and algorithmically complex systems lies in getting the technicalities right. Therefore we strongly believe that supporting any aspect of their development must be primarily *code-driven*.

Appendix A

The Full Representational Mapping

This appendix lists the complete representational mapping for representing a subset of C program elements to their internal representations. The mapping is based on a C grammar as developed for the Ansi C standard grammar[Deg].

Table 5.1 describes the direct mapping of function definitions, function declarations, macro definitions and macro declarations into separate logic facts (the table was shown already in section 5.2.5; it is repeated here for the sake of completeness).

C Construct	**Logic Representation**
Function definition	`functionDefinition(FileName,Ret,Nam,Pars,Body).`
Declaration	`declaration(FileName,DeclSpecifs,InitDecLst).`
Macro definition	`macroDefinition(FileName,Name,Value).`
Function macro	`functionMacro(FileName,Name,Parameters,Body).`
C file	`baseFile(FileName).`
Header file	`includedFile(FileName).`
C project path	`projectPath(PathString).`
User include	`userInclude(FileName,IncludedFileName).`
System include	`systemInclude(FileName,SystemFileName).`

The following functors capture the representation of types, operators, declarators and program statements:

C Construct	**Logic Representation**
+	`addition(LeftExpression,RightExpression).`
&&	`and(LeftExpression,RightExpression).`
&	`bitwiseAnd(LeftExpression,RightExpression).`
ˆ	`bitwiseExclusiveOr(LeftExpr,RightExpr).`
|	`bitwiseInclusiveOr(LeftExpr,RightExpr).`
/	`division(LeftExpression,RightExpression).`
==	`equal(LeftExpression,RightExpression).`
>	`greater(LeftExpression,RightExpression).`
>=	`greaterOrEqual(LeftExpression,RightExpression).`
<<	`leftShift(LeftExpression,RightExpression).`
%	`modulo(LeftExpression,RightExpression).`
*	`multiplication(LeftExpression,RightExpression).`
! =	`notEqual(LeftExpression,RightExpression).`
||	`or(LeftExpression,RightExpression).`
->	`pointerAccess(PostFixExpression,Identifier).`
>>	`rightShift(LeftExpression,RightExpression).`
<	`smaller(LeftExpression,RightExpression).`
<=	`smallerOrEqual(LeftExpression,RightExpression).`
−	`substraction(LeftExpression,RightExpression).`
Expression–	`postfixDecrement(Expression).`
Expression++	`postfixIncrement(Expression).`
–Expression	`prefixDecrement(Expression).`
++Expression	`prefixIncrement(Expression).`
Abstract declarator	`abstractDeclarator(Pointer,DirectAbstrDecl).`
Array Access P[E]	`arrayAcces(P,E).`
Array Declarator	`arrayDeclarator(DirectDeclarator,ConstantExpr).`
Assignment expr	`assignment(LeftExpr,AssgnOp,RightExpr).`
Assgn operators	=, >>=, <<=, + =. − =, ⋆=, /=, %=, &=, ˆ=, |=
Case statement	`case(ConstantExpression,Statement).`
Cast	`cast(Type,Expression).`
Constant	`constant(Something).`
Declarator	`declarator(Pointer,DirectDeclarator).`
DoWhile statement	`doWhile(Statement,Expression).`
Enumerator	`enumerator(Identifier,ConstantExpression).`
Enumspecifier	`enumSpecifier(Identifier,EnumeratorList).`
For Statement	`for(Init,Cond,UpdateExpression,Statement).`
Function body	`compoundStatement(Declarations,Statements).`

C Construct	Logic Representation
Function call	`functionCall(PostFixExpression,Arguments).`
Function declarator	`functionDeclarator(DirectDeclarator,List).`
Function type declarator	`functionTypeDeclarator(Declarator).`
Goto	`goto(Identifier).`
Identifier	`identifier(Symbol).`
If statement	`ifThenElse(Expression,Statement,ElseStat).`
Init declarator	`initDeclarator(Declarator,Initializer).`
Initializer	`initializer(ListOrAssignment).`
Label	`label(Symbol,Statement).`
MacroCall	`macroCall(Name,Arguments).`
Macro	`macro(Symbol).`
PostFixExpr.Identifier	`memberAccess(PostFixExpr,Identifier).`
Parameter Decl.	`parameterDeclaration(DeclSpecs,Declarator).`
Parentheses (Expr)	`parenthesedExpression(Expr).`
Pointer	`pointer(TypeQualifierList,AnotherPointer).`
Return	`return(Expression).`
If without else	`shortIf(Expression,Statement).`
Sizeof	`sizeof(Expression).`
Storage class spec.	`typedef, extern, static, auto, register`
String	`stringLiteral(Something).`
Struct declaration	`structDeclaration(SpecQualLst,StructDecLst).`
Struct declarator	`structDeclarator(Declarator,ConstantExpr).`
Struct	`struct(Identifier,StructDeclarationList).`
Switch statement	`switch(Expression,Statement).`
Typename	`typeName(SpecQualList,AbstractDeclarator).`
Self-defined type	`typeNameSymbol(Something).`
Unary operation	`unaryOperation(Operator,Expression).`
Union	`union(Identifier,StructDeclarationList).`
While statement	`while(Expression,Statement).`

Appendix B
BEHAVE Instrumentation Module

This appendix contains the logic predicates contained in the BEHAVE instrumentation module of the BEHAVE architecture as depicted in figure 5.5. This layer contains the predicates needed to generate instrumented source code for a base language program parsed and reified in the BEHAVE reification layer. As explained in section 5.4.3, this module consists of two layers: the generation layer and the aspect layer. The generation layer generates C code for each of the logical representations of all C parse tree nodes. The *aspect layer* contains the predicates to check for each parse tree node if it needs to be instrumented *before* or *after* that particular node.

