AARHUS
UNIVERSITY
SCIENCE AND TECHNOLOGY

On the Extensibility of Formal Methods Tools

Department of Engineering

Luís Diogo Couto

PhD Dissertation

On the Extensibility of Formal Methods Tools

PhD Dissertation

Luís Diogo Couto

September 30, 2015

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

On the Extensibility of Formal Methods Tools

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Luís Diogo Couto
January 8, 2016

EBOOK ISBN: 978-87-7507-360-3
DOI: 10.7146/aul.118.106

Abstract

Modern software systems often have long lifespans over which they must continually evolve to meet new, and sometimes unforeseen, requirements. One way to effectively deal with this is by developing the system as a series of extensions. As requirements change, the system evolves through the addition of new extensions and, potentially, the removal of existing extensions. In order for this kind of development process to thrive, it is necessary that the system have a high level of extensibility. Extensibility is the capability of a system to support the gradual addition of new, unplanned functionalities. This dissertation investigates extensibility of software systems and focuses on a particular class of software: formal methods tools. The approach is broad in scope. Extensibility of systems is addressed in terms of design, analysis and improvement, which are carried out in terms of source code and software architecture. For additional perspective, extensibility is also considered in the context of formal modelling. The work carried out in this dissertation led to the development of various extensions to the Overture tool supporting the Vienna Development Method, including a new proof obligation generator and integration with theorem provers. Additionally, the extensibility of Overture itself was also improved and it now better supports the development and integration of various kinds of extensions. Finally, extensibility techniques have been applied to formal modelling, leading to an extensible architectural style for formal models.

Resumé

Moderne softwaresystemer har ofte en lang levetid, hvor de løbende bliver udviklet for at imødekomme nye, og sommetider uforudsete, krav. En effektiv måde at håndtere dette på er ved at udvikle systemet som en række udvidelser. I takt med at krav ændrer sig, udvikler systemet sig ved tilføjelse af nye udvidelser og potentielt set, fjernelse af eksisterende funktionalitet. For at få succes med denne form for udviklingsproces er det nødvendigt at systemet i høj grad er forberedt på udvidelser. Et systems udvidbarhed beskriver dets evne til at understøtte gradvis tilføjelse af ny, uplanlagt funktionalitet. Denne afhandling undersøger software systemers udvidbarhed og fokuserer på en særlig klasse software: værktøjer til formelle metoder. Fremgangsmåden er omfangsrig med mange forskellige vinkler. Systemers udvidbarhed undersøges i forhold til design, analyse og forbedring og gennemføres i form af kilde-kode og software arkitektur. For at få et yderligere perspektiv betragtes udvidbarhed også i en formel modelleringskontekst. Arbejdet udført i denne afhandling har ført til udviklingen af forskellige udvidelser til software-værktøjet, Overture, som understøtter udviklingsmetoden Vienna Development Method, inklusiv en ny generator af bevisforpligtigelser samt integration af værktø jer til at bevise egenskaber. Yderligere er Overtures udvidbarhed blevet forbedret, så værktøjet nu bedre understøtter udvikling og integration af forskellige former for udvidelser. Endelig er teknikker til udvidelighed blevet anvendt til formel modellering, hvilket har ført til en arkitektonisk stil for formelle modeller, der er nemmere at udvide.

Acknowledgements

The work reported in this thesis would not have been possible without the collaboration and support of several people. My thanks to all of them but I would like to single out the following.

I thank my supervisor, Peter Gorm Larsen, for the opportunity to pursue this PhD. His feedback and suggestions were always valuable. I also thank my co-supervisor, Joey W. Coleman, for his collaboration and feedback.

I thank Nick Battle, Stefan Hallerstede, and Victor Bandur for reviewing this thesis.

Thanks to the members of the Software Engineering Group, especially my two officemates Peter W. V. Tran-Jørgensen and Kenneth Lausdahl, with whom I had many fruitful discussions and collaborations.

Thanks to Richard Payne and Simon Foster for their collaboration on the COMPASS project.

On a personal note, I would like to thank Augusto Ribeiro who was a good friend when I first moved to Denmark and continues to be so.

Finally, I would like to thank my parents and sister for their love and support.

This work was partially supported by the EU Framework 7 Integrated Project “Comprehensive Modelling for Advanced Systems of Systems”.

Contents

Abstract i

Resumé iii

Acknowledgements v

Contents ix

Acronyms xi

I Summary 1

1 Introduction 3
- 1.1 Context . . . 3
- 1.2 Motivation . . . 5
- 1.3 Research Objectives . . . 6
- 1.4 Research Method . . . 7
- 1.5 Evaluation Criteria . . . 8
- 1.6 Publications . . . 9
- 1.7 Outline and reading guide . . . 10

2 Background 13
- 2.1 Extensibility and related terms . . . 13
- 2.2 Other approaches to extensibility . . . 14
- 2.3 Software Architecture . . . 16
- 2.4 Design patterns and data structures . . . 17
- 2.5 Overture . . . 18
- 2.6 Semantics . . . 19
- 2.7 Other tools and case studies . . . 20

3 Specific Applications of Extensibility **23**
3.1 New Extensions 24
3.2 Combining Extensions 27
3.3 Improving Extensibility through Software Architecture . . . 29
3.4 Extensibility in Formal Models 37

4 Generalisations of Extensibility **41**
4.1 Generalising the new POG contributions 42
4.2 Technical aspects of extension combination 42
4.3 Reusability principles for formal analyses 44
4.4 Extensible architecture for formal language tools 46
4.5 Strategy pattern-based Modelling 49

5 Foundational Contributions **53**
5.1 Extending Proof Rules of VDM 53
5.2 Integrating Proof Automation 57

6 Conclusion **61**
6.1 Summary of Contributions 61
6.2 Assessing Contributions 63
6.3 Future Work 65

II Publications **69**

7 The COMPASS Proof Obligation Generator: A Test Case of Overture Extensibility **71**

8 Towards Verification of Constituent Systems through Automated Proof **79**

9 Migrating to an Extensible Architecture for Abstract Syntax Trees **87**

10 Extending the Overture code generator towards Isabelle syntax **89**

11 Towards Enabling Overture as a Platform for Formal Notation IDEs **103**

12 Principles for Reuse in Formal Language Tools **119**

13 LPF-Aware Proof Obligation Generation in VDM/Overture 121

14 Combining Harvesting Operation Optimisations using Strategy-based Simulation 137

Bibliography 147

Acronyms

ACSL ANSI/ISO C Specification Langage

ASM Abstract State Machine

AST Abstract Syntax Tree

CGP Code Generation Platform

CML COMPASS Modelling Language

CSP Communicating Sequential Processes

FM Formal Method

IDE Integrated Development Environment

IR Intermediate Representation

LPF Logic of Partial Functions

PO Proof Obligation

POG Proof Obligation Generator

RCP Rich Client Platform

SA Software Architecture

SMT Satisfiability Modulo Theories

TLAPS TLA$^+$ Proof System

TPP Theorem-Proving Plug-in

UI User Interface

UTP Unifying Theories of Programming

VDM Vienna Development Method

Part I

Summary

1

Introduction

> "A hallmark — if not the hallmark — of good object oriented design is that you can modify and extend a system by adding code rather than hacking it. In short, change is additive, not invasive. Additive change is potentially easier, more localized, less error-prone, and ultimately more maintainable than invasive change."
>
> *–John Vlissides, The C++ Report, February 1998*

1.1 Context

Modern software systems stay in use for a very long period of time and are continuously evolved to deal with new, and possibly unforeseen, requirements. Developing systems based on extensions is a way to deal with this. Extensions may be added or removed from the system over time to deal with evolving needs over the lifetime of the system. However, in order for extension-based development to be successful, the system must be designed and developed to support it. In other words, it must have a high level of extensibility.

This dissertation examines extensibility of software systems with a focus on a particular class of software: Formal Methods (FMs) tools. The desire for extension-based development is easy to understand in an FM setting, both for experimental research purposes and to better support the evolution of a method and its supporting tools. Extensibility can also promote heavy reuse of the existing system which increases developer efficiency. Additionally, extensions can be a good way to achieve interaction between multiple FM tools by using dedicated extensions to drive the interaction and allow the tools themselves to evolve independently.

However, the construction of extensions to FM tools can pose particular challenges. Consider Figure 1.1 which represents an arbitrary FM tool, *FM-*

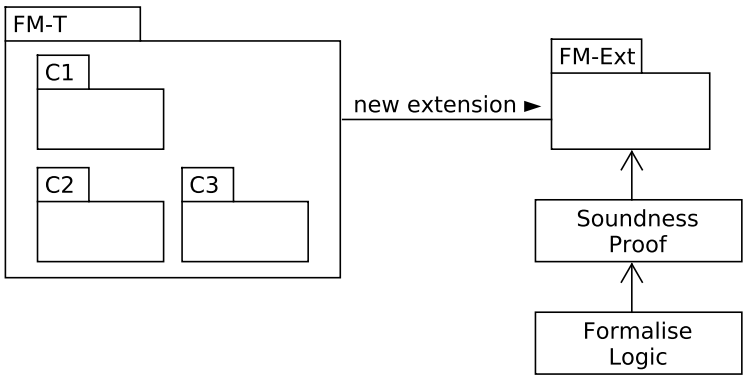


Figure 1.1: Extending an FM tool.

T, and a proposed extension to it, *FM-Ext*. In a non-formal setting, one may simply employ a plugin-based architecture (or similar) for the system being extended and directly implement the extension. This may present complications on its own — for example, the system may not have a plugin-based architecture or otherwise have low extensibility — but in formal settings, to ensure the soundness of the functionality being provided by the new extension, we must provide a soundness proof which may even necessitate formalising the underlying logic. Furthermore, we must ensure that the new extension is consistent with the existing ones and is implemented efficiently due to the extra time and effort already spent ensuring soundness.

In order to achieve high-extensibility FM tools which support consistent and efficient implementation of extensions, we must understand both the concept of extensibility itself and how to develop extensible systems. Additionally, and in keeping with the theme of reuse, we focus on existing systems and how to analyse and, most crucially, improve their extensibility.

We examine software extensibility from two points of view: source code and Software Architecture (SA). Source code relates to specific, technical challenges to particular extensions of a system. SA is about the larger picture of how software is organized and interconnected and affects most qualities of a software system, including extensibility. The two points of view are connected but it is worth distinguishing between the micro-level (source code) and the macro-level (SA) issues.

This work is carried out within a research group that focuses on tooling for model-based FMs. The extensibility research utilises the tools developed at the group as case studies and, as part of the work, the extensibility of those tools is improved. However, we also consider extensibility in the context of formal modelling — a good way to gain insights into a topic by abstracting away unnecessary information through the construction of models. Finally, as formalists, we are concerned with the foundations of our work and while that is not a focus, foundational work is performed as necessary.

1.2 Motivation

Extensibility and reuse are frequently desired in software development [17]. When it is possible to reuse and extend existing software artefacts (e.g., source code), then new software systems can be developed faster and more efficiently [38].

Extensibility can increase reusability of a system by promoting the reuse of the base functionality of the system across its various extensions. This can lead to a reduction in code duplication as the base functionality is properly reused rather than copied [63]. Code duplication negatively affects software maintainability [45] — bugs have to be fixed twice and if one is not aware of the duplicates, a given bug will not be completely fixed. Therefore, reducing code duplicates can make software maintenance easier.

Extensibility can also be beneficial to the software development process, if a system is organised into a core plus extensions. The core contains the crucial, heavily used components for which maintenance efforts can be prioritised. New features can be added incrementally and independently as extensions [15]. Furthermore, if such features do not prove worthwhile, they can be easily removed them from the system. The same holds for experimental research extensions. Another advantage is that multi-developer teams can divide extensions among their members, allowing each of them to work independently. This is particularly useful for geographically diverse teams or teams whose members have significantly different technical backgrounds and expertise and are thus well suited for developing a particular kind of feature.

While extensibility has several advantages, it can also have disadvantages. Namely, extensible code may be more complex than its non-extensible equivalent due to an additional cost involved in making the code extensible. This increase in complexity may raise issues of code maintenance or performance. Therefore, we must fully understand extensibility in order to make well-

informed decisions about which kinds of trade-offs to make when developing a system.

The focus on extensibility allows us to improve the reusability of the developed tools and that is a clear benefit. But we also wish to contribute to actual new functionality of those tools in order to address research challenges in other areas. By developing these new functionalities as extensions, not only is the new feature provided but the extensibility of the system is exercised and possibly improved. This is an approach with a high level of synergy as it not only addresses technical challenges in diverse areas but it enhances our understanding of extensibility in general and of the tools being extended.

1.3 Research Objectives

This section presents the main research objectives. The general objective for is to gain insight into the phenomenon of extensibility, what influences it and what are its advantages and disadvantages. We also aim to understand extensibility from various perspectives. Therefore, the following objectives have been determined:

Objective 1 — Extensibility analysis of software systems. The 1st objective relates to extensibility of existing software systems. It consists of analysing existing software systems in order to assess their extensibility by identifying specific extensibility issues and their causes.

Objective 2 — Extensibility improvements to software systems. The 2nd objective is also applied to existing systems and complements the one above by improving extensibility of a software system by providing solutions or recommendations of how to fix specific extensibility issues.

Objective 3 — Well founded extensions for FM tools. The third objective is to develop contributions to existing FM tools by implementing extensions for them. This objective allows us to contribute to challenges in a variety of scientific areas in a way that advances core research on extensibility. As users of FM, we are interested in these extensions having solid theoretical foundations. Where such foundations do not exist, we develop them.

Objective 4 — Extensibility in formal models. Finally, on the topic of FM, there is a smaller 4th objective of combining extensibility with formal modelling. The idea is not to model extensibility itself and formally

analyse it. Rather, we take extensibility principles from software development and apply them to the construction of formal models to show that standard software engineering extensibility practices can be applied to the development of formal models.

1.4 Research Method

This thesis is in support of an engineering PhD and the work is carried out in a research environment that is driven significantly by tool development. Therefore, the research to be carried out is practical and usually applied to addressing concrete and specific issues. The research method reflects this.

The research method works in breadth and consists of several distinct but loosely connected studies regarding extensibility in order to draw and present results from the whole of them. Some studies go in depth, the work in each building on previous work. Other studies are smaller and their work stands alone. This approach allows us to be involved in a variety of research areas and see how extensibility can be applied to them.

The research group in which this work has been conducted has a track record of using FM. This plays an important part in the research work, although the focus lies on the use of existing FM rather than the development of new ones.

Much of this work is driven by the creation of new extensions that allow us to exercise and address most of the research objectives. These extensions are based on formal theoretical foundations. However, we do not necessarily develop those theories but instead rely on existing foundations, reusing as much as possible. We contribute to foundations selectively when it is necessary to support the main research work. The overall methodology for development of extensions is as follows:

1. *Identify* and *design* a new extension. This supports tooling to address a specific challenge in one of the research areas the group is involved in.

2. *Develop theoretic foundations*, where necessary. If an extension provides a feature that is not supported by an underlying theory we begin by working out the theory before developing the extension.

3. *Develop a new extension*, focusing on reuse of the base system as much as possible. The reuse focus is crucial as it allows us to narrow in on extensibility. Indeed, it can be more important to ensure that reuse is heavily carried out, than simply providing the functionality.

4. Throughout the development process, whenever a technical issue is encountered, it is *documented* and *analysed*, particularly to uncover any influence it may have on extensibility. This ensures that we carry out an analysis of the base system as part of the development of the extension.

5. If possible, *propose* and *implement* a solution for the extensibility issue. This step is carried out iteratively with the previous two, leading to cycles of develop→identify→fix. While any kind of issue may be addressed, focus is on issues in the base system that directly prevent or hamper development of the extension.

6. *Analyse* all documented issues and their solutions. Once the extension has been developed, we focus on identifying which issues were particularly relevant from an extensibility perspective, why were they relevant, and what were their causes. Additionally, these issues and their solutions may be specific instances of a more general problem. If so, an effort is made to generalise the problem and its solution.

7. Once work is completed, it is important to *report* what has been achieved, focusing particularly on extensibility issues encountered and their solutions. This ensures that we *reflect* on the work and properly organize it. The reflection, in particular may lead to new avenues of research. This also allows us to crystallize the work by formulating it in the form of concrete contributions.

1.5 Evaluation Criteria

The output of this work is synthesised as a set of contributions. In order to assess these contributions, the following evaluation criteria have been chosen. The first two criteria both support research objectives 1 and 2 but they do so in different ways, thus ensuring our overall goal of having multiple perspectives on extensibility. The last criterion (and the objective it supports) also contributes towards having multiple perspectives on extensibility.

Source Code improving and/or analysing extensibility from the perspective of the source code and implementation of the software system. This criterion relates to research objectives 1 and 2.

Software Architecture addressing — improvement, analysis or both – extensibility through a SA perspective. This criterion relates to research objectives 1 and 2.

New Extension providing or directly enabling a new feature for existing FM tools. This criterion relates to research objective 3.

Foundation theoretical foundation work supporting other, practical contributions. This criterion relates to research objective 3.

Modelling apply extensibility to formal modelling contexts. This criterion relates to research objective 4.

An overview and an assessment of all contributions are carried out in Sections 6.1 and 6.2, respectively.

1.6 Publications

This section lists work selected for publication throughout the PhD.

1.6.1 Published

The following have been published and are included in this thesis in part II.

[P24] Luís Diogo Couto and Richard Payne. *The COMPASS Proof Obligation Generator: A Test Case of Overture Extensibility*. 11th Overture Workshop, August 2013.

[P22] Luís Diogo Couto, Simon Foster and Richard Payne. *Towards Verification of Constituent Systems through Automated Proof*. Workshop on Engineering Dependable Systems of Systems (EDSoS), May 2014.

[P26] Luís Diogo Couto, Peter W. V. Tran-Jørgensen, Joey W. Coleman and Kenneth Lausdahl. *Migrating to an Extensible Architecture for Abstract Syntax Trees*. 12th Working IEEE / IFIP Conference on Software Architecture, May 2015.

[P25] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. *Extending the Overture code generator towards Isabelle syntax*. 13th Overture Workshop, June 2015.

[P23] Luís Diogo Couto, Peter Gorm Larsen, Miran Hasanagić, Georgios Kanakis, Kenneth Lausdahl and Peter W. V. Tran-Jørgensen. *Towards Enabling Overture as a Platform for Formal Notation IDEs*. 2nd Workshop on Formal Integrated Development Environment (F-IDE), June 2015.

1.6.2 Submitted

The following has been submitted for publication and is included in this thesis in part II.

[P28] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Kenneth Lausdahl. *Principles for Reuse in Formal Language Tools*. 31st ACM Symposium on Applied Computing (SAC 2016), April 2016.

1.6.3 Planned

The following are planned for submission and are included in this thesis in part II.

[P21] Luís Diogo Couto, Nick Battle and Peter Gorm Larsen. *LPF-Aware Proof Obligation Generation in VDM/Overture*. To be submitted to the 5th International ABZ Conference (ABZ 2016), May 2016.

[P27] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Gareth Edwards. *Combining Harvesting Operation Optimisations using Strategy-based Simulation*. To be submitted to the International Journal of Computers and Electronics in Agriculture, 2016.

1.6.4 Not included

The following has not been included in this thesis but is available from its publisher.

[20] Luís Diogo Couto. *On Extensibility of Software Systems*. Department of Engineering – Electrical and Computer Engineering, Aarhus University, April 2014.

1.6.5 Other

In addition to the above, the author of this thesis is also co-author of [49, 57, 64, 48] and sole author of [19], but these publications do not form part of the thesis.

1.7 Outline and reading guide

This thesis is organised in two parts: part I gives a summary of the work carried out in the PhD mainly based on excerpts from a selection of publi-

cations carried out during the PhD. Part II contains a selection of complete publications that formed the basis of the PhD.

Part I is meant to give an overview of the contributions of the PhD. Contributions are numbered — e.g. Contribution 1 — and framed so as to stand out in the text. Part I also presents background material and an evaluation of the work. As it is mainly based on publications, in order to distinguish the publications of the PhD from other references, they are prefaced with "P" in their citation identifiers – e.g. [P24].[1]

Part I is structured as follows: after this introduction chapter, Chapter 2 presents relevant background material for the rest of the thesis. Then, the main applied contributions of interest to specific communities are presented in Chapter 3. Some of these contributions can be generalised and be made of interest to broader audiences. This is done in Chapter 4, together with discussions of other contributions of more general interest. Chapter 5 presents the theoretical foundation work that was necessary to support some of the other contributions. Finally, Chapter 6 concludes part I by assessing the contributions and discussing future avenues of research.

Part II contains a selection of papers that have been written by the author of the PhD thesis in collaboration with others. Each chapter presents a single publication and is prefaced by the bibliographic entry of the publication used in the rest of the thesis and a brief description of where it was published. Following that, the publication is presented in its original form.

[1] There is one exception: [20], which constitutes the report for the qualifying exam of the PhD.

2

Background

This chapter presents related work and background information used in the thesis. Section 2.1 introduces and defines extensibility and Section 2.2 introduces other approaches to the extensibility of formal methods tools. The remaining sections present background material underlying the work in this thesis. Most of the topics covered here are independent of each other, facilitating the use of this chapter as a reference.

2.1 Extensibility and related terms

The idea of extending software has existed for several decades — some of the foundations towards it can be seen in the late 1960s [30] and early 1970s [44]. At the 1968 and 1969 NATO Conferences on Software Engineering there is discussion on both extensibility as a design concept [70] and on how to develop extensible systems [10]. Towards the end of the 1970s, extensibility begins to become a concern in its own right, and the idea of explicitly designing for extensibility [75] emerges.

Broadly speaking, extensibility is the ability to contribute new, unplanned functionalities or features to a software system without changing the existing ones. These new features can be achieved through customisation, redefinition, etc. In this thesis, the following definition of extensibility is used:

> Software is extensible if it can be adapted to new, possibly unforeseen requirements by addition of new source code and without modifying the existing sources.

There are two software properties that are closely related with extensibility and must be discussed: modifiability and reusability. Modifiability measures the extent to which it is possible to modify a system effectively and efficiently [50]. Reusability measures the ability to use an existing software component when developing a different software system [60, 68].

Modifiability can be viewed as a counterpart of extensibility. Both properties seek to address a system's ability to cope with changing needs or requirements. However, extensibility seeks to do this through augmentation and without altering the existing source code of the system. To put it another way, modifiability means it is easy to change the existing implementation, whereas extensibility means that it is unnecessary to change the existing implementation.

In this thesis, extensibility is broadly grouped in two kinds: functional extensibility and data extensibility. *Functional extensibility* refers to extending a system in order to provide new functionalities. A fairly common example from the real world are web browsers and their plugins. On the other hand, *data extensibility* is the ability to extend a system in order to cope with additional types of data. The existing functionalities of the system (or a subset of them) thus become available for the new data types, which may be input, output or both.

This thesis considers extensibility according to two perspectives: source code and SA. Source code, of course, refers to the implementation of the software system. SA refers to how a software system is structured, organised and intra-connected [6]. These two perspectives are connected as extensible architectures are realized through source code. Nonetheless, we distinguish the extensibility-enabling aspects of both. In both cases, the presence of certain characteristics and design decisions can help achieve extensibility.

2.2 Other approaches to extensibility

In this section we briefly describe approaches to extensibility taken by other language-specific FM tools and briefly compare them to the approach taken by Overture (see Section 2.5 for more about Overture). Due to our focus on model-based FM, we focus predominantly on those kinds of tools, although we also describe one instance of generic theorem proving and program language verification.

The Rodin [5] tool supports the Event-B method [4] for the development and analysis of system models. Rodin is based on the Eclipse platform [41] and is extensible by means of the plugin-based architecture of Eclipse. Furthermore, in Rodin, models are stored in a repository. Rather than storing a model as an Abstract Syntax Tree (AST), Rodin stores and manipulates the model in a XML-based database. The model repository was designed with ease of extensibility in mind. In particular, the repository-based approach means there is no need to change the syntax of the language in order to extend

the Event-B notation. By contrast, in Overture, models are stored as ASTs and notation extensibility is based on extending the AST itself.

The TLA Toolbox is an Integrated Development Environment (IDE) for the TLA$^+$ tools. TLA$^+$ is a specification language for modelling and verifying concurrent systems [61]. The TLA$^+$ tools do not appear to have a particular focus towards extensibility although some extensions have been carried out. For example, the TLA$^+$ Proof System (TLAPS) [18] has been extended by connecting it with Satisfiability Modulo Theories (SMT) solvers via translation of the TLA$^+$ specification syntax into the input language of the SMT solvers [66, 67]. In Overture, the implementation of such translations is facilitated by using a dedicated Code Generation Platform (CGP) (see Section 2.5).

CoreASM [33] is a tool environment and language for the development and execution of Abstract State Machines (ASMs) [9]. ASM is a multi-purpose notation suitable for modelling algorithms, protocols, systems, etc. The CoreASM engine executes CoreASM specifications and has an extensible plugin-based architecture [34]. The plugins progressively extend the language from the core constructs defined in the interpreter. Example plugins include a debugging component [32] and support for aspect-oriented specification [31]. The plugin architecture is also used to define the ASM language itself: a kernel provides only the bare essentials for executing the most basic ASMs and everything else is contributed via plug-ins. By contrast, in Overture, the complete Vienna Development Method (VDM) languages are part of the core of the tool.

In the Isabelle theorem prover [72], extensibility is dealt with through conservative extensions: a logical theory is a conservative extension of another theory if every formula common to both theories that is provable in the extension is also provable in the base theory. Isabelle contains native language mechanisms for constructing new theories as conservative extensions of existing ones. Additionally, the Sledgehammer tool [76] allows for connecting other theorem provers and external tools through translation and reconstruction of received proofs. In Overture, new constructs can be added to the language independently. Additionally, it is possible to redefine (or remove) existing language constructs. On the other hand, Isabelle can prove the soundness of its extensions whereas this must be done separately in Overture.

Frama-C is a platform for the verification of C programs [29]. The platform consists of a kernel providing core services and common data structures shared by all platform plugins. Additionally, plugins support and interact with each other through the ANSI/ISO C Specification Langage (ACSL) [7].

Extensions to Frama-C consist of new plugins built on top of the platform, reusing the kernel and data structures and supporting ACSL through generation or validation of annotations. Overture has a similar architecture with core functionality and data structure shared by all plug-ins. However, in Overture, plugins interact by operating directly on the shared data structure rather than through annotations.

2.3 Software Architecture

SA can be defined as:

> "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." [6]

There are various ways to visualise the SA of a system [59]. As an example, the functionalities of the Overture tool are summarised in Figure 2.1. Overture is further described in Section 2.5.

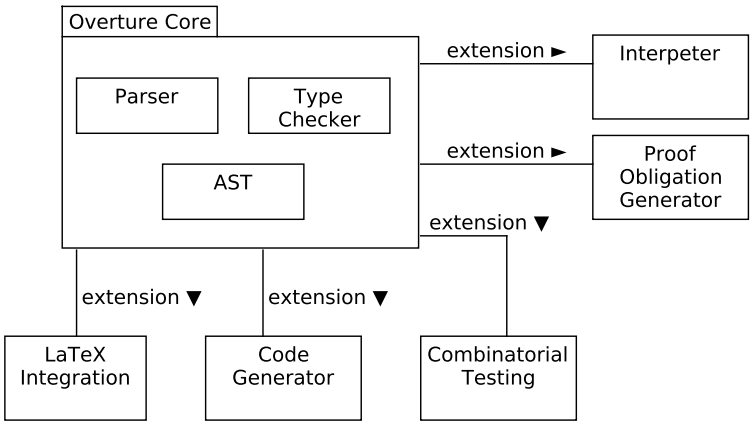


Figure 2.1: Logical view of the Overture architecture.

SA is connected to the notion of software quality and quality attributes. In particular SA greatly influences most of a system's quality attributes [14] either by preventing or enabling the system from realising said attributes.

One of the quality attributes influenced by SA is extensibility. Specifically, SA influences the extent to which a system is extensible in both its ability to support new extensions and the ease with which that can be done. Conversely, the desire for high extensibility influences decisions made about the architecture of a system.

In this thesis, we are more interested in how SA influences extensibility. Specifically, we are interested in how an SA limits the extensibility of a system, how such limitations can be addressed and in the design of SAs that enable extensibility.

2.4 Design patterns and data structures

Design patterns can be an effective means of achieving extensible software. A design pattern is a reusable solution to a known recurring problem [40]. The work of this thesis makes significant use of two design patterns, summarised below:

- The *strategy pattern* allows the behaviour of an algorithm to be selected at runtime. It defines a family of interchangeable algorithms and encapsulates them inside a general strategy. New algorithms can be contributed to the family at any time, provided they follow the definition of the strategy.

- The *visitor pattern* allows for detaching an algorithm from the data structure it is executed on. It also enables adding additional functionality to a class without changing the implementation of said class.

Data structures also play an important role in the extensibility work reported in this thesis. In terms of extensibility, data structures must both enable functional extensions and be amenable to data extensibility. When speaking of data extensibility, we introduce the notion of a hybrid data structure. The base data structure refers to the data structure with only base elements and the extended data structure refers to the data structure that contains the elements introduced as part of the extension. A hybrid data structure is a data structure where some of its elements are drawn from the base structure and some are drawn from the extended structure.

2.5 Overture

Overture is the primary software system upon which the extensibility research reported in this thesis was carried out. As such, an introduction to the tool, its goals and uses may be helpful to better understand why particular design decisions were made while carrying out extensibility work.

Overture is an open source tool for the development and analysis of VDM models [56, 35] and is freely available online.[1] A VDM model consists of a series of definitions including data and behaviour.

VDM models are defined via textual sources so the initial processing of a model is similar to that of a programming language — parsing and type checking. But the kinds of analyses done with VDM tend to be broader in variety. This is a key reason for desiring extensibility in Overture. New kinds of analyses can be contributed to the tool as extensions.

In Overture, most analyses are implemented over the Overture AST. An AST is a data structure that represents the abstract syntax of the VDM model in tree form, hence its name. From an extensibility perspective, this can be generalised further. An AST is seen as a tree where the nodes have various fields for storing information. Nodes are not generic but can belong to one of several families. In the Overture AST, nodes are sometimes called `INodes` after the name of the Java interface that is implemented by all nodes in the tree and defines a node at the most abstract level.

The Proof Obligation (PO) generation analysis was chosen as an object of particular study for this thesis as the variety of kinds of obligation is expected to provide a fertile ground for extensibility experimentation. In VDM, type checking is statically undecidable. This issue dealt with by generating various POs — logical predicates that, if discharged, ensure the type-correctness of a VDM model. In Overture, PO generation is the responsibility of the Proof Obligation Generator (POG). Like most other analyses in Overture, the POG is implemented as visitors over the AST.

Another relevant component of Overture is the CGP, used for the development of various code generators and model translations. From the VDM AST, the CGP constructs an Intermediate Representation (IR), which forms a tree structure that is independent of any particular target language.

Initially, each node in the IR has a one-to-one correspondence to a node in the VDM AST. Subsequently, the IR is subjected to a series of *transformations* in order to change the tree structure into a new form that is easier for a particular code generator to produce code from.

[1] `http://overturetool.org/`

After the IR has been fully transformed, it is handed over to a language-specific backend generator in order to finalise the code generation process. The CGP provides a framework for syntax generation that serves to facilitate production of code in the target language. This framework is based on the Apache Velocity template engine and is used for mapping each node in the IR into concrete syntax [1].

2.6 Semantics

The foundation work carried out in this thesis often relies on the semantics of VDM. A complete treatment of the semantics is unnecessary for this thesis (refer to [56] and [8]) but a brief discussion of undefinedness in VDM is in order.