B.1 Generation Layer

The predicates of the generation layer are listed below. *All* predicates contained in that layer are included, however not all declarations of the `generateAux2/3` predicate are listed. There is a `generateAux2` declaration for every C language element as described in the full representational mapping in appendix A. We included only a few representative examples.

```
generateProject(Project) :-
        findall(Filename,isProjectFilename(Filename),Filenames),
        foreach(Filenames,generateFile).

generateFile(Filename) :-
        generatePath(Path),
        buildPath(Path,Filename,Fullpath),
        zopen(Fullpath,Stream),
        includeBehave(Stream),
        generateAllIncludes(Filename,Stream),
        generateAllMacros(Filename,Stream),
        generateAllDeclarations(Filename,Stream),
        generateAllFunctionDefinitions(Filename,Stream),
        znewline(Stream),
        zclose(Stream).

%% Logic fact set by the user through the user interface
generatePath("/Users/imichiel/Desktop/BEHAVE-needs/Pico-1.0/generated/").

buildPath(Path,Filename,Fullpath) :-
        appendStrings(Path,Filename,Fullpath).

includeBehave(Str) :-
        aspectsEnabled,
        zwrite("#include \"behave.h\"",Str).

includeBehave(_).

generateAllIncludes(Filename,Stream) :-
        generateSystemIncludes(Filename,Stream),
        generateUserIncludes(Filename,Stream).

generateAllIncludes(Filename,Stream) :-
        zwrite("INCLUDES FAILED",Stream).

generateSystemIncludes(Filename,Stream) :-
        zwritenl("/* System includes */",Stream),
        forall(systemIncludeHasFilename(Include,Filename),
               generateSystemInclude(Include,Stream)),
        znewline(Stream).

generateUserIncludes(Filename,Stream) :-
        zwritenl("/* User includes */",Stream),
        forall(userIncludeHasFilename(Include,Filename),
               generateUserInclude(Include,Stream)),
        znewline(Stream).

generateSystemInclude(Include,Stream) :-
        zwrite("#include <",Stream),
        systemIncludeHasFile(Include,Filename),
        zwrite(Filename,Stream),
        zwritenl(">",Stream).

generateUserInclude(Include,Stream) :-
        zwrite("#include \"",Stream),
        userIncludeHasFile(Include,Filename),
        zwrite(Filename,Stream),
        zwritenl("\"",Stream).

generateAllMacros(Filename,Stream) :-
        generateAllMacroDefinitions(Filename,Stream),
        generateAllFunctionMacros(Filename,Stream).

generateAllMacroDefinitions(Filename,Stream) :-
        zwritenl("/* Macro definitions */",Stream),
        forall(macroDefinitionHasFilename(MacroDefinition,Filename),
               generateMacroDefinition(MacroDefinition,Stream)),
        znewline(Stream).

generateAllFunctionMacros(Filename,Stream) :-
        zwritenl("/* Function macros */",Stream),
        forall(functionMacroHasFilename(FunctionMacro,Filename),
               generateFunctionMacro(FunctionMacro,Stream)),
        znewline(Stream).

generateMacroDefinition(MacroDefinition,Stream) :-
        macroDefinitionHasName(MacroDefinition,Name),
        zwrite("#define ",Stream),
        zwrite(Name,Stream),
        zwrite(" ",Stream),
        macroDefinitionHasValue(MacroDefinition,Value),
        zwritenl(Value,Stream).

generateFunctionMacro(FunctionMacro,Stream) :-
        functionMacroHasName(FunctionMacro,Name),
        zwrite("#define ",Stream),
        zwrite(Name,Stream),
        functionMacroHasParameters(FunctionMacro,Parameters),
        zwrite("(",Stream),
        generateCommaList(Parameters,[FunctionMacro],Stream),
        zwritenl(")\\",Stream),
        functionMacroHasBody(FunctionMacro,Body),
        zwrite("  ",Stream),
        zwritenl(Body,Stream).

generateAllDeclarations(Filename,Stream) :-
        zwritenl("/* Declarations */",Stream),
        forall(declarationHasFilename(Declaration,Filename),
               generateDeclaration(Declaration,Stream)),
        znewline(Stream).

generateDeclaration(Declaration,Path,Stream) :-
        isSimpleDeclaration(Declaration),
        declarationHasDeclarationSpecifiers(Declaration,DeclSpecs),
        generateSpaceList(DeclSpecs,[Declaration|Path],Stream),
        zwritenl(";",Stream).

generateDeclaration(Declaration,Path,Stream) :-
        declarationHasDeclarationSpecifiers(Declaration,DeclSpecs),
        generateSpaceList(DeclSpecs,[Declaration|Path],Stream),
        zwrite(" ",Stream),
        declarationHasInitDeclaratorList(Declaration,InitDecList),
        generateCommaList(InitDecList,[Declaration|Path],Stream),
        zwritenl(";",Stream).

generateAllDeclarations(Filename,Stream) :-
```

```
        zwrite("DECLARATIONS FAILED",Stream).

generateAllFunctionDefinitions(Filename,Stream) :-
        zwritenl("/* Function definitions */",Stream),
        forall(functionDefinitionHasFilename(Function,Filename),
               generateFunctionDefinition(Function,Stream)),
        znewline(Stream).

generateAllFunctionDefinitions(Filename,Stream) :-
        zwrite("FUNCTIONS FAILED",Stream).

generateFunctionDefinition(Function,Stream) :-
        generateFunctionSignature(Function,Stream),
        functionDefinitionHasBody(Function,Body),
        generateAux(Body,[Body,Function],Stream),
        znewline(Stream),
        znewline(Stream).

generateFunctionSignature(Function,Stream) :-
        functionDefinitionHasReturn(Function,Return),
        generateSpaceList(Return,[Function],Stream),
        functionDefinitionHasName(Function,Name),
        zwrite(" ",Stream),
        zwrite(Name,Stream),
        functionDefinitionHasParameters(Function,Parameters),
        zwrite("(",Stream),
        generateCommaList(Parameters,[Function],Stream),
        zwrite(")",Stream).

generateAuxStatement(E,Path,Stream) :-
        generateAux(E,Path,Stream),
        zwritenl(";",Stream).

zforeach(List,Predicate,Args) :-
        list(List),
        list(Args),
        forall(member(Item,List),apply(Predicate,[Item|Args])).

generateDeclarationList(List,Path,Str) :-
        list(List),
        forall(member(Declaration,List),
               generateDeclaration(Declaration,[List|Path],Str)).

generateSeparatedList([Last],Separator,Path,Stream) :-
        generateAux(Last,[Last|Path],Stream).

generateSeparatedList([First|Rest],Separator,Path,Stream) :-
        generateAux(First,[First|Path],Stream),
        zwrite(Separator,Stream),
        generateSeparatedList(Rest,Separator,Path,Stream).
generateSeparatedList([],Separator,Path,Stream).

transformWhat(Construct,Path,What,NewWhat) :-
        termToCode(Construct,Path,What,Code),
        equals(Dc,[[";{"],Code,["behaveLog(\".\\n\")}"]]),
        flatten(Dc,DecoratedCode),
        equals(NewWhat,behaveCodeList(DecoratedCode)).

termToCode(Construct,Path,Term,Code) :-
        atom(Term),
        !,
        keyword(Construct,Path,Term,Code).

termToCode(Construct,Path,Term,Code) :-
        equalsStructureList(Term,[Functor|Arguments]),
        addCommas(Arguments,List),
        maplist(termToCode(Construct,Path),List,TransformedArgs),
        concat([" behaveLog(\"",Functor,"(\");"],First),
        equals(Last," behaveLog(\")\");"),
        equals(C,[[First],TransformedArgs,[Last]]),
        flatten(C,Code).

keyword(topOfStack,
        Construct,
        Path,
        "behaveLog(\"%i\", _stack_[_top_ - 1]);").

keyword(sizeOfStack,
        Construct,
        Path,
        "behaveLog(\"%i\", _top_);").

keyword(time,
        Construct,
        Path,
        "behaveLog(\"%i\", TIME++);").

keyword(comma,Construct,Path,"behaveLog(\",\");").

keyword(functionName,Construct,Path,Result) :-
        functionName(Construct,Path,Name),
        concat(["behaveLog(\"'",Name,"'\");"],Result).

keyword(var(V),Construct,Path,Result) :-
        concat(["behaveLog(\"",V,"\");"],Result).

keyword(assVar,Construct,Path,Result) :-
        assignmentToVariable(Construct,Var),
        keyword(var(Var),Construct,Path,Result).

keyword(macroName,Construct,Path,Result) :-
```

```
        macroCallHasName(Construct,Name),
        concat(["behaveLog(\"'",Name,"'\");"],Result).

keyword(AnythingElse,Construct,Path,Result) :-
        concat(["behaveLog(\"",AnythingElse,"\");"],Result).

generate(Construct,Str) :-
        generateAux(Construct,[Construct],Str).

generateStatementList(List,Path,Str) :-
        list(List),
        forall(member(Constr,List),
               generateAuxStatement(Constr,[Constr,List|Path],Str)).

generateCommaList(List,Path,Str) :-
        generateSeparatedList(List,",",Path,Str).

generateSpaceList(List,Path,Str) :-
        generateSeparatedList(List," ",Path,Str).

generateSummary(Function,Stream) :-
        functionDefinitionHasFilename(Function,Filename),
        zwrite(Filename,Stream),
        zwrite(": ",Stream),
        generateFunctionSignature(Function,Stream).

generateSummary(Unknown,Stream) :-
        zwrite("no summary available",Stream).

generateDeclaration(Declaration,Stream) :-
        generateDeclaration(Declaration,[],Stream).

addCommas([],[]).

addCommas([X],[X]).

addCommas([First|Rest],[First,comma|NewRest]) :-
        not(equals(Rest,[])),
        addCommas(Rest,NewRest).

generateAuxWrap(CompoundStatement,Path,Str) :-
        isCompoundStatement(CompoundStatement),
        generateAux(CompoundStatement,Path,Str).

generateAuxWrap(Construct,Path,Str) :-
        zwrite("{",Str),
        generateAux(Construct,Path,Str),
        zwrite(";}",Str).

generateAux(Construct,Path,Stream) :-
        aspectsEnabled,
        generateBeforeAspects(Construct,Path,Stream),
        generateAux2(Construct,Path,Stream),
        generateAfterAspects(Construct,Path,Stream).

generateAux(Construct,Path,Stream) :-
        generateAux2(Construct,Path,Stream).

generateAux2(Construct,Path,Stream) :-
        not(equals([Construct|_],Path)),
        error("generateAux2/3 convention violated:
               path needs to already include first argument").

generateAux2(Addition,Path,Stream) :-
        additionHasLeftExpression(Addition,Left),
        generateAux(Left,[Left|Path],Stream),
        zwrite(" + ",Stream),
        additionHasRightExpression(Addition,Right),
        generateAux(Right,[Right|Path],Stream).

generateAux2(And,Path,Stream) :-
        andHasLeftExpression(And,Left),
        generateAux(Left,[Left|Path],Stream),
        zwrite(" && ",Stream),
        andHasRightExpression(And,Right),
        generateAux(Right,[Right|Path],Stream).

generateAux2(Assignment,Path,Stream) :-
        assignmentHasLeftExpression(Assignment,Left),
        generateAux(Left,[Left|Path],Stream),
        assignmentHasOperator(Assignment,Op),
        generateAux(Op,[Op|Path],Stream),
        assignmentHasRightExpression(Assignment,Right),
        generateAux(Right,[Right|Path],Stream).

generateAux2(CompoundStatement,Path,Stream) :-
        isCompoundStatement(CompoundStatement),
        zwritenl("{",Stream),
        zwritenl("/* declarations */",Stream),
        compoundStatementHasDeclarations(CompoundStatement,Decls),
        generateDeclarationList(Decls,Path,Stream),
        zwritenl("/* statements */",Stream),
        compoundStatementHasStatements(CompoundStatement,Stats),
        generateStatementList(Stats,Path,Stream),
        zwrite("}",Stream).
```

B.2 Aspect Layer

This section contains all predicates of the aspect layer.

```
generateBeforeAspects(Construct,Path,Stream) :-
        forall(isBeforeAspect(Before),
               generateBeforeAspect(Before,Construct,Path,Stream)).

generateAfterAspects(Construct,Path,Stream) :-
        forall(isAfterAspect(After),
               generateAfterAspect(After,Construct,Path,Stream)).

generateBeforeAspect(Before,Construct,Path,Stream) :-
        equals(Before,before(Construct,Path,Result,Joinpoint,Code)),
        Joinpoint,
        Code,
        generateAux2(Result,[Result|Path],Stream).

generateBeforeAspect(Before,Construct,Path,Stream).

generateAfterAspect(After,Construct,Path,Stream) :-
        equals(After,after(Construct,Path,Result,Joinpoint,Code)),
        Joinpoint,
        Code,
        generateAux2(Result,[Result|Path],Stream).

generateAfterAspect(After,Construct,Path,Stream).

isBeforeAspect(Before) :-
        nonvar(Before),
        equals(Before,before(Construct,Path,Result,Joinpoint,Code)).

isBeforeAspect(Before) :-
        var(Before),
        captureEvent(Construct,Path,When,CodeSpec),
        equals(Before,
               before(Construct,
                      Path,
                      Result,
                      When,
                      transformWhat(Construct,Path,CodeSpec,Result))).

isBeforeAspect(Before) :-
        var(Before),
        equals(Before,before(Construct,Path,Result,Joinpoint,Code)),
        Before.

isAfterAspect(After) :-
        nonvar(After),
        equals(After,after(Construct,Path,Result,Joinpoint,Code)).

isAfterAspect(After) :-
        var(After),
        equals(After,after(Construct,Path,Result,Joinpoint,Code)),
        After.

isAfterAspect(After) :-
        var(After),
        captureEventAfter(Construct,Path,When,CodeSpec),
        equals(After,
               after(Construct,
                     Path,
                     Result,
                     When,
                     transformWhat(Construct,Path,CodeSpec,Result))).

behaveCodeListHasCodeList(BehaveCodeList,CodeList) :-
        equals(BehaveCodeList,behaveCodeList(CodeList)).

isBehaveCode(BehaveCode) :-
        equals(BehaveCode,behaveCode(Code)).

behaveCodeHasCode(BehaveCode,Code) :-
        equals(BehaveCode,behaveCode(Code)).

isBehaveCodeList(BehaveCodeList) :-
        equals(BehaveCodeList,behaveCodeList(CodeList)).

captureEvent(Construct,Path,Translated,RecordAs) :-
        intercept(before,Functor,RecordAs),
        Translated =.. [Functor,Construct,Path].

captureEventAfter(Construct,Path,Translated,RecordAs) :-
        intercept(after,Functor,RecordAs),
        Translated =.. [Functor,Construct,Path].
```

Appendix C

BEHAVE Temporal Logic Meta Interpreter

This appendix is intended to provide documentation and the complete implementation of the temporal logic meta interpreter used within the context of this dissertation. The meta interpreter was implemented by Coen De Roover. It is written as a Prolog extension making use of the Prolog library `clp/bounds`.

C.1 Documentation

These are the temporal operators the temporal logic meta interpreter supports:

`not(A)`	not A
`next(A)`	A at the next point in time
`next(C,A)`	A at C steps in the future - C can be a variable
`previous(A)`	A at previous point in time
`previous(C,A)`	A at C steps in the past - C can be a variable

Table C.1: `Not, Next` and `Previous` Operators

`sometime(-,A)`	A sometime in the past
`sometime(+,A)`	A sometime in the future
`sometime_past(W,A)`	A must be true within now and W steps in the past
`sometime_future(W,A)`	A must be true within now and W steps in the future
`sometime(C,A)`	A must be true within C steps(C can be +/-)
`sometime(A)`	A is true sometime during the whole timeline

Table C.2: `Sometime` Operators

`always(-,A)`	A must always be true in the past
`always(+,A)`	A must always be true in the future
`always_past(A)`	A must always be true in the past
`always_future(A)`	A must always be true in the future
`always(A)`	A must be true during the whole timeline
`always_future(W,A)`	A must be true within W time steps in the future
`always_past(W,A)`	A must be true within W time steps in the past

Table C.3: `Always` Operators

`when(C1,C2)`	If C1 is true at a certain time, then C2 must be true as well
`or(A,B)`	A or B must be true
`and(A,B)`	A and B must be true
`until(A,B)`	B must always be true until at some point A is true

Table C.4: Compound Operators

C.2 Implementation of the meta interpreter

C.2.1 Marking the timeline

```
% SWI-Prolog constraint library
:- use_module(library('clp/bounds')).

max_list([X], X).
max_list([X|Xs], Y) :- max_list(Xs, Z), Y is max(X, Z).


% beginning of time
bot(-1).

% end of time
eot(Time) :-
        findall(T, event(T, _), Ts),
        max_list(Ts, Time).
```

C.2.2 The Temporal Logic Operators

```
solve(A) :-
        prove(A, 0).

prove(not(A), T) :- !,
        not(prove(A, T)).

prove(next(A), T) :- !,
        NT #= T + 1,
        prove(A, NT).

prove(next(C, A), T) :- !,
        C #> 0,
        NT #= T + C,
        prove(A, NT).

prove(previous(A), T) :- !,
        NT #= T - 1,
        prove(A, NT).

prove(previous(C, A), T) :- !,
        C #> 0,
        NT #= T - C,
        prove(A, NT).

prove(sometime(X, A), T) :-
        atom(X),
        X = -,
        prove(sometime_past(_, A), T).

prove(sometime_past(Within, A), T) :-
        Steps #=< 0,
        Steps #>= -Within,
        prove(sometime(Steps, A), T).

prove(sometime(X, A), T) :-
        atom(X),
        X = +,
        prove(sometime_future(_, A), T).

prove(sometime_future(Within,A), T) :-
        Steps #>= 0,
        Steps #=< Within,
        prove(sometime(Steps, A), T).

prove(sometime(C, A), T) :-
        C#>=0,
        bot(Bot),
        eot(Tot),
        NT in Bot..Tot,
        NT #>= T,
        NT #=< T+C,
        prove(A, NT).

prove(sometime(C,A), T) :-
        C #=< 0,
        bot(Bot),
        eot(Tot),
        NT in Bot..Tot,
        NT #>= T + C,
        NT #=< T,
        prove(A, NT).

prove(sometime(A), _) :-
        bot(Bot),
        eot(Tot),
        C in Bot..Tot,
        prove(A, C).

prove(always_future(A), T) :-
        eot(EOT),
        T #= EOT, !,
        prove(A, T).

prove(always_future(A), T) :-
        copy_term(A, AC),
        prove(A, T),
        NT #= T + 1,
        prove(always_future(AC), NT).

prove(always_past(A), T) :-
        bot(BOT),
        T #= BOT, !,
        prove(A, T).

prove(always_past(A), T) :-
        copy_term(A, AC),
        prove(A, T),
        NT #= T - 1,
        prove(always_past(AC), NT).

prove(always(X, A), T) :-
        atom(X),
        X = +,
        prove(always_future(A), T).

prove(always(X, A), T) :-
        atom(X),
        X = -,
        prove(always_past(A), T).

prove(always(A), T) :-
        copy_term(A, AC),
        prove(always_past(A), T),
        prove(always_future(AC), T).

prove(always_future(Within, A), T) :-
         Within #= 0, !,
         prove(A,T).

prove(always_future(Within, A), T) :-
```

```
         copy_term(A, AC),
         prove(A, T),
         NT #= T + 1,
         NW #= Within - 1,
         prove(always_future(NW, AC), NT).

prove(always_past(Within, A), T) :-
        Within #= 0, !,
        prove(A,T).

prove(always_past(Within, A), T) :-
        copy_term(A, AC),
        prove(A, T),
        NT #= T - 1,
        NW #= Within - 1,
        prove(always_past(NW, AC), NT).

prove(when(C1, C2), T) :-
        prove(C1,T) -> prove(C2,T) ; true.

prove(X #= Y, _) :-
        X #= Y.

prove(or(A,B), T) :- !,
        (prove(A, T) ; prove(B, T)).

prove(and(A,B), T) :- !,
        prove(A, T),
        prove(B, T).

until(A,B) :-
        sometime_future(S, A),
        S1 #= S - 1,
        always_future(S1, B).
```

C.2.3 The `prove/2` predicate

Clause A is a built-in predicate:

```
prove(A, _) :-
        predicate_property(A, built_in), !,
        call(A).
```

Clause A is found in the normal repository:

```
prove(A, T) :-
        clause(A, B),
        %prove(B, 0).
        prove(B, T).
```

Clause A is found in the execution trace repository where predicates have an extra argument:

```
prove(A, T) :-
        A =.. [Predicate | Arguments],
        append([Predicate, T], Arguments, Extended),
        Term =.. Extended,
        clause(Term, B),
        writeln(Term),
        prove(B, T).
```

Appendix D

BEHAVE for Pico 1.0

This appendix is intended as a BEHAVE reuse framework for reasoning about the behaviour of Pico 1.0. We list here all used predicates for documenting and verifying the design invariants from chapter 6.