Undefined values have many sources such as partial operators like division or, more generally, partial functions. The field has a long history [62] and there are various approaches to dealing with undefinedness such as Owe's "weak logic" [74], Partial Function Logic [77], Logic of Computable Functions [69, 42] or underspecification [43]. In VDM, undefinedness is dealt with using the three-valued Logic of Partial Functions (LPF) [11, 53] that is based on Kleene semantics [58] (see [12] for a comparison of approaches to partial functions).

An important aspect of the Kleene semantic model is the commutativity laws for logical operators. This is in contrast to McCarthy logic [65] which also deals with undefinedness but does not have commutativity laws — expressions are evaluated left-to-right. For an example, consider Table 2.1 where the truth table for disjunction for both McCarthy and Kleene logics is given, and the row highlighted in grey shows how the two logics differ. Note that Kleene produces a defined value in a situation where McCarthy does not.

Undefinedness is a source of runtime errors during model interpretation. Hence, one of the tasks of the POG is to generate POs that guard against it. For all sources of undefinedness, it is possible to write a definedness predicate that ensures safe use of the operator. For example, if x is a divisor then such a definedness predicate is $x \neq 0$. These predicates are used by the POG to generate POs that guard against undefinedness. Because the interpreter is based on McCarthy logic, so is the POG, even though this introduces a disconnect between the semantics of VDM and those of the obligations.

Generating POs is, naturally, insufficient to ensure runtime safety of the model. In order to achieve this, the POs must be discharged. In order to discharge POs we rely on the tool Isabelle [72], a framework for implement-

A	B	A **or** B (McCarthy)	A **or** B (Kleene)
true	**true**	**true**	**true**
true	**false**	**true**	**true**
true	⊥	**true**	**true**
false	**true**	**true**	**true**
false	**false**	**false**	**false**
false	⊥	⊥	⊥
⊥	**true**	⊥	**true**
⊥	**false**	⊥	⊥
⊥	⊥	⊥	⊥

Table 2.1: Truth table for OR operator in McCarthy and Kleene logics

ing logical formalisms. One such formalism is Isabelle/UTP [37], a mechanisation of Unifying Theories of Programming (UTP) [47] in Isabelle. Isabelle/UTP allows for the construction and combination of theories and proofs of their properties. It is upon this framework that the semantics of VDM are mechanised in order to provide proof support.

2.7 Other tools and case studies

The development of Overture is supported in part by the *AstCreator* tool [2]. *AstCreator* generates ASTs from specification files. In addition to the AST nodes, the tool also generates mechanisms to traverse and manipulate the tree using visitors [40].

AstCreator also allows for the addition of new nodes to an AST by using its extension mechanism. This mechanism allows *AstCreator* to produce nodes and visitors that allow construction and traversal of hybrid trees, i.e. tree structures composed of both original AST nodes and new nodes contributed via an AST specification extension file. In addition to adding new nodes, it is also possible to extend existing nodes by adding new fields to them.

In terms of case studies, from a tool perspective the main case study for extensibility carried out in this PhD was the Symphony tool [16], developed as a large set of extensions to Overture. Symphony is a tool for model-based analysis of Systems-of-Systems and supports the COMPASS Modelling Language (CML) [79] — a combination of VDM and Communicating Sequential Processes (CSP) [46]. From an extensibility perspective, the most relevant aspect of Symphony is how it reuses and extends Overture components in order

to process the VDM elements of CML. Symphony provides cases of data extensibility (the new notation must be supported with a data structure that extends the existing one) and functional extensibility (Symphony provides new features that are not available in Overture).

A secondary case study for this PhD was carried out from a modelling perspective. In order to step away from the software-centric analysis of the rest of the work and to try and gain a more abstract view on extensibility, a model-based approach to the problem was made. The goal of this case study is not the construction of extensible models (or rather, that is not the main goal). The goal is to model an extensible system with the expectation being that this more abstract approach yields higher-level insights into the notion of extensibility.

The problem addressed is the optimisation of harvest operations in the research area of mechanised agriculture. For a given crop field, there is an optimal way to harvest it by dividing the field into several rows and harvesting them in a particular order — a route. In addition to the order of the rows, it is also important to consider service wagons as the harvester cannot harvest the entire field — the yield of the field is many times greater than the capacity of the harvester — and must make multiple offloads while harvesting the field.

From the perspective of extensibility, the solution must be able to support multiple optimisation algorithms for route planning of the harvester and the service wagons. Additionally, the system should be extensible to cope with harvesting of various kinds of crops and eventually other kinds of field operations.

3

Specific Applications of Extensibility

This chapter contains the practical contributions of the thesis — the specific problems that were tackled, and their solution, from an extensibility perspective. These contributions address issues of interest to specific communities such as the Overture/VDM community. There are three main areas of contributions: the Overture tool, its extension Symphony, and a case study based on formal modelling of harvest operations. The first two areas allow us to focus on software extensibility while the formal modelling case study complements this by providing a more general perspective on extensibility. The first two areas are intrinsically linked and multiple contributions affect them both. As such, these contributions are discussed and grouped in a thematic order rather than the area they target. Certain paragraphs of these discussions go into significant technical detail and are prefaced with "**Technical implementation details**". The chapter concludes by presenting the formal modelling work as it is sufficiently independent from the rest that it can be adequately presented on its own.

This chapter contains material originally reported in the following publications:

[P24] Luís Diogo Couto and Richard Payne. *The COMPASS Proof Obligation Generator: A Test Case of Overture Extensibility*. 11th Overture Workshop, August 2013.

[P22] Luís Diogo Couto, Simon Foster and Richard Payne. *Towards Verification of Constituent Systems through Automated Proof*. Workshop on Engineering Dependable Systems of Systems (EDSoS), May 2014.

[P25] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. *Extending the Overture code generator towards Isabelle syntax*. 13th Overture Workshop, June 2015.

[P26] Luís Diogo Couto, Peter W. V. Tran-Jørgensen, Joey W. Coleman and Kenneth Lausdahl. *Migrating to an Extensible Architecture for Abstract Syntax Trees*. 12th Working IEEE / IFIP Conference on Software Architecture, May 2015.

[P27] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Gareth Edwards. *Combining Harvesting Operation Optimisations using Strategy-based Simulation*. To be submitted to the International Journal of Computers and Electronics in Agriculture, 2016.

[20] Luís Diogo Couto. *On Extensibility of Software Systems*. Department of Engineering – Electrical and Computer Engineering, Aarhus University, April 2014.

3.1 New Extensions

One of our goals with respect to Overture is to assess and improve its extensibility. Our approach to assessing extensibility is via the construction of a new extension. Although it is a labour-intensive process, it not only yields a new extension, but provide feedback on the extensibility of a system and constitutes the basis for an extensibility analysis. Driving the analysis with the development of a new extension is also effective at uncovering subtle extensibility issues as they are encountered directly during the development process.

This kind of extensibility exploration is primarily oriented towards implementation since one is working directly with the source code, although the analysis is still capable of uncovering extensibility issues related to an architectural perspective. There is also a potential issue of familiarity with the source code as someone who is familiar with the source code may succeed at constructing an extension where someone new to the code would fail. This can be taken as an extensibility issue since a system that requires deep knowledge in order to be extended cannot be considered to be easily extensible, although this particular aspect of extensibility is quite challenging to measure.

In terms of new extensions, there are various dimensions along which one can build a new extension:

- One may construct an extension that provides new functionality that is conceptually related to the rest of the system and relies on the core functionality of the system to a lesser or greater extent.

- One may provide an alternative version of existing functionality or extension, perhaps in a complimentary way that allows both versions to coexist.

- One may also create an extension that implements existing functionality over a new data type.

The first extension we carried out extended existing functionality over a new data type. More concretely, the Overture POG for VDM was extended to support CML as part of the Symphony tool extension of Overture. The main goal was to have a high level of reuse and directly utilise the Overture POG to generate all the POs from VDM constructs inside CML directly. Because of this, the overall structure and behaviour of the extension is heavily influenced by the base Overture POG and follows the same visitor-based approach.

When developing a new extension, as one identifies and reports on extensibility issues, it is important to distinguish between extensibility-specific issues and other, more general, issues that can also negatively impact the process of constructing an extension. The latter class of issues should not be reported as extensibility issues. For example, while poor source code readability can hamper the construction of new extensions, it is a more general issue that reduces overall source code maintainability. Therefore, it should be documented as such and not as an extensibility-specific issue. In terms of extensibility-specific issues, however, these can be broadly grouped into two categories: extensibility-blocking issues and extensibility-hampering issues.

- *Extensibility-blocking* issues are the more serious group: these are issues that somehow prevent the construction of the extension in the desired way.

- *Extensibility-hampering* issues do not prevent the construction of the extension but they significantly impede it and the workarounds reduce the technical quality or efficiency of the final code.

When documenting an extensibility issue and ways to address it, in addition to the group it belongs to, there are several points worth noting:

- the degree to which the issue reduces or affects extensibility;

- where in the source code of the base system the issue lies;

- what exactly it is that the issue prevents one from achieving;

- why that issue prevents extensibility.

In terms of addressing uncovered issues, it is important to propose a fix and describe how said fix addresses the issue. Ideally, an implementation of the fix should also be carried out, though that is not always possible due to, for example, lack of modification rights over the code base of the base system.

Technical implementation details Working on the POG extension revealed two significant issues with the extensibility of Overture, namely in situations where the extensions contain visitors that must rely on the base Overture visitors to process parts of the AST.

1. There is no way to identify a node belonging to the base or extension tree without using explicit `instanceof` checks in a manually implemented decision method. One must use the default cases of visitors to work around this limitation. This issue is further compounded by the absence of a default case for base nodes (there are only default cases that match all nodes and extended nodes). This specific issue can be traced back to the Overture AST itself and not just the Overture POG.

2. When a base visitor takes over processing of a node, the extended visitor is no longer in control. The control of execution remains with the base Overture visitors and that control is never relinquished back to the extended visitor. This becomes a problem for ASTs where a base node has extended nodes as its children, and this case happens frequently in the Symphony tool. The hybrid tree is passed to the base visitors, but they are not capable of processing the extended nodes, causing the POG to terminate in error.

Contribution 1. Extensibility analysis of the Overture POG carried out via construction of a new extension. Analysis reveals extensibility-blocking issues in terms of data type extensibility. The issue lies in implementation of the visitor pattern and its inability to cope with hybrid data structures.

In addition to identifying extensibility issues in Overture, it was also possible to address these issues thus improving the extensibility of the tool. This was achieved by modifying the original implementation of the visitor pattern in Overture to allow the base visitors to release execution control back to the extension visitors.

Technical implementation details In order to address this issue, the notion of a main visitor was introduced. All visitor applications are now routed through the main visitor whereas they were previously locked to the visitor performing the initial dispatch due to use of the **`this`** keyword. Under the main visitor, applications are realised with a reference (`mainVisitor`) to the visitor to apply. The main visitor is a parameter in the base Overture visitors. When there is no need to utilise the extended visitors, the Overture visitors simply receive references to themselves as the main visitor parameters.

Technical implementation details When the base visitors are used by the extension, the extension visitor is set as the main visitor parameter. This means that every visitor application returns the AST to the control of the extension. The base visitor simply inspects a node, generates any relevant POs, and applies the main (i.e. extension) visitor to any sub-nodes. In effect, the base visitor is called for the use of only one method at a time.

Contribution 2. Extensibility improvement of Overture POG by modifying the implementation of the visitor pattern it used.

This contribution was generalised in Contribution 9 discussed in Section 4.1.

3.2 Combining Extensions

An important aspect of extensible systems is the ability of the extensions of the system to interact in constructive ways. The combination of extensions is particularly constrained by SA as that is, to some extent, what ultimately defines how extensions (which are components of the system) are organized and interact with each other. Provided that the SA enables such combinations, significant synergistic benefits may be available by having the extensions collaborate. However, combining extensions presents its own set of challenges, some of which are discussed here.

The principal idea behind combination of extensions is to use the combination to provide new or improved functionalities. One way to achieve this is by chaining functionalities together. However, in order to achieve this, each extension must be capable of processing data in accordance with a common format.

However, it may be that, for various reasons, an extension is not capable of using a common data format. Perhaps it is a legacy extension, or it was simply never designed with this kind of interoperability in mind, particularly in terms of its output. In these situations, the extension itself must be modified in order to support the common format.

In the Overture tool, this issue occurred when attempting to combine the POG with other extensions. The POG was capable of reading the common data format (the Overture AST) but its output was VDM concrete syntax encoded as strings. This made it challenging to combine the POG as an intermediary link in a functionality chain. The approach used to address this issue was to modify the POG so that it produced an output in the common format by converting the output format from string to AST.

The conversion from strings to specialized ASTs was possible because the AST matches the VDM language grammar and the PO predicates themselves are expressed as VDM predicates. This meant that no changes had to be made to the POs but only to the variables representing the predicates.

Technical implementation details Additionally, no changes were made to the behaviour of the POG (expressed in the visitor classes) since, to the outside, the PO classes did not change. The conversion work involved changing the type of the PO predicates from strings to `INode` (the default type of an AST node) and rewriting the constructor code of each obligation. Because all the necessary information for constructing the predicate was already passed to the PO, the changes simply focused on how that information was used in constructing the predicate. Therefore, these changes were localised and no further work was needed to accommodate them. However, the changes themselves were significant as the structure of the AST classes required more complex code when constructing a predicate.

Contribution 3. Improvement of extensibility of the Overture POG by converting its output from a string to an AST. This improvement was reflected in the Symphony POG at no extra effort since the Symphony POG was itself an extension of the Overture POG.

The successful conversion of the POs to ASTs enabled the POG to be combined with a Theorem-Proving Plug-in (TPP) to provide extended static checking of CML models. This was a particularly strong example of extensibility enabling team-based development as the POG and theorem proving extensions were developed by different people in different research groups.

While the POG is responsible for generation of POs, the task of discharging them falls to the TPP. At its core, the TPP consists of a mechanised semantic model for CML within *Isabelle/UTP*. It is essentially a deep embedding of CML, in that an explicit semantics is given to each of the constructs of CML within Isabelle.

The TPP processes a CML model and its associated POs and automatically generates Isabelle theory files for them by means of syntax translation. These theory files can then be submitted to Isabelle for discharging through various automated proof tactics such as auto and sledgehammer, or the custom-written cml_tac tactic that maps a CML formula onto a HOL formula.

The TPP offers a fully automated mode of interaction with Isabelle where users simply choose which PO to discharge and all inner workings (such as tactic selection and result collection) are hidden from them. Furthermore, because the TPP connects to the *Isabelle/Eclipse* plugin, the full functionality of that plugin and, by extension, Isabelle itself also becomes available.

Contribution 4. Combination of the POG and the TPP — separately developed extensions in the Symphony tool — to provide extended static checking of CML models.

Contribution 3 and Contribution 4 are generalised in Section 4.2 as Contribution 10 and one potential issue with it is reported as Contribution 11.

3.3 Improving Extensibility through Software Architecture

In Section 3.1, we reported on extensibility on a micro-scale: improving extensibility of a system component by addressing specific implementation details. In this section, we present extensibility analysis and improvement in a macro-scale: addressing larger-scale SA extensibility issues by means of architectural migration and refactoring. The migration involves restructuring and redistribution of the various system components, so source code modification is still present. The source code considerations are not related to specific implementation details but to the larger picture of how the code is distributed.[1]

[1] For the rest of this section, we use the term *original* to refer to the original pre-migration architecture and *extensible* to refer to the post-migration architecture.

Causes for the poor extensibility of Overture were found in the design of the AST. A given instantiation of the AST is an internal representation of a VDM model, and is composed of nodes used to represent the various language constructs of VDM. The structure of the AST has a deep influence on the overall architecture of the tool as every core component depends directly on it.

Technical implementation details The nodes composing the AST in the original architecture are handwritten and use a centralised design where core functionality, such as type-checking and evaluation, is implemented directly in each node class. As shown in Figure 3.1 any node that can be type-checked has a `typeCheck()` method, any node that can be evaluated has an `eval()` method, and so on. The type-checking process is invoked directly on the nodes. Therefore, the original architecture follows the standard approach to cohesive OO design.

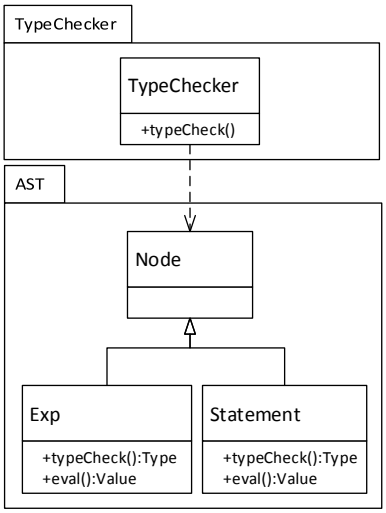


Figure 3.1: Static view of the original architecture showing part of the AST and the type-checker module.

Similarly to what was described in Section 3.1, issues were encountered when attempting to extend the tool. Moreover, these issues were encountered

repeatedly while attempting to construct different extensions, particularly when extending the underlying data structure of the tool and attempting to extend the various existing features to support it. These problems were rooted in larger issues with the original SA.

- adding or changing tool functionality that interacts with the AST requires changes to the nodes themselves. This is problematic since most of the functionalities of the tool interact with the AST and, therefore, the AST would be under constant change, imposing the risk of inadvertently breaking the functionality of other components.

- node classes become very large as the number of different tool functionalities increases. For example, introducing a code generator for VDM in the original architecture would require a new `codegen()` method to be added to all nodes, mixed in with all other methods in those nodes.

- it is challenging to preserve an existing functionality when it is being replaced incrementally, or when an alternative version of the functionality is being developed. This is because the only way to replace a given functionality without destroying its existing implementation is to directly alter the invocation of its respective method at every relevant place in the code.

- maintenance is challenging since modifications to functionality require updating every affected node. In the case of functionality that applies to all nodes, this means updating the entire AST. Since the grammar of the VDM language has a large number of constructs, updating the entire AST is a tedious and error-prone process that is costly in terms of development resources.

Contribution 5. Extensibility analysis of the SA of the Overture tool and identification of various extensibility issues, particularly related to data structure-driven extensibility.

The recurrence of issues when extending the tool confirmed that the extensibility problems ran deeper than technical implementation and had to be addressed at the SA level. To do so, a new SA for the tool was conceived and the existing code base was migrated towards it. Alternatively, it would have

been possible to re-develop the system from scratch but due to the volume of the code base, it was decided against this.

The extensible architecture was inspired by the *SableCC* parser generator [39], specifically its generation of tree nodes and mechanisms to walk the tree. It was also guided by previous experience developing functionality in Overture. The goal was to develop a common shared AST for Overture [78, 71], and heavy use of the standard visitor pattern was made to achieve this.

Technical implementation details In the extensible SA, each class that represents a node of the AST has methods for field access and is equipped with template-based `apply()` methods that are specific to the generated AST. These `apply()` methods have four variants that support various analysis interfaces.

Technical implementation details The analysis interfaces contain multiple `case`*`NodeType`*`()` methods, corresponding to each type of node in the generated AST. For example, an `ANotExp` node has a `caseANotExp()` method corresponding to it. Each of these `case`*`NodeType`*`()` methods is invoked when its corresponding AST node is visited. These interfaces make it possible to implement any analysis that had been implemented in the original design by deriving `case`*`NodeType`*`()` visitor methods from the methods originally embedded in the AST classes. This can be seen in Figure 3.2, which shows part of the extensible architecture of the AST and type-checker modules, the latter now subclassing a visitor in the AST module.

Data structure extensibility is achieved by means of the *AstCreator* tool and its feature to extend ASTs by adding new fields or nodes via AST specifications. An extension is entirely new code, and duplicates none of the existing AST; the extension typically depends upon the base AST through inheritance. When an extension is generated it creates a set of Java classes for only the new and the extended nodes. It also includes a new set of analysis interfaces that handle the new additions.

Technical implementation details The extension does not affect existing visitors that implement analyses of the base AST, and these visitors may also be applied to the extended AST. As such, the extended AST nodes need to support application of both the base and the extended visitors. In order to achieve this, the `apply` method of extended AST nodes contains logic to

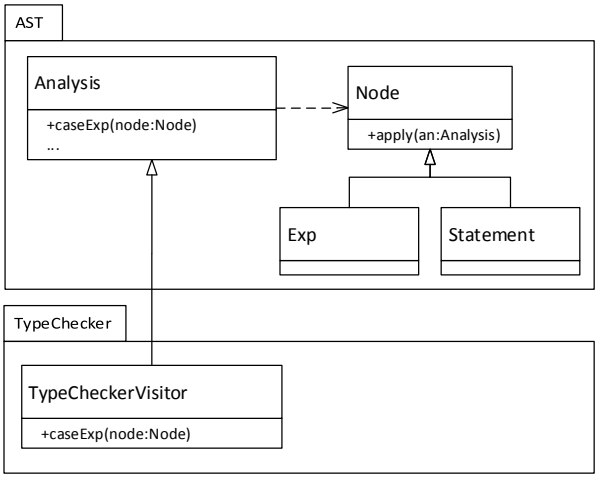


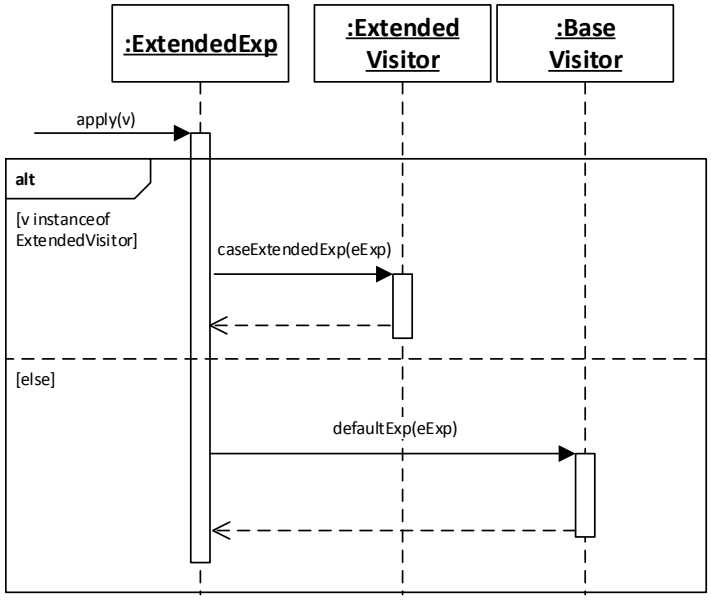
Figure 3.2: Static view of the extensible architecture showing part of the AST and the type-checker module.

detect if a visitor was written for the extended AST. When an extended visitor is applied to an extended node the dispatching proceeds as normal and the relevant case method is called (see Figure 3.3a). However, if a base visitor is applied then the closest default method in the base hierarchy is called (see Figure 3.3b). Naturally, extended visitors work as normal when applied to base nodes.

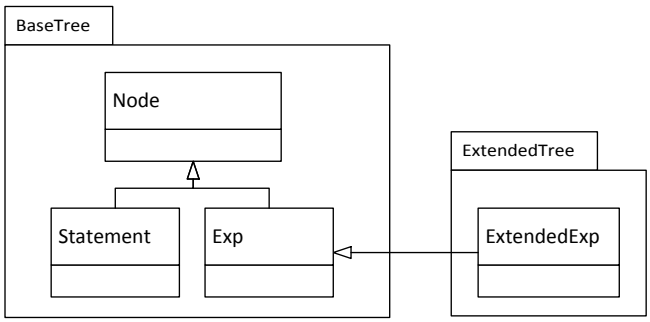
As for the migration process itself, it did not need to use the AST extension features of *AstCreator*, but did attempt to migrate the original functionality in a way that would allow for future extension (though more needed to be done after the migration was notionally complete — for example, the work reported in Section 3.1).

Given that the extensible design is visitor-based, tree traversal and analysis were moved out of the AST, and core functionality such as type-checking and evaluation were placed in separate modules. The old parser was an exception to the plan of applying a visitor-based design to the migrated functionality, given that it instantiated the AST rather than traversing or analysing it. It was nevertheless moved to a separate module but its design did not become visitor-based.

Technical implementation details Because all the components were structured in a similar way, it was clear that as long as the new architecture supported the type-checker, it would support all of the other components. Thus the migration of the type-checker was particularly crucial for the overall mi-



(a) Visitor dispatching.



(b) Extended AST.

Figure 3.3: Example of visitor dispatching for an extended AST.

gration process. The type-checker migration moved the type-checker's functionality out of the nodes and into a separate visitor. The migration of the functionality started by moving the `typeCheck()` methods out from inside the old nodes and into the corresponding `case`*`NodeType`*`()` method of the visitor. The Java compiler was then relied upon to identify auxiliary methods that were used in each of the `case`*`NodeType`*`()` methods that were missing; these auxiliary methods were placed into an assistant class. While the assistant classes were useful during the migration to ensure that functionality was preserved correctly, the use of these assistants was determined to be an anti-pattern and their class structure has since been simplified.

The migration phases that followed that of the type-checker continued in the same manner. All of the following phases required less effort to migrate than the type-checker; ensuring that the details of the target code structure for the migration were correct was more difficult than actually migrating the code. Furthermore, no fundamental changes were necessary to migrate the interpreter in either the structure of the AST in the extensible architecture or in the functionality of the type-checker.

Contribution 6. Architectural migration of Overture code base. A new, more extensible SA was conceived for the Overture tool and the code base was migrated towards it. The new SA successfully supported various Overture extensions, particularly the Symphony tool.

In terms of assessing the post-migration SA, it can be be considered successful from an extensibility perspective, as witnessed by the successful development of the Symphony tool [16] as an extension of Overture supporting the CML notation which is itself an extension of VDM. To further assess the migration, the original and extensible architectures were compared based on volume and various OO metrics [13].

Volume is used to compare code distribution across both architectures. To facilitate this comparison, we grouped the Java packages of both architectures into several modules. Each module corresponds to a single well-defined functionality, although the package groupings in the original version is somewhat subjective since those packages implement multiple functionalities. In addition, the comparison only includes functionalities that are present in both architectures. The `ast` module corresponds to the VDM source tree and the `core` module provides User Interface (UI) and common utilities. The remaining modules provide the functionality for which they are named.

The volumes of the modules are presented in Table 3.1. The data shows the decentralisation effort, with code migrating from the `ast` and `core` modules to the other functionality modules, primarily the `type-checker` and the `interpreter`. The overall volumes of both architectures, discounting generated code, are similar.

Module	pre-[LoC]	post-[LoC]
core	3747	0
ast	29998	7311
interpreter	18048	35213
parser	7945	7027
pog	2484	5003
type-checker	5226	17488
combinatorial testing	1210	1746
Total	68658	73788

Table 3.1: Volume distributions pre- and post-migration.

Metric	pre-	post-
Methods per class	5	6
Depth of Inheritance	0	0
Number of Children	0	0
Efferent Coupling	10	8
Response for Class	20	21
Lack of Cohesion	1	7
Afferent Coupling	3	2
Public Methods	5	5

Table 3.2: OO metrics pre- and post-migration.

The OO metrics for both SAs are summarised in Table 3.2. For all metrics, the median value is presented. The changes in the metrics correspond to design differences between the two SAs such as in efferent and afferent coupling due to a looser architecture or a large increase in lack of cohesion due to the introduction of the visitor pattern. Other metrics have changed little, in part due to similarities between both architectures — both are deeply influenced by the VDM AST and consist of largely the same code, albeit distributed differently. Of the metrics with little change, the number of public methods was somewhat unexpected since the methods are distributed through

more classes now. Possible explanations for this are the fact that the new public traversal methods overwhelm the measure or that the redistribution of methods has forced additional methods to be made public.

Contribution 7. Architectural comparison of Overture code base before and after architectural migration based on source code metrics.

The extensible architecture has continued to support the development of Overture and has been generalised into a SA proposal for the development of IDEs for formal languages as reported in Section 4.4 as Contribution 13.

3.4 Extensibility in Formal Models

One of the reasons for investigating extensibility from a non-software perspective is to try to gain additional, different kinds of insights into the phenomenon of extensibility. By approaching it from a more abstract perspective — formal modelling, in this case — we gain more abstract and fundamental insights into extensibility. Furthermore, the contributions produced are more fundamental and applicable to a more diverse range of situations. However, this is a rather vague goal and it is also difficult to assess. Therefore, to help focus the research work, we established the goal of producing extensible models and used the harvest optimisation case study (see Section 2.7).

The production of an extensible model shares similarities with that of extensible software. We are interested in preparing the model for future evolution by enabling it to be extended with new kinds of functionality, data, and properties.

In the harvest operation case study, the formal model is trying to capture properties of the operation and rigorously define the participants and the relations between them. This formal model is also used to support the application of various optimisation algorithms to the harvest operation problem. From an extensibility viewpoint, it is this last objective that is most relevant: we support multiple optimisation algorithms including unforeseen ones. Therefore, our model is extensible in terms of optimisation algorithms.

Optimisations for the harvest operation vary not only in terms of algorithms but also in terms of what aspect of the operation is being optimised. This includes the route planning of the harvester and the load planning for harvesters and other vehicles. Because of the variety of aspects that can be optimised, the model not only supports multiple algorithms to optimise the same

aspects of the problem, but also supports interaction between optimisations of different aspects.

The solution proposed lies in strategy pattern-based modelling. This is an architectural design pattern (or architectural style) for formal modelling and constitutes Contribution 14, as discussed in Section 4.5. Essentially, it consists of organising a model into:

- an *Execution Engine* responsible for advancing model execution;
- a *State* component responsible for representing domain entities; and
- a set of *Strategies* responsible for executing the more complex calculations and decisions in the model.

Unlike other general interest contributions, Contribution 14 was not generalised from a narrower one. It was conceived independently and then applied to this case study. It is the opposite progression of other contribution pairs. Therefore, a reader may consider reading Section 4.5 for further details about the overall architecture of the model.

The model was developed according to the structure recommended by the architectural style in Section 4.5. In this model, the Execution Engine advances the simulation by taking the largest possible step that all participants in the simulation can take before needing to compute new instructions.

In terms of domain entities, the harvesters are the primary units of the operation. Routes and coordinated offload points for the harvesters are built by the employed strategies. The grain vehicles are the service units of the operation and their main objective is to convey material from the harvesters to out-of-field storage. The service points coordinate when and where the wagons must meet the harvesters in order for material to be passed between the two.

The strategy classes define how certain aspects of the harvest operation are to be executed, namely in terms of route planning. There are three classes of strategies: route, deconflict and load.

- A *route strategy* is responsible for constructing the routes for harvesters. The routes direct the harvester from its location to a point where it will next require service.
- A *deconflict strategy* is responsible for the infield coordination of the vehicles. It is possible that conflicts can arise when a vehicle may block the path of another vehicle. In this case the deconflict strategy is employed

to determine what course of action (such as planning a new route, or waiting for the obstruction to pass) is to be taken.

- A *load strategy* is responsible for assisting the route strategy to find a location where the harvester can be serviced and for constructing a route for the service unit from its current position to the service point and then to the out of field storage.

The interaction between strategies is as follows:

- The deconflict strategy is consulted by the execution engine to assess whether vehicles can move and to reroute them if they cannot. The deconflict strategy takes as input the route plans produced by the route and load strategies but does not otherwise directly interact with these strategies.
- The load strategy is responsible for assisting the route strategy to find a location where the harvester can be serviced and for constructing a route for the service unit from its current position to the service point and then to the out of field storage. This is done through three specific functions of the load strategy that are called by the route strategy.

Contribution 8. Formal model of harvest operation optimisation. The model enables the application of multiple optimisation algorithms to the harvest operation and the interaction of the optimisation of different aspects of the harvest operation. The model is built by application of the strategy-based modelling architectural style.