Section D.1 first documents all reasoning predicates and the used keywords, while section D.2 provides the declarations that implement these predicates.

D.1 Documentation

Table D.1 lists the Pico 1.0 reasoning predicates which were directly used for intercepting run-time events. The declarations implementing these predicates are shown in section D.2.2. Note that the pointcut descriptions used in the `intercept` predicates (denoted in bold in section D.2.1) appear as atoms. However, BEHAVE adds variables `C` and `P` which represent a C construct and the parse tree path respectively. So these pointcut descriptions always represent a predicate of the form `predicate(C,P)` (as can be seen from table D.1) and in an `intercept` predicate they are denoted as the atom `predicate`.

`continuationEntry(C,P)`	Construct C on path P is at the entry of a continuation
`continuationExit(C,P)`	Construct C on path P is at the exit of a continuation
`peekExp(C,P)`	Construct C peeks the Pico expression stack
`popExp(C,P)`	Construct C does a pop from the Pico expression stack
`varAssignment(C,P)`	Construct C represents a variable assignment
`tempVarUsed(C,P)`	The value of temporary variable C has been used
`pushRetCnt(C,P)`	Construct C pushes a RET cont. on the CNT stack
`pokeRetCnt(C,P)`	Construct C pokes a RET cont. on the CNT stack

Table D.1: Pico Reasoning Predicates

Table D.2 lists other Pico reasoning predicates which are used in the bodies of the above mentioned predicates in table D.1.

`inContinuation(Path)`	Path is the path of a continuation
`inContinuation(Path,Cont)`	Path is the path of a continuation Cont
`functionEntry(C,P)`	Construct C with path P is the entry of a function
`functionExit(C,P)`	Construct C with path P is the exit of a function
`continuation(Construct)`	Construct C is a continuation
`manipulatesPicoStack(C)`	Construct C uses one of the pico stacks
`cntStack(C)`	Construct C uses the continuation stack
`expStack(C)`	Construct C uses the expression stack
`tempVariable(C,P)`	Construct C is a temporary variable
`assignmentToVariable(A,V)`	A is an assignment expr. of variable V
`macroCallArgument(M,A)`	M is a macro call with A as first argument

Table D.2: Other Pico Reasoning Predicates

`time`	time stamp
`cntName`	name of a Pico continuation function
`cntPointer`	pointer of a continuation
`cntStack`	state of the entire continuation stack
`assVar`	name of a variable being assigned
`macroVar`	name of a variable which is the argument of a macro call
`var(V)`	a variable V
`topCnt`	top of the CNT stack
`uTopCnt`	element right under the top of the CNT stack

Table D.3: Pico 1.0 keywords

Table D.3 lists all keywords representing Pico run-time values which can easily be reused for associating them with other Pico high-level run-time events. They return values depending on the *context* at the time they are called and the keywords log their value at that time during execution.

D.2 Implementation of all predicates

D.2.1 User-defined `Intercept/3` predicates

```
intercept(before,continuationEntry,
   event(time,cntEntered(cntName,cntPointer,cntStack))).
```

```
intercept(after,continuationExit,
   event(time,cntExited(cntName,cntPointer,cntStack))).

intercept(before,gcPossible,
   event(time,gcPossible(cntName))).

intercept(after,peekExp,
   event(time,tempUpdated(cntName,macroVar))).

intercept(after,popExp,
   event(time,tempUpdated(cntName,macroVar))).

intercept(after,varAssignment,
   event(time,tempUpdated(cntName,assVar))).

intercept(instead,tempVarUsed,
   event(time,tempUsed(cntName,varName))).

intercept(before,pushRetCnt,
   event(time,pushReturnOnCntStack(cntName,topCnt))).

intercept(before,pokeRetCnt,
   event(time,pokeReturnOnCntStack(cntName,utopCnt))).
```

D.2.2 Reasoning Predicates

This section lists all reasoning predicates which are specifically used for Pico. Other predicates used here which belong to the basic layer are not repeated here. Predicates which were used directly as pointcut description in the `intercept` predicates in section D.2.1 are denoted in bold.

```
continuationEntry(Construct,Path) :-
   functionEntry(Construct,Path),
   inContinuation(Path).

continuationExit(Construct,Path) :-
   functionExit(Construct,Path),
   inContinuation(Path).

inContinuation(Path) :-
   inContinuation(Path,_).

inContinuation(Path,Continuation) :-
   last(Path,Continuation),
   continuation(Continuation).

functionEntry(Construct,Path) :-
   listAt(4,Path,Function),
   isFunctionDefinition(Function),
   listAt(3,Path,Body),
   compoundStatementHasStatements(Body,Statements),
   first(Statements,Construct).

functionExit(Construct,Path) :-
   listAt(4,Path,Function),
   isFunctionDefinition(Function),
   listAt(3,Path,Body),
   compoundStatementHasStatements(Body,Statements),
   last(Statements,Construct).

continuation(Construct) :-
   isFunctionDefinition(Construct),
   expressionIn(Construct,Expression,_),
   manipulatesPicoStack(Expression).

manipulatesPicoStack(Expression) :-
   cntStack(Expression).
manipulatesPicoStack(Expression) :-
   expStack(Expression).

cntStack(Construct) :-
   peekCnt(Construct).
cntStack(Construct) :-
   pokeCnt(Construct).
cntStack(Construct) :-
   popCnt(Construct).
cntStack(Construct) :-
   pushCnt(Construct).
cntStack(Construct) :-
   zapCnt(Construct).

expStack(Construct) :-
        peekExp(Construct).
expStack(Construct) :-
        pokeExp(Construct).
expStack(Construct) :-
        popExp(Construct).
expStack(Construct) :-
        pushExp(Construct).
expStack(Construct) :-
        zapExp(Construct).

gcPossible(Construct,Path) :-
        macroCallHasName(Construct,'_stk_claim_').
gcPossible(Construct,Path) :-
        macroCallHasName(Construct,'_mem_claim_').
gcPossible(Construct,Path) :-
        macroCallHasName(Construct,'_mem_claim_SIZ_').
gcPossible(Construct,Path) :-
```

```
        macroCallHasName(Construct,'_mem_claim_STR_').

peekExp(Construct,Path) :-
   macroCallHasName(Construct,'_stk_peek_EXP_').

popExp(Construct,Path) :-
   macroCallHasName(Construct,'_stk_pop_EXP_').

varAssignment(Construct,Path) :-
   assignmentHasLeftExpression(Construct,Var),
   tempVar(Var).

tempVarUsed(Construct,Path) :-
   tempVariable(Construct,Path),
   not(leftValue(Construct,Path)),
   inContinuation(Path).

tempVariable(Construct,Path) :-
   identifierHasSymbol(Construct,Var),
   declaredVariableAs(Path,Var,'_EXP_TYPE_').

assignmentToVariable(Assignment,Variable) :-
   assignmentHasLeftExpression(Assignment,Identifier),
   identifierHasSymbol(Identifier,Variable).

macroCallArgument(MacroCall,Argument) :-
   macroCallHasArguments(MacroCall,ArgumentList),
   first(ArgumentList,Argument).

pushRetCnt(Construct) :-
   macroCallHasName(Construct,'_stk_push_CNT_'),
   macroCallHasArguments(Construct,Lst),
   member(identifier('RET'),Lst).

pokeRetCnt(Construct) :-
   macroCallHasName(Construct,'_stk_poke_CNT_'),
   macroCallHasArguments(Construct,Lst),
   member(identifier('RET'),Lst).
```

D.2.3 Behavioural Models

This section lists the three behavioural models for the three Pico design invariants.

Design Invariant 1: Active Behavioural Documentation

```
cntDocumented('ASS',['ASS'|R],R).
cntDocumented('REF',['REF'|R],['REF','APL'|R]).
...

behaviouralModel :-
$\Box$(when(cntExecuted(Name,Before,After),
       cntDocumented(Name,Before,After))).

cntExecuted(Name,StackBefore,StackAfter) :-
   cntExited(Name,_,StackAfter),
   $\bullet^{t}$cntEntered(Name,_,StackBefore).
```

Design Invariant 2: Garbage Collection

```
possibleGc(Cont) :-
   event(possibleGc(Cont)).
tempUsed(Cont,Var) :-
   event(tempUsed(Cont,Var)).
tempUpdated(Cont,Var) :-
   event(tempUpdated(Cont,Var)).

safeToUseTemp(Cont,Var) :-
    $\bullet^{t}$tempUpdated(Cont,Var),
    $\neg\Diamond^{-t}$possibleGc(_).

unsafeUseOfTemp(Cont,Var) :-
   tempUsed(Cont,Var),
   $\neg$safeToUseTemp(Cont,Var).

behaviouralModel(Continuation(C),variable(V)) :-
    $\Diamond$unsafeUseOfTemp(C,V).
```

Design Invariant 3: Tail Recursion

```
continuationExecute(Name,Pointer) :-
  event(continuationEntered(Name,Pointer,_)).

installReturnOnContinuationStack(Cntname,Topcnt) :-
   event(pushReturnOnCntStack(Cntname,CntPointer,Topcnt)).
installReturnOnContinuationStack(Cntname,Topcnt) :-
  event(pokeReturnOnCntStack(Cntname,CntPointer,Topcnt)).

possibleTailRecursionOptimization(Cntname) :-
   continuationExecute('RET',Pointer),
   $\Diamond^{-}$installReturnOnContinuationStack(Cntname,Pointer)).
```

D.2.4 Keywords

This section lists all keywords used for verifying the three design invariants for Pico 1.0. They represent Pico run-time values to be associated with high-level events. They can be reused for capturing run-time values associated with other design invariants for Pico 1.0.

```
keyword(time,Construct,Path,"behaveLog(\"%i\", TIME++);").

keyword(comma,Construct,Path,"behaveLog(\",\");").

keyword(cntName,C,P,Result) :-
   continuationName(C,P,Name),
   concat(['behaveLog("',Name,'");'],Result).

keyword(cntPointer,Construct,Path,Result) :-
   continuationName(Construct,Path,Name),
   concat(["behaveLog(\"%i\",",Name,");"],Result).

keyword(cntStack,Construct,Path,Result) :-
   equals(Result,
   ";{/* cntStack dump Pico 1.0 */
   _EXP_TYPE_ exp;
   _UNS_TYPE_ idx, siz;
   behaveLog(\"[\");
   siz = _ag_get_TAB_SIZ_(_STK_);
   idx = ((_CNT_ - (_UNS_TYPE_)(_mem_STORE_+_STK_.ptr.ofs))
          /_EXP_SIZE_);
   for ( ; idx < siz ; idx++) {
       exp = _ag_get_TAB_EXP_(_STK_, idx);
       behaveLog(\"%i,\", exp.cnt);
   }
   exp = _ag_get_TAB_EXP_(_STK_, siz);
   behaveLog(\"%i]\", exp);
   };").

keyword(var(V),Construct,Path,Result) :-
   concat(["behaveLog("",V,"\");"],Result).

keyword(assVar,Construct,Path,Result) :-
   assignmentToVariable(Construct,Var),
   keyword(var(Var),Construct,Path,Result).

keyword(macroVar,Construct,Path,Result) :-
   macroCallArgument(Construct,Arg),
   keyword(var(Arg),Construct,Path,Result).

keyword(topCnt,Construct,Path,Result) :-
   equals(Result,
   ";{/* cntStackpeektop Pico 1.0 */
   _CNT_TYPE_ cnt;
   _stk_peek_CNT_(cnt);
   behaveLog(\"%i\",cnt);
   };").

keyword(utopCnt,Construct,Path,Result) :-
   equals(Result,
   ";{/* cntStackpeekundertop Pico 1.0 */
   _CNT_TYPE_ cnt,temp;
   _stk_pop_CNT_(temp);
   _stk_peek_CNT_(cnt);
   _stk_push_CNT_(temp);
   behaveLog(\"%i\",cnt);
   };").
```

The two keywords below represent alternative versions of **topCnt** and **utopCnt** that are more efficient, i.e. they do not influence continuation stack behaviour. The above versions are more intuitive, but they use the continuation stack at run-time.

```
keyword('topCnt2',Construct,Path,Result) :-
   equals(Result,
   ";{/* cntStack dump Pico 1.0 */
   _EXP_TYPE_ exp;
   _UNS_TYPE_ idx;
   idx = ((_CNT_ - (_UNS_TYPE_)(_mem_STORE_+_STK_.ptr.ofs))/_EXP_SIZE_);
   exp = _ag_get_TAB_EXP_(_STK_, idx);
   behaveLog(\"%i\", exp.cnt);
   };").

keyword('utopCnt2',Construct,Path,Result) :-
   equals(Result,
   ";{/* cntStack dump Pico 1.0 */
   _EXP_TYPE_ exp;
   _UNS_TYPE_ idx;
   idx = ((_CNT_ - (_UNS_TYPE_)(_mem_STORE_+_STK_.ptr.ofs))/_EXP_SIZE_);
   idx = idx+1;
   exp = _ag_get_TAB_EXP_(_STK_, idx);
   behaveLog(\"%i\",exp.cnt);
   };").
```

Appendix E
BEHAVE generated code excerpt

This appendix contains an example of generated code of the BEHAVE platform. It shows an excerpt of the file PicoEva.c after performing the code instrumentation for verifying the tail recursion design invariant from section 6.5. The instrumentation code is put in **bold**. On the first line, the file `behave.h` is included which contains the implementation for the `behaveLog` function used in the instrumentation code which writes strings to a file.

```
intercept(before,
          pushRetCnt,
          event(time,pushReturnOnCntStack(cntName,topCnt))).
intercept(before,
          pokeRetCnt,
          event(time,pokeReturnOnCntStack(cntName,utopCnt))).
intercept(before,
          continuationEntry,
          event(time,cntEntered(cntName,cntPointer,cntStack))).
```