4

Generalisations of Extensibility

This chapter presents more general contributions of interest to a broader audience. Some of the contributions presented here are, in fact, generalisations of contributions presented in Chapter 3. The idea is that some of the specific problems solved are in fact instances of more general problems and the solutions proposed can be generalised and thus they can be of greater value to those outside the communities targeted by the previous contributions. We also attempt to identify the situations in which most contributions in this chapter are most relevant, under the heading "***When to use?***". There are three main areas of contributions: generalising specific extensibility improvements, combinations of extensions and extensibility in formal models. The specific extensibility improvement is presented first as it is a stand-alone contribution. The discussion of extension combinations is broader and takes up three sections, presenting different perspectives on it. The chapter concludes with the discussion on extensibility in formal models, which is independent from the rest.

This chapter contains and adapts material originally reported in the following publications:

[P23] Luís Diogo Couto, Peter Gorm Larsen, Miran Hasanagić, Georgios Kanakis, Kenneth Lausdahl and Peter W. V. Tran-Jørgensen. *Towards Enabling Overture as a Platform for Formal Notation IDEs*. 2nd Workshop on Formal Integrated Development Environment (F-IDE), June 2015.

[P28] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Kenneth Lausdahl. *Principles for Reuse in Formal Language Tools*. 31st ACM Symposium on Applied Computing (SAC 2016), April 2016.

[P27] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Gareth Edwards. *Combining Harvesting Operation Optimisations using Strategy-based Simulation*. To be submitted to the International Journal of Computers and Electronics in Agriculture, 2016.

4.1 Generalising the new POG contributions

The improvement of the extensibility of Overture described in Contribution 2 was in fact an improvement to the extensibility of the visitor pattern. This improvement can, in general, be applied to other cases where the visitor pattern is used. Specifically, in cases where the pattern is being extended to support hybrid data structures.

When to use? This contribution is based on an improvement of the visitor pattern and, as such, should be considered in situations where the visitor pattern is appropriate. A more thorough discussion of the visitor pattern is available in [40] but, in general, its use is worth considering when there is a need to detach an algorithm from the class it runs on — in other words, to separate data and behaviour. As for the extensibility improvement, it is worth considering in situations where the base data structure is extended and analyses are created that run on the hybrid data structure while reusing existing base analyses to process the base elements of the hybrid structure.

Contribution 9. Extensibility improvement of visitor pattern in terms of its ability to cope with extensions that require or rely on hybrid data structures.

4.2 Technical aspects of extension combination

Combining extensions can provide significant synergistic benefits and the combination presented in Section 3.2 is just one example of such. In general, extensions can be combined to provide higher-level functionality. These combinations can be along a chain of processing or with multiple extensions processing the same initial data or even passing data back and forth between them.

In all cases of combinations, one of the key technical aspects to achieve success lies in the format of the data structure(s) used as the input and output of the various extensions. Ideally, all extensions share a common format for both input and output in order to maximise combination possibilities. However, this is complicated by various issues:

1. Some extensions present information to users which may mean that the output they produce is in a format for reading by humans but unsuitable for further processing (e.g. strings). One way to address this issue is to

modify the extension to output its data in the common data structure format. Another component can then be created to produce human-readable outputs from the common data structure.

2. More generally, existing extensions may not produce (or consume) data according to the common format. This can be addressed by modifying said extensions in order to support the common format. If this is not possible for some reason (for example, poorly understood legacy code or efficiency concerns), one alternative is to develop a wrapper that converts between the data format used by the extension and the common data format.

3. The common data format itself may not support the needs of all extensions. As extensions increase in number and diversity, more and more demands are placed upon the common data format. This can eventually lead to an inefficient or unmaintainable data format. Therefore, it is important to separate fundamental data that is needed by most extensions and more specific data that only a small subset of extensions requires. This extension-specific data can be placed either in auxiliary data structures or the common format itself can be extended while under use by extensions that require the additional data.

All three issues can affect the successful combination of extensions in a system and in particular issues 2 and 3 need to be balanced against each other in terms of ensuring that extensions do share the common format, but also that the format itself does not end up being excessively heavy. Nonetheless, all issues are addressable thus enabling extension combinations.

When to use? This kind of combination is worth considering when developing systems that perform distinct but related analyses on the same data. This allows for a single, efficient data structure to be used in support of the combination. If several fields of the data structure are used by most analyses, this solution is relevant. On the other hand, if there is little overlap in the elements of the data structure used by the different extensions, then this approach is not ideal.

Contribution 10. Supporting extension combinations through common data structures. The data structure itself should be flexible enough to support the needs of various extensions. The same extensions themselves

should be able to consume (and ideally produce) data according to the format of the common data structure.

When using this kind of extension combination, a common way to address issue 1 is through the use of pretty-printer backends that read the common data structure and produce human-readable output. When working with extensible common data structures, care must be taken when pretty printing them as the extended data may have a different human-readable format – for example, the decorations may change, even for the elements already present in the base structure. In this case, it is possible to develop brand new backends for the extended data structures. However, this can lead to bloated code in terms of its volume and unnecessary maintainability issues. This should be avoided (for example, using extensible backends with parametrizable decorations), since the presence of backends with redundant code constitutes an anti-pattern.

Contribution 11. Multiple backend anti-pattern. When developing backends for extensible data structures where the extended structure redefines the way base elements are to be processed, there is a danger of introducing an anti-pattern in the system by means of multiple, redundant backends.

4.3 Reusability principles for formal analyses

The combination of extensions presented in Section 4.2 can be generalised further into a set of reuse principles for formal analysis tools. In addition to providing an effective way of achieving extension combinations, these principles enable the reuse of the implementations of the formal analyses at the core of these extensions. Reuse is achieved both within the context of a single formal language-specific tool as well as when extending the language itself. The principles are supported by the *AstCreator* tool and are described below.

1. Specification-driven ASTs In language processing, every analysis interacts with the AST, either as input, output or both. For example: a type checker analyses a tree and produces a tree annotated with type information; then an interpreter analyses and evaluates it. This makes the AST a central

component that every analysis depends on. Therefore a language building tool should provide a convenient way to design, specify and maintain an AST, i.e. describing its structure as well as its constructs and their relationships.

Keeping the specification of the AST separate from its implementation follows the notion of separation of concerns. This approach makes it easier to maintain and extend an AST as the focus is the design of the tree rather than the implementation of it. Another advantage of this approach is that a single AST design can, in principle, be implemented in different languages, thus improving reuse of the AST design. The recommended way to implement such ASTs is via tool support that automatically generates the implementation of the tree from the specification. Crucially, such a tool differs from parser generators in its support of the remaining principles.

2. Contract-based AST analysis Contract-based analysis is crucial to enable runtime reuse and further ensure analysis interoperability. The formal tool is aware of the contract and thus is capable of applying any analysis that conforms to the contract. Therefore, to achieve reuse, all formal analyses should conform to the same contract. The contract must specify the input and output of the analysis as well as the means through which the analysis handles each kind of node in the AST. However, the input and output may not be the same for all kinds of analysis so the contract must be parameterisable in terms of input and output. One possible way to define the contract is via interfaces in object-oriented settings. Like the AST implementation, the contract should be generated from the AST specification.

3. Hybrid tree support Extending a language implies reusing the base tree *as is* without modifying it. Extensions can only add elements to the AST. They can neither change the existing structure nor delete elements from it. Language extensions should also be specification-driven. A language extension specification either adds additional attributes to nodes already defined by the base language or it adds new syntactic categories or extends existing ones. Together the specification of the base and the extended tree define nodes that can be used to form *hybrid trees*: tree structures that consist of nodes defined by both specifications.

4. Compatibility between base and extended analyses Extending a language also means that the formal analyses for the language also need to be extended. In this situation, it must be possible to define extended analyses that

reuse the base analyses without modifying them. Furthermore, we require analyses to be *compatible* — meaning that the base and extended analyses should be able to combine in order to process the hybrid tree. The contract for an extended analysis must be compatible with the contract of the base analysis and is also derived from the extension specification. Alternatively, the extended analysis may process the base nodes in a different way that is implemented from scratch.

When to use? This contribution is of particular interest to FM tool builders interested in language extensions and, more generally, to developers of language-based tools who are interested in performing language extensions with direct reuse of complex, handwritten software components that process the language.

Contribution 12. Set of tool-supported principles that enable reusability of implementations of formal analyses.

4.4 Extensible architecture for formal language tools

The combination of extensions discussed in Section 4.2 is particularly well suited for the development of tools for FM notations. In this section, we present a way to use these principles in combination with a strong reuse philosophy as described in Section 4.3 to propose an extensible platform-based architecture for the development of FM IDEs.

The key idea is to use extension combinations based around an AST as the common data structure to provide the actual functionalities of the tool. However, these principles are taken significantly further in providing higher reuse capabilities. Additionally, an IDE platform should also provide capability to construct the UI features of a tool.

In order to facilitate development of functionalities as extensions, a platform should provide a common data structure as well as mechanisms to construct it (normally, a parser) and manipulate it, thus ensuring consistent implementation of the various extensions. Together, these three elements constitute what we call the language core of the platform. Similarly, a platform should enable the construction of a UI for the FM tool. Due to the inherent complexities in developing UIs, the recommendation is to use an existing UI framework and provide a way to simplify access and use of the relevant features of said framework (such as editors or error messages).

The language core encapsulates and handles any language and notation-related concerns, including parsing, representation and analysis, in order to facilitate decoupling between the implementations of the core language and the UI. In addition to the general benefits of separation of concerns, the language core also opens the possibility of migrating the IDE implementation to another UI technology as well as providing the base tool functionalities for command line access, batch processing or as an external tool to be accessed by others.

The language core consists of a set of classes that provide an extensible AST and are automatically generated by the *AstCreator* tool, as well as a parser for constructing concrete ASTs from model sources. In addition, *AstCreator* also generates machinery for traversing and processing trees in a consistent way in the form of a visitor framework [40]. Most kinds of analyses of the AST such as type checking or interpretation are implemented using the visitor framework.

In the Overture platform, UI features are provided by the Eclipse Rich Client Platform (RCP). As part of the platform, Overture contains a set of extensions to the Eclipse RCP that are used to help build the UI components of the IDE. The Eclipse RCP is a generic framework for building rich client applications using the Eclipse OSGi [3] plugin model and UI toolkits. It is powerful and generic but comes with a cost: significant amounts of boilerplate source code and configuration files must be written in order to prepare it to build an IDE.

The Overture Eclipse extensions automate some of the configuration and preparation work by providing the aforementioned boilerplate code targeting FM notations. The extensions provide an extensible application framework on top of the RCP. It significantly reduces the amount of code that needs to be written in order to contribute an extension to the IDE. To put it another way, the RCP API is very wide and the Overture Eclipse Extensions summarise a portion of it, thus giving developers faster access to the functionality at the cost of some flexibility. However, the Overture extensions are fully interoperable with the RCP so any other extension that requires direct access to the RCP can still be used. Broadly speaking, the Overture Eclipse extensions can be divided into three groups:

- a set of UI elements for editors, launch configurations, etc. that interact directly with the Eclipse RCP.
- a set of project elements that represent the FM model and associated concepts such as source units, according to the Eclipse project model.

Also included are connectors and providers for accessing these various entities from within the IDE.

- a set of builders that interact with the language core in order to process language sources to construct an internal representation of the model and load it into the project elements.

Both the builders and the project elements are developed according to standard Eclipse conventions so that new versions of these packages for other notations may be contributed.

In order to further increase the usefulness of the platform, we introduce several extensibility features to the language core. The idea is to allow the same core (and associated analyses) to support a base language as well as extensions to the language itself, thus promoting even more reuse. These features are based on the set of reusability principles described in Section 4.3.

The basic principles of extensibility in the Overture language core are related to the generation of ASTs from specification files, similar to parser generators like SableCC [39]. In addition to generating the classes representing the tree structure, it is important to generate auxiliary machinery to allow developers to implement analyses of the AST in a consistent manner.

The main way to construct extensions in the language core is by extending the AST. Generally speaking, an AST is extended by adding new subtrees that are either entirely new or that contain some existing base nodes. In addition, the extended tree needs to reuse the existing base node classes wherever possible.

In addition to extending the tree itself, it is important to also extend the analysis machinery. Particularly, this extended machinery needs to be able to analyse trees made up of extension and base nodes. Furthermore, the extended analysis machinery needs to reuse the base machinery when processing base nodes — this is essential for achieving reuse of functionalities already implemented as base analyses.

Whether speaking of a tree made of only extension nodes or a hybrid tree with extension and base nodes or even a base tree, the AST classes have a limited ability to enforce the structure of each particular instantiation of the tree. It is the syntax of the language, as encoded in the parser, that ultimately controls which trees are admissible. Along the same lines, it is the parser that controls which base nodes are reused when constructing hybrid trees as the extended tree specification can only set an upper limit on this.

This platform-based architecture has been successful in supporting the development of the Overture IDE and allowed not only its continued evo-

lution but it has also supported the development of new IDEs on top of the platform such as the Symphony IDE.

When to use? Platform-based IDEs are suited for FM notations, particularly when there is a need for high reuse and extensions to the formal notation itself. One aspect that makes formal notations particularly suited to platform-based IDEs is that FM frequently consist of a multitude of distinct analyses to be performed on the same formal model and it is desirable to have these analyses implemented in a consistent way. Non-FM tools in similar situations may also benefit from the use of platform-based architectures.

Contribution 13. Platform-based architectural description for development of formal notation IDEs. The platform consists of a core that ensures consistent implementation of various formal analyses and framework extensions that facilitate access and usage of a UI framework. A key feature of the platform are the extensibility principles that enable a single platform to support multiple formal notations.

4.5 Strategy pattern-based Modelling

In this section we present an architectural style for formal modelling based on the strategy pattern [40]. The main goal of the style is to enable the development of formal models where the primary kinds of extensions are alternative realisations of the same functionality. The style was developed for the VDM++ notation but it can be used with other formal notations that provide support for OO features.

The style divides the elements of the data into separate data elements, execution elements and processing elements. These elements correspond to three sets of classes, as shown in Figure 4.1.

Execution Engine is responsible for coordinating and advancing simulation of the model. It is also responsible for connecting the other sets of classes.

State classes represent data in the model and are used to model the domain entities of the problem. They hold minimal functionality other than basic data access. Due to the use of formal modelling notations, we can use features such as invariants to express and verify properties about the data itself and thus gain valuable insights about it.

Strategies are responsible for processing data and making decisions that are needed during model execution. Various kinds of strategies may be defined to handle different aspects of the problem. Strategies are defined by their contracts and once again, we can leverage the features of a formal notation such as pre- and post-conditions to formally define the contract. This enables us to initially verify the feasibility of the contract and when concrete versions of a strategy have been implemented, we can verify their adherence to the contract.

Broadly speaking, the three sets of classes interact as shown in Figure 4.2. The Execution Engine is responsible for advancing the simulation of the model. It uses the State and domain classes to populate the model with initial data and then progresses the simulation according to the execution rules that have been implemented (this varies greatly depending on the model). Whenever a non-trivial decision or result must be calculated, the model extracts the relevant data from the State and passes it to the relevant strategy. The strategy calculates an answer that is used by the engine to update the state or progress the simulation as appropriate.

A particular model may have multiple kinds of strategies defined and these may interact directly by calling operations from each other. However, it is worth noting that this kind of interaction is typically very difficult or impossible to enforce[1] so it is recommended to clearly define which operations of a strategy are meant to be called by other strategies and document their purpose and assumptions on their use.

[1] This holds for formal notations in the style of VDM++. Other notations may be capable of enforcing the interactions.

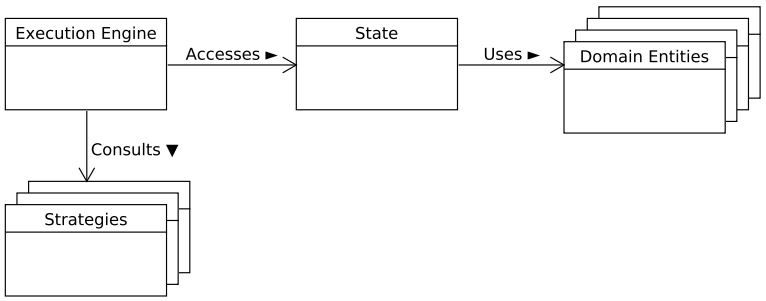


Figure 4.1: Model structure realised as a UML class diagram

When to use? The main advantage of this style lies in its use of the strategy pattern as it allows for different alternatives to the same functionality to be explored. For example, different algorithms may be compared for performance or for the quality of their results (see Section 3.4 for an example with optimisation). Another possibility is to develop different versions of an algorithm that are more amenable to different kinds of analyses such as theorem proving or model checking. Therefore, this architectural style may be

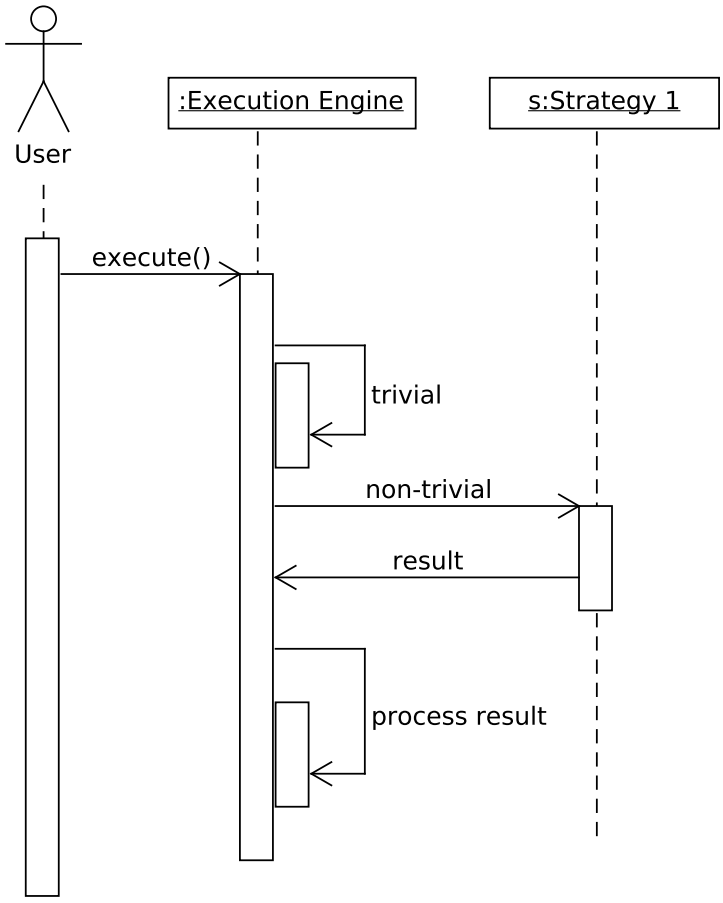


Figure 4.2: Model execution realised as a UML sequence diagram

used whenever doing formal models that aim to explore different alternatives in terms of algorithms for the same functionality.

Contribution 14. Strategy-based architectural style for formal modelling. The style consists of separation between data, execution and data processing and facilitates formal analysis and comparison of different alternatives for the same functionality.

5

Foundational Contributions

This chapter discusses contributions at a more theoretical or foundational level and how these can be used to support extensibility at more applied levels. There are two areas of contributions: a theoretical extension of the proof rules for VDM and leveraging of a mechanisation of the semantics of VDM as a way to help provide automated proof support. The two topics are independent of each other and are presented as such. The extension to the proof rules is presented first, followed by the discussion of proof support.

This chapter contains material originally reported in the following publications:

[P25] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. *Extending the Overture code generator towards Isabelle syntax*. 13th Overture Workshop, June 2015.

[P21] Luís Diogo Couto, Nick Battle and Peter Gorm Larsen. *LPF-Aware Proof Obligation Generation in VDM/Overture*. To be submitted to the 5th International ABZ Conference (ABZ 2016), May 2016.

5.1 Extending Proof Rules of VDM

The Overture POG was the target of an extension providing additional functionality. Specifically, the POG was extended to provide support for generation of POs according to the principles of Kleene-based LPF. The original POG followed McCarthy logic with left-to-right evaluation of boolean operators and "short-circuiting" evaluation of expressions when the left branch is sufficient to determine the overall value. This is because McCarthy's is the logic used in the Overture interpreter and the POG is used, in part, to protect against interpretation errors in specification execution.

In essence, the extension provides an alternative and complementary functionality — it is not akin to providing a different implementation or algorithm

since the generated POs are different. In order to support this extension, however, it was necessary to work out the proof rules for this alternate way of calculating proof obligations.

The fundamental change in the generation of LPF POs lies in how composite boolean expressions are manipulated. This means that the behavior of the POG must be altered when it is applied to elements such as:

- **`and`** binary expressions
- **`or`** binary expressions
- **`=>`** binary expressions
- **`forall`** quantified expressions
- **`exists`** quantified expressions
- **`if then else`** and **`cases`** expressions

The proof rules for the various boolean operators are summarised in Table 5.1 along with their respective definedness predicates. We omit any discussion of the negation operator as it is the same in both versions of the POG since the definedness of negation is identical in McCarthy and LPF. Note that, for the conjunction, disjunction and implication operators, two separate predicates must be generated for McCarthy logic.

The conjunction (**`and`**) and disjunction (**`or`**) operators are duals of each other. So, whereas a disjunction holds if either of its members is true, a conjunction is false if either of its operands is false. Extending this to LPF, a conjunction is defined if either of its operands is false or if all of the operands are defined whereas a disjunction is defined if either of its operands is true or if all operands are defined. The definedness predicates for conjunction and disjunction are shown in rows 1 and 2 of Table 5.1.

The implication operator (**`=>`**), unlike disjunction and conjunction, does not have an absorbing element and therefore we cannot apply the LPF extension directly. However an implication can be unfolded into disjunctive normal form through the following equivalence: $P \Rightarrow Q \equiv \neg P \vee Q$. From there, it follows that the definedness predicate for the implication operator is as shown in row 3 of Table 5.1.

The universal and existential quantifiers are generalizations of the conjunction and disjunction operators respectively so the rules for handling them are also generalizations of rules for dealing with conjunction and disjunction.

logic expression	LPF (Kleene)	McCarthy
δ(P **and** Q)	$\neg P \vee \neg Q \vee (\delta(P) \wedge \delta(Q))$	$\delta(P)$ $P \rightarrow \delta(Q)$
δ(P **or** Q)	$P \vee Q \vee (\delta(P) \wedge \delta(Q))$	$\delta(P)$ $\neg P \rightarrow \delta(Q)$
δ(P **=>** Q)	$\neg P \vee Q \vee (\delta(P) \wedge \delta(Q))$	$\delta(P)$ $P \rightarrow \delta(Q)$
δ(**forall** x : T(x) & P(x))	$(\exists x : T_x \cdot \neg P(x))$ $\vee \forall x : T_x \cdot \delta(P(x))$	$\forall x : T_x \cdot \delta(P(x))$
δ(**exists** x : T(x) & P(x))	$(\exists x : T_x \cdot P(x))$ $\vee \forall x : T_x \cdot \delta(P(x))$	$\forall x : T_x \cdot \delta(P(x))$

Table 5.1: Logical expressions and their LPF and McCarthy definedness predicates.

A universally quantified expression is defined if it is **false** for at least one of its values or if it is defined for all values. We write its definedness predicate as shown in row 4 of Table 5.1. Conversely, the rule for existential quantifiers is an extension of the rule for disjunction where just one of the expressions is **true**. Its definedness predicate is shown in the last row of Table 5.1.

We also consider the case of nested binary operators. In the McCarthy POG, these are not particularly interesting as the operators (and their subexpressions) are simply processed left to right. As the subexpressions are encountered, any relevant POs are generated and additional context information is generated and prepended to subsequent definedness predicates. This context information ensures the subexpression has the relevant truth value that ensures the subsequent subexpressions are evaluated. The overall effect of this is that the further to the "right" an inconsistency is, the more complex the resulting PO is.

Nested operators are handled differently by the LPF POG. In LPF subexpressions are not treated separately. Thus for an expression with nested operators, a single PO is still generated. The difference lies in the definedness condition which now apply to more complex operands. As an example, for A **and** B **and** C the definedness predicate is $\neg A \vee \neg(B \wedge C) \vee \delta(A) \wedge \delta(B$ **and** $C)$. Of course, $\delta(B$ **and** $C)$ can be unfolded to: $\neg B \vee \neg C \vee \delta(B) \wedge \delta(C)$. In that sense, there is still a degree of left to right processing taking place. However, this is only in the PO generation phase. At the proof stage,

the PO may be manipulated in such a way that, for example, proving $\neg C$ is sufficient to discharge the whole PO. In a McCarthy setting, with multiple POs, this simple proof is not sufficient to discharge all of them. A direct comparison between the multiple McCarthy POs and the single LPF PO can be seen in Table 5.2.

Expression	A **and** B **and** C
McCarthy	$\delta(A)$ $A => \delta(B)$ $A => B => \delta(C)$
Kleene	$\neg A \vee \neg(B \wedge C) \vee \delta(A) \wedge (\neg B \vee \neg C \vee \delta(B) \wedge \delta(C))$

Table 5.2: POG comparison for nested conjunctions.

In terms of implementation, it required fairly small adjustments to the base Overture POG to change the logic supported from McCarthy to LPF. With LPF it is possible to avoid the use of certain "guarding expressions". The main advantage of supporting LPF is that it is more powerful to use LPF in proofs. In general, fewer POs are generated using LPF logic. However, some of them are more complex and closer to the semantic definition of definedness in LPF. The main drawback of using LPF for the POG is that the Overture interpreter is currently not able to use LPF and thus that even though LPF PO generation and discharge ensures the consistency of a specification, its interpretation may still yield runtime errors and specification writers must be aware of this issue, particularly when executing elements of a model that contain expressions that are handled differently in LPF. Also, LPF POs are, in general, longer and more verbose than their McCarthy counterparts. In any case, the work carried out serves as a valuable test of POG extensibility in terms of alternate functionalities and may be valuable input for others who have existing POGs about the relatively minor adjustments needed to be able to explore other logics supporting undefinedness.

Contribution 15. Proof rules for PO generation according to the semantics of LPF supporting a new extension for the Overture POG.

5.2 Integrating Proof Automation

In VDM, POs can act as a check on the intuition and designs of the modeller, ensuring that a model is both feasible and can be executed in the interpreter without yielding runtime errors. This allows the modeller to focus on design considerations and delay the proving of properties and model consistency to when PO discharge is attempted. While this makes POs valuable, in order for them to be truly useful as part of a tool-supported workflow, there is a need for automated proof support.

In order to achieve automated proof support, and keeping in line with the general theme of reuse, the Overture tool support for VDM was connected with an existing theorem prover, namely Isabelle [72]. There are various reasons for choosing Isabelle (see Section 2.6), but the most relevant one is the existence of a CML embedding — *HOL-UTP-CML* [36] — that, due to its close connection to VDM, can be used to help provide the desired proof support.

In order to provide Isabelle-based proof support for VDM, it is necessary to export a VDM model (and its associated POs) to Isabelle. The approach chosen is to adapt and reuse the *HOL-UTP-CML* embedding to create *HOL-UTP-VDM*, an Isabelle embedding of the VDM-SL dialect.

Contribution 16. Adaptation of HOL-UTP-VDM Isabelle embedding, based on existing HOL-UTP theories, thus providing automated proof support for VDM.

In order to take advantage of the embedding in the context of the Overture tool and VDM in general, it is necessary to translate VDM models into it. While it would have been possible to adapt the existing translation of CML models, this was a good opportunity to both explore the extensibility of the code generation platform of Overture and, furthermore, assess whether it is beneficial to use the platform to perform model translations rather than using handwritten translations. As such, the existing translation was adapted and re-implemented by means of extending the Overture CGP to translate VDM models into Isabelle syntax. As part of this work, an extensibility analysis of the Overture CGP was also carried out, similar to the analysis described in Contribution 1. Various issues were uncovered by this analysis and subsequently addressed.

The translation targets the HOL-UTP-VDM Isabelle embedding that is very similar to VDM in the sense that the majority of constructs in VDM are

present in the embedding. Therefore, after constructing the initial IR for the VDM model, there is a relatively small number of operations that need to be performed over the IR in order to carry out the translation.

The first set of operations is also the simplest and most common: direct syntax translations. These translations can be applied directly to the initial IR nodes that already map directly to a construct in the embedding. A few of them are shown in Table 5.3. In general, the syntax of a construct in HOL-UTP-VDM is the same as the one in VDM, with the following differences:

- all constructs are delimited by `"` to identify them as user-defined syntax in Isabelle
- variable names are delimited by `^` to mark them as model variables
- types are prefixed by `@` to mark them as model types
- string literals are delimited by `''`
- certain operators (such as `in @set`) are prefixed by `@` to further disambiguate them

VDM	Isabelle embedding
`x`	`"^x^"`
`int`	`"@int"`
`f(1)`	`"f(1)"`
`"foo"`	`"''foo''"`
`if b then s1 else s2`	`"if ^b^ then ^s1^ else ^s2^"`
`x in set y`	`"^x^ in @set ^y^"`

Table 5.3: Example VDM constructs and their HOL-UTP-VDM counterparts.

The second set of operations consists of tree transformations, of which the first is reordering of definitions. Isabelle does not allow forward referencing in its definitions so any dependency of a definition must be processed before the definition itself. When generating the syntax, the CGP processes definitions in the IR in the order in which they appear so it is necessary to reorder the IR nodes according to their dependency relation.

The final operation over the IR is also related to dependency handling, specifically the dependencies between mutually recursive functions. It is possible to define mutually recursive functions in Isabelle but they must be placed

together in a single definition block and identified as being mutually recursive. Because VDM has no such restrictions, mutually recursive functions may be spread through the definition list. As part of the translation to the embedding, these functions are identified, grouped and placed into an IR node that has been newly-defined for this purpose.

Contribution 17. Translation rules for VDM models into the HOL-UTP-VDM Isabelle embedding.

6

Conclusion

6.1 Summary of Contributions

In this section, the contributions of the PhD are summarised and visualised in order to provide an overview. The contributions are visualised as a graph in Figure 6.1. Each contribution is a vertex in the graph. Three kinds of relations are established between contributions, all as edges:

generalise contribution has been developed by generalising its predecessor;

instantiate the opposite relation of generalise; and

build on contribution builds on top of work reported as its predecessor.

There are 6 distinct graphs, surrounded by boxes. Each corresponds, to some extent, to an independent vein of work. There are few connections between contributions of different blocks. This is reflective of the discrete nature and the research methodology which consisted of several independent experiments.

Contribution 15 is in a standalone block. A reason for this is that the contribution itself is somewhat coarse as it represents both the foundation work to support an experimental extension and the extension itself. One could divide it into two, with the extension building on the theory. However, there is nothing particularly worthwhile in the extension itself so it was not separated.

Contributions 5 to 7 represent the deepest continuous work in the PhD. Together, they culminated in the fairly large refactoring effort which led to the revised, extensible architecture for Overture — a major piece of work.

Contributions 1, 2 and 9 also represent a continuous flow of work, with 2 building upon 1 — essentially, Contribution 1 identifies various issues which Contribution 2 addresses — and Contribution 9 generalises the solutions reported as Contribution 2.