High-level events specification

The above `intercept` predicates were evaluated for generating the following instrumented code excerpt:

```
#include "behave.h"

/* System includes */

/* User includes */
#include "Pico.h"
#include "PicoMai.h"
#include "PicoEnv.h"
#include "PicoMem.h"
#include "PicoNat.h"
#include "PicoEva.h"

/* Macro definitions */
#define CAL _eval_CAL_
#define EXP _eval_EXP_
#define NAT _eval_NAT_
```

```
/* Function macros */

/* Declarations */
static _NIL_TYPE_  APL(_NIL_TYPE_);
static _NIL_TYPE_  ASS(_NIL_TYPE_);
...

/* Function definitions */

static _NIL_TYPE_ ASS(_NIL_TYPE_){
/* declarations */
_EXP_TYPE_  dct, val;
/* statements */
;{ behaveLog("event(");behaveLog("%i", TIME++);behaveLog(",");
behaveLog("continuationEntered(");behaveLog("'ASS'");
behaveLog(",");behaveLog("%i",ASS);behaveLog(",");
;{/* cntStack dump Pico 1.0 */
_EXP_TYPE_ exp;
_UNS_TYPE_ idx, siz;
 behaveLog("[")
siz = _ag_get_TAB_SIZ_(_STK_);
idx = ((_CNT_ - (_UNS_TYPE_)(_mem_STORE_+_STK_.ptr.ofs))/_EXP_SIZE_);
for ( ; idx < siz ; idx++) {
    exp = _ag_get_TAB_EXP_(_STK_, idx);
    behaveLog("%i,", exp.cnt);
}
exp = _ag_get_TAB_EXP_(_STK_, siz);
behaveLog("%i]", exp);
}; behaveLog(")"); behaveLog(")");behaveLog(".\n")}
_stk_pop_EXP_(val);
_stk_peek_EXP_(dct);
_ag_set_DCT_VAL_(dct,val);
_ag_set_DCT_DCT_(dct,_DCT_);
_DCT_ = dct;
_stk_poke_EXP_(val);
_stk_zap_CNT_();
}

static _NIL_TYPE_ ATA(_NIL_TYPE_){
/* declarations */
_EXP_TYPE_  act, apl, arg, dct, exp, fun, nam, nbr, par, tab;
_CNT_TYPE_  cnt;
_UNS_TYPE_  ctr, siz;
/* statements */
;{ behaveLog("event(");behaveLog("%i", TIME++);behaveLog(",");
behaveLog("continuationEntered(");behaveLog("'ASS'");
behaveLog(",");behaveLog("%i",ASS);behaveLog(",");
;{/* cntStack dump Pico 1.0 */
_EXP_TYPE_ exp;
_UNS_TYPE_ idx, siz;
 behaveLog("[")
siz = _ag_get_TAB_SIZ_(_STK_);
idx = ((_CNT_ - (_UNS_TYPE_)(_mem_STORE_+_STK_.ptr.ofs))/_EXP_SIZE_);
for ( ; idx < siz ; idx++) {
    exp = _ag_get_TAB_EXP_(_STK_, idx);
    behaveLog("%i,", exp.cnt);
}
exp = _ag_get_TAB_EXP_(_STK_, siz);
behaveLog("%i]", exp);
}; behaveLog(")"); behaveLog(")");behaveLog(".\n")}
_mem_claim_();
fun = _ag_make_FUN_();
_stk_pop_EXP_(apl);
_stk_pop_EXP_(nbr);
_stk_pop_EXP_(tab);
_stk_pop_EXP_(arg);
_stk_peek_EXP_(dct);
siz = _ag_get_TAB_SIZ_(arg);
ctr = _ag_get_NBU_(nbr);
act = _ag_get_TAB_EXP_(arg,ctr);
nam = _ag_get_APL_NAM_(apl);
par = _ag_get_APL_ARG_(apl);
_ag_set_FUN_NAM_(fun,nam);
_ag_set_FUN_ARG_(fun,par);
_ag_set_FUN_EXP_(fun,act);
_ag_set_FUN_DCT_(fun,dct);
_ag_set_TAB_EXP_(tab,ctr,fun);
if (ctr < siz) {
/* declarations */
/* statements */
_stk_push_EXP_(arg);
_stk_push_EXP_(tab);
nbr = _ag_succ_NBR_(nbr);
_stk_push_EXP_(nbr);
_stk_push_EXP_(apl);
} else {
/* declarations */
/* statements */
_stk_zap_EXP_();
_stk_peek_EXP_(exp);
_ag_set_DCT_VAL_(dct,tab);
_stk_poke_EXP_(_DCT_);
_stk_push_EXP_(exp);
_stk_pop_CNT_(cnt);
;{ behaveLog("event(");behaveLog("%i", TIME++);behaveLog(",");
behaveLog("pushReturnOnCntStack(");behaveLog("'ATA'");
behaveLog(",");
;{/* cntStackpeektop Pico 1.0 */
_CNT_TYPE_ cnt;
_stk_peek_CNT_(cnt);
behaveLog("%i",cnt);
}; behaveLog(")"); behaveLog(")");behaveLog(".\n")}
_stk_push_CNT_(RET);
_stk_push_CNT_(EXP);
_DCT_ = dct;
};
}

static _NIL_TYPE_ ATV(_NIL_TYPE_){
/* declarations */
_EXP_TYPE_  act, arg, dct, exp, nbr, tab, val;
_CNT_TYPE_  cnt;
_UNS_TYPE_  ctr, siz;
/* statements */
;{ behaveLog("event(");behaveLog("%i", TIME++);behaveLog(",");
behaveLog("continuationEntered(");behaveLog("'ASS'");
behaveLog(",");behaveLog("%i",ASS);behaveLog(",");
;{/* cntStack dump Pico 1.0 */
_EXP_TYPE_ exp;
_UNS_TYPE_ idx, siz;
 behaveLog("[")
siz = _ag_get_TAB_SIZ_(_STK_);
idx = ((_CNT_ - (_UNS_TYPE_)(_mem_STORE_+_STK_.ptr.ofs))/_EXP_SIZE_);
for ( ; idx < siz ; idx++) {
    exp = _ag_get_TAB_EXP_(_STK_, idx);
    behaveLog("%i,", exp.cnt);
}
exp = _ag_get_TAB_EXP_(_STK_, siz);
behaveLog("%i]", exp);
}; behaveLog(")"); behaveLog(")");behaveLog(".\n")}
_stk_claim_();
_stk_pop_EXP_(val);
_stk_pop_EXP_(nbr);
_stk_pop_EXP_(tab);
_stk_peek_EXP_(arg);
siz = _ag_get_TAB_SIZ_(arg);
ctr = _ag_get_NBU_(nbr);
_ag_set_TAB_EXP_(tab,ctr,val);
if (ctr < siz) {
/* declarations */
/* statements */
act = _ag_get_TAB_EXP_(arg,ctr + 1);
_stk_push_EXP_(tab);
nbr = _ag_succ_NBR_(nbr);
_stk_push_EXP_(nbr);
_stk_push_EXP_(act);
_stk_push_CNT_(EXP);
} else {
/* declarations */
/* statements */
_stk_zap_EXP_();
_stk_pop_EXP_(dct);
_ag_set_DCT_VAL_(dct,tab);
_stk_peek_EXP_(exp);
_stk_poke_EXP_(_DCT_);
_DCT_ = dct;
_stk_push_EXP_(exp);
_stk_pop_CNT_(cnt);
;{ behaveLog("event(");behaveLog("%i", TIME++);behaveLog(",");
behaveLog("pushReturnOnCntStack(");behaveLog("'ATV'");
behaveLog(",");;{/* cntStackpeektop Pico 1.0 */
_CNT_TYPE_ cnt;
_stk_peek_CNT_(cnt);
behaveLog("%i",cnt);
}; behaveLog(")"); behaveLog(")");behaveLog(".\n")}
_stk_push_CNT_(RET);
_stk_push_CNT_(EXP);
};
}

static _NIL_TYPE_ BND(_NIL_TYPE_){
/* declarations */
_EXP_TYPE_  act, arg, dct, exp, fun, frm, nam, nbr, par, val, xdc;
_CNT_TYPE_  cnt;
_TAG_TYPE_  tag;
_UNS_TYPE_  ctr, siz;
/* statements */
;{ behaveLog("event(");behaveLog("%i", TIME++);behaveLog(",");
behaveLog("continuationEntered(");behaveLog("'ASS'");
behaveLog(",");behaveLog("%i",ASS);behaveLog(",");
;{/* cntStack dump Pico 1.0 */
_EXP_TYPE_ exp;
_UNS_TYPE_ idx, siz;
 behaveLog("[")
siz = _ag_get_TAB_SIZ_(_STK_);
idx = ((_CNT_ - (_UNS_TYPE_)(_mem_STORE_+_STK_.ptr.ofs))/_EXP_SIZE_);
for ( ; idx < siz ; idx++) {
    exp = _ag_get_TAB_EXP_(_STK_, idx);
    behaveLog("%i,", exp.cnt);
}
exp = _ag_get_TAB_EXP_(_STK_, siz);
behaveLog("%i]", exp);
}; behaveLog(")"); behaveLog(")");behaveLog(".\n")}
_stk_claim_();
```

```
_mem_claim_();
_stk_pop_EXP_(val);
_stk_pop_EXP_(dct);
_ag_set_DCT_VAL_(dct,val);
_stk_pop_EXP_(nbr);
_stk_pop_EXP_(arg);
siz = _ag_get_TAB_SIZ_(arg);
ctr = _ag_get_NBU_(nbr);
if (ctr == siz) {
/* declarations */
/* statements */
_stk_zap_EXP_();
_stk_zap_CNT_();
_stk_peek_CNT_(cnt);
if (cnt != RET) {
/* declarations */
/* statements */
_stk_peek_EXP_(exp);
_stk_poke_EXP_(_DCT_);
_stk_push_EXP_(exp);
;{ behaveLog("event(");behaveLog("%i", TIME++);behaveLog(",");
behaveLog("pushReturnOnCntStack(");behaveLog("'BND'");
behaveLog(",");
;{/* cntStackpeektop Pico 1.0 */
_CNT_TYPE_ cnt;
_stk_peek_CNT_(cnt);
behaveLog("%i",cnt);
}; behaveLog(")"); behaveLog(")");behaveLog(".\n")}
_stk_push_CNT_(RET);
};
_stk_push_CNT_(EXP);
_DCT_ = dct;
} else {
...
};
};
}
...

static _NIL_TYPE_ RET(_NIL_TYPE_){
/* declarations */
_EXP_TYPE_  val;
/* statements */
;{ behaveLog("event(");behaveLog("%i", TIME++);behaveLog(",");
behaveLog("continuationEntered(");behaveLog("'ASS'");
behaveLog(",");behaveLog("%i",ASS);behaveLog(",");
;{/* cntStack dump Pico 1.0 */
_EXP_TYPE_ exp;
_UNS_TYPE_ idx, siz;
 behaveLog("[")
siz = _ag_get_TAB_SIZ_(_STK_);
idx = ((_CNT_ - (_UNS_TYPE_)(_mem_STORE_+_STK_.ptr.ofs))/_EXP_SIZE_);
for ( ; idx < siz ; idx++) {
    exp = _ag_get_TAB_EXP_(_STK_, idx);
    behaveLog("%i,", exp.cnt);
}
exp = _ag_get_TAB_EXP_(_STK_, siz);
behaveLog("%i]", exp);
}; behaveLog(")"); behaveLog(")");behaveLog(".\n")}
_stk_pop_EXP_(val);
_stk_peek_EXP_(_DCT_);
_stk_poke_EXP_(val);
_stk_zap_CNT_();
_ESCAPE_;
}
```

Bibliography

[ABE$^+$04] Bengt Ahlgren, Marcus Brunner, Lars Eggert, Robert Hancock, and Stefan Schmid. Invariants: A New Design Methodology for Network Architectures. In *FDNA '04: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 65–70, New York, NY, USA, 2004. ACM Press.

[AJU02] Mikhail Auguston, Clinton Jeffery, and Scott Underwood. A Framework for Automatic Debugging. In *Proc. of the 17th IEEE Intl. Conf. on Automated Software Engineering (ASE02)*, page 217, 2002.

[And98] James H. Andrews. Testing using Log File Analysis: Tools, Methods, and Issues. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, page 157, Washington, DC, USA, 1998. IEEE Computer Society.

[ARTZ03] Paul Anderson, Thomas Reps, Tim Teitelbaum, and Mark Zarins. Tool Support for Fine-Grained Software Inspection. *IEEE Software*, 20(4):42–50, 2003.

[AS84] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.

[Asp] AspectJ general website - http://www.eclipse.org/aspectj/.

[AT01] Paul Anderson and Tim Teitelbaum. Software Inspection using Codesurfer. In *Proceedings of the 1st Workshop on Inspection in Software Engineering (WISE01)*, pages 4–11, 2001.

[Aug98] Mikhail Auguston. Building Program Behavior Models. In *Proc. of the European Conf. on Artificial Intelligence Worksh. on Spatial and Temporal Reasoning (ECAI98)*, pages 19–26, 1998.

[Aug00] Mikhail Auguston. Tools for Program Dynamic Analysis, Testing, and Debugging Based on Event Grammars. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 159–166, July 6-8 2000.

[Bal] Thomas Ball. The Essence of Dynamic Analysis. Powerpoint Presentation - (part of) a course on Software Reliablility, Purdue University.

[Bal99] Thomas Ball. The Concept of Dynamic Analysis. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 216–234, London, UK, 1999. Springer-Verlag.

[BBF$^+$99] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification - Model Checking Techniques and Tools - chapter 2.* Springer, 1999.

[BDF$^+$04] Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology, vol. 3 no. 6, Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs*, June 2004.

[Bei90] Boris Beizer. *Software Testing Techniques*, chapter 1 (Introductory concepts). John Wiley & Sons, Inc., New York, NY, USA, 1990.

[BMBL05] Jürgen Börstler, Isabel Michiels, Kim B. Bruce, and Morten Lindholm. Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts. In *Object Oriented Technology ECOOP 2005 Workshop Reader*, Lecture Notes in Computer Science, 2005.

[BMF04] Jürgen Börstler, Isabel Michiels, and Annita Fjuk. Eighth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts. In J. Malenfant and B. M. Østvold, editors, *Object Oriented Technology ECOOP 2004 Workshop Reader*, volume 3344 of *Lecture Notes in Computer Science*, pages 36–48, 2004.

[Bol04] Tommaso Bolognesi. A Conceptual Framework for State-Based and Event-Based Formal Behavioural Specification Languages. In *ICECCS '04: Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age (ICECCS'04)*, pages 107–116, Washington, DC, USA, 2004. IEEE Computer Society.

[BR] John Brant and Don Roberts. The Refactory, inc. - TOOLS at www.refactory.com - SmaCC.

[Bri05] Johan Brichau. *Integrative Composition of Program Generators*. PhD thesis, Vrije Universiteit Brussel, 2005.

[Brz95] Christoph Brzoska. Temporal Logic Programming with Metric and Past Operators. In *Proceedings of the Workshop on Executable Modal and Temporal Logics*, pages 21–39, 1995.

[BY01] Luciano Baresi and Michal Young. Test Oracles. Technical report cis-tr01 -02, Department of Computer and Information Science, University of Oregon, 2001.

[CGP02] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software Model Checking in Practice: An Industrial Case Study. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 431–441, New York, NY, USA, 2002. ACM Press.

[cin] Cincom Smalltalk - http://smalltalk.cincom.com.

[CWA$^+$96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. of the 21st Intl. Conf. on Software Engineering (ICSE99)*, pages 411–420, 1999.

[Deg] Jutta Degener. Ansi C Yacc Grammar - http://www.lysator.liu.se/c/ansi-c-grammar-y.html.

[DFW04] Stéphane Ducasse, Michael Freidig, and Roel Wuyts. Logic and Trace-based Object-Oriented Application Testing. In *Proc. of the Intl. Worksh. on Object-Oriented Reengineering (WOOR04)*, 2004.

[DGD05] Coen De Roover, Kris Gybels, and Theo D'Hondt. Towards Abstract Interpretation for Recovering Design Information. In *Proc. of the 1st Intl. Worksh. on Abstract Interpretation of Object-oriented Languages (AIOOL05)*, volume 131 of *Electronic Notes in Theoretical Computer Science*, pages 15–25, 2005.

[DM] Theo D'Hondt and Wolfgang De Meuter. The Pico Programming Language Interpreter, http://pico.vub.ac.be.

[DM00] Theo D'Hondt and Isabel Michiels. Combatting the Paucity of Paradigms in Current OOP Teaching. In *ECOOP 2000 Workshop on Tools and Environments for Understanding Object-Oriented Concepts*, 2000.

[Duc99] Mireille Ducassé. Coca: An Automated Debugger for C. In *Proc. of the 21st Intl. Conf. on Software engineering (ICSE99)*, pages 504–513, 1999.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[EP06] David Evans and Michael Peck. Inculcating Invariants in Introductory Courses. In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, pages 673–678, New York, NY, USA, 2006. ACM Press.

[EPG$^+$07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 2007.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants - Chapter 1.* PhD thesis, University of Washington, Department of Computer Science and Engineering, Seattle, Washington, USA, August 2000.

[Ern03] Michael D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In ICSE Workshop on Dynamic Analysis (WODA), Portland, Oregon, USA, May 2003., 2003.

[Eva01] David Evans. Teaching Software Engineering Using Lightweight Analysis. Selected proposal for the NSF Course, Curriculum and Laboratory Improvement (CCLI) program, 2001.

[Fla94] Peter Flach. *Simply Logical.* John Wiley, 1994.

[FM04] Johan Fabry and Tom Mens. Language-Independent Detection of Object-Oriented Design Patterns. *Elsevier International Journal on Computer Languages, Systems & Structures - Proceedsings of the ESUG 2004 Conference.*, 30(1-2):21–33, 2004.

[FPB87] Jr. Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[GB03] Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM Press.

[GDJ02] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – A Tool for Dynamic Analysis of Java Programs. In *Proc. of the 17th Conf. on Automated Software Engineering*, pages 117–126, 2002.

[GOA05] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational Queries Over Program Traces. In *Proc. of the 20th annual ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages and Applications (OOPSLA05)*, pages 385–402, 2005.

[Gri87] David Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1987.

[Gro06] *The Object Constraint Language Specification, version 2.0*, chapter 7 - OCL Language Description. Object Management Group - OMG, 2006.

[Guo06] Philip Jia Guo. A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs - chapter 1. Master's thesis, Massachusetts Institute of Technology (MIT), 2006.

[Gyb05] Kim Gybels. A Declarative Meta-Programming Platform for the Domain-specific Run-time Verification of Imperative Programs. Master's thesis, Vrije Universiteit Brussel, Belgium, September 2005.

[Ham04] Paul Hamill. *Unit Test Frameworks*. O'Reilly, 2004.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A System and Language for Building System-Specific, Static Analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, New York, NY, USA, 2002. ACM Press.

[HL95] Walter Hürsch and Cristina Lopes. Separation of Concerns. Technical report, Northeastern University, Boston, February 1995.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM Press.

[HLL04] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A Survey of Trace Exploration Tools and Techniques. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[Hoa69] Charles A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] Charles A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*, chapter 1. Prentice Hall, 1991.

[Hol02a] Gerard J. Holzmann. The Logic of Bugs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 81–87, New York, NY, USA, 2002. ACM Press.

[Hol02b] Gerard J. Holzmann. Static Source Code Checking for User-Defined Properties. In *Proceedings of IDPT 2002*, Pasadena, CA, USA, 2002.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Pages 109-119. Addison Wesley, 2000.

[JR00] Daniel Jackson and Martin Rinard. Software Analysis: A Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 133–145, New York, NY, USA, 2000. ACM Press.

[KF04] Murat Karaorman and Jay Freeman. jMonitor: Java Runtime Event Specification and Monitoring Library. In *Proc. of the 4th Intl. Worksh. on Run-time Verification (RV04)*, volume 113 of *Electronic Notes in Theoretical Computer Science*, pages 181–200, 2004.

[KM05a] Gregor Kiczales and Mira Mezini. Aspect-Oriented Programming and Modular Reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, 2005.

[KM05b] Gregor Kiczales and Mira Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Pringer Berlin / Heidelberg, 2005.

[LC92] Yingsha Liao and Donald Cohen. A Specificational Approach to High-Level Program Monitoring and Measuring. *IEEE Transactions on Software Engineering*, 18(11):969–978, 1992.

[LG00] Guy Leduc and François Germeau. Verification of Security Protocols using LOTOS – Method and Application. *Computer Communications*, 2000.

[LM04] Rustan M. Leino and Peter Müller. Object Invariants in Dynamic Contexts. In *Proceedings of the Eighteenth European Conference on Object-Oriented Programming (ECOOP 2004)*, 2004.

[MBB02] Isabel Michiels, Jürgen Börstler, and Kim B. Bruce. Sixth Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts. In J. Hernández and A. Moreira, editors, *Object Oriented Technology ECOOP 2002 Workshop Reader*, volume 2548 of *Lecture Notes in Computer Science*, pages 30–43, 2002.

[MBB03] Isabel Michiels, Jürgen Börstler, and Kim B. Bruce. Seventh Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts. In Frank Buschmann and Alejandro Buchmann, editors, *Object Oriented Technology ECOOP 2003 Workshop Reader*, volume 3013 of *Lecture Notes in Computer Science*, pages 119–129, 2003.

[MBFP00] Isabel Michiels, Jürgen Börstler, Alejandro Fernandez, and Maximo Prieto. Tools and Environments for Understanding Object-Oriented Concepts. In J. Malenfant and S. Moisan, editors, *Object Oriented Technology ECOOP 2000 Workshop Reader*, volume 1964 of *Lecture Notes in Computer Science*, pages 65–77, 2000.