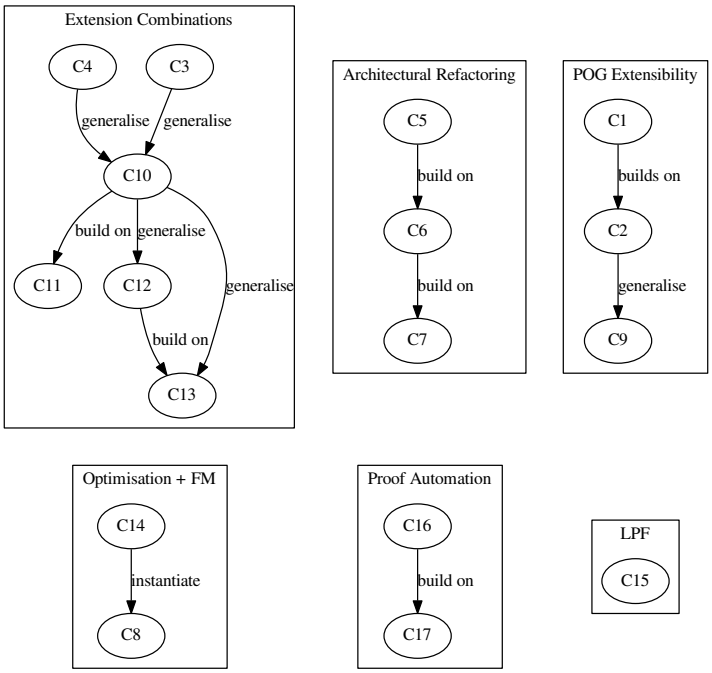


Figure 6.1: Contributions visualised as graphs.

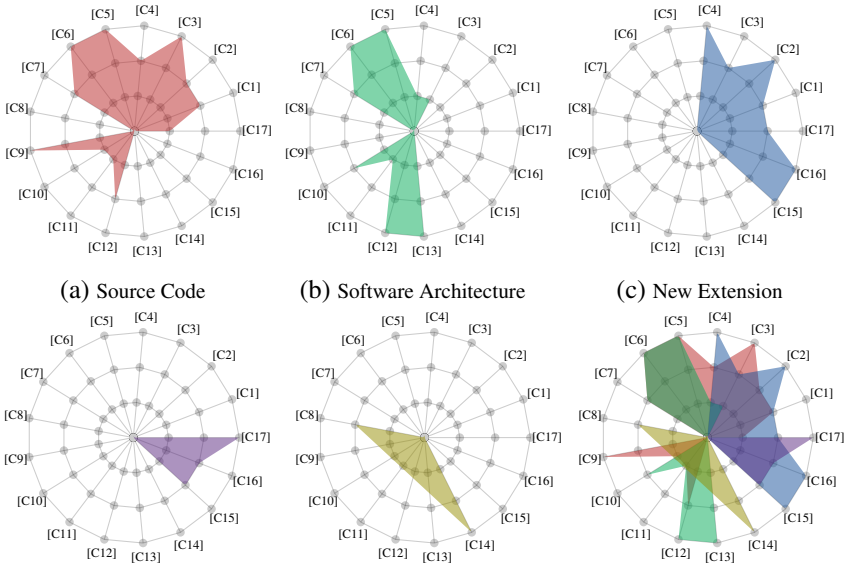
Finally, Contributions 3, 4 and 10 to 13 represent the single largest chunk of work in the PhD. Contributions 3 and 4 combined to address specific extensibility issues in the Overture code base. Together those combinations led to a general solution for combining extensions, which constitutes Contribution 10. Specific issues that must be considered when attempting this solution were formulated as Contribution 11. Finally, the solution eventually led to a set of principles for reuse in FM tools (Contribution 12) and an architectural proposal for platform-based IDEs (Contribution 13). Therefore, we began with an Overture IDE with extensibility issues and through addressing those issues and reflecting on those solutions, we eventually reached a general architectural solution for extensible IDEs which not only underpins the development of Overture but can be used by other FM tool builders.

6.2 Assessing Contributions

This section assesses the contributions of the thesis in several ways. First, in Figure 6.2, contributions are related individually with all evaluation criteria. Conversely, Figure 6.3, shows how all contributions together are distributed across the evaluation criteria. The section is concluded by a qualitative assessment of the various contributions.

Contributions are related to the evaluation criteria of Section 1.5 in Figure 6.2. The figure presents an informal ranking that indicates how each individual contribution fulfils the criteria. The scale indicates to what extent each criterion is fulfilled — the closer to the edge of the circle, the greater the contribution. The process of evaluating contributions was done by the author of the thesis so it is naturally subjective. It is based primarily on the author's insights although, when possible external information such as subsequent uses or feedback from publication reviewers has been taken into account.

Figures 6.2a to 6.2e show the assessments for each individual evaluation criterion. Figure 6.2f overlays all the previous figures and thus shows an overall assessment for all criteria.



(a) Source Code (b) Software Architecture (c) New Extension

(d) Foundation (e) Modelling (f) Combined Criteria

Figure 6.2: Relation between individual contributions and evaluation criteria.

Figure 6.3 provides an additional relation between contributions and evaluation criteria, in this case showing how the various contributions combined are distributed across the evaluation criteria.

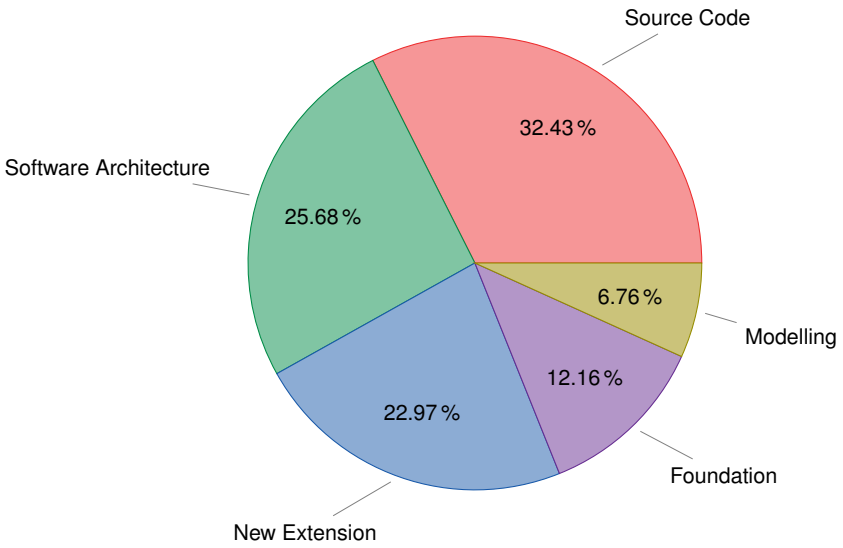


Figure 6.3: Combined contributions distributed across evaluation criteria.

We now go through the contribution for the various evaluation criteria and provide qualitative assessments for each criterion. The two most successfully fulfilled criteria are *SA* and *Source Code*. These were the two primary objectives of the PhD, particularly considering the applied nature of the PhD and the focus on tool development.

In terms of *Source Code*, some of the most important contributions are Contribution 2 and its generalisation (Contribution 9). In fact, it was this contribution that provided the impetus for the idea of generalising certain contributions. In terms of impact, perhaps the most significant contribution was number 3. It supported several pieces of additional work by third parties around POs. For example, work has been carried out to translate VDM-SL models into Alloy [51] in order to discharge certain POs, which is made possible by the new internal PO format [73].

SA was perhaps the most successfully fulfilled evaluation criterion. The main branch of work that stands out here is the one represented by Contributions 5 to 7. It explored various dimensions of extensibility in SA including

analysis, design, migration and comparison. The work has also had impact as it was fundamental to several subsequent developments of Overture including the construction of the Symphony IDE.

In terms of *New Extensions*, the thesis provided three kinds of extensions: data-based extensions, where an existing feature was extended to a new data structure (Contribution 2), a functional extension where an alternate version of existing functionality was provided (Contribution 15) and extensions where new functionality was provided (Contributions 4 and 17).

Modelling and *Foundation* were fulfilled by fewer contributions. However, both criteria were assessing secondary and supporting research goals. In the case of *Foundation*, the work that was carried out, while a small portion of the PhD, fitted quite well with the rest and satisfied the stated goal of selective development of foundational work. For Contribution 15, we have an ideal example of a well-founded extension — the theory work was relatively straightforward and the new tool extension supporting it was also developed rapidly, thus taking advantage of the increased implementation speed afforded by extensibility. Contributions 16 and 17 represented significantly larger foundational work, although extensibility was also leveraged when implementing the translation rules once they had been developed.

As for *Modelling*, it was the most isolated piece of work in the PhD. We were successful in terms of applying extensibility principles to formal modelling and demonstrating it through a case study application. However, that work never fed or contributed to any other branch of the PhD. Perhaps it would have been better to design a formal modelling goal that would have ensured more interaction with the other areas of the PhD.

To summarise, we feel as though three of the evaluation criteria were quite successfully fulfilled, with *SA* being perhaps the strongest one. On the other hand, the fulfilment of the *Modelling* criterion left something to be desired, in terms of feeding into other areas of the PhD and the topic of extensibility in general.

6.3 Future Work

The possibilities for future work following up on the work presented here fall in two categories:

- technical work that directly builds on some of the existing contributions;
- new challenges or directions uncovered during the PhD studies.

In terms of *technical work* to progress, there is a chance to have a great impact by evolving the Isabelle integration with VDM. For starters, the embedding itself can be taken further by moving to a dedicated VDM embedding that includes support for the remaining VDM dialects. This would be a large piece of work, including theoretical foundations as there are many issues with the semantics of VDM++ and VDM-RT that need clarification. From an extensibility perspective, it would allow the exploration of extensibility in the context of theorem proving by attempting to use extensibility techniques in the development of embeddings to support the three VDM dialects. Work on a new embedding is already under way but it is not yet sufficiently mature to form part of this thesis.

Further work can also be done on the Overture side, in terms of improving the integration. Currently, Overture generates a set of theories and proof goals for Isabelle. But the proof results are not communicated back to Overture. One can envisage a tool-supported workflow where POs are discharged directly through the Overture UI, with all the Isabelle interaction hidden from the user in the background. On the theoretical side, there is an interesting challenge in terms of translating results from Isabelle back to the syntax of VDM, which would be useful to present counter-examples or to provide meaningful proof states for failed discharge attempts.

Another branch of work that can serve as basis for future research is the platform-based architecture for IDEs. The original publication suggested a few avenues for future work and of those, we would highlight the development of standard mechanisms for integrating external tools. For additional work, the UI side of the platform is under-developed when compared to the core. A possible evolution would be decoupling the UI from the Eclipse framework as it brings significant maintenance costs when the Eclipse framework evolves. Migrating the platform to another UI framework would also be a good way to assess the portability of the core. An alternative to another UI framework would be migrating the platform to the web and providing browser-based tooling instead. One advantage of this approach would be to make it easier to enable and deploy new extensions and make them available to users faster. It would also be easier to track usage of extensions to judge whether they should be kept or removed.

In terms of *future challenges*, we would like to focus on the combination of extensibility and formal models. In this PhD we have investigated how to take extensibility principles from software development and apply them to the construction of formal models. The next challenge to address is how these models feed back into software. We propose taking the extensibility

principles, applying them to models and then take those models into realised systems in order to investigate how to preserve extensibility. In other words, how to answer the question of how extensible models can lead to extensible software. If we opt for solutions based on code generation, the extensibility of generated code is another interesting avenue for research. Finally, the production of these extensible models leading to extensible systems could be formulated as a series of methodological guidelines or as part of a system development workflow.

More generically, there are many other experimental extensions that come to mind, some of which could be developed as part of further extensibility analyses. Examples include integration of Rely/Guarantee [54, 55, 52] techniques in Overture or a plugin for computing metrics for VDM models (this would require foundational work). But what we would like to end on is the idea that development of these extensions, and others we cannot foresee, is greatly supported by the various extensibility improvements to Overture that have been carried out over the course of this PhD.

Part II

Publications

7

The COMPASS Proof Obligation Generator: A Test Case of Overture Extensibility

The paper presented in this chapter has been accepted as a peer-reviewed workshop paper.

[P24] Luís Diogo Couto and Richard Payne. *The COMPASS Proof Obligation Generator: A Test Case of Overture Extensibility*. 11th Overture Workshop, August 2013.

The COMPASS Proof Obligation Generator: A test case of Overture Extensibility

Luis Diogo Couto[1] and Richard Payne[2]

[1] Aarhus University
lcouto@iha.dk
[2] Newcastle University
richard.payne@ncl.ac.uk

Abstract. Proof obligation generation is used as a compliment to type checking for the verification of consistency of VDM specifications. The Overture toolset includes a Proof Obligation Generator (POG). Overture is designed to be a highly extensible platform. CML, a new language designed for modelling systems of systems is based in part on VDM. The CML tools are themselves built on Overture. We evaluate the extensibility and potential for reuse of Overture by reporting our experiences in developing a POG for CML as an extension of the Overture POG. During this process, we alter the existing Overture POG visitors in order to make them more extensible and reusable.

1 Introduction

Type checking is statically undecidable in VDM [1]. VDM specifications can be generally divided into 3 sets: on the one end we have correct or "good" specifications; on the other end we have incorrect or "bad" specifications; and between these two ends, we have undecidable specifications.

The VDM type checker can handle the first 2 sets on its own (it accepts correct specifications and rejects incorrect ones). Specifications from these 2 sets will not have any associated proof obligations. But for the third set, the undecidable specifications, we need the assistance of a Proof Obligation Generator (POG).

The POG therefore picks up where the type checker leaves off and generates a series of proof obligations related to the elements that make the specification undecidable. Discharging these obligations helps prove the internal consistency and correctness of the specification.

The Overture platform, an open source tool for VDM, has a POG for VDM as part of its toolset, although there is no support yet for discharging proof obligations [9].

The COMPASS project seeks to develop tools and practices for modelling Systems of Systems (SoS) [4], including the COMPASS Modelling Language (CML) and a supporting toolset built on top of Overture [3]. Part of the COMPASS toolset will include a POG for CML, developed as an extension of the Overture one.

In this paper, we consider the extensibility of the Overture POG and discuss the issues in the reuse of the Overture toolset. In Section 2, we provide a brief introduction to CML, Section 3 describes the CML POG, we discuss the extensibility of the Overture POG and issues for future development effort in Section 4. Counclusions are drawn in Section 5.

2 The COMPASS Modelling Language

The CML is the first language to be designed specifically for the modelling and analysis of SoS [10]. It is based on the languages VDM [6], CSP [7] and Circus [11]. A CML model comprises a collection of types, functions, channels and processes. Each process encapsulates a state and operations written in VDM and interacts with the environment via synchronous communications in CSP. A semantic model for CML using UTP [8] is in development as part of the COMPASS project [2].

As CML and the COMPASS tool platform are based upon VDM and Overture, the Abstract Syntax Tree (AST) generated by the COMPASS parser is extended from the Overture AST. The ASTCreator tool, a part of the Overture platform, is used to automatically generate ASTs for VDM dialects, which is extended to support CML. This reuse allows us to directly reuse elements of the Overture platform, including the type checker, interpreter and POG.

Being partly based upon VDM, the CML POG will generate those VDM Proof Obligation (PO)s generated by the Overture platform. As such, we aim to reuse and extend the Overture POG.

3 The COMPASS Proof Obligation Generator

3.1 Structure

The COMPASS POG is built on two sets of classes: visitors [5] and proof obligations. This structure was inherited from the existing Overture POG.

The ProofObligation class and its various subclasses are responsible for holding proof obligation data. Each different type of proof obligation has its own subclass (for example NonZeroObligation is a class for representing proof obligations that an expression must evaluate to something other than zero). There are also a related set of classes for storing data related to the proof obligation context. For example, the POFunctionContextDefintion stores the various syntactic elements of a function required for function-related proof obligations.

The other set of classes are the visitors. They are responsible for traversing the CML AST and generating the various proof obligations. Whereas the proof obligation classes can be thought of as holding the data, the visitor classes implement the behavior of the POG. Unlike the proof obligation classes, whose type hierarchy is dictated by the proof obligations we want to generate, the visitor hierarchy reflects the CML ast. We have 4 kinds of visitors, each responsible for a subset of AST nodes (POGProcessVisitor is responsible for traversing processes, etc.). At runtime we need an instance of each visitor type and we also need to move between them and so every visitor has a pointer to its parent visitor.

3.2 Behavior

The COMPASS POG is built as a series of visitors. The overall behaviour is relatively simple. The main visitor (ProofObligationGenerator) initializes the various

sub-visitors and applies them to the AST. Whenever one of the sub-visitors encounters a node it cannot handle (e.g. the process visitor encounters an expression) it will pass the node up to the main Visitor who will then re-apply the correct sub-visitor.

This behavior is shown in the SysML sequence diagram in Figure 1.

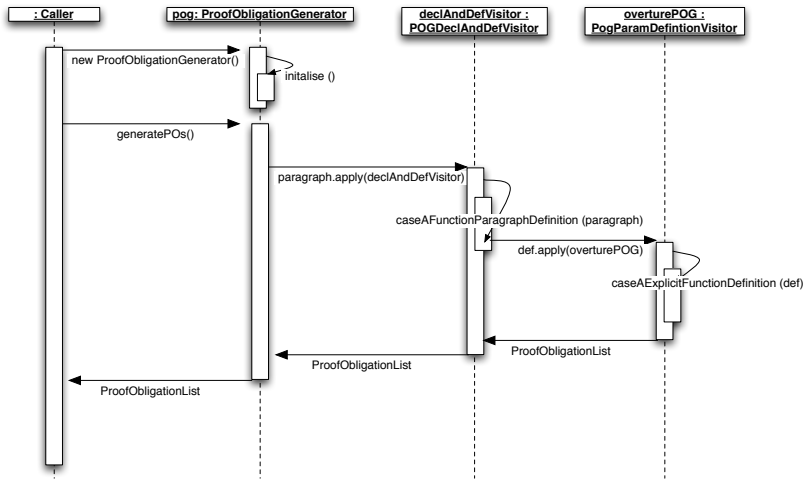


Fig. 1: Sequence diagram representing COMPASS POG visit

3.3 Reuse

Our main goal for reuse was to be able to directly utilise the Overture POG to generate all the Proof Obligations from VDM constructs directly. Because of this, the overall structure and behavior of the COMPASS POG are heavily influenced by the Overture POG. The entire visitor style of passing AST nodes between the various is lifted from Overture.

However, rather then simply passing a node up to their root, CML visitors must pass the node up to the Overture visitors. For example, the CML expression visitor must handle new CML expressions and then call the Overture expression visitor to handle the VDM expressions. There are two main issues with this approach.

The first issue is that there is no way to immediately identify a node as being from Overture or CML without using `instanceof` checks in a manually implemented decision method. One must use the default cases of visitors to work around this limitation. We can set up a for default case for CML nodes and another default case for all nodes (including the extended ones) . This of course limits our ability to handle default cases.

It would be good if we had 3 default cases available: extended, non-extended and all nodes. This limitation seems to be in the AST itself and not the Overture POG

The second issue we encountered was with the Overture POG visitors. When we pass a node to the Overture visitors, we are no longer able to control what happens. The AST goes under control of the Overture visitors and that is never relinquished. Their default cases are to call the root Overture visitor and its default case is to simply return `null`. The issue of course comes when you have both VDM and non-VDM nodes in a branch of the AST, which happens quite often. When the AST is passed to Overture, its visitors will not know how to handle the VDM nodes. Of course, this means that at best our POG will be unable to produce the proof obligations for these hybrid trees and at worst, it will die (this will be the most frequent outcome).

To handle this second issue, we had to alter the existing Overture POG to enable its visitors to release the AST back to COMPASS. We introduce the notion of a main visitor. The main visitor is the one that is called on most (any non-parent) calls of the `apply()` method. Previously these calls were of form `node.apply(this)`. Now they become `node.apply(mainVisitor)`. This main visitor becomes a parameter in the Overture visitors. To preserve compatibility with existing Overture plugins, we rename the altered visitors to ParamVisitor and create new subclasses of these parametrized visitors with the old visitor names. In these cases, the visitor receives a reference to itself as the main visitor parameter.

When the Overture visitors are used by COMPASS the COMPASS visitor is set as the main visitor parameter. This means that every `apply()` method will return the AST to COMPASS. Now, all decisions belong to COMPASS. The Overture visitor will simply unpack the node, generate any relevant proof obligations and apply the COMPASS visitor to any sub-nodes. In effect, the Overture visitor is called for the use of only one method at a time.

4 Discussion

The current version of the COMPASS POG generates the majority of VDM POs as generated in Overture. This is due to the reuse of the Overture Expression visitor, the ability to reuse the majority of the Overture declaration and definition visitors (apart from the Operation syntactic elements which differ in CML), and the reuse of ProofObligation and POContext classes. As mentioned above, this to reuse these elements required some effort. Whilst this reuse has been useful and reduced the amount of effort to generate VDM-related POs, there are two main dimensions in which the reuse is insufficient for a CML POG.

- We shall need to address the CSP syntactic elements of CML and the resultant POs not covered in VDM. The CML visitors currently have placeholders for most of the process and action CML language elements, influenced by the Overture visitors. Further language development effort is required to define the POs resulting from CML, not present in VDM.
- The current format of storing POs is adequate when their use is limited to printing to the screen. However, as the POs will be used by other analysis tools, storing POs

as strings is not appropriate. This is due to the fact that storing POs in this way allows only one form of PO representation, limiting the use the toolset can make from the generated POs. To address this issue, the PO representation format will be reimplemented in the form of its own AST, which will be an extended subset of the existing CML expression nodes. This new PO format will be composed of one PO expression (the assertion to be proved) and a set of PO expressions holding the context information. Work on this new format is underway, beginning with its implementation in Overture.

When tackling these issues, we should consider how much effort should be made in making changes in the Overture POG (which can be reused in the COMPASS tool platform) and how much is COMPASS-specific. Effort placed in the former case may slow down development of the COMPASS POG, however this will aid in future Overture reuse. However, we must be careful not to add complexity to Overture where it is not necessary for VDM. Our initial thought would be to make COMPASS-specific POG changes for the first issue above, and make changes in the Overture for the second issue.

The COMPASS toolset proposes the incorporation of several analysis tools as plugins to reason over properties of a CML model. The POG, therefore, is a clear source of such properties and thus the proof obligations generated must be made available to the analysis plugins and the analysis results must be related to the PO in the COMPASS toolset. Different plugins will need the proof obligation in different syntaxes and the new AST format will help with that. We can simply develop new visitors that traverse the PO AST and generate the relevant syntax. A clear example of this need for extensibility is the use of the proof assistant Isabelle[3]. To be of use, the proof obligations must be made available in Isabelle compatible syntax, refer to the relevant part of the CML model, and be associated with the result of any proof generated in Isabelle. The connection between proof obligations and their respective Isabelle proofs, particularly across multiple versions of a model is a problem currently under study.

5 Conclusion

We have presented a POG for CML, developed as an extension of the Overture POG. In developing, we have gained insight into the current extensibility and potential for reuse of Overture.

Overall, reuse is definitely possible and is quite powerful. However, it is not a particularly easy task. There were several issues with extending the Overture POG and were it not for existing familiarity with Overture, the task would have been extremely complicated.

We also benefited greatly from being able to alter existing Overture code. The visitor context swaps (particularly, return going from Overture back to COMPASS) were very challenging and without changes to the existing code, it would have been impossible to implement the COMPASS POG with proper reuse. It is clear to us that more work must be done to improve the extensibility of Overture.

[3] http://isabelle.in.tum.de

It is also worth mentioning that the development of these extended versions of Overture plugins can be quite challenging. It will be interesting to see how the combination of all Overture and COMPASS plugins turns out.

Acknowledgements

The authors wish to thank Peter Gorm Larsen and Joey Coleman for reviews on the manuscript. Nick Battle implemented the original Overture POG and is currently working on the AST version. His work is greatly appreciated. Simon Foster is developing the Isabelle plugin for COMPASS and his ideas on the format for proof obligations have been a great help.

The work presented here is supported by the EU Framework 7 Integrated Project "Comprehensive Modelling for Advanced Systems of Systems" (COMPASS, Grant Agreement 287829). For more information see http://www.compass-research.eu.

References

1. Hans Bruun, Flemming Damm, and Bo Stig Hansen. An Approach to the Static Semantics of VDM-SL. In *VDM '91: Formal Software Development Methods*, pages 220–253. VDM Europe, Springer-Verlag, October 1991.
2. Jeremy Bryans, Andy Galloway, and Jim Woodcock. CML definition 1. Technical report, COMPASS Deliverable, D23.2, September 2012.
3. Joey W. Coleman, Anders Kaels Malmos, Peter Gorm Larsen, Jan Peleska, Ralph Hains, Zoe Andrews, Richard Payne, Simon Foster, Alvaro Miyazawa, Cristiano Bertolini, and André Didier. COMPASS Tool Vision for a System of Systems Collaborative Development Environment. In *Proceedings of the 7th International Conference on System of System Engineering, IEEE SoSE 2012*, volume 6 of *IEEE Systems Journal*, pages 451–456, July 2012.
4. Comprehensive Modelling for Advanced Systems of Systems, 2011. http://www.compass-research.eu/.
5. R.Johnson E.Gamma, R.Helm and J.Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.
6. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.
7. Tony Hoare. *Communication Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.
8. Tony Hoare and Hi Jifeng. *Unifying Theories of Programming*. Prentice Hall, April 1998.
9. Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1), January 2010.
10. J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: a Formal Modelling Language for Systems of Systems. In *Proceedings of the 7th International Conference on System of System Engineering*, volume 6 of *IEEE Systems Journal*. IEEE, July 2012.
11. Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 184–203, London, UK, UK, 2002. Springer-Verlag.

8

Towards Verification of Constituent Systems through Automated Proof

The paper in this chapter has been accepted as a peer-reviewed workshop paper.

[P22] Luís Diogo Couto, Simon Foster and Richard Payne. *Towards Verification of Constituent Systems through Automated Proof*. Workshop on Engineering Dependable Systems of Systems (EDSoS), May 2014.

Towards Verification of Constituent Systems through Automated Proof

Luís Diogo Couto
Aarhus University, Denmark
ldc@eng.au.dk

Simon Foster
University of York, United Kingdom
simon.foster@york.ac.uk

Richard Payne
Newcastle University, United Kingdom
richard.payne@ncl.ac.uk

***Abstract*—This paper explores verification of constituent systems within the context of the *Symphony* tool platform for Systems of Systems (SoS). Our SoS modelling language, CML, supports various contractual specification elements, such as state invariants and operation preconditions, which can be used to specify contractual obligations on the constituent systems of a SoS. To support verification of these obligations we have developed a proof obligation generator and theorem prover plugin for Symphony. The latter uses the *Isabelle/HOL* theorem prover to automatically discharge the proof obligations arising from a CML model. Our hope is that the resulting proofs can then be used to formally verify the conformance of each constituent system, which is turn would result in a dependable SoS.**

I. Introduction

A System of Systems (SoS) [1] is a collection of semantically heterogeneous, independent, and distributed constituent systems (CSs) which are co-ordinated to achieve an overall goal. Independence means that no CS can exert control on another CS, only influence its behaviour by offering potential opportunities should synergy be reached. Since CSs are dynamic and heterogeneous, often changing their capabilities and services, such synergy is achieved by negotiation of *contracts* between a set of CSs, which impose binding conditions on the behaviour of each CS. Since failure of such an agreement will result in degradation of the SoS, it is important that each CS has some measure of certainty in its ability to fulfil its requirements, which in turn will lead to a dependable SoS.

System of Systems Engineering (SoSE) therefore requires languages with which we can accurately model CSs to predict their behaviour, and tools which enable their verification. Such languages should have a sound theoretical background to ensure that they can be assigned a consistent behaviour, and the ability to handle the composition of heterogeneous constituents. To this end the COMPASS Modelling Language (CML) [2] has been developed, a formal modelling language for SoSs. CML reproduces the style of the VDM-SL [3] formal specification language, whilst integrating CSP [4] process modelling constructs from the *Circus* [5] language. CML has a formal semantics based in Hoare and He's Unifying Theories of Programming (UTP) [6], in its denotational, operational, and axiomatic flavours. Along with the associated *Symphony*[1] tool platform, CML allows SoSs and CSs to be formally modelled, tested, and verified in a controlled environment.

This paper focuses on two closely related components of Symphony, the Theorem Prover Plugin (TPP) and Proof Obligation Generator (POG). A theorem prover can be applied to *verify* a software system, that is mathematically demonstrate that required properties are met through mechanically verified proof. In the case of CSs, we need to verify that the internal functionality and pattern of interaction is guaranteed to fulfil the contract. In Symphony this verification can be facilitated through the POG which generates proof goals upon which the correctness of the CS model depends. CML has a number of facilities for specifying contractual obligations, such as type invariants, pre- and post-conditions for functions and operations, and system state invariants. These can then variously be used to specify contractual obligations for a CS model, and the application of the POG in concert with the TPP can be used to verify that the system satisfies those obligations. Our thesis is, therefore, that these technologies provide a way forward for mechanically verifying that a CS model fulfils its contractual obligations to the wider SoS.

In the remainder we outline our contributions. Section II gives more background to our baseline technology. Section III discusses related work. Section IV discusses the combined POG and TPP framework, and how it can be used to verify a CS model. Section V demonstrates an example CS, and how we envisage verifying it for a wider SoS. Finally in Section VI we conclude and outline future work.

II. Background

CML is a language for modelling constituent systems and their composition in an SoS. Systems are modelled using CML *processes*, which are stateful reactive entities that can be executed concurrently, and exchange messages over *channels* in the style of the CSP process calculus [4]. A CML model consists of a collection of user defined *types*, *functions*, *channels*, and *processes*. A process, in turn, consists of private *state variables*, *operations* that act on these variables, and *actions* that specify reactive behaviour using operators from CSP. CML processes can be parallel composed to represent concurrent execution, enabling description of a complete SoS.

CML has a formal mathematical foundation [7] based in the UTP semantic framework [6], which allows processes to be given a precise semantics. UTP allows us to tackle semantic heterogeneity in SoSE by decomposing a modelling language semantics into its theoretical building blocks, such as state, concurrency, discrete time, and mobility, which can then be formalised as *"UTP theories"*. UTP theories then act as components with which we can construct semantic models for languages and provide links between similar languages based on common theoretical factors.

[1] Symphony can be downloaded from http://symphonytool.org/

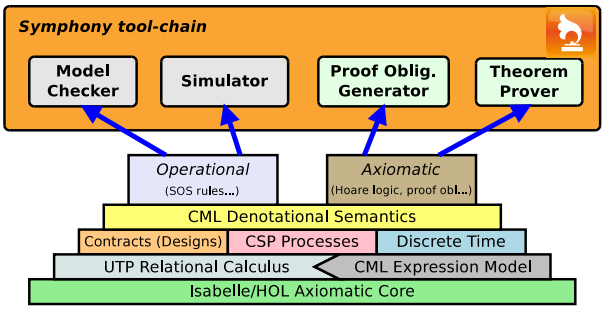


Fig. 1: Semantically supported Symphony tool platform

Development of CML models is aided through the associated *Symphony* tool platform. Symphony is an Eclipse-based development environment that provides a parser, syntax highlighting, a type checker, simulator, model checker, and a variety of other tools for variously constructing and verifying CML models. Though consisting of independently developed components, the different tools share a common semantic foundation given by the UTP denotational model. This "semantic stack" is shown in Figure 1, with the Symphony platform and associated components positioned on top. Within the UTP relational calculus several computational paradigms have been formalised as UTP theories (Contracts, Processes, etc.), and these have been in turn composed to produce the CML denotational semantics. Finally, reference semantics have been produced that underlie the various tools, including the operational semantics, which underlies the simulator and model checker, and various axiomatic semantics, such as a Hoare calculus [8]. Since this formal link exists from each tool down into the unified semantic basis we can have a degree of certainty that the various evidences produced can be consistently composed to verify a CS model.

To support such verifications we have created a theorem prover plugin, based on the *Isabelle/HOL* [9] interactive theorem prover. Isabelle/HOL is ideal for this kind of verification since proofs can be independently checked with respect to a secure axiomatic core; a facet of the *"LCF architecture"*. Our theorem prover is based in a mechanised semantic framework for UTP called *Isabelle/UTP* [10] that provides a strong theoretical grounding for CML, ensuring its consistency. We have mechanised a partial semantic model for CML in *Isabelle/UTP*, a collection of associated proof tactics, and a visitor that translates the CML Abstract Syntax Tree (AST) into Isabelle definitions that can then be used to support proof. Our current approach to proof in CML is to, where possible, convert CML to equivalent HOL formulae, and perform the proof using Isabelle's variety of existing tactics and laws, effectively transferring results from HOL to UTP. In line with the UTP framework, the theorem prover is fully *extensible*: we can add support for additional programming concepts, and associated tactics as required in the future.

Alongside the theorem prover, Symphony contains a POG that generates, for a given CML document, a collection of proof goals that must be satisfied to prove certain high level properties of the model, such as internal consistency, contractual correctness of operations, and termination. The TPP can then be used to attempt discharge of these proof obligations, resulting in concrete proof objects for the properties. We are currently working towards a formal axiomatic semantics for these proof obligations based on the current implementation, which will allow the integration of these proof objects with other evidences in the tool-chain.

III. Related Work

The Symphony tool platform is an extension of the open source *Overture* IDE [11] for VDM based modelling. The Symphony POG is an adaptation of the POG for Overture [12] to also handle CML proof obligations. Previous efforts to generate and discharge Proof Obligations (POs) for VDM include [13] and [14], which connect VDMTools and Overture POs respectively to the HOL4 theorem prover [15]. These attempts were limited to the functional subset of VDM. We use a similar mapping for CML types and expressions, whilst adding support for CML's imperative and concurrent constructs.

The area of theorem proving tools includes a number of options including Isabelle/HOL [9] (which we use); PVS[2] combining a specification language with a theorem prover; Coq [16], a proof assistant based on intuitionistic logic; specialised verification systems such as *Spec*♯, which is based on the Boogie verification language [17] and supported by the *Z3* SMT solver [18]; and the *Rodin*[3] tool for Event-B which includes an automated theorem prover.

We choose Isabelle for several reasons. It is based on *Higher Order Logic* which is ideal for embedding a language like CML. The LCF architecture ensures proofs are correct with respect to a secure logical core. It has a large library of mathematical structures related to program verification, such as relational calculus and lattice theory. It integrates powerful proof facilities, such as the auto tactic for automated deduction, integration of first-order automated theorem provers (like *Z3*) in the sledgehammer tool [19], and counterexample generators like nitpick. We can directly harness many of these proof facilities by our transfer based proof tactics. Finally, Isabelle has been integrated into Eclipse in the form of *Isabelle/Eclipse*[4], an IDE which we reuse in Symphony.

IV. PO Generation and Discharge in Symphony

The Symphony POG is an extension of the Overture POG for the Vienna Development Method (VDM) [12], and therefore many proof goals generated are derived from VDM. However, the POG has been developed with an extensible visitor [20] based architecture that will enable the addition of further goals as they are researched. The current proof goals fall broadly into the following categories: safe usage of partial operators; safe usage of functions with pre-conditions; type compatibility due to union types, type invariants and subtypes; and satisfiability of implicitly defined functions and operations.

To illustrate the use of POs in Symphony, we present a simple example based on a well known partial operator case: division by zero. Consider the following CML function:

```
division : int * int → real
division (x,y) == x / y
```

[2] http://pvs.cdl.sri.com
[3] http://event-b.org
[4] http://andriusvelykis.github.io/isabelle-eclipse/

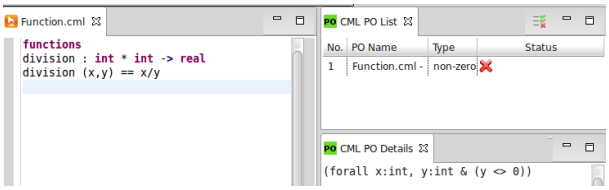


Fig. 2: Failed PO discharge.

For this function, the POG generates an obligation stating that, for all inputs to the function, the value of the divisor (y) will not be 0, thus ensuring the function executes successfully. This is represented in the logical formula below:

```
PO1: forall x:int, y:int & (y <> 0)
```

This states that for all variables x and y of type int, y is not equal to 0. This is not satisfiable, and so an attempt to discharge this PO will fail as shown in Figure 2. The function must be enriched with a pre-condition, using the **pre** keyword, in order for the discharge to be possible:

```
division : int * int → real
division (x,y) == x / y
pre y <> 0
```

The additional information offered by the pre-condition alters the PO and now the theorem prover plug-in is able to discharge the revised PO as shown in Figure 3. It is of course not possible to prove that an arbitrary integer is different from zero, but it is trivial to prove that a non-zero integer is different from zero.

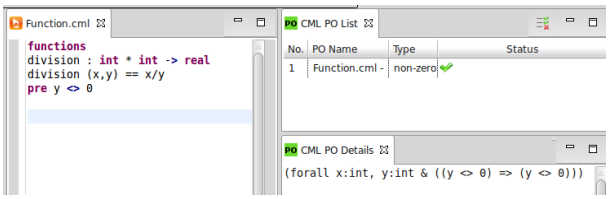


Fig. 3: Successful PO discharge.

The aforementioned example was trivial but, in general, it is quite important to ensure that, when adding a pre-condition, said pre-condition is sufficient to allow the discharge of any POs generated for the function.

The addition of the pre-condition has another effect. One must now ensure that, whenever the function is called its pre-condition is respected. Therefore, a new kind of PO is generated. Consider the following function and PO:

```
divby2: int → real
divby2 (x) == division(x,2)
```

```
PO2: forall x : int & pre_division(x,2)
```

Since divby2 calls division, a PO is generated to ensure that the pre-condition of division, given by pre_division, is satisfied. This kind of obligation, called pre-condition obligation is generated at all points in the model where the function is called.

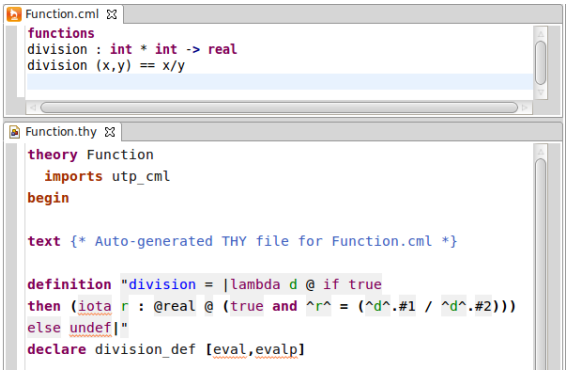


Fig. 4: Auto-generated theory files.

While the example shown was very simple, pre-conditions (and invariants) can be as complex as necessary. They are expressed in the functional subset CML and thus have the full expressive power of CML's first-order logic. In addition to helping ensure consistency of the model, pre-conditions and invariants are also used to specify additional properties

In fact, the methodology we propose is based precisely on specifying desired properties and requirements of a model through pre- and post-conditions as well as invariants. The POs, once generated and discharged, stand as proof that the model respects the specified properties.

Discharging POs is the task of the TPP. At its core, the TPP consists of a mechanised semantic model for CML within *Isabelle/UTP*. It is essentially a deep embedding of CML, in that we give an explicit semantics to each of the operators of CML processes within Isabelle.

The TPP will process a CML model and its associated POs and automatically generate Isabelle theory files for them (see Figure 4). These theory files can then be submitted to Isabelle for discharging through various automated proof tactics such as auto and sledgehammer, or the cml_tac tactic that maps a CML formula onto a HOL formula.

Because the TPP connects to the *Isabelle/Eclipse* plug-in, the full functionality of that plug-in and, by extension, Isabelle is available to the user . This includes the ability to write and discharge model-specific conjectures directly in the Isabelle encoding of the model. However, to perform this kind of work requires significant knowledge of Isabelle and its syntax.

Therefore, the POG will be the primary source of goals to discharge. Furthermore, the TPP offers a fully automated mode of interaction with Isabelle where users simply choose which PO to discharge and all inner workings (such as tactic selection and result collection) are hidden from them.

We envisage two main functionalities for the plug-ins *Quick check* and *Proof Session*. *Quick check* will be a fully automated process, simply presenting a list of Proof Obligations (POs) (including their predicates) in CML and linked to the relevant model elements. The process of generating POs is quick, therefore this may be performed frequently during initial model development, to gain useful feedback about the model.

The *Proof Session* will be the main functionality of the plug-ins. It creates a snapshot of the model (a timestamped,

read-only copy of the model's CML sources), generates POs and translates the model and POs into Isabelle theories. The POs are displayed in a similar manner to the quick check version but can now be submitted to the TPP for discharge. At the moment only cml_tac is available, though we hope to enable automated use of additional tactics when attempting to discharge POs. Regardless, the output of a proof attempt will be captured from Isabelle and displayed to the user. Also, the results of a proof session will be stored along with the model snapshot, thus verifying the model's correctness.

These two functionalities combine to form the following work-flow: as a user works on a model, he can quick check for POs as a way to gain early insights into the of the model. Each PO can be seen of as a possible inconsistency and merely by manual inspection they can guide the user in terms of adding necessary pre-conditions or guards to the model.

Once a set of changes has been completed, the user may use the proof session functionality to verify the model's correctness. Each set of POs and their associated proofs are only valid for the particular version of the model they were generated from so it makes little sense to attempt manual proofs on a volatile model. Regardless of when it is attempted, the proof session for the average user will be fully automated. The user simply initiates a proof session and selects POs for discharging either manually or in batch. Typically some POs will be successfully discharged whereas others will fail to discharge. These should indicate a problem with the model and action must be taken by the user (for example, by adding a guard or correcting program logic) to alter the model in a way that allows the PO to be discharged. Then, the set of completed PO goals can be used as a formal proof of the constituent system's correctness.

For advanced users who are comfortable interacting with *Isabelle/Eclipse* directly, the full theorem proving perspective gives them direct access to the tool so that manual proofs may be attempted. Users can also specify and discharge additional model-specific conjectures.

V. Verification of Example Constituent System

We illustrate the use of the Symphony tool platform POG and TPP with a simple example CS from a Railway Signal System of Systems (SoS). The SoS in question aims to ensure the safe and correct movement of trains on a section of railway tack. Naturally such a SoS poses several dependability concerns and the integrator of the SoS requires several safety properties to hold throughout the life of each of the systems.

The *Railway Signal* SoS comprises several constituents including a *Route Rule Engine*, several *Track Actuators*, *Trains* and *Dwarf Signal* systems. In this paper, we look at one of the constituent systems – the *Dwarf Signal* system – in detail and consider the safety properties of that system.

From the perspective of the SoS integrator, there is a requirement that the procured constituent systems provide a safe service. The constituent system designer must, therefore, provide evidence of this safety. Using model-based techniques, we define a formal model of the *Dwarf Signal* – which may be used as a contract to which the the signals must conform. The *Dwarf* model used in this paper is based upon that introduced in [21], and a typical signal may be seen in Figure 5.

Fig. 5: Picture of railway signal, with lamps indicated

The *Dwarf Signal* model is defined in CML with several datatypes, functions, a single `Dwarf` process with state variables, operations and actions. The main datatype, `DwarfType` shown below, has several fields relating to the transitions which are to be made in the *Dwarf Signal*. For example, the `currentstate` field dictates the collection of lamps currently lit, and the `desiredproperstate` field represents the next state the *Dwarf Signal* should reach. The set of possible signal states that may be reached is defined by the `ProperState` datatype, which is constrained to be one of for constant values: `dark`, `stop`, `warning` and `drive` – each a set of lamps.

```
types
 LampId      = <L1> | <L2> | <L3>
 Signal      = set of LampId
 ProperState = Signal
 inv ps == ps in set {dark, stop, warning, drive}

 DwarfType :: lastproperstate    : ProperState
              turnoff            : set of LampId
              turnon             : set of LampId
              laststate          : Signal
              currentstate       : Signal
              desiredproperstate : ProperState
 inv d == NeverShowAll(d) and MaxOneLampChange(d)
      and ForbidStopToDrive(d) and DarkOnlyToStop(d)
      and DarkOnlyFromStop(d)

values
 dark: Signal    = {}
 stop: Signal    = {<L1>, <L2>}
 warning: Signal = {<L1>, <L3>}
 drive: Signal   = {<L2>, <L3>}
```

There are several safety properties to which the *Dwarf Signal* must adhere. These are defined in terms of functions referred to in the `DwarfType` invariant – including, for example, `NeverShowAll` which requires that the `currentstate` should never have all three lamps lit. The `Dwarf` process, outlined below has a single state variable: `dw` of type `DwarfType`, and four operations: `Init`, which initialises the `dw` state variable; `SetNewProperState`, allowing the next desired `properstate` to be set; and two operations for changing the lamps lit in the signal – `TurnOn` and `TurnOff`.

```
process Dwarf = begin
state
  dw : DwarfType

operations
  Init : () ==> ()
  Init() == (...)

  SetNewProperState: (ProperState) ==> ()
  SetNewProperState(st) == (...)

  TurnOn: (LampId) ==> ()
  TurnOn(l) == (...)

  TurnOff : (LampId) ==> ()
  TurnOff(l) == (...)
 ... end
```

Each operation is defined in more detail in terms of pre- and post-conditions, dictating the conditions in which the operation may be called and the guarantees it makes if those conditions are met. The `Init` operation, defined in more detail below, has a body which initialises the `dw` state variable, with a post-condition requiring that various fields of the `dw` variable are updated. The operation body – an assignment to the `dw` state variable – must respect the safety properties of the *Dwarf Signal*, in the form of the type invariant described above.

```
Init : () ==> ()
Init() ==
  dw := mk_DwarfType(stop, {}, {}, stop, stop, stop)
post dw.lastproperstate = stop and dw.turnoff = {}
       and dw.turnon = {} and dw.laststate = stop
       and dw.currentstate = stop
       and dw.desiredproperstate = stop
```

The remainder of the CML operations are defined in a similar manner – with pre- and post- conditions. In addition to these operation definitions, the CML model contains *actions* which dictate the ordering of internal events and operation calls. At present, the POG does not handle these features of CML, and thus they are omitted from this paper.

Executing the Symphony POG, we obtain several POs, which are generated by the `Init`, `SetNewProperState`, `TurnOn` and `TurnOff` operations. The POs fall into two PO types: ensuring that the postcondition holds given the body of the operation; and ensuring subtype consistency. It is the second of these which ensures that the `DwarfType` type invariant (and thus the safety properties of the *Dwarf Signal*) holds when setting a new value of the `dw` variable. The generated subtype POs (`PO1` and `PO2`) and postcondition PO (`PO3`) for the `Init` operation are shown below.

```
PO1: inv_ProperState(stop)

PO2: ((inv_DwarfType(mk_DwarfType(stop, {}, {}, stop
     , stop, stop)) and inv_ProperState(stop)) and
     inv_ProperState(stop))

PO3: (((dw.lastproperstate) = stop) and (((dw.
     turnoff) = {}) and (((dw.turnon) = {}) and (((dw
     .laststate) = stop) and (((dw.currentstate) =
     stop) and ((dw.desiredproperstate) = stop))))))
```

Using the Symphony tool platform, we generate these POs, and attempt to discharge them. In Figure 6 below, we show Symphony in the POG perspective with the POs represented in the Isabelle syntax used by the TPP. In the figure, the list of POs is given in the right hand pane, with a pane showing the PO definition in CML below. In the figure, we see that several of the POs have been discharged – these relate to the *Init* operation above – as denoted by the green ticks.

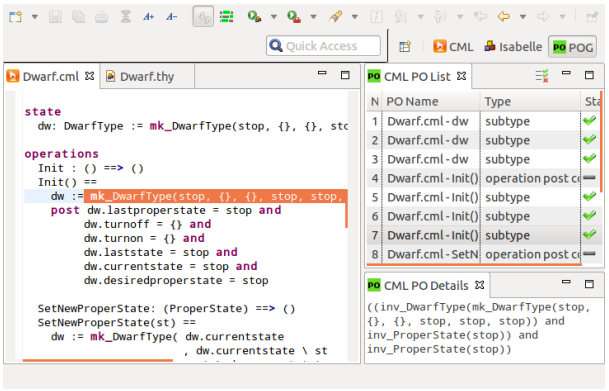


Fig. 6: Progress on POs generated for Dwarf model

By discharging all POs for the *Dwarf Signal* model, we provide a contractual model which is verified to be both internally consistent and, through encoding the safety properties which must be met by a signal, is safe with respect to the requirements placed on that contract. At present, whilst those POs shown are successfully discharged by the Symphony TPP, several are not. These relate to those POs which rely upon the value of the *Dwarf* process state variable *dw* at a given point of time. This may be either an issue with the PO expressions themselves (where the VDM-based PO expressions require further adaption to CML), or due to the early stage of development of the TPP proof tactics. We discuss these areas as future work in the next section.

VI. Conclusion and Future Work

In this paper, we have outlined two Symphony tool platform plugins which enable automated proof support for CML POs. The Symphony POG reuses and expands upon the Overture POG, also resulting in improvements in the Overture Tool. The TPP is the first attempt at tightly integrating a theorem prover into the VDM family of tools, providing a more useful tool for users wishing to discharge CML POs and general theorems. We have also shown how both plug-ins can be used in combination to formally verify the integrity of constituent systems of an SoS specified with CML. There are clearly many areas of future work, both short-term improvements to the two plugins, and also longer-term directions and scoping of the work in the fields of SoS and dependability-related issues. Below, we discuss several such directions for further work.

This paper demonstrated the verification of constituent system *models*. An interesting issue would be the verification of an actual implementation, with respect to realistic system properties. Whilst clearly not in the scope of this paper, we would consider the work of this paper in context with other system engineering activities. In particular; positioning constituent system verification with respect to the work of Holt et al [22] on SoS requirements engineering and the

specification of SysML contracts and translation to CML [23] may provide the means to more realistic property verification.

The current CML TPP focuses on VDM-style proof obligations which deal with issues such as subtyping and internal consistency. In the future we will extend this with a more comprehensive calculus, such as a Hoare logic [8] or a weakest precondition calculus, which would both expand on the existing proof obligations. This would also allow us to reason directly about CML process and state behaviour, and therefore provide fuller support for reasoning about contracts.

Another extension to the proof obligations relates to scaling our approach from the level of constituent systems to the SoS-level, thus ensuring that the verified constituent systems interact in a manner that ensures the desired behaviour of the overall SoS.

Just as pre-conditions and invariants can be used to specify the properties that the POG and TPP verify, we need a mechanism that allows these tools to reason about correctness at the SoS level. We see two distinct possibilities here: the first is to introduce a new CML construct that allows one to specify invariants over the entire SoS, thus being able to "see" inside all constituents. The second approach is to take the existing POs that verifies a system and use them to also verify the interface of a constituent. Afterwards, one must establish a means by which these verified interfaces can be combined to establish global SoS properties. Of the two approaches, the second one seems closer to the spirit of SoS engineering, and we believe CS refinement provides a way forward here.

We are also currently working on a tool for CS refinement, which combines with the theorem prover and can be used to formally demonstrate contractual satisfaction. This will reuse the POG to enumerate and discharge refinement provisos which must often be satisfied to ensure validity of a refinement step. Refinement will be principally supported by Isabelle, though we are also exploring the use of model generation tools to aid automation. For example, the Maude rewriting logic engine [24] has previously been applied to automated refinement [25], which we hope to adapt for CML. Such advances over the current technology are feasible because of our extensible approach to semantics provided by UTP.

Finally, though both plug-ins presented here are still at an early development stage, work is ongoing on various improvements. While the TPP and its associated Isabelle theories support a significant subset of CML (types, expressions, functions, and operations), work is ongoing on increasing the coverage of the plug-in. The proof tactics are also under further development in order to discharge increasingly complex goals. Moreover we wish to expose more of Isabelle's native proof facilities in the TPP, such as sledgehammer and nitpick, so as to bring their full weight to bear in discharging or refuting proof obligations. Parallel to this there is work to formalise the proof obligations in Isabelle with respect to the CML semantics. Finally, we hope to produce guidance to the user of how to interpret failure when a PO cannot be discharged.

Acknowledgements

This work is supported by EU Framework 7 Integrated Project "Comprehensive Modelling for Advanced Systems of Systems" (COMPASS, Grant Agreement 287829). For more information see http://www.compass-research.eu.

References

[1] H. Kopetz, "System-of-Systems Complexity," in *Proc. 1st Workshop on Advances in Systems of Systems*, 2013, pp. 35–39.

[2] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry, "Features of CML: a Formal Modelling Language for Systems of Systems," in *Proc. 7th Intl. Conference on Systems of Systems Engineering (SoSE)*. IEEE, July 2012.

[3] C. B. Jones, *Systematic Software Development Using VDM*. Prentice-Hall, 1990.

[4] T. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] J. C. P. Woodcock and A. L. C. Cavalcanti, "A Concurrent Language for Refinement," in *IWFM'01: 5th Irish Workshop in Formal Methods*, ser. BCS Electronic Workshops in Computing, July 2001.

[6] T. Hoare and J. He, *Unifying Theories of Programming*. Prentice Hall, 1998.

[7] J. Fitzgerald, P. G. Larsen, and J. Woodcock, "Foundations for Model-based Engineering of Systems of Systems," in *Complex Systems Design and Management*, M. A. et al., Ed. Springer, January 2014, pp. 1–19.

[8] S. Canham and J. Woodcock, "CML Definition 3 — Hoare Logic," COMPASS Deliverable, D23.4d, Tech. Rep., September 2013.

[9] T. Nipkow, M. Wenzel, and L. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[10] S. Foster, F. Zeyda, and J. Woodcock, "Isabelle/UTP: A mechanised theory engineering framework," in *5th Intl. Symposium on Unifying Theories of Programming*, 2014.

[11] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The Overture Initiative – Integrating Tools for VDM," *SIGSOFT Software Engineering Notes*, vol. 35, no. 1, Jan 2010.

[12] L. Couto and R. Payne, "The COMPASS Proof Obligation Generator: A test case of Overture Extensibility," in *Proc. 11th Overture Workshop*, 2013.

[13] S. Agerholm and K. Sunesen, "Reasoning about VDM-SL Proof Obligations in HOL," IFAD, Tech. Rep., 1999.

[14] S. Vermolen, "Automatically Discharging VDM Proof Obligations using HOL," Master's thesis, Radboud University Nijmegen, Computer Science Dept., August 2007.

[15] K. Slind and M. Norrish, "A brief overview of HOL4," in *TPHOLs*, ser. LNCS, vol. 5170. Springer, 2008, pp. 28–32.

[16] Y. Bertot and P. Castéran, *Coq'Art: the calculus of inductive constructions*. Springer, 2004.

[17] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs," in *Formal Methods for Components and Objects*. Springer, 2006.

[18] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008.

[19] J. C. Blanchette, L. Bulwahn, and T. Nipkow, "Automatic proof and disproof in Isabelle/HOL," in *FroCoS*, ser. LNCS, vol. 6989. Springer, 2011, pp. 12–27.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software.*, ser. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[21] S. Foster and J. Woodcock, "A Dwarf Signal in CML," COMPASS Whitepaper WP04, Tech. Rep., September 2013.

[22] J. Holt, "Model-based Requirements Engineering for System of Systems," in *Proc. 7th Intl. Conference on Systems of Systems Engineering (SoSE)*. IEEE, July 2012.

[23] J. Bryans, J. Fitzgerald, R. Payne, and K. Kristensen, "SysML Contracts for Systems of Systems," June 2014, to appear in IEEE SoSE 2014.

[24] M. Clavel, F. Duán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, "Maude: specification and programming in rewriting logic," *Theor. Comp. Sci.*, vol. 285, no. 2, pp. 187–243, 2002.

[25] A. Griesmayer, Z. Liu, C. Morisset, and S. Wang, "A framework for automated and certified refinement steps," *Innovations in Systems and Software Engineering*, vol. 9, no. 1, pp. 3–16, 2013.

9

Migrating to an Extensible Architecture for Abstract Syntax Trees

The paper in this chapter has been accepted as a peer-reviewed conference paper.

> [P26] Luís Diogo Couto, Peter W. V. Tran-Jørgensen, Joey W. Coleman and Kenneth Lausdahl. *Migrating to an Extensible Architecture for Abstract Syntax Trees*. 12th Working IEEE / IFIP Conference on Software Architecture, May 2015.

10

Extending the Overture code generator towards Isabelle syntax

The paper in this chapter has been accepted as a peer-reviewed workshop paper.

[P25] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. *Extending the Overture code generator towards Isabelle syntax*. 13th Overture Workshop, June 2015.

Extending the Overture code generator towards Isabelle syntax

Luís Diogo Couto and Peter W. V. Tran-Jørgensen

Department of Engineering, Aarhus University, Denmark
{ldc,pvj}@eng.au.dk

Abstract. Overture has a Code Generation Platform (CGP), designed with extensibility in mind but this extensibility has never been thoroughly tested before. In this paper, we explore the extensibility of the Overture CGP by developing code generation support targeting an Isabelle embedding of VDM. We compare our solution to an existing hand-coded VDM to Isabelle translation based on direct traversals of the VDM AST and show that using the CGP led to a decrease in code volume of 86%. We also report various extensibility improvements that have been incorporated into the CGP as part of our work.

Keywords: VDM, code generation, Isabelle, extensibility

1 Introduction

The Overture tool[1] for VDM [6] has a Code Generation Platform (CGP) that was originally developed targeting the Java language but was designed with extensibility in mind. The intent of the CGP is to make it easy to contribute new Code Generation (CG) support for new languages to Overture [12]. Currently, the CGP supports the original Java code generation as well as an experimental generation of C++. The extensibility features of the CGP have never been thoroughly tested since C++ generation is similar to Java generation.

In this paper, we further explore the extensibility of the CGP by developing experimental support for generation of Isabelle syntax, which differs from Java more significantly than C++ does. The reason for this is that Java and C++ are both imperative OO languages and Isabelle is not. The process for developing this translation is also generalised into a standard methodology for developing CGP extensions.

There are two reasons for choosing Isabelle: there is already a usable existing embedding of VDM in Isabelle that we can reuse and a corresponding translation that runs on Overture models [3]. This translation was handwritten and as such will provide a good basis of comparison to see if it is really worthwhile to use the CGP. The comparison shows that using the CGP leads to a code volume reduction of 86%.

The remainder of this paper is structured as follows: the code generation platform as well as the existing Isabelle embedding and translation are described in section 2. The steps taken by the developer to construct the new CG extension are described in section 3. Relevant details of the Isabelle translation are discussed in section 4. The results

[1] http://overturetool.org

of the work in terms of the new Isabelle translation and extensibility improvements to the CGP are reported in section 5 and evaluated in section 6. Finally, we discuss future work in section 7 and conclude in section 8.

2 Background

2.1 Isabelle Embedding

This subsection presents the target language of the translation: an Isabelle embedding of VDM. Isabelle [13] is a framework for implementing logical formalisms and the VDM embedding being targeted is one such formalism. It was originally developed for the COMPASS Modelling Language (CML) [15] in the COMPASS project [7] and is built on an Isabelle mechanisation [8] of the UTP semantics used for CML [10].

CML is a combination of VDM and CSP [9]. In particular, the types, values, expressions and functions of CML are lifted from VDM. State is similar although it is handled somewhat differently – state in CML is composed of multiple independent variables much like VDM++ rather than a single record structure. Additionally, CML does not support the **`let be st`** construct due to its non-deterministic nature. The remaining differences between CML and VDM are related to the reactive and Object Oriented (OO) features of the language. Neither are relevant for this translation.

The Isabelle embedding of CML/VDM is a deep embedding, which means that it gives an explicit semantics to each construct of CML/VDM in Isabelle. In other words, rather than translating from VDM to another formalism, each construct in VDM is defined in the embedding and then given a semantics using formalisms available in Isabelle – specifically, higher-order logic.

Furthermore, the parsing capabilities of Isabelle give significant flexibility when defining the syntax of the VDM constructs in the embedding. The end result is that the embedding has its own syntax which is quite similar to that of the VDM language itself. The primary differences lie in separator characters such as `"` to distinguish between Isabelle and VDM syntax, `^` to identify VDM variables and `@` to identify VDM types.

In addition to the syntactical similarities there is also a near one-to-one correspondence between constructs in the source and target languages which facilitates the translation process. However, while CML has OO features the embedding does not support OO so it is suitable for representing VDM-SL models only.

Finally, we briefly describe the manually written existing translation, based on the visitor framework of the Abstract Syntax Tree (AST). The translation visitors traverse the AST and produce an intermediate data structure used to store relevant translation information for each node including its syntax and dependencies. Afterwards, the data structure is used to generate the Isabelle syntax, either with direct conversion to strings or with auxiliary methods and classes for the processing of more complex nodes. Further details about the existing Isabelle translation as well as the embedding are available in [7].

2.2 Code Generation Platform

The reason for using the CGP, and what makes it a viable solution for developing code generators, is found in the way the CGP represents and works with the generated code.

From the VDM AST the CGP constructs an Intermediate Representation (IR) of the generated code, which forms a tree structure that is independent of any particular target language.

Initially, each node in the IR has a one-to-one correspondence to a node in the VDM AST. Subsequently, the IR is subjected to a series of *transformations* in order to change the tree structure into a new form that is easier for a particular code generator to produce code from. More specifically, each transformation represents a rewriting of the IR with the purpose of changing the IR into a form where each node in the resulting tree structure maps easily into the target language. One advantage of this approach is that transformations operate directly on the IR, and therefore they can be shared among code generators. As an example, the Java and C++ code generators use many of the same transformations to eliminate functional-styled constructs in the IR such as quantified expressions and collection comprehensions.

The IR is generated from an AST specification file using the *AstCreator* tool [1]. In addition to the IR nodes, the *AstCreator* also generates mechanisms to walk the tree using visitors [5] as well as functionality to change the tree structure by allowing parts of it to be replaced. Transformations are themselves implemented as extensions to the visitors generated by the *AstCreator*. What characterises a transformation is that in addition to traversing the tree structure, it also manipulates it.

After the IR has been fully transformed, it is handed over to a language-specific backend generator in order to finalise the code generation process. The CGP provides a framework for syntax generation that serves to facilitate production of code in the target language. This framework is based on the Apache Velocity template engine and used for mapping each node in the IR into concrete syntax [14]. This is handled by the *template manager*, which associates each type of IR node to a template file, that describes the code to be produced.

Code generators extending the CGP may need extra nodes in addition to those already defined by the platform. Therefore, the CGP allows new nodes to be added via the *AstCreator* extension mechanism [4]. This mechanism allows the *AstCreator* to produce nodes and visitors that allow construction and traversal of hybrid trees, .i.e. tree structures composed of both IR nodes defined within the CGP and new nodes contributed via an AST specification extension file. In addition to adding new nodes, the CGP also allows existing IR nodes to be extended to include new fields. Finally, the template manager can be redefined to support syntax generation of new nodes added by the user.

3 Methodology

Based on the description of the CGP in subsection 2.2 we now outline the steps used to develop the Isabelle syntax generator. These steps constitute a general methodology for development of code generation support in Overture using the CGP. Others who want to use the CGP to develop code generation support for another target language may benefit from following these steps.

We start out by listing the steps to be carried out by the developer and afterwards we elaborate on each of them.

1. Set up the CGP extension
2. Add new nodes
3. Transform the IR
4. Generate syntax
5. Validate the translation

The first step in the process is only necessary once. The remaining steps are done in an iterative manner. The approach is to start with a very small VDM example and go through the steps until the example is completely translated. Afterwards, the example should be expanded as little as possible and the steps repeated. This is done iteratively until the new CGP extension is complete.