[MBZR03] Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. Towards a Taxonomy of Software Evolution. Technical report vub-prog-tr-02-05, VUB, Programming Technology Lab, Brussels, Belgium., 2003.

[MDD04] Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker. Pico: Scheme for Mere Mortals. In *Online Proceedings of the 1st European Lisp and Scheme Workshop*, 2004.

[MDTZ03] Isabel Michiels, Dirk Deridder, Herman Tromp, and Andy Zaidman. Identifying Problems in Legacy Software - Preliminary Findings of the ARRIBA Project. In *ELISA Workshop at the International Conference on Software Maintenance (ICSM)*, 2003.

[Mey92a] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[Mey92b] Bertrand Meyer. Design by Contract: Building Bug-Free OO Software. *Hotline on Object-Oriented Technology - Revised version (2000) online at eiffel.com*, 4(2):4–8, December 1992.

[Mic98] Isabel Michiels. Using Logic Meta-Programming for Building Sophisticated Development Tools. Master's thesis, Vrije Universiteit Brussel, 1998.

[Min93] Naftaly H. Minsky. Regularities in Software Systems. In *ICSE Workshop on Studies of Software Design*, pages 49–63, 1993.

[Min96a] Naftaly H. Minsky. Independent On-line Monitoring of Evolving Systems. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 134–143, Washington, DC, USA, 1996. IEEE Computer Society.

[Min96b] Naftaly H. Minsky. Law-Governed Regularities in Object Systems. Part 1: An Abstract Model. *Theory and Practice of Object Systems (TAPOS)*, 2(4):283–301, 1996.

[Min00] Naftaly H. Minsky. Towards Architectural Invariants of Evolving Systems. *SIGSOFT Softw. Eng. Notes*, 25(1):65, 2000.

[Min01] Naftaly H. Minsky. Establishing Accounting Principles as Invariants of Financial Systems. In *Proceedings of the Fourth International IFIP TC-11 WG 11.5 Conference on Integrity and Internal Control in Information Systems*, 2001.

[MK06] Kim Mens and Andy Kellens. IntensiVE, a Toolsuite for Documenting and Checking Structural Source-Code Regularities. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 239–248, Washington, DC, USA, 2006. IEEE Computer Society.

[MMW02] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting Software Development Through Declaratively Codified Programming Patterns. *Elsevier Journal on Expert Systems with Applications*, pages 405–431, November 2002.

[MRB$^+$06] Isabel Michiels, Coen De Roover, Johan Brichau, Elisa Gonzalez Boix, and Theo D'Hondt. Program Testing Using High-Level Property-Driven Models. In *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE 2006)*, pages 489–494. Knowledge Systems Institute, 2006.

[MS03] Grant Martin and Sandeep Shukla. Hierarchical and Incremental Verification for System Level Design: Challenges and Accomplishments. In *MEMOCODE '03: Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*

(MEMOCODE'03), page 97, Washington, DC, USA, 2003. IEEE Computer Society.

[OM94] Mehmet A. Orgun and Wanli Ma. An Overview of Temporal and Modal Logic Programming. In *Proc. of the 1st Intl. Conf. on Temporal Logic (ICTL94)*, pages 445–479, 1994.

[Org94] Mehmet A. Orgun. Temporal and Modal Logic Programming: an Annotated Bibliography. *SIGART Bull.*, 5(3):52–59, 1994.

[Pal04] Girish Keshav Palshikar. An Introduction to Model Checking. *Embedded Systems Programming*, February 2004.

[PN81] Bernhard Plattner and Jürg Nievergelt. Monitoring Program Execution: A Survey. *IEEE Computer*, 14(11):76–93, 1981.

[Raj06] Sriram K. Rajamani. Automatic Property Checking for Software: Past, Present and Future. *sefm*, 0:18–20, 2006.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-Based Test Oracles for Reactive Systems. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 105–118, New York, NY, USA, 1992. ACM Press.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM99)*, pages 13–22, 1999.

[RD02] Tamar Richner and Stéphane Ducasse. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, number IAM-01-007, page 34. IEEE Computer Society, 2002.

[Ric02] Tamar Richner. *Recovering Behavioral Design Views: a Query Based Approach.* PhD thesis, Universität Bern, Philosophisch-naturwissenschaftlichen Fakultät, Bern, Swiss, 2002.

[RMG$^+$06] Coen De Roover, Isabel Michiels, Kim Gybels, Kris Gybels, and Theo D'Hondt. An Approach to High-Level Behavioral Program Documentation Allowing Lightweight Verification. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006)*, pages 202–211, 2006.

[Roo04] Coen De Roover. Incorporating Dynamic Analysis and Approximate Reasoning in Declarative Meta-Programming to Support Software Reengineering. Master's thesis, Vrije Universiteit Brussel, 2004.

[Ros95] David S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[SB05] Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. In *Proc. of the 5th International Workshop on Run-time Verification (RV05)*, 2005.

[Sch] Michael I. Schwartzbach. *Lecture Notes on Static Analysis*. BRICS, Department of Computer Science, University of Aarhus, Denmark.

[sou] The Smalltalk Open Unification Language (SOUL) - http://prog.vub.ac.be/soul/index.html.

[Spi89] J. Michael Spivey. *The Z notation: a reference manual - Chapters 1 and 5 (Introduction and Sequential Systems resp.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Spl] Splint - http://www.splint.org - Secure Programming Lint - Annotation-assisted Lightweight Static Checking.

[TJ98] Kevin S. Templer and Clinton L. Jeffery. A Configurable Automatic Instrumentation Tool for ANSI C. In *Proc. of the 13th IEEE Intl. Conf. on Automated Software Engineering (ASE98)*, page 249, 1998.

[vL00] Axel van Lamsweerde. Formal Specification: A Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA, 2000. ACM Press.

[Vol98] Kris De Volder. *Type-Oriented Logic Meta Programming - Chapter 5 (Logic Meta Programming)*. PhD thesis, Vrije Universiteit Brussel, 1998.

[Wie07] Jan Wielemaker. *SWI-Prolog 5.6 Reference Manuel*, January 2007.

[WM06] Roel Wuyts and Kim Mens. Codifying Structural Regularities of Object-Oriented Programs. Technical report, Département d'Ingénierie Informatique, Université Catholique de Louvain, Belgium, 2006.

[WMFB$^+$98] Robert J. Walker, Gail C. Murphy, Bjorn N. Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Conference on Object-Oriented*, pages 271–283, 1998.

[Wuy98] Roel Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

[Wuy01] Roel Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.

[YB94] Hwei Yin and James M. Bieman. Improving Software Testability with Assertion Insertion. In *International Test Conference*, pages 831–839, 1994.

[YE04] Jinlin Yang and David Evans. Automatically Inferring Temporal Properties for Program Evolution. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, New York, NY, USA, 2004. ACM Press.

Index

BEHAVE, 79, 90
    architecture, 93
    aspect layer, 101
    generation layer, 98
    instrumentation module, 98
    reification module, 95

abstract data types, 24
advice, 53
analysis of code, 39
analysis time, 49
aspect, 101
Aspect-Oriented Programming (AOP), 53
aspectJ, 53
assertion checking, 51
assertions, 17
automatic memory management, 120

behavioural specifications, 24
branching time logic, 26

characteristics of logic programming, 84
class invariants, 18
closures, 120
code generation, 98, 195
communication protocols, 23
critical section, 26

DAIKON, 21
DBC, 19
debuggers, 50
Design By Contract (DBC), 19
design invariants, 28, 29, 34
DIDUCE, 21
documentation, 15
dynamic analysis, 45, 90
dynamic invariants, 21
dynamic program analysis, 36

embedded assertions, 17
error detection tools, 51
execution model, 120

finite state machine, 38
formal specification, 27
formal verification, 36

garbage collection, 133
generic assertions, 23

invariants, 11, 12
    detection of, 21
    uses of, 15

join point, 53

late binding, 50
Law-Governed Architecture (LGA), 22
legacy systems, 50
linear time logic (LTL), 26
Logic Meta Programming (LMP), 81
logic programming, 81

meta-circular, 120
model checking, 37
  difficulties with, 39

object Invariants
  problems with, 18, 20
OCL, 27
oracles, 52

Pico, 120, 187
pointcut, 53
polymorphism, 50
post-conditions, 19
pre-conditions, 19
program analysis, 35
program monitors, 49
Prolog, 93
  terminology, 82

quicksort algorithm, 146

regularities in software, 22
reification, 95
representational mapping, 87, 167

Smalltalk Open Unification Language (SOUL), 86
software behaviour, 13
software complexity, 12
static analysis, 41
static program analysis, 35, 36

tail call optimisation, 140
tail recursion, 140
tail-recursive, 120
temporal logic, 25
temporal logic meta interpreter, 106, 181
temporal logic operators, 182
temporal logic programming, 86
temporal operators, 26
temporal quantifiers, 26
test oracles, 52
theorem proving, 37

XUnit framework, 28

Z notation, 24, 25