Step 1 - Set up the CGP extension: Broadly speaking, the setting up of the CGP extension consists of subclassing the base code generator class – `CodeGenBase` – that is the common extension point of the CGP. The base code generator is responsible for driving the code generation and providing access to the IR and various settings. It is also responsible for storing data used and generated throughout the code generation process.

Next, it is necessary to construct a new template manager for the extension. This can be done by subclassing the base template manager. This will provide access to the basic CGP template structure which manages an initial collection of template locations. If additional template locations are necessary, the template manager can be used to configure them.

Finally, it is worth setting up a basic test infrastructure to drive the development process. This test infrastructure is responsible for processing a VDM source, passing the respective AST to the code generator and validating the translation outcome.

Step 2 - Add new nodes: If the target language construct being translated is sufficiently different from those of the base IR, then it is likely that a code generator needs extra nodes. If necessary, these can be provided by extending the IR as described in subsection 2.2. Once the extension is defined, the *AstCreator* tool must be invoked in order to generate the extension nodes.

Step 3 - Transform the IR: Constructs that are not supported by the code generator need to be transformed away, using either base IR nodes or extended nodes generated in the previous step. This is done by implementing one or more necessary transformations. It is recommended that transformations be as small as possible so that each transformation only changes the IR in terms of one concept such as removing comprehensions or reordering definitions.

Step 4 - Generate syntax: Once the IR is in a form suitable for code generation, syntax can be generated using the syntax generation framework of the CGP. This is done by creating the Apache Velocity template files for each of the nodes that is to be translated and updating the template manager accordingly.

Step 5 - Validate the translation: Validation of the translation should be done by means of the test infrastructure by comparing the translation output to a reference. Alternatively, executable translated code may also be compiled and executed to ensure it produces the right result. This test should then be stored to use as regression in the continued development of the CGP extension.

4 Translations and Transformations

The Isabelle embedding we are targeting is very similar to VDM in the sense that most constructs in VDM are present in the embedding. As such, the initial version of the IR is already close to what is needed for generation – most nodes in the IR already map directly to a construct in the target language. Therefore, there is a relatively small number of operations that need to be performed over the tree.

The first set of operations is also the simplest and most common: direct syntax translations. These translations can be applied directly to the initial IR nodes that already map directly to a construct in the embedding. A few of them are shown in Table 1. These translations take advantage of the fact that the Isabelle embedding of VDM defines its own syntax which is quite close to that of VDM. In general, the syntax is the same as that of source VDM, except for the following:

- all constructs are delimited by `"` to identify them as user-defined syntax in Isabelle
- variables names are delimited by `^` to mark them as model variables
- types are prefixed by `@` to mark them as model types
- string literals are delimited by `''`

VDM	**Isabelle embedding**
`x`	`"^x^"`
`int`	`"@int"`
`f(1)`	`"f(1)"`
`"foo"`	`"''foo''"`
`if b then s1 else s2`	`"if ^b^ then ^s1^ else ^s2^"`

Table 1: VDM constructs and their Isabelle embedding counterparts.

To achieve these translations, all that is necessary is to specify the target syntax in the Velocity templates and the CGP handles everything else. Most templates are simple since most translations only need to add minor pieces of Isabelle syntax. A few translations require some extra logic – for example, sequences of type `char` are handled differently from all other sequences – and this is achieved through a handful of auxiliary static methods callable from within the template engine.

The second set of operations consists of tree transformations, of which the first is reordering of definitions. Isabelle does not allow forward referencing in its definitions so any dependency of a definition must be processed before the definition itself. When

generating syntax, the CGP processes definitions in the IR in the order in which they appear so it is necessary to reorder the IR nodes according to their dependency relation. For example, consider the VDM functions shown in Listing 1.1. The initial IR generated for this example would have to be re-ordered as shown in Figure 1.

```
1 f : int -> int
2 f (x) == if x = 0 then 0 else g(x);
3
4 g : int -> int
5 g (x) == x/x;
```

Listing 1.1: A simple forward dependency example.

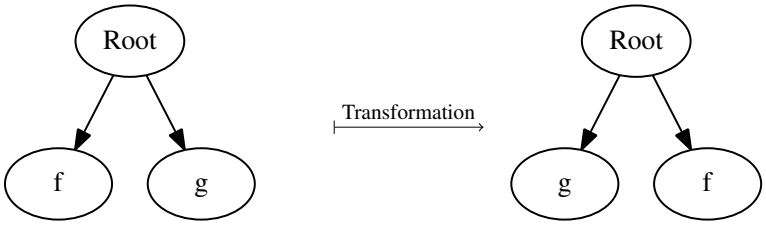


Fig. 1: Dependency sorting transformation.

Dependency sorting is implemented as a CGP transformation that takes an IR module node (the top level element of the IR), constructs a dependency graph of its definitions and then applies a topological sort algorithm [2].

The final operation over the IR is also related to dependency handling, specifically the dependencies between mutually recursive functions. Isabelle can cope with mutually recursive functions but these must be identified as such and grouped together for processing.[2] In order to provide grouping of mutually recursive functions, we construct another transformation that constructs a dependency graph for the function definitions and afterwards applies an algorithm for computing strongly connected components [11]. Thus, the VDM functions in Listing 1.2 would be transformed as shown in Figure 2.

[2] Although Isabelle supports them, the VDM embedding cannot currently cope with mutually recursive functions. However, we have implemented the transformation nonetheless as it was a good way to test the extensibility of the CGP.

```
1 odd: nat -> bool
2 odd (x) == if x = 0
3     then false
4     else even(x-1);
5
6 even : nat -> bool
7 even (x) == if x = 0
8     then true
9     else odd(x-1);
```

Listing 1.2: A simple example of mutual recursion.

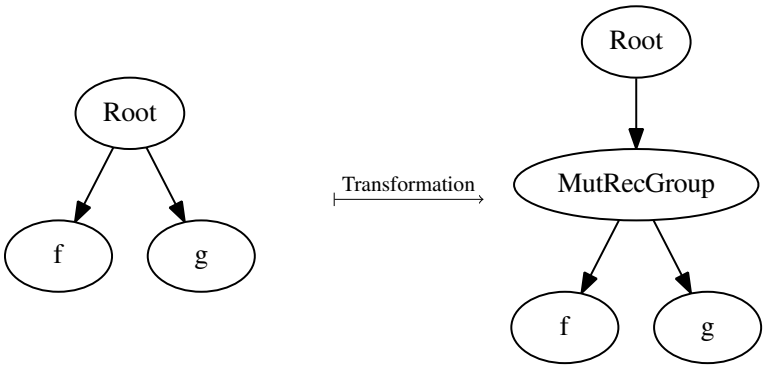


Fig. 2: Mutual recursion grouping transformation.

It is worth noting that the base IR module node does not support mutual recursion groups. As such, we extended the IR to add a new field for it. The mutual recursion transformation takes a base module node as its input and produces an extended module node.

5 Results

5.1 New Isabelle Generation

This section presents the translation from VDM to Isabelle. The translation is demonstrated by means of a complete example, shown in Listing 1.3. Much of the translation is straightforward syntax conversion, however, the example demonstrates the two main issues discussed in section 4: reordering definitions due to dependencies and grouping mutually recursive functions.

Functions `g()` and `f()` shown in lines 3-7 of the VDM model are translated to functions `f()` and `g()` in the Isabelle embedding shown in lines 5-13. Note that the two functions have changed to that `f()` comes before `g()` in the Isabelle source. This is because `f()` is a dependency of `g()` and so must be processed first.

Functions `odd()` and `even()` shown in lines 9-17 of the VDM model are also translated to functions in the Isabelle embedding, shown in lines 15-31. However, the functions in the embedding are delimited by the **`begin_mutrec`** and **`end_mutrec`** keywords which identify them as a block of mutually recursive functions. In Isabelle, such functions must be delimited as they are processed together.

5.2 Code Generation Extensibility Improvements

In addition to constructing the new extension, a series of improvements to the extensibility of the CGP were also carried out. The first set of extensibility improvements had minor impact on the CGP and was related to changing the visibility of various classes and class members. Prior to this work, we were uncertain of which parts of the CGP needed to be exposed to extensions. While it would have been possible to simply expose everything, that would make the CGP too complex to use. By carrying out this work we were able to discover which features to expose and were able to safely keep the rest encapsulated inside the CGP.

As an example of the above, the template manager has a field that defines the folder structure used so store template files. This field was not visible to extensions and that forced an extension to follow the same structure as the base CG with no ability to redefine it. By making the field visible to subclasses, it became possible for each extension to define its own template folder structure.

The second change to increase extensibility had greater impact on the design of the CGP and was related to transformation application. Originally, the CGP was only capable of transforming the internal part of a node. In other words, the root node of the tree could not be changed. This was insufficient for our extension because it was necessary to have a different class at the root of the tree. To address this, the CGP was modified to support transformations that convert between different node types at the root of the tree and thus it became possible to perform transformations between any two arbitrary trees. This new kind of transformation was named *total transformation* and the existing ones were preserved as *partial transformations*. One advantage of the *partial transformation* is that it can rely on the root node of the tree to remain the same and know what kind of node it is. This reduces the amount of conversions that are required to perform the transformation. The *total transformation* is more powerful but will always take as input and produce as output a generic tree node. The CGP was enriched with functionality to help cope with this by converting between generic and specific root nodes via the adapter pattern [5].

6 Evaluation

To assess the effectiveness of using the CGP for Isabelle translation, a simple comparison of volume – measured in Lines of Code (LoC) – was performed between the two

```
functions

g : nat -> nat
g (x) == f(x);

f : nat -> nat
f (x) == x;

odd: nat -> bool
odd (x) == if x = 0
    then false
    else even(x-1);

even : nat -> bool
even (x) == if x = 0
    then true
    else odd(x-1);
```

(a) VDM model.

```
theory A
  imports utp_cml
begin

cmlefun f
  inp x :: "@nat"
  out "@nat"
  is "^x^"

cmlefun g
  inp x :: "@nat"
  out "@nat"
  is "f(^x^)"

begin_mutrec

cmlefun odd
  inp x :: "@nat"
  out "@bool"
  is "if (^x^ = 0)
    then false
    else even((^x^ - 1))"

cmlefun even
  inp x :: "@nat"
  out "@bool"
  is "if (^x^ = 0)
    then true
    else odd((^x^ - 1))"

end_mutrec

end
```

(b) Isabelle translation.

Listing 1.3: VDM model and respective Isabelle translation.

versions. LoC is an imperfect measure of volume and does not particularly capture effort or productivity. However, it can be effectively and accurately measured and does provide a reasonable measure of the size of an implementation, which is sufficient for our comparison.

Table 2 presents a summary of results. In this table, *Manual* refers to the original visitor-based translation and *CGP* refers to the translation we have implemented. The comparison does not consider components from the original translation that are re-

sponsible for processing CML-exclusive elements that have no counterpart in VDM. To facilitate comparison, we have broadly grouped the sources of both versions into three groupings:

data Refers to classes implementing the intermediary data representation between source and target syntax

process Refers to classes that are used to help process or analyse the intermediary representation

syntax Refers to classes that provide or define the target syntax for final translation printing

	Manual	CGP	ΔLoC_{abs}	ΔLoC_{rel}
data	981	27	954	97.25%
process	2427	538	1889	77.83%
syntax	1395	86	1309	93.84%
Total	4803	651	4152	86.45%

Table 2: Volume comparison between translation implementations measured in LoC.

Looking at the data in Table 2, it is clear that utilising the CGP allows for an implementation with much less volume – a reduction of 86%. There are gains in every grouping but the largest ones are in the internal representation – 97%. This is because the *Manual* version utilises a handwritten data structure, whereas the *CGP* version reuses the IR and the only code necessary is that for defining the necessary data extensions. Likewise, most of the machinery for processing both the source language and the IR is reused from the CGP. Particularly, the construction of the IR from the source AST is handled entirely by the CGP. The *syntax* grouping is also much smaller in the *CGP* version – a reduction of 93% –since it uses the template engine in the CGP which allows for significantly more concise expression of syntax.

7 Future Work

In the future, there are two main avenues for improving this work: the translation itself and the extensibility of the CGP. Beginning with the translation, the most immediate improvement is to expand the coverage of VDM constructs. This is to some extent tied to the support of the embedding but there is a significant number of supported constructs that are not translated. For most of these it is only a matter of adding the relevant templates, although there is also the matter of making the dependency calculator more generic, which should not present a problem.

On the topic of the embedding, it would be worthwhile to switch to a pure VDM embedding. While the similarities between CML and VDM make the current embedding suitable for an initial translation, it would be beneficial to migrate to a dedicated

embedding for VDM that could be maintained and evolved separately as necessary. Furthermore, the current embedding contains multiple definitions supporting the reactive aspects of CML that are unnecessary from a VDM perspective. Finally, a dedicated VDM embedding would allow for syntax that is even closer to that of VDM. Work is already underway on adapting the CML embedding into a pure VDM one.

Returning to the translation, there is a potentially problematic issue in that it is only possible to generate syntax for all definitions of the same kind together in one pass. This is a problem when needing to print definitions of various kinds according to the order of dependencies. The issue is related to the IR being structured as lists of definitions of the same kind. It would need to be altered to support generic definition lists – for example, type and function definitions would be stored in the same list.

In terms of translation, it would also be worthwhile to translate proof obligations along with the model thus allowing them to be discharged in Isabelle. The proof obligations are encoded as ASTs using only the expression subset of VDM. Therefore it should be possible to translate them with the existing machinery and require only some additional syntax to turn them into proof goals for Isabelle.

With regards to the CGP itself, the work presented in this paper has suggested two improvements to be carried out. The first is an architectural refactoring of the CGP. At the moment the CGP is directly tied to the Java code generator and that component must be reused as part of reusing the CGP. While this does not limit the ability to construct new extensions, it does expose a significant amount of Java-related functionality that is not necessary. Therefore, it would be beneficial to refactor the code generator into a *core* component that provides the CGP and a *javacg* component that provides code generation to Java.

Another improvement is related to transformations of the IR. Currently, a new extension must provide all of its transformations and develop them from scratch. It stands to reason that some translations are required for multiple extensions – for example, dependency sorting may be needed in other target languages – so it would be beneficial to reuse existing transformation. However, most transformations make assumptions about the target language and the order in which transformations are applied. This makes it quite challenging to reuse them since none of these assumptions hold in all situations.

8 Conclusion

This paper has presented a VDM to Isabelle translation using the code generation platform of Overture. The initial results show that the translation can be written with a significantly smaller amount of code (86%). Additionally, the use of the platform confers various benefits such as improved maintainability of the intermediary data structure and more easily adjustable syntax (via templates instead of Java strings). Also, any general improvements made to the CGP will be propagated to the translation as well.

The successful development of the Isabelle translation stands as proof of the extensibility of the CGP. Some issues were identified and addressed in order to increase extensibility. Specifically, a more generic transformation mechanism was implemented with support for changing the root node of the tree.

Our initial results show that it is quite worthwhile and beneficial to use the CGP for syntactical translations. The work presented here not only validates the extensibility of the CGP but it also provides a good basis for developing a complete VDM to Isabelle translation.

Acknowledgements

The authors would like to thank Simon Foster, Richard Payne and Sune Wolff for their valuable insight and comments on this paper.

References

1. ASTCreator (2014), https://github.com/overturetool/astcreator
2. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, 2nd edn. (2001)
3. Couto, L.D., Foster, S., Payne, R.: Towards Certification of Constituent Systems through Automated Proof. In: Workshop on Engineering Dependable Systems of Systems (EDSoS) (May 2014)
4. Couto, L.D., Tran-Jørgensen, P.W.V., Lausdahl, J.W.C.K.: Migrating to an Extensible Architecture for Abstract Syntax Trees. In: 12th Working IEEE / IFIP Conference on Software Architecture (May 2015)
5. E.Gamma, R.Helm, R., J.Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company (1995)
6. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), http://overturetool.org/publications/books/vdoos/
7. Foster, S., Payne, R.J.: Theorem Proving Support - Developers Manual. Tech. rep., COMPASS Deliverable, D33.2b (September 2013)
8. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: Unifying Theories of Programming, pp. 21–41. Springer (2015)
9. Hoare, T.: Communication Sequential Processes. Prentice-Hall International, Englewood Cliffs, New Jersey 07632 (1985)
10. Hoare, T., Jifeng, H.: Unifying Theorieses of Programming. Prentice Hall (April 1998)
11. JGraphT (2015), http://jgrapht.org/
12. Jørgensen, P.W., Couto, L.D., Larsen, M.: A Code Generation Platform for VDM. In: The Overture 2014 workshop (June 2014)
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
14. Apache Velocity (2015), http://velocity.apache.org/
15. Woodcock, J., Cavalcanti, A., Fitzgerald, J., Larsen, P., Miyazawa, A., Perry, S.: Features of CML: a Formal Modelling Language for Systems of Systems. In: Proceedings of the 7th International Conference on System of System Engineering. IEEE (July 2012)

11

Towards Enabling Overture as a Platform for Formal Notation IDEs

The paper in this chapter has been accepted as a peer-reviewed workshop paper.

[P23] Luís Diogo Couto, Peter Gorm Larsen, Miran Hasanagić, Georgios Kanakis, Kenneth Lausdahl and Peter W. V. Tran-Jørgensen. *Towards Enabling Overture as a Platform for Formal Notation IDEs*. 2nd Workshop on Formal Integrated Development Environment (F-IDE), June 2015.

Towards Enabling Overture as a Platform for Formal Notation IDEs

Luís Diogo Couto Peter Gorm Larsen Miran Hasanagić Georgios Kanakis
Kenneth Lausdahl Peter W. V. Tran-Jørgensen
Department of Engineering, Aarhus University,
Finlandsgade 22, 8200 Aarhus N, Denmark
{ldc,pgl,miran.hasanagic,gkanos,lausdahl,pvj}@eng.au.dk

Formal Methods tools will never have as many users as tools for popular programming languages and so the effort spent on constructing Integrated Development Environments (IDEs) will be orders of magnitudes lower than that of programming languages such as Java. This means newcomers to formal methods do not get the same user experience as with their favourite programming IDE. In order to improve this situation it is essential that efforts are combined so it is possible to reuse common features and thus not start from scratch every time. This paper presents the Overture platform where such a reuse philosophy is present. We give an overview of the platform itself as well as the extensibility principles that enable much of the reuse. The paper also contains several examples platform extensions, both in the form of new features and a new IDE supporting a new language.

1 Introduction

The Vienna Development Method (VDM) is one of the most mature Formal Methods (FM) [26, 15]. The method focuses on the development and analysis of a system model expressed in a formal language. The formality of the language enables developers to use a wide range of analytic techniques, from testing to mathematical proof, to verify the consistency of a model and its correctness with respect to an existing statement of requirements. The VDM modelling language has been gradually extended over time. Its most basic form (VDM-SL), standardised by ISO [29] supports the modelling of the functionality of sequential systems. Extensions support object-oriented modelling and concurrency (VDM++) [16], real-time computations [36] and distributed systems (VDM-RT) [43, 42]. All these dialects of VDM are supported by the Overture platform [30].[1]

The mission of the Overture open-source project is twofold:

- To provide an industrial-strength tool that supports the use of precise abstract models in any VDM dialect for software development.
- To foster an environment that allows researchers and other interested parties to experiment with modifications and extensions to the tool and the different VDM dialects.

As is the case with other FM tools, the Overture Integrated Development Environment (IDE) consists of a common Abstract Syntax Tree (AST) representing the model and various plug-ins providing the different kinds of analysis available in VDM as shown in fig. 1. The broad variety of analysis possible is common in many formal methods. In such cases, it is important to ensure that all analyses are implemented in a consistent way to facilitate maintenance. Such consistency would also aid in the development and integration of new functional extensions.

[1] See http://overturetool.org.

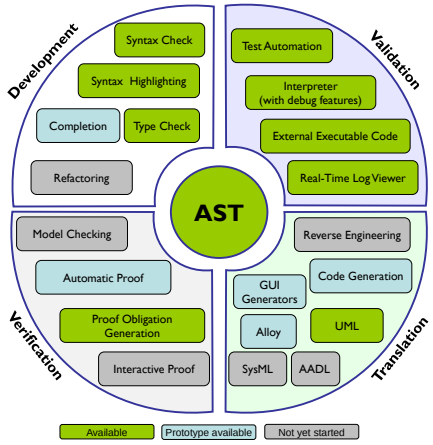


Figure 1: Overture Tool Components

In addition to the above, this platform-based architecture allows for the reuse of common features across all extensions. This reuse can be taken further by supporting language extensions that would allow other formal notations to reuse the same platform. Recently, Overture has been re-factored to enable such a reuse [8, 12]. The main contribution reported in this paper is the Overture platform itself and its extensibility principles which are described in sections 2 and 3. An extensible platform facilitates the development of new features for an IDE and in section 4 we demonstrate how several features of the Overture IDE have been developed on top of the platform. Furthermore, in section 5 we demonstrate how the platform has been integrated with an external tool.

The extensibility principles of the Overture platform also affect the notation itself. The platform is capable of supporting a base language (VDM in the case of Overture) as well as multiple notation extensions. This allows for the development of IDEs for new notations with heavy reuse of common features. Section 6 describes one such IDE, which also includes integration with several external tools.

Open issues remain in the platform, most notably in terms of integration with external tools. Section 7 lays out future work for addressing some of these issues and also summarises the paper. It is our hope that this paper demonstrates the advantages of platform-based IDE development and that it can be beneficial for multiple FM tool builders to share a common platform.

Other examples of FM platforms with comparable functionalities include the Asmeta tool set for ASM [4, 3], the Rodin platform for Event-B [1, 2, 14] or TLAToolbox for TLA$^+$ [28, 41]. The extension philosophies of the software tools differ as do the actual extensions that are available. A detailed comparison is beyond the scope of this article, but more information about these platforms and the modelling languages they support can be found by following the references provided. The general philosophy of reuse has also been employed effectively for theorem provers [39].

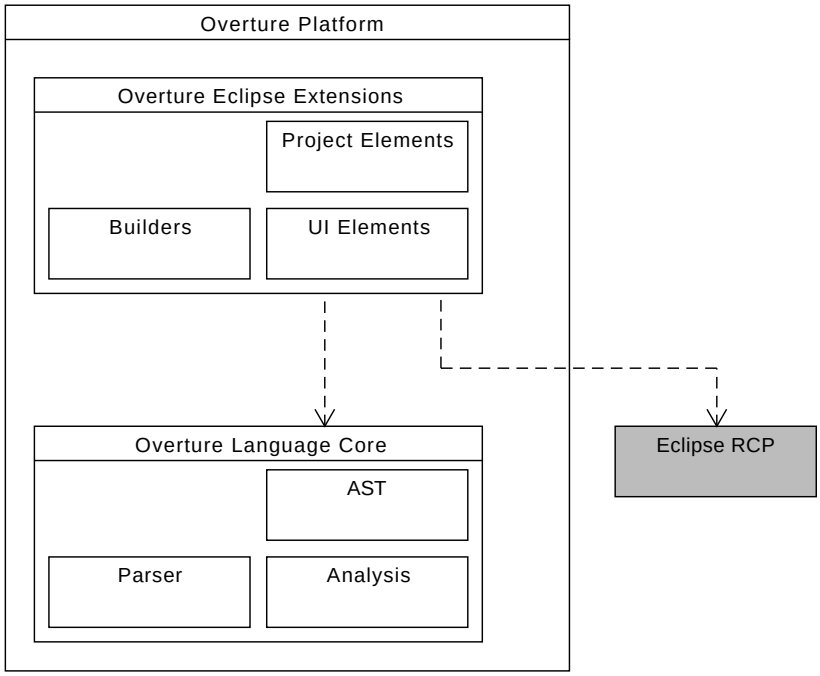


Figure 2: The Overture Platform.

2 The Overture Platform

2.1 Overview

The Overture platform supports the development of FM IDEs. It was originally developed to support the development of the Overture IDE for VDM but has since evolved into a more general platform. It is comprised of two parts: the *Overture Language Core* and the *Overture Eclipse Extensions*, as shown in fig. 2.

2.2 Overture Language Core

The language core encapsulates and handles any language and notation-related concerns, including parsing, representation and analysis, in order to facilitate decoupling between the core language and User Interface (UI) implementations. In addition to the general benefits of separation of concerns, the language core also opens the possibility of migrating the IDE implementation to another UI technology as well as providing the base tool functionalities for command line access, batch processing or as an external tool to be accessed by others.

The language core consists of an extensible AST that is automatically generated by the *AstCreator* tool[2], as well as a parser for constructing the AST from model sources. In addition, *AstCreator* also

[2]See `http://github.com/overturetool/astcreator`

generates machinery for traversing and processing trees in a consistent way in the form of a visitor framework [21]. Any kind of analysis of the AST such as type checking or interpretation should be implemented using the visitor framework.

One of the key features of the language core is its extensibility mechanism which allows language extensions or new languages to be implemented in the Overture platform while reusing as much existing code as possible. This mechanism is described in further detail in section 3

2.3 Overture Eclipse Extensions

The Overture platform also consists of a set of extensions to the Eclipse Rich Client Platform (RCP) that are used to help build the UI components of the IDE. The Eclipse RCP is a generic framework for building rich client applications using the Eclipse OSGi plug-in model and UI toolkits. It is is powerful and generic but comes with a cost: significant amounts of boilerplate source code and configuration files must be written in order to prepare it to build an IDE.

The Overture Eclipse extensions automate some of the configuration and preparation work by providing the aforementioned boilerplate code targeting FM notations. The extensions provides an extensible application framework on top of the RCP. It significantly reduces the amount of code that needs to be written in order to contribute an extension to the IDE. To put it another way, the RCP API is very wide and the Overture Eclipse Extensions summarise a portion of it, thus giving developers faster access to the functionality at the cost of some flexibility. However, the Overture extensions are fully interoperable with the RCP so any other extension that requires direct access to the RCP can still be used.

There are other frameworks similar to the Overture extensions in the Eclipse project, such as the Dynamic Languages Toolkit (DLTK) [13] and *Xtext* [45]. DLTK is designed to support the implementation of IDEs for dynamic programming languages and *Xtext* is designed to support the implementation of IDEs for for Domain-Specific Languages (DSLs) or small programming languages. Neither framework is particularly suitable for VDM – VDM is similar in notation to a statically typed general-purpose programming language – which was the original target language to be supported by the Overture IDE.

Broadly speaking, the Overture Eclipse extensions can be divided into three groups:

- a set of UI elements for editors, launch configurations, etc. that interact directly with the Eclipse RCP.
- a set of project elements that represent the FM model and associated concepts such as source units, according to the Eclipse project model. Also included are connectors and providers for accessing these various entities from within the IDE.
- a set of builders that interact with the language core in order to process language sources to construct an internal representation of the model and load it into the project elements.

Both the builders and the project elements are developed according to standard Eclipse conventions so that new versions of these packages for other notations may be contributed.

Currently, the Overture Platform primarily supports the Overture IDE. The Overture IDE is comprised of Eclipse plug-ins that use components implemented with the language core to perform analysis of the VDM AST and UI components that wrap the analysis and use the Eclipse extensions to implement the interaction with the user.

3 Extension Principles of the Overture Language Core

The basic principles of extensibility in the Overture language core are related to the generation of ASTs from specification files, similar to parser generators like SableCC [20]. In addition to generating the classes representing the tree structure, it is important to generate auxiliary machinery to allow developers to implement analysis of the AST in a consistent manner.

The main way to construct extensions in the language core is by extending the AST. Generally speaking, an AST is extended by adding new subtrees that are either entirely new or that contain some existing base nodes. In addition, the extended tree needs to reuse the existing base node classes wherever possible.

In addition to extending the tree itself, it is important to also extend the analysis machinery. Particularly, this extended machinery needs to be able to analyse trees made up of extension and base nodes. Furthermore, the extended analysis machinery needs to reuse the base machinery when processing base nodes – this is essential for achieving reuse of functionalities already implemented as base analysis.

Whether speaking of a tree made of only extension nodes or a hybrid tree with extension and base nodes or even a base tree, the AST classes have a limited ability to enforce the structure of each particular instantiation of the tree. It is the syntax of the language, as encoded in the parser, that ultimately controls which trees are admissible. Along the same lines, it is the parser that controls which base nodes are reused when constructing hybrid trees as the extended tree specification can only set an upper limit on this.

The extensibility principles of the Overture language core are primarily realised through the *AstCreator* tool. *AstCreator* provides provides an automated way of generating trees and auxiliary machinery from specification files as shown in fig. 3. The tool is capable of taking an existing AST as well as an extension specification and generating the extension nodes and visitor framework upon which to implement analyses.[3] Both nodes and visitors are aware of the base classes thus ensuring interoperability with the base trees.

It is also possible to use *AstCreator* to build a completely new AST supporting a language that is unrelated to VDM (see section 4.1 for an example). In this case, a new base tree and visitor framework will be produced and it will not be possible to reuse existing components of the language core. As such, we focus on the case where the new language being supported is an extension of an existing notation where it is possible to reuse parts of the base AST, and the corresponding analysis. Typically, constructs like arithmetic or logical expressions or imperative statements can be reused. This leads to hybrid trees where nodes from the base and extended trees are blended together.

The semantics of such a language extension should be implemented as various AST analyses such as type checking or interpretation. Each analysis should be implemented as an independent component that processes the tree in a consistent way. The visitor framework that is generated as part of the extension provides a way to achieve this. Since the visitor framework itself is extension-aware it enables selective and controlled reuse of existing base analyses as necessary. The extension-aware visitor is illustrated in fig. 4.

An example of these extension principles at work can be seen in the Symphony IDE, as described in section 6.

[3] *AstCreator* is also capable taking a base and extension specification and producing both sets of classes, though this is done less frequently.

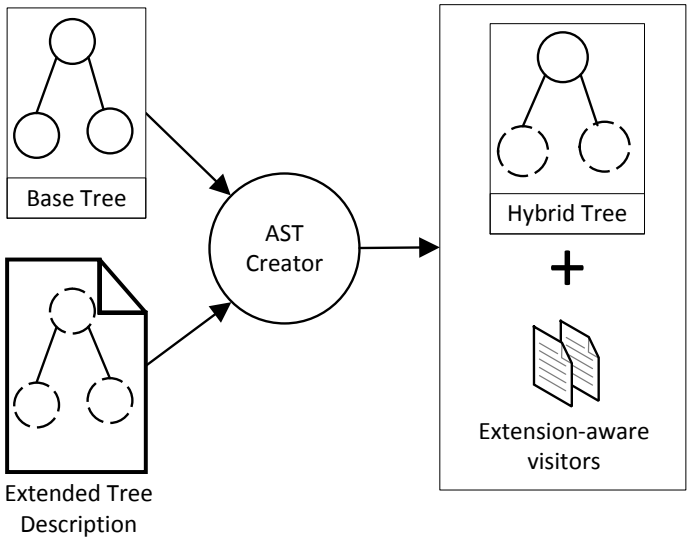


Figure 3: Extending AST specifications.

4 Functionality extensions

New functionality can be contributed in the Overture platform either by using the language core, the Eclipse extensions or a combination thereof. The language core provides the necessary mechanisms to interact with the AST as well as extending it, whereas the Eclipse extensions provide the means to expose functionality to the user. In this section, we provide examples of how both can be used to add new functionality to Overture.

4.1 The code generation platform

The code generation platform aims to facilitate integration of VDM code generators into Overture with minimum effort [27]. Like many other Overture components, the code generation platform interacts with the language core by analysing a type checked VDM AST in order to generate code in some target language. Currently, the code generation platform is used to develop VDM code generation support to Java and C++, and in addition, there is ongoing work on generating Isabelle/HOL syntax [11].

In order to promote reuse the code generation platform works with an Intermediate Representation (IR) of the generated code, which is independent of any particular target language. In addition, the code generation platform provides mechanisms for rewriting or *transforming* the IR into a semantically preserving form that is easier for a particular backend to code generate. Furthermore, since transformations work directly on the IR it becomes easier for different backends to use and contribute new

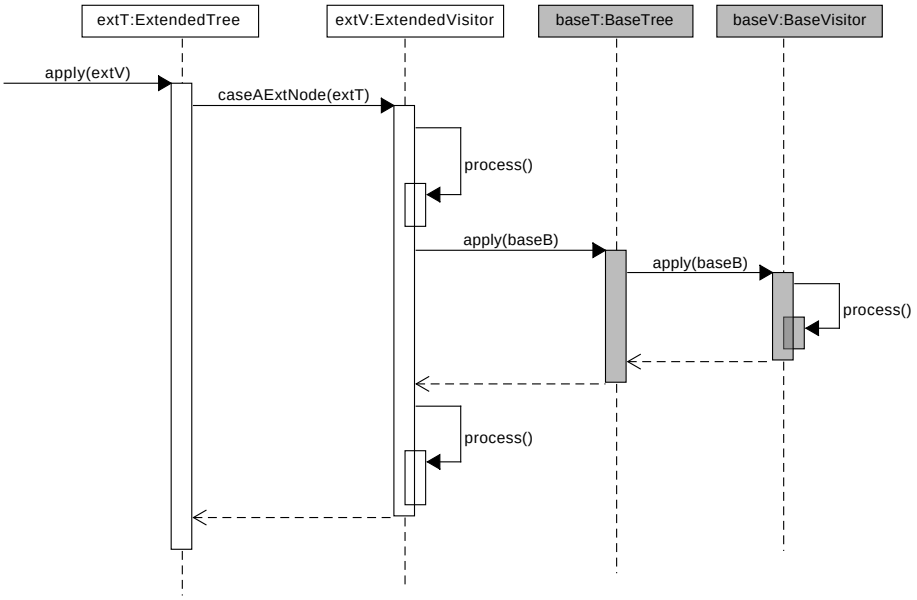


Figure 4: Extended AST and analysis.

functionality to analyse and modify the IR.

The code generation platform allows new nodes to be added to the IR as well as extending existing nodes with additional fields as enabled by *AstCreator*, which is used for the specification of the IR. The Isabelle/HOL code generator exploits this, since mutually recursive functions must be grouped explicitly in Isabelle/HOL and therefore the code generator adds function groups to the IR. This is done in a supplementary tree extension file and demonstrates how users of the code generation platform can extend and change the IR as needed. Although most of the work involved in developing code generation support includes traversing and transforming the IR, thus interacting with the language core, the Eclipse extensions provide the necessary mechanisms to read preferences and configure the code generation process.

4.2 Interpreting implicit specifications using ProB

In VDM functions and operations can be either explicitly or implicitly (using pre and post conditions) defined. An explicit description defines how the output is obtained from the input, which enables the description to be evaluated directly in the VDM interpreter [33]. Implicit descriptions, on the other hand, only specify the constraints that must be met but without defining how the output is obtained. Therefore, attempting to evaluate an implicit description in the interpreter yields a runtime error. To avoid restricting analysis of implicit descriptions to static analysis only, Overture has recently integrated the ProB constraint solver in order to enable evaluation of implicit descriptions [32].

Interpretation of implicit descriptions adds an additional step to the model execution where the pre and post state as well as the constraints imposed by the implicit description, is converted to ProB syntax to form a formula, which is submitted to the ProB constraint solver. This formula is constructed as a

string by analysing the type checked AST.

If ProB is able to find a solution to the given problem the solution is converted back to VDM format and used throughout subsequent execution of the model. Intercepting the interpretation of implicit descriptions is primarily enabled through extension of the language core, and interacting with ProB via an external Java API.

4.3 VDMTools integration

VDMTools [17] is an industrial strength IDE, maintained by the SCSK corporation, for analysing models written in VDM. Among many features also supported by Overture, VDMTools provides extensive semantic checking, execution and Java/C++ code generation of models written in VDM. To facilitate use of different IDEs, Overture provides an option to export an Overture VDM project to a VDMTools compatible format. This plugin is developed through the Eclipse extensions that are used to convert meta data from an Overture project to a format compatible with VDMTools.

4.4 Combinatorial Testing

Combinatorial Testing (CT) in VDM provides automated generation and execution of a large collection of tests as an extension to the language core functionality [31]. The addition of CT in Overture has trigged several changes to the language core components. First, the AST was extended to support the trace nodes. Second, the type checker was updated to support type checking of both traces and generated tests. Finally, the interpreter was extended to support trace expansion as well as test execution.

In addition to extending the language core a CT view has been added to Overture as a new Overture Eclipse plugin extension. This plugin serves to provide a convenient way for users to inspect the test execution results, filtering large collections of tests in order to obtain a reduced representable subset of tests, and re-executing tests individually.

5 Building a Co-Simulation tool with the Overture Platform

The Crescendo tool supports collaborative modelling and co-simulation of Cyber Physical Systems (CPSs) [18], and has been developed by extending the Overture platofrm. This extension enables co-simulation between co-models, which are composed of a discrete time model described in the VDM-RT language, and a continues time model described using differential equations. The extension is composed of a co-simulation engine that connects an extended version of the VDM interpreter [33] from the Overture tool with the simulator in 20-Sim [9].

The Crescendo tool primarily extends the Overture platform using ordinary Eclipse extension points for: builders, debug related UI and views. However, it also uses Overture Eclipse extensions for e.g. editors, and debugging related components.

An extension was also made to the language core by extending the VDM interpreter used for evaluating and debugging with two main features: *a*) the ability to only simulate until a certain time bound, and *b*) the ability to detect when a shared co-simulation variable is accessed. This is necessary in order to support co-simulation such that the two simulators can synchronize their time steps.

6 Building a new IDE with the Overture Platform

Thus far, this paper has shown how to contribute extensions to the Overture IDE and how to extend existing components to support co-simulation with an external tool. These examples consist of extensions that either make very small or no changes to the VDM language, in terms of new syntax, the semantics thereof or the concepts introduced. This makes the extensions relatively simple to support in comparison to an extension for a new full-blown FM notation, especially considering the wide variety of formal notations as well as their associated semantics, paradigms and problem domains.

It is possible to use the Overture platform to build an IDE for a new notation that shares nothing with the VDM language. However, this means that the new IDE will be unable to reuse much of the language core since the AST and associated analyses will be entirely different. On the other hand, when building an IDE for a notation that reuses or shares parts of VDM, then the relevant parts of the language core can be reused. The remainder of this section shows how such reuse was achieved in the construction of the Symphony tool [7] in the COMPASS EU FP7 Project. Symphony supports the COMPASS Modelling Language (CML) notation [44] that was introduced in the COMPASS project and combines VDM with Communicating Sequential Processes (CSP) [22].

The syntax of CML differs significantly from that of VDM, especially as it relates to the new constructs inherited from CSP. As such, it was necessary to construct a parser to recognize CML notation. Tools such as ANTLR [38] greatly aid in parser construction and Symphony has an ANTLR parser built from scratch that processes CML sources to construct ASTs that are compliant with the Overture language core.

The static analysis of CML ASTs (type checking and proof obligation generation) significantly reuses relevant Overture components [10]. In the case of proof obligations, reuse led to reduction in lines of code from 2596 to 978 as well as a reduction in duplicate code from 37.2% to 3.1%. In general, any existing analysis for VDM was reused whenever possible. A good example lies in the processing of VDM expressions inside CSP actions – also an example of hybrid tree processing.

The validation of CML models could not reuse Overture components so easily since the paradigms of CML notation are different from those of VDM. In particular, CML is a process algebra and its models are interpreted as sequences of events, as opposed to VDM's imperative approach based on state transformations.

Due to the difference in paradigms between the languages, significant portions of the Symphony interpreter had to be built from scratch. However, in spite of the differences in behaviour, the Symphony interpreter still manages to reuse the Overture one for evaluating expressions and reused statements.

For all cases of reuse in Symphony, the same basic principle applies: the extended analysis processes the hybrid tree and when it encounters a base node, it submits the node to its counterpart base analysis, with a mechanism in place for the extension to re-assume control and preventing the base analysis from hijacking the analysis of the remaining tree.

Finally, it is important to discuss the underlying semantics of the various analyses as it should be ensured that consistent semantics are in place across all components of the tool, lest errors be introduced in the overall results due to gaps between the various semantics. In the COMPASS project this was addressed by using the Unifying Theories of Programming [23] that provides a common framework for the various semantic models used in the project. This work eventually led to a mechanisation of a subset of CML in Isabelle [19].

Most functionalities of the Symphony IDE were implemented as Eclipse plug-ins using a combination of the Overture language core (exported via a counterpart Symphony core) and the Overture Eclipse extensions (used to build the main UI components of the Symphony IDE). In addition to

its native functionalities, the Symphony IDE also uses its various plug-ins to integrate externals tools such as Maude [5, 6], Isabelle [37] (via Isabelle/Eclipse [24]), FORMULA [25], RT-Tester [40] and ProB[34, 35]. The most relevant external tool integrations are shown in fig. 5. Note how this aims at following the principle of reusing existing functionality rather than re-developing from scratch.

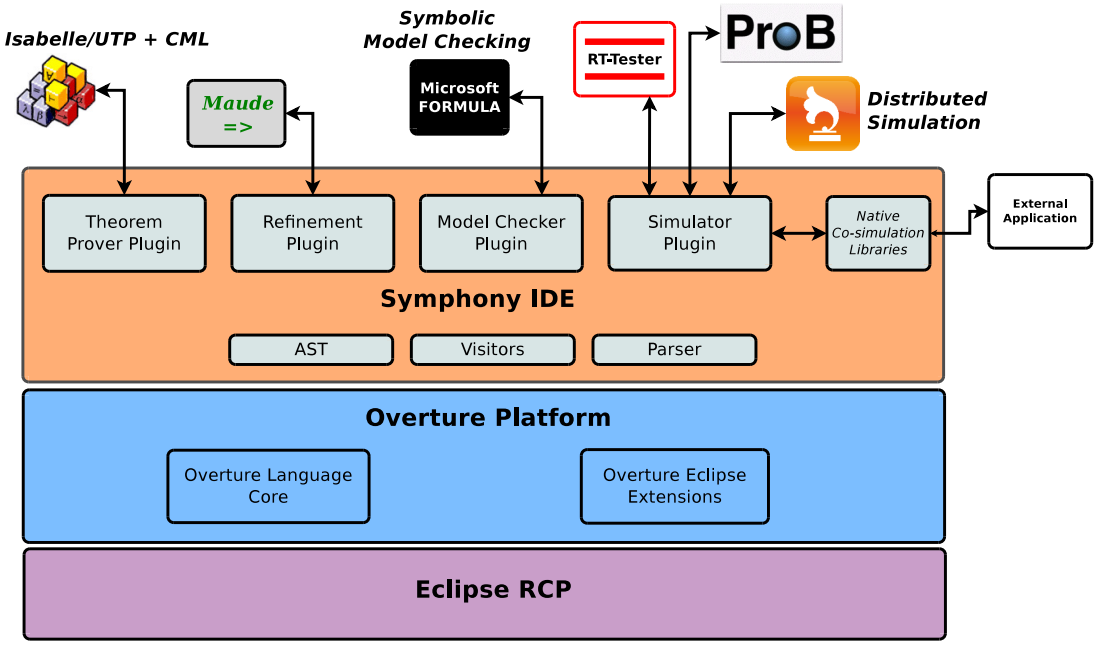


Figure 5: The COMPASS tools

7 Concluding Remarks and Future Work

This paper has described the Overture IDE and its underlying platform. We have shown the extensibility principles of the platform and demonstrated how they support multiple functional extension plug-ins. Furthermore, we have demonstrated how the platform can support notation extensions and, as such, be used as a basis platform by other FM tool builders. The ability to reuse existing functionality and build on the work of other teams can help improve the quality of FM tools in general.

Going forward, there are various potential improvements that can be made to the Overture platform and we discuss a few of them here. The first improvement is in terms of the *AstCreator* tool's specification files. At the moment, *AstCreator* is only capable of generating the Java code for the AST from a fairly simple tree specification file. This is by design. *AstCreator* does not aim to address issues of parsing when tools such as ANTLR already do an excellent job of it. However, it may be beneficial to integrate *AstCreator* with parser generators. Either by deriving an *AstCreator* specification file from the parser generator grammar or by creating a stub grammar from the *AstCreator* specification.

Another potential improvement lies in making more use of the code generation platform when integrating external tools. Integration with external tools often consists of translating the VDM syntax into

that of the external tool and submitting it to the tool as is done for example in the ProB integration. These translations are often implemented manually using the visitor framework. However, by using the code generation platform, significant gains may be attained in terms of the amount of code that is necessary. We are currently undertaking work in this direction and early results are very promising [11].

The final improvement under consideration is also related to external tool integration, but is a somewhat open-ended question at the moment. Integration of external tools is currently done on a case-by-case basis. Each external tool is integrated in its own way with entirely handwritten code. While the syntax translation issue may be addressed, the invocation of the external tool, passing of data to it and collection of results is completely non-standardized. This is mostly a consequence of all external tools having different ways of accessing them. However, at a high-level, most external tool interactions can be reduced to a general case such as external command invocation, protocol-based communication or API access. It would be beneficial to have mechanisms in the platform to help deal with each of the general cases. Another alternative would be a methodological approach where guidelines are produced to help developers implement each kind of integration in a consistent manner.

Acknowledgments

The authors wish to thank Stefan Hallerstede for valuable feedback. Partial funding for the work reported here was provided by the COMPASS project (Grant Agreement 287829) as well as the INTO-CPS project (Grant Agreement 644047).

References

[1] Jean-Raymond Abrial (2010): *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, doi:10.1017/CBO9781139195881. Available at `http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521895569`.

[2] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta & Laurent Voisin (2010): *Rodin: an open toolset for modelling and reasoning in Event-B*. *STTT* 12(6), pp. 447–466, doi:10.1007/s10009-010-0145-y.

[3] (2015): *The Asmeta tool set for ASM*. `http://asmeta.sourceforge.net`.

[4] Egon Börger & Robert F. Stärk (2003): *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, doi:10.1007/978-3-642-18216-7.

[5] M. Clavel, F. Durn, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer & J. F. Quesada (1999): *The Maude System*. In: *Rewriting Techniques and Applications*, Springer, LNCS1631, doi:10.1007/3-540-48685-2_18.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. *Lecture Notes of Computer Science* 4350, Springer-Verlag, doi:10.1007/978-3-540-71999-1.

[7] Joey W. Coleman, Anders Kaels Malmos, Peter Gorm Larsen, Jan Peleska, Ralph Hains, Zoe Andrews, Richard Payne, Simon Foster, Alvaro Miyazawa, Cristiano Bertolini & André Didier (2012): *COMPASS Tool Vision for a System of Systems Collaborative Development Environment*. In: *Proceedings of the 7th International Conference on System of System Engineering, IEEE SoSE 2012*, pp. 451–456, doi:10.1109/SYSoSE.2012.6384150.

[8] Joey W. Coleman, Anders Kaels Malmos, Claus Ballegaard Nielsen & Peter Gorm Larsen (2012): *Evolution of the Overture Tool Platform*. In: *Proceedings of the 10th Overture Workshop 2012*, School of Computing Science, Newcastle University.

[9] Controllab products (2013): *http://www.20sim.com/*. 20-Sim official website.

[10] Luís Diogo Couto & Richard Payne (2013): *The COMPASS Proof Obligation Generator: A test case of Overture Extensibility*. In: *Proceedings of the 11th Overture Workshop*.

[11] Luís Diogo Couto & Peter W. V. Tran-Jørgensen (2015): *Extending the Overture code generator towards Isabelle syntax*. In: *Proceedings of the 13th Overture Workshop*, Center for Global Research in Advanced Software Science and Engineering, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan, pp. 48–59. Available at `http://grace-center.jp/wp-content/uploads/2012/05/13thOverture-Proceedings.pdf`. GRACE-TR-2015-06.

[12] Luís Diogo Couto, Peter W. V. Tran-Jørgensen, Joey W. Coleman & Kenneth Lausdahl (2015): *Migrating to an Extensible Architecture for Abstract Syntax Trees*. In: *12th Working IEEE / IFIP Conference on Software Architecture*.

[13] Eclipse (2015): *Dynamic Languages Toolkit*. Available at `http://eclipse.org/dltk/`.

[14] (2015): *Event-B and the Rodin Platform*. `http://www.event-b.org`.

[15] John Fitzgerald & Peter Gorm Larsen (2009): *Modelling Systems – Practical Tools and Techniques in Software Development*, Second edition. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, doi:10.1017/CBO9780511626975. ISBN 0-521-62348-0.

[16] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat & Marcel Verhoef (2005): *Validated Designs for Object–oriented Systems*. Springer, New York, doi:10.1007/b138800. Available at `http://overturetool.org/publications/books/vdoos/`.

[17] John Fitzgerald, Peter Gorm Larsen & Shin Sahara (2008): *VDMTools: Advances in Support for Formal Modeling in VDM*. *ACM Sigplan Notices* 43(2), pp. 3–11, doi:10.1145/1361213.1361214.

[18] John Fitzgerald, Peter Gorm Larsen & Marcel Verhoef, editors (2014): *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, doi:10.1007/978-3-642-54118-6. Available at `http://link.springer.com/book/10.1007/978-3-642-54118-6`.

[19] Simon Foster, Frank Zeyda & Jim Woodcock (2015): *Isabelle/UTP: A mechanised theory engineering framework*. In: *Unifying Theories of Programming*, Springer, pp. 21–41, doi:10.1007/978-3-319-14806-9_2.

[20] Etienne M. Gagnon & Laurie J. Hendren (1998): *SableCC, an Object-Oriented Compiler Framework*. In: *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '98, IEEE Computer Society, Washington, DC, USA, pp. 140–154, doi:10.1109/TOOLS.1998.711009.

[21] E. Gamma, R. Helm, R. Johnson & R. Vlissides (1995): *Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series* , Addison-Wesley Publishing Company.

[22] C.A.R Hoare (1978): *Communicating Sequential Processes*. *Communications of the ACM* 21(8), doi:10.1145/359576.359585.

[23] Tony Hoare & He Jifeng (1998): *Unifying Theories of Programming*. Prentice Hall, doi:10.1007/11768173.

[24] Isabelle/Eclipse (2015): *Isabelle/Eclipse*. Available at `http://andriusvelykis.github.io/isabelle-eclipse/`.

[25] Ethan K. Jackson, Dirk Seifert, Markus Dahlweid, Thomas Santen, Nikolaj Bjørner & Wolfram Schulte (2009): *Specifying and Composing Non-functional Requirements in Model-Based Development*. In Alexandre Bergel & Johan Fabry, editors: *Software Composition*, *Lecture Notes in Computer Science* 5634, Springer Berlin Heidelberg, pp. 72–89, doi:10.1007/978-3-642-02655-3_7.

[26] Cliff B. Jones (1999): *Scientific Decisions which Characterize VDM*. In J.M. Wing, J.C.P. Woodcock & J. Davies, editors: *FM'99 - Formal Methods*, Springer-Verlag, pp. 28–47, doi:10.1007/3-540-48119-2_2. Lecture Notes in Computer Science 1708.

[27] Peter W. V. Jørgensen, Luís D. Couto & Morten Larsen (2014): *A Code Generation Platform for VDM*. In: *The Overture 2014 workshop*.

[28] Leslie Lamport (2002): *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. Available at `http://research.microsoft.com/users/lamport/tla/book.html`.

[29] P. G. Larsen, B. S. Hansen et al. (1996): *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*. International Standard ISO/IEC 13817-1.

[30] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl & Marcel Verhoef (2010): *The Overture Initiative – Integrating Tools for VDM*. *SIGSOFT Softw. Eng. Notes* 35(1), pp. 1–6, doi:10.1145/1668862.1668864.

[31] Peter Gorm Larsen, Kenneth Lausdahl & Nick Battle (2010): *Combinatorial Testing for VDM*. In: *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, IEEE Computer Society, Washington, DC, USA, pp. 278–285, doi:10.1109/SEFM.2010.32. ISBN 978-0-7695-4153-2.

[32] Kenneth Lausdahl, Hiroshi Ishikawa & Peter Gorm Larsen (2015): *Interpreting Implicit VDM Specifications using ProB*. In: *Proceedings of the 12th Overture Workshop*, *Technical Report Series* CS-TR-1446, Computing Science, Newcastle University, pp. 1–15. Available at `http://www.cs.ncl.ac.uk/publications/trs/papers/1446.pdf`.

[33] Kenneth Lausdahl, Peter Gorm Larsen & Nick Battle (2011): *A Deterministic Interpreter Simulating A Distributed real time system using VDM*. In Shengchao Qin & Zongyan Qiu, editors: *Proceedings of the 13th international conference on Formal methods and software engineering*, *Lecture Notes in Computer Science* 6991, Springer-Verlag, Berlin, Heidelberg, pp. 179–194, doi:10.1007/978-3-642-24559-6_14. Available at `http://dl.acm.org/citation.cfm?id=2075089.2075107`. ISBN 978-3-642-24558-9.

[34] Michael Leuschel & Michael Butler (2003): *ProB: A model checker for B*. In: *FME 2003: Formal Methods*, Springer, pp. 855–874, doi:10.1007/978-3-540-45236-2_46.

[35] Michael Leuschel & Michael Butler (2005): *Automatic refinement checking for B*. In: *Formal Methods and Software Engineering*, Springer, pp. 345–359, doi:10.1007/11576280_24.

[36] Paul Mukherjee, Fabien Bousquet, Jérôme Delabre, Stephen Paynter & Peter Gorm Larsen (2000): *Exploring Timing Properties Using VDM++ on an Industrial Application*. In J.C. Bicarregui & J.S. Fitzgerald, editors: *Proceedings of the Second VDM Workshop*. Available at www.vdmportal.org.

[37] Tobias Nipkow, Lawrence C Paulson & Markus Wenzel (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. 2283, Springer Science & Business Media, doi:10.1007/3-540-45949-9.

[38] Terence Parr (2007): *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf.

[39] Lawrence C. Paulson (2010): *Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers*. In Renate A. Schmidt, Stephan Schulz & Boris Konev, editors: *Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010*, *EPiC Series* 9, pp. 1–10.

[40] Jan Peleska, Elena Vorobev & Florian Lapschies (2011): *Automated Test Case Generation with SMT-Solving and Abstract Interpretation*. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *Nasa Formal Methods, Third International Symposium, NFM 2011*, NASA, Springer LNCS 6617, Pasadena, CA, USA, pp. 298–312, doi:10.1007/978-3-642-20398-5_22.

[41] (2015): *The TLA Toolbox*. `http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html`.

[42] Marcel Verhoef (2009): *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Ph.D. thesis, Radboud University Nijmegen.

[43] Marcel Verhoef, Peter Gorm Larsen & Jozef Hooman (2006): *Modeling and Validating Distributed Embedded Real-Time Systems with VDM++*. In Jayadev Misra, Tobias Nipkow & Emil Sekerinski, editors:

FM 2006: Formal Methods, Lecture Notes in Computer Science 4085, Springer-Verlag, pp. 147–162, doi:10.1007/11813040_11.

[44] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa & S. Perry (2012): *Features of CML: a Formal Modelling Language for Systems of Systems*. In: *Proceedings of the 7th International Conference on System of System Engineering*, IEEE, doi:10.1109/SYSoSE.2012.6384144.

[45] Xtext (2015): *Xtext*. Available at `https://eclipse.org/Xtext/`.

12

Principles for Reuse in Formal Language Tools

The paper in this chapter has been accepted as a peer-reviewed conference paper.

[P28] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Kenneth Lausdahl. *Principles for Reuse in Formal Language Tools*. 31st ACM Symposium on Applied Computing (SAC 2016), April 2016.

13

LPF-Aware Proof Obligation Generation in VDM/Overture

The paper in this chapter is planned for submission to a peer-reviewed conference.

[P21] Luís Diogo Couto, Nick Battle and Peter Gorm Larsen. *LPF-Aware Proof Obligation Generation in VDM/Overture*. To be submitted to the 5th International ABZ Conference (ABZ 2016), May 2016.

LPF-Aware Proof Obligation Generation in VDM/Overture

Luís Diogo Couto[1], Nick Battle[2], and Peter Gorm Larsen[1]

[1] Aarhus University, Department of Engineering
{ldc,pgl}@eng.au.dk
[2] Fujitsu UK
nick.battle@uk.fujitsu.com

Abstract. For specification languages such as VDM, the presence of union types and invariants makes type-checking statically undecidable and thus insufficient to ensure consistency of specifications. VDM uses a 3-valued logic and undefinedness is dealt with by automatically generating Proof Obligations (POs) that, once discharged, ensure internal consistency of a specification, i.e. undefined values do not occur. The Overture/VDM tool has a Proof Obligation Generator (POG) but until now it has used McCarthy's 3-valued logic. This follows the left to right evaluation approach used by most programming languages. However, standard VDM uses the Logic of Partial Functions (LPF) with commutativity for logic operators that is convenient for carrying out proofs. This paper investigates the possibility of extending the existing Overture POG so that it generates POs that ensure consistency of the model under LPF. The work presented here should also be applicable to other formal methods that include McCarthy style generation of POs.
Keywords: VDM, LPF, Proof obligations, logics, tool support

1 Introduction

Dealing with undefined values in the analysis of formal specifications has been an active area of research for many years, with different semantic approaches [1]. For most computer-based languages the expressiveness of the language makes it impossible for static type checking to determine whether all terms are defined. In order to ensure that terms denote it is possible to automatically generate *Proof Obligations (POs)* that, if dischargeable, guarantee this. In this paper the difference in POs generated due to different three-valued semantic models (McCarthy [2] versus Kleene [3]) is explored using VDM.

Proof Obligation Generators (POGs) [4,5] have been developed for both major VDM tools: VDMTools [6] and Overture [7]. These POGs use the left-to-right McCarthy logic, the same as the interpreters for the executable subsets of VDM [8,9]. However, the formal semantics of VDM [10] uses Logic of Partial Functions (LPF) [11,12,13] which is based on the Kleene semantics with commutativity for logic operators. From a proof perspective this introduces a disconnect between the model and the obligations being discharged since the semantics of the two are not identical. Therefore, it

is worthwhile considering whether it is possible to provide an option for the POG to accommodate LPF semantics.

Many other formal methods also have the notion of proof obligations although some of them give them different names. For example in the Prototype Verification System (PVS) they call them Type-Correctness Conditions (TCCs) [14]. Here definedness predicates are also used as guards in exactly the same way as is done for the existing Overture POG using the McCarthy logic. Another example is Event-B where proof obligations are primarily used to verify properties of a model [15].

In this paper we use VDM both for the object level and for the meta level languages. The object language will be used to present examples and sources of undefinedness. The meta-language applies to objects and will describe definedness conditions and behaviour of the POG. so in order to distinguish them we use the mathematical VDM syntax –$\forall x : T_x \cdot P(x)$– for the meta-level and the ASCII syntax –`forall x : T(x) & P(x)` – for the object-level[3].

The next section provides the basic background information about three-valued logics to enable the reader to understand the rest of the paper. Afterwards, Section 3 introduces examples of POs for both logical approaches. The main contributions of this paper are discussed in Section 4, where we explore the new option for generating POs using the LPF approach that has been implemented in the Overture POG as a prototype. Afterwards Section 5 presents future work which could take the results from this paper further. Finally, Section 6 provides concluding remarks.

2 Three-Valued Logics

Partial operators such as division exist in most computer-based languages. For all partial operators it is possible to write a definedness predicate that will ensure safe use of the operator. If for example `x` is used as a divisor that definedness predicate is "`x <> 0`" whereas if we lookup the `n`'th element in a sequence `list` by writing "`list(n)`" the definedness predicate would be "`n in set inds list`" to ensure that `n` is a valid index in the sequence.

McCarthy logic evaluates expressions from left-to-right and if the value of a logical expression can be determined by a subexpression, the evaluation of the remaining subexpression(s) is not carried out. This kind of short-circuiting strategy is used in most programming languages and also in the current versions of the VDM interpreters. This means that users need to "guard" the use of partial operators by writing a "guarding expression" on the left-hand-side. Thus if one used for example division by a variable "`x`" then one would write "`x <> 0 and ...`" to guard it. From a user perspective this can be useful, but for proving properties about specifications it would be simpler if there was more symmetry in the evaluation of logical operators. This is symmetry is not present in McCarthy logic and that can be seen for example from the truth table comparing McCarthy logic and LPF for disjunction (where $\perp$ indicates undefined), shown in Table 1:

[3] Note that `T(x)` and T_x indicate the type information of `x` and x, including invariant restrictions, in the object and meta-level languages respectively.

A	B	A **or** B (McCarthy)	A **or** B (Kleene)
true	**true**	**true**	**true**
true	**false**	**true**	**true**
true	⊥	**true**	**true**
false	**true**	**true**	**true**
false	**false**	**false**	**false**
false	⊥	⊥	⊥
⊥	**true**	⊥	**true**
⊥	**false**	⊥	⊥
⊥	⊥	⊥	⊥

Table 1: Truth table for OR operator in McCarthy and Kleene logics

where the row highlighted in grey shows where LPF differs from McCarthy logic. Similar differences can be seen in the truth tables for the other logic operators and their quantified generalisations. It is natural to expect that fewer proof obligations would be generated if one uses LPF but it is also clear that the nature of POs generated will be different. We explore these issues further in the rest of this paper.

3 Proof Obligations in McCarthy and Kleene Logics

The current Overture POG generates POs that guard against inconsistency in a VDM specification. It does so by constructing a series of definedness predicates (in the form of VDM boolean expressions) that ensure undefined values do not occur and thus ensuring the consistency of the specification.

The POG generates various different kinds of obligation, each of which checks for a particular kind of inconsistency. Fundamentally, a PO will consist of the definedness predicate that guards against that particular inconsistency and the necessary scoping information (such as quantification of variables) to allow the predicate to be discharged.

This PO-driven approach has several benefits, particularly from a user's perspective as parallels can be established between POs and "potential problems" in a specification. As such, POs can be used as a quick check of the potential inconsistencies present in a specification. Further, manual inspection of POs may yield useful insights into the specification and guide the user's work on it.

However, because the POG is driven by POs, care must be taken to ensure that the obligations properly cover all possible inconsistencies. To achieve this, one must go through all VDM elements and identify all possible sources of inconsistency. Typically, a source of inconsistency corresponds to a possibly undefined value in a VDM element. These sources of inconsistency are then related to a definedness predicate. Given a possibly undefined element E, we utilize the notation $\delta(E)$[16] as a meta-operator to refer to the predicate that ensures the definedness of E. The compilation of a reference document for all undefined elements is ongoing in Overture and an example of it can be seen in Figure 1.

Division by Zero
Description: Division is undefined for divisors of value 0. A PO must guard against this by forcing the divisor to be different from 0.
Occurrences: Any VDM expression representing a division: either `x / y` or `x div y`, no matter where it occurs.
Definedness: δ(e/x) $\iff$ `x <> 0`

Fig. 1: Inconsistency Reference for Division by Zero

It should be noted that some elements of VDM are undefined under similar circumstances. For example, the **hd** (head) and **tl** (tail) sequence operators are both undefined for empty sequences. Because multiple inconsistencies can arise from the same kind of source, they can be guarded against with the same kind of obligation and definedness predicate.

This PO-driven approach also influences reasoning about the POG itself, which is frequently done in terms of missing obligations, incorrect obligations and so forth. While this can be an intuitive way of reasoning, one must be careful to ensure that all relevant aspects of PO generation are considered, particularly the coverage of inconsistency sources.

As stated in Section 1, the current version of the Overture POG (henceforth referred to as the McCarthy POG) is implemented according to McCarthy's 3-valued logic [2] because the interpreter also uses that logic and the POG is used, in part, to protect against interpretation errors in specification execution.

Due to its choice of logic, the McCarthy POG must address each inconsistency source independently. So, for each source of inconsistency, a distinct PO will be generated. For example, the VDM function shown in Listing 1, taken from [17], yields two separate POs, as shown in Listing 2.

Listing 1: A VDM function with two sources of inconsistency.

```
f: int -> bool
f(x) == x/x = 1 or (x+1)/(x+1) = 1
```

Listing 2: Two non-zero proof obligtations.

```
PO1: (forall x:int &
       x <> 0)

PO2: (forall x:int &
       (not ((x / x) = 1) =>
         (x + 1) <> 0))
```

The reasoning behind the first PO is clear. An occurrence of division by `x` introduces a possibly undefined value that must be guarded against. `PO2` is similar but there is an important distinction: the first part of the implication, which corresponds to the negated left hand side of the body of `f`. In essence, `PO2` is only relevant if the first part of

the disjunction in the body of `f` is false because otherwise the right side will not be evaluated. When generating `PO2`, the POG takes into account the McCarthy-style left-to-right evaluation of the expression. It is also worth noting that both POs have the same kind of definedness predicate (`x <> 0`), which is natural since they both protect against the same kind of inconsistency (division by zero).

Neither PO in Listing 2 can be discharged. For example, for `PO1` it is not possible to prove that an arbitrary integer is different from zero. The only way to discharge these POs would be to expand the specification with guards such as the ones shown in Listing 3. This would be obfuscating the specification with additional checks that are unrelated to the behavior and properties of the specification but simply help ensure run-time consistency.

Listing 3: A VDM function with guards on its sources of inconsistency.

```
f_guards: int -> bool
f_guards(x) ==
  (x <> 0 => x/x =1) or (x+1 <> 0 => (x+1)/(x+1) = 1)
```

The function in Listing 3 would indeed yield POs that can be discharged but a cost has been paid in the form of the guard conditions. Thus, it could be interesting if one can avoid such guards. Particularly if the only reason for their existence is the current choice of logic, but still be able to generate POs that can be successfully discharged. The way to achieve this is by utilizing the LPF logic.

The primary difference between the LPF and McCarthy lies in operators that combine logical expressions. In McCarthy logic, expressions are evaluated left to right with short-circuiting and that affects the undefinedness of composite logical expressions. It is for this reason that the McCarthy POG must take a stepwise approach. Each subexpression must be processed individually and each source of undefinedness must be addressed by itself with a dedicated PO that protects against it. The connection between subexpressions in a composite expression is then handled with contexts that restrict the variables of the subsequent POs (note again how this connects with left to right evaluation).

To continue our illustration, consider once again the function in Listing 1. In the McCarthy POG, this yielded two obligations (see Listing 2), even though it contains a a single expression. However, if the function is analyzed with the LPF POG, a single PO would be produced, as shown in Listing 4.

Listing 4: LPF proof obligation.

```
(forall x:int &
   ((((x / x) = 1)
   or (((x+1) / (x+1)) = 1) )
   or ((x <> 0) and ((x+1) <> 0))))
```

For LPF, there is only one obligation that is somewhat more complex than either of the two McCarthy POs and that can be discharged. This is because, in the LPF POG, the definedness predicate is generated for the entire disjunction rather than simply for each division by zero subexpression. This is the main advantage of LPF: there are more

sophisticated ways to specify definedness and the ability to do so for more elements of VDM. In the case of simple expressions (such as a single `x/x` expression) the LPF POG behaves in the same manner as the McCarthy POG. However, for composite logic expressions, the truth tables for each operator can be used to enforce the definedness of the entire expression.

An LPF definedness predicate is shown in the PO in Listing 4. In LPF, a disjunction is defined if either of its operands is true or if both are defined. Therefore, the definedness predicate must enforce that. The sub-predicate `x<>0 and (x+1)<>0` has an obvious mapping to the 2 McCarthy obligations shown in Listing 2 but the other clauses are new.

It is worth noting that, in a McCarthy setting, the evaluation of the PO shown in Listing 4 could be problematic as it may involve undefined calculations. The solution is to discharge these LPF obligations with a proof tool that also supports LPF. Regardless, the result of the LPF POG is a PO that can potentially be discharged without the need for guards. However, to further reinforce this point, imagine now that the function is altered in the manner shown in Listing 5.

Listing 5: A VDM function with one source of inconsistency.

```
functions
f : int -> bool
f (x) == 1/x = 1 or true
```

Now there is only one source of inconsistency and the McCarthy POG generates just one obligation, shown in Listing 6 but the same problem as before remains: the obligation cannot be discharged.

Listing 6: Non-zero proof obligation for revised function.

```
(forall x:int & x <> 0)
```

However, the LPF POG has a more interesting output, shown in Listing 7. It is not possible to prove `x<>0`, but **`true`** can certainly (and trivially) be proven.

Listing 7: LPF Proof Obligation for revised function.

```
(forall x:int &
    (((((1 / x) = 1) )
    or (true))
    or (x <> 0)))
```

The example function in Listing 5 is certainly artificial but it illustrates the larger point: LPF has more sophisticated ways of dealing with undefinedness. If that sophistication is properly leveraged, it is possible to generate dischargeable obligations in situations where otherwise it would not be.

4 Generating Proof Obligations

The difference between the LPF and McCarthy approaches lies in the POG execution for composite expressions. In particular, the approaches differ in how they process con-

texts. Thus, the notion of contexts needs a careful examination. Most VDM elements have the potential to be extracted as contextual information. In essence, context information is used by the POG to incorporate any restrictions on values present at any given point in the specification, i.e. what context an expression shall be considered in. For example, when analyzing the **`else`** clause of an **`if then else`** expression, the context information will tell the POG that the test condition for the **`if`** expression is false.

As the POG traverses a VDM specification, contextual information is collected as necessary. Then, once an inconsistency is found, any relevant contextual information is combined with the definedness predicate. In the example above, the left-hand-side of the disjunction was extracted as contextual information and when a PO was generated for the right-hand-side, the context was prepended to the definedness predicate, as an implication.

A VDM element may be a source of both context and inconsistency so the flow of execution for the POG is to analyze an element, generate any relevant obligations and afterwards, add the element to the context, if appropriate.

In addition to being used to constrain the values of variables, contexts are used for another purpose: construction of scoping information. When the POG encounters a source of inconsistency, it is only aware of that particular VDM element so the actual PO that is produced at that point will simply be the definedness predicate guarding against the inconsistency (for example `x` **`<>`** `0`). However, in order for the predicate to be valid and dischargeable, its variables must be quantified and contexts are used for this. Whenever the POG encounters an element that introduces variables (such as in a function declaration), context information is extracted to quantify the introduced variables in any subsequent obligations variables (for example, in the case of a function declaration this takes the form of a universal quantification over the arguments of the function and their respective types).

When the inconsistency source is encountered and the definedness predicate extracted, the context information will be prepended to the predicate to form the final PO. So given a definedness predicate $\delta(\overline{x})$, and contextual information of the form $\forall \overline{x} : T_{\overline{x}} \cdot$, the final PO will be $\forall \overline{x} : T_{\overline{x}} \cdot \delta(\overline{x})$. Note that the capture of variables is intentional as it gives meaning to the final PO and ensures that it is dischargeable

In order to ensure that context generation is correct in Overture, a reference for contexts is maintained, similar to the work carried out for inconsistencies. This reference relates every VDM element with any restricting or scoping contexts it extracts [18]. An example of such a reference is shown in Figure 2. Note that handling of context extraction at top-level (e.g. for functions) is exactly the same for McCarthy and LPF POGs.

4.1 LPF Extension

In this section, we present the main contribution of our work: generation of VDM POs in an LPF context and how this changes the behavior of the POG compared to the McCarthy version. The fundamental change in the generation of LPF POs lies in how composite boolean expressions are manipulated. This means that the behavior of the POG must be altered when it is applied to elements such as:

Explicit Function Definitions

Description: Function parameters (and return values) must be introduced in the context in order for any POs resulting from the function body to be valid.

Type: Scoping

Forms to prepend:

- For parameters `x1`, `x2`, ... "**`forall`** `x1:T(x1), x2:T(x2) ... &`" should be prepended to the definedness predicate.
- If the function has a return value that is referred to in the post condition, then "**`exists`** `r : T(r) &`", where `r` stands for the result value, should also be prepended.

Fig. 2: Context Reference for Explicit Function Definition

- **`and`** binary expressions
- **`or`** binary expressions
- **`=>`** binary expressions
- **`forall`** quantified expressions
- **`exists`** quantified expressions
- **`if then else`** and **`cases`** expressions

For any composite expression, the McCarthy version of the POG will process the left subexpression, then generate any relevant context information, and use it while processing the right subexpression. This flow of execution maps directly onto the left to right stepwise evaluation of expressions and is shown below for the function[4] that processes disjunctions:

$$PogMcCarthy\_OR : Exp \times Ctxt \to PO\text{-set}$$

$$\begin{array}{l} PogMcCarthy\_OR\,(\mathsf{mk\_}(l, \mathrm{OR}, r), c) \triangleq \\ \quad \mathsf{let}\ newc = MakeImpliesCtxt\,(l, c), \\ \qquad lpos = PogMcCarthy\,(l, c)\ \mathsf{in} \\ \quad lpos \cup PogMcCarthy\,(r, newc); \end{array}$$

$$MakeImpliesCtxt : Exp \times Ctxt \to Exp$$

$$\begin{array}{l} MakeImpliesCtxt\,(e, c) \triangleq \\ \quad \mathsf{let}\ imp = \mathsf{mk\_}(e, \mathrm{IMP}, \mathrm{NIL}), \\ \qquad \mathsf{mk\_}(ce, op, \mathrm{NIL}) = c\ \mathsf{in} \\ \quad \mathsf{mk\_}(ce, op, imp); \end{array}$$

The treatment of individual expressions and generation of definedness predicates (the *pogUnaryExp* function) does not change as the sources of inconsistency remain

[4] The *PogMcCarthy* function simply dispatches the input to the appropriate sub-function.

the same. The difference lies in the treatment of composite expressions. When a composite expression is analysed, the LPF POG will construct a definedness predicate for the entire expression rather than processing it in a stepwise fashion. Because of this, subexpressions are never added to the context in the LPF POG They are instead analysed and any relevant definedness predicates are produced. This can be seen below (note that absence of context generation at the expression level):

$$
\begin{array}{l}
PogLPF\_OR : Exp \times Ctxt \xrightarrow{\sim} PO\text{-set} \\
PogLPF\_OR\,(\mathsf{mk\_}(l, \mathrm{OR}, r), c) \triangleq \\
\quad \mathsf{let}\ popreds = PogLPF\,(l) \cup PogLPF\,(r)\ \mathsf{in} \\
\quad \mathsf{if}\ popreds \neq \{\} \\
\quad \mathsf{then}\ [MakeOrLPF\,(l, r, popreds, c)] \\
\quad \mathsf{else}\ \{\}
\end{array}
$$

The actual generation of an LPF PO for any given operator is fairly straightforward. In general, all operators follow the same pattern: either a subset of the subexpressions has a specific truth value or all subexpressions must be defined. Therefore, one must generate predicates that force whatever particular truth values are needed (these are typically constant for each operator). One must also generate the definedness predicates for all subexpressions and combine them, typically by means of a conjunction (the $chain$ function). The entire process of generating a PO is shown (for the **`or`** operator) below[5]. Essentially this is the construction of the POs in a disjunctive normal form.

$$
\begin{array}{l}
MakeOrLPF : Exp \times Exp \times PO\text{-set} \times Ctxt \xrightarrow{\sim} PO \\
MakeOrLPF\,(l, r, preds, c) \triangleq \\
\quad \mathsf{let}\ lf = makeEquals\,(l, \mathsf{true}), \\
\quad\quad rf = makeEquals\,(r, \mathsf{true}), \\
\quad\quad alld = Chain\,(preds, \mathrm{AND})\ \mathsf{in} \\
\quad \mathsf{let}\ orpo = \mathsf{mk\_}(lf, \mathrm{OR}, \mathsf{mk\_}(rf, \mathrm{OR}, alld))\ \mathsf{in} \\
\quad AddScope\,(orpo, c);
\end{array}
$$

$$
\begin{array}{l}
Chain : Exp\text{-set} \times BOP \xrightarrow{\sim} Exp \\
Chain\,(s, op) \triangleq \\
\quad \mathsf{if\ card}\ s = 1 \\
\quad \mathsf{then\ let}\ e \in s\ \mathsf{in} \\
\quad\quad e \\
\quad \mathsf{else\ let}\ e \in s\ \mathsf{in} \\
\quad\quad \mathsf{mk\_}(e, op, Chain\,(s \setminus \{e\}, op));
\end{array}
$$

$$
\begin{array}{l}
MakeEquals : Exp \times Exp \xrightarrow{\sim} Exp \\
MakeEquals\,(e1, e2) \triangleq \\
\quad \mathsf{mk\_}(e1, \mathrm{EQUALS}, e2);
\end{array}
$$

[5] The $addScope$ function simply adds quantifiers over any variables introduced — the scoping context functionality is retained.

4.2 Handling the Remaining Logical Operators

For the remaining logical operators in VDM, we will not go into as much detail as we did for disjunction. However, in general, what has been presented in the previous subsection also applies to the remaining logic operators. These are shown in Table 2 along with their respective definedness predicates. We omit any discussion of the negation operator as it is the same in both versions of the POG since the definedness of negation is identical in McCarthy and LPF.

logic expression	LPF Definedness Predicate
$\delta($`P` **`and`** `Q`$)$	$\neg P \vee \neg Q \vee (\delta(P) \wedge \delta(Q))$
$\delta($`P` **`or`** `Q`$)$	$P \vee Q \vee (\delta(P) \wedge \delta(Q))$
$\delta($`P` **`=>`** `Q`$)$	$\neg P \vee Q \vee (\delta(P) \wedge \delta(Q))$
$\delta($**`forall`** `x : T(x) & P(x)`$)$	$(\exists x : T_x \cdot \neg P(x)) \vee \forall x : T_x \cdot \delta(P(x))$
$\delta($**`exists`** `x : T(x) & P(x)`$)$	$(\exists x : T_x \cdot P(x)) \vee \forall x : T_x \cdot \delta(P(x))$

Table 2: Logical expressions and their LPF definedness predicates.

The conjunction operator (**`and`**) is the dual of the disjunction operator (**`or`**) shown before. So, whereas a disjunction holds if either of its members is true, a conjunction is false if either of its operands is false. Extending this to LPF and considering the truth tables, a disjunction is defined if either of its operands is false or if all of the operands are defined. The definedness predicate for conjunction is shown in row 1 of Table 2.

The implication operator (=>), unlike disjunction and conjunction does not have an absorbing element and therefore we cannot apply the LPF extension directly. However an implication can be unfolded to a disjunctive normal term in the following manner: $P \Rightarrow Q \equiv \neg P \vee Q$. From there, it follows that the definedness predicate for the implication operator is as shown in row 2 of Table 2.

The universal and existential quantifiers are generalizations of the conjunction and disjunction operators respectively so the rules for handling them will also be generalizations of rules for dealing with conjunction and disjunction. A universally quantified expression will be defined if it is **`false`** for at least one of its values or if it is defined for all values. We write its definedness predicate as shown in row 3 ofTable 2. Conversely, the rule for existential quantifiers is an extension of the rule for disjunction where just one of the expressions is **`true`**. Its definedness predicate is shown in row 4 of Table 2.

However, when dealing with quantified expressions there is another issue that must be consider: that of the quantifier bindings themselves. Take, for example, the following predicate: **`forall`** `x` **`in set`** `{y | y:` **`int`** `& y >0} & P(x)`. The POG will generate an obligation to ensure that the binding set is finite since the semantics of VDM does not allow infinite sets as values[6]. That is one of many possible

[6] Please note that this finiteness restriction does not apply to the POs themselves. Only to values in the specification so that it may be executed.

inconsistencies in the bindings of a predicate[7]. Due to this, for the McCarthy POG, bindings are the first element of a quantified expression to be analyzed. Afterwards, they are added to the context scope and the predicate of the quantified expression is analyzed. When going from a McCarthy POG to an LPF-based one the handling of the bindings is unchanged. In the semantics of VDM, a quantified expression is considered undefined if its bindings are undefined [10].

We also consider the case of nested binary operators. In the McCarthy POG, these are not particularly interesting as the operators (and their subexpressions) are simply processed left to right. Any relevant POs are generated as normal and context information (of the restricting kind) is generated and prepended to each definedness predicate as necessary. This has the effect that, further to the "right" an inconsistency is, the more complex the resulting PO shall be.

Nested operators are handled differently by the LPF POG. In LPF sub-expressions are not treated separately. Thus for an expression with nested operators, a single PO will still be generated. The difference lies in the definedness condition which will now apply to more complex operands. As an example, for A **and** B **and** C the definedness predicate will be $\neg A \vee \neg(B \wedge C) \vee \delta(A) \wedge \delta(B$ **and** $C)$. Of course, $\delta(B$ **and** $C)$ can be unfolded to: $\neg B \vee \neg C \vee \delta(B) \wedge \delta(C)$. In that sense, there is still a degree of left to right processing taking place. However, this is only in the PO generation phase. At the proof stage, the PO may be manipulated in such a way that, for example, proving $\neg C$ is sufficient to discharge the whole PO. In a McCarthy setting, with multiple POs, this simple proof would not be sufficient to discharge all of them. A direct comparison between the multiple McCarthy POs and the single LPF[8] PO can be seen in Table 3.

Expression	A **and** B **and** C
McCarthy	$\delta(A)$ $A \Rightarrow \delta(B)$ $A \Rightarrow B \Rightarrow \delta(C)$
Kleene	$\neg A \vee \neg(B \wedge C) \vee \delta(A) \wedge (\neg B \vee \neg C \vee \delta(B) \wedge \delta(C))$

Table 3: POG comparison for nested conjunctions.

4.3 Other Expressions

Finally, we consider non-logical operators such as the **`if then else`**. In general, the definedness of such elements is independent from the logical semantics being used. The definedness predicates for these expressions must be extracted directly from the VDM semantics. In the case of **`if then else`**, we say that it is defined if the test condition is defined and either the **`then`** or **`else`** clauses are defined [10], depending on the value

[7] Since one can utilize most VDM elements when specifying the bindings for a predicate, most kinds of inconsistencies can occur in bindings.

[8] Kleene refers to the semantic treatment of LPF that is used in VDM.

of the test condition: $\delta(\textbf{if}\ \text{A}\ \textbf{then}\ \text{B}\ \textbf{else}\ \text{C}) \iff \delta(A) \wedge (A \wedge \delta(B) \vee \neg A \wedge \delta(C))$. This is the treatment given to **if then else** expressions by the McCarthy POG so there is no change when moving to LPF. The **cases** expression is defined as a series of **if then else** expressions so it is also unchanged. As for other VDM elements, in general, if they are not defined in terms of logical operators, their treatment by the POG remains unchanged. When handling terminination of recursive functions [5] the POs are also the same in McCarth and LPF settings. Thus, the difference between the two POGs is really only with respect to the logical expressions.

5 Future Work

As a part of the COMPASS project [19][9] the POG for the CML language (containing both VDM and CSP elements) is being connected to the Isabelle theorem prover [20]. In the future we plan to examine the difference in proving the POs generated using McCarthy logic and the ones generated based on LPF logic by using this theorem prover. We believe that this will be rather straigthforward as its semantics is defined in Unified Theory of Programming (UTP) and both McCarthy and LPF logics have been incorporated already [21].

Another possibility is to use this work as the basis for exploring POs in VDM++ and the ways Kleene logic affects them. This becomes particularly challenging if one also considers aspects of inheritance and concurrency present in VDM++.

The main follow-up to this work is to enable LPF support in the interpreter for the executable subset of VDM. Since one of the primary uses of the POG is to ensure that runtime errors do not occur when interpreting a specification, the interpreter must utilize the same logic system as the POG. The work presented here is a good starting point for such improvements as it identifies the main points in VDM where LPF and McCarthy differ and what those differences are. When extending the interpreter to deal with LPF, one would also need to concurrently evaluate the multiple possibilities of an expression and then not report undefinedness errors until it is clear that none of them will be sufficient to yield a defined value. If a defined value is found, then the remaining possibilities must be discarded. This substantially increases the complexity of the interpreter and, since we would still like it to be deterministic. A preliminary possibility would be to wait until all evaluations terminate and then resort to a predetermined order when selecting which result to present. However, this approach may reduce the performance of the interpreter and would not be feasible for dealing with quantifiers over infinite sets.

6 Concluding Remarks

In this paper we have demonstrated that it requires fairly small adjustments to the Overture VDM POG to change the logic supported from McCarthy to LPF. With LPF there are "guarding expressions" which can be avoided as the example from Listing 1 demonstrates. The main advantage of supporting LPF is clearly that it is more powerful to use

[9] See also `http://www.compass-research.eu/` for information about this project.

LPF in proofs. In general fewer POs are generated using LPF logic. However, some of them are more complex and closer to the semantic definition of definedness in LPF. The main drawback of using LPF for the POG is that the Overture interpreter is currently not able to use LPF and thus that even though LPF PO generation and discharge ensures the consistency of a specification, its interpretation may still yield runtime errors and specification writers must be aware of this issue, particularly when executing elements of a model that contain expressions that are handled differently in LPF. Also, LPF POs are, in general, longer and more verbose than their McCarthy counterparts. In any case we hope that the work presented here may be valuable input for others who have existing pogs about the relatively minor adjustments needed to be able explore other logics supporting undefinedness.

Acknowledgments The work described in this paper is partially supported by the commission of the European communities Project Comprehensive Modelling for Advanced Systems of Systems (COMPASS, grant agreement 287829). The authors are grateful to their many collaborators in the project. We would also like to thank Joey Coleman for feedback on the work reported in this paper.

References

1. Łukasiewicz, J.: O logice trójwartościowej. Ruch Filozoficzny (1920) 169–171 Translated as (On three-valued logic) in Polish Logic 1920–39, S. McCall (ed.), Oxford U.P., 1967.
2. McCarthy, J.: A Basis for a Mathematical Theory of Computation. In: Western Joint Computer Conference. (1961) Then published in: Computer Programming and Formal Systems (P.Braffort, D.Hirstberg eds.) North Holland 1967, 33–70.
3. Kleene, S.: Introduction to Mathematics. North Holland (1952) Republished in 1957, 59, 62, 64, 71.
4. Aichernig, B.K., Larsen, P.G.: A Proof Obligation Generator for VDM-SL. In Fitzgerald, J.S., Jones, C.B., Lucas, P., eds.: FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997). Volume 1313 of Lecture Notes in Computer Science., Springer-Verlag (September 1997) 338–357 ISBN 3-540-63533-5.
5. Ribeiro, A., Larsen, P.G.: Proof Obligation Generation and Discharging for Recursive Definitions in VDM. In Song, J., Huibiao, eds.: The 12th International Conference on Formal Engineering Methods (ICFEM 2010), Springer-Verlag (November 2010)
6. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. ACM Sigplan Notices **43**(2) (February 2008) 3–11
7. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes **35**(1) (January 2010) 1–6
8. Larsen, P.G., Lassen, P.B.: An Executable Subset of Meta-IV with Loose Specification. In: VDM '91: Formal Software Development Methods, VDM Europe, Springer-Verlag (March 1991)
9. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In Qin, S., Qiu, Z., eds.: Proceedings of the 13th international conference on Formal methods and software engineering. Volume 6991 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer-Verlag (October 2011) 179–194 ISBN 978-3-642-24558-9.

10. ISO: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (December 1996)
11. Cheng, J.: A Logic for Partial Functions. PhD thesis, Department of Computer Science, University of Manchester (1986) UMCS-86-7-1.
12. Jones, C.B., Middelburg, K.: A typed logic of partial functions reconstructed classically. Technical Report 89, Department of Philosophy, Utrecht University (April 1993)
13. Fitzgerald, J.S., Jones, C.B.: The connection between two ways of reasoning about partial functions. Inf. Process. Lett. **107**(3-4) (July 2008) 128–132
14. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In Kapur, D., ed.: 11th International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Artificial Intelligence., Saratoga, NY, Springer-Verlag (June 1992) 748–752
15. Hallerstede, S.: On the purpose of event-b proof obligations. Formal Aspects of Computing **23**(1) (2011) 133–150
16. Jones, C.: Systematic Software Development Using VDM. Prentice-Hall (1986)
17. Jones, C., Lovert, M., Steggles, J.: A Semantic Analysis of Logics That Cope with Partial Terms. In Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E., eds.: Abstract State Machines, Alloy, B, VDM, and Z. Volume 7316 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 252–265
18. Couto, L.D., Larsen, P.G., Battle, N.: Overture Proof Obligations Reference Guide (in preparation). Technical Report TR-008, The Overture Project, www.overturetool.org
19. Fitzgerald, J., Larsen, P.G., Woodcock, J.: Modelling and Analysis Technology for Systems of Systems Engineering: Research Challenges. In: INCOSE, Rome, Italy (July 2012)
20. Foster, S., Woodcock, J.: Unifying Theories of Programming in Isabelle. In Liu, Z., Woodcock, J., Zhu, H., eds.: Unifying Theories of Programming and Formal Engineering Methods, International Training School on Software Engineering, Held at ICTAC 2013, Shanghai, China, 26–30 August, 2013, Advanced Lectures. Volume 8050 of Lecture Notes in Computer Science., Springer (2013) 109–155
21. Bandur, V., Woodcock, J.: Unifying theories of logic and specification. In Iyoda, J., de Moura, L.M., eds.: Formal Methods: Foundations and Applications, 16th Brazilian Symposium, SBMF 2013, Brasilia, Brazil, 29 September–4 October, 2013, Proceedings. Volume 8195 of Lecture Notes in Computer Science., Springer (2013) 18–33

14

Combining Harvesting Operation Optimisations using Strategy-based Simulation

The paper in this chapter is planned for submission to a peer-reviewed journal.

[P27] Luís Diogo Couto, Peter W. V. Tran-Jørgensen and Gareth Edwards. *Combining Harvesting Operation Optimisations using Strategy-based Simulation*. To be submitted to the International Journal of Computers and Electronics in Agriculture, 2016.

Combining Harvesting Operation Optimisations using Strategy-based Simulation

Luis Diogo Couto[1], Peter W. V. Tran-Jørgensen[1], and Gareth T. C. Edwards[2]

[1]Aarhus University, Department of Engineering , {ldc,pvj}@eng.au.dk
[2]Kongskilde , gtce@kongskilde.com

October 15, 2015

Abstract

Modelling and simulation can help with decision support or planning activities by allowing efficient exploration of multiple scenarios in a situation where testing in a real setting is impractical. This exploration is normally done by varying numerical parameters in the model such as physical dimensions or speed in order to find the optimal configuration. When calculating optimised routes for harvesters and other vehicles in a harvest operation, the choice optimisation algorithm is an important part of the problem, but traditional modelling and simulation techniques do not allow us to vary the algorithm across simulations as easily. In this paper the strategy pattern from the field of software engineering is used to address this problem. The strategy pattern allows experiments with different combinations of planning algorithms to be analysed effectively. This approach can be adopted in other planning activities where multiple algorithms need to be considered.

Keywords: Harvesting Operation; Optimisation; Strategy Pattern; Design patterns; Vienna Development Method; Modelling; Simulation

1 Introduction

There are various steps to calculating optimised solutions for harvest operations, including partitioning of the field and calculating optimised coverage plans for harvesters and route plans for other vehicles. One approach to the problem often involves the use of various optimisation algorithms that produce coverage plans for the harvesters [16, 3]. However, planning of harvester routes is just one part of the harvesting operation. Path planning for wagons (or other similar) that service the harvesters must often also be developed. Algorithms exist for optimising service plans [10]but they are independent from those of harvesters. This independence makes it difficult to explore in detail how the various types of algorithms interact and combine to produce a complete solution for the harvest operation.

As an example, little research has previously been conducted into how harvesting and loading algorithms can affect operational execution times of harvesting operations. Examples of planning tools for operations often employ a single algorithm; such as in-field unloading [14] or single point unloading [4]. Farmers will generally choose a plan with which they are familiar without considering alternatives.

In this paper, we seek to explore how different optimization algorithms can be combined. We will explore this using a model based on mathematical formalisms in combination with the strategy pattern from software engineering. The strategy pattern is used in the model to encode different optimization algorithms. A novel aspect here is that the strategies representing the different kinds of algorithms (har-

vest routing and wagon path planning) co-exist and collaborate to produce the final solution.

1.1 Background

From an operational research perspective, the harvest operation is an example of an output material flow (OMF) operation where material is removed from the field and transported to another location [1]. The machinery utilised within the OMF operation can be divided into two groups; Primary Units (PUs) which perform the main task i.e. harvesting the crop, and Service Units (SUs) which service the PUs by receiving harvested material and transporting it away. The capacity of the PU is many times smaller than the expected yield of the field, and therefore a PU unloads either to a nearby SU or directly or to an out of field storage point.

The planning of the tasks of the PUs and SUs are often considered separately [9], with coverage plans being developed for PUs [16, 3] and path plans being developed for SUs [10]. However, the tasks are spatially and temporally dependant on one another, so in order for efficient plans to be produced the plans must be developed concurrently [15].

To assist with the planning of in-field operations, fields can be decomposed in to a number of tracks or rows. Many methods have been proposed for the decomposition of fields [14, 11, 18, 8]. Fields are typically divided into headlands which encircle the field and can be used for turning and working rows which transect the main area of the field. By confining all field traffic to drive along these predefined rows, the trafficked area of the field can be limited which has been shown to produce benefits on increased yield and better soil structure [17].

In the above mentioned approaches, the planning for the various kinds of vehicles is performend indepdently as well the decomposition of the field. In our work, we consider all vehicles simultaneously when planning, although field decomposition is still done separately.

A different approach to optimisation was carried out in a EU project called DESTECS, using VDM to perform design space exploration by sweeping parameters of models of cyber-physical systems [6]. Among other things, the DESTECS project proposes methodological guidelines for modelling fault-tolerant cyber-physical systems, which also involve the use of the strategy pattern to model faulty behaviour as well guarding against it [2]. This is similar to the presented approach, in that the strategy pattern is used in the DESTECS project to explore different behaviours of a system. However, while the DESTECS project used the strategy pattern to make a system more fault-tolerant, in this work the strategy pattern is used to help find optimised solutions to use in a harvest operation.

The strategy pattern is a design pattern [7] with two key features. First, the strategy pattern allows selection of different algorithms to be done at execution time and; secondly, it defines a family of interchangeable algorithms. Essentially this allows one to execute the same functionality in different ways. Broadly speaking, the strategy pattern consists of a contract that defines the functions of a strategy in terms of their inputs and outputs including the properties that these functions may have. Given this contract, a specific strategy must provide an implementation of the function that obeys the input and output properties of the contract, but which is free to use whatever algorithms are desired.

The strategy pattern is based on object-oriented (OO) features [13], as enabled by the VDM++ formal modelling language [5]. VDM++ is the OO dialect of the Vienna Development Method (VDM). VDM is one longest-established formal methods for the development of computer-based systems. The method focuses on the development and analysis of a system model expressed in a formal language. Broadly speaking, a VDM++ model consists of a series of definitions for types, functions, operations, etc. The OO features of VDM++ allow for structuring the model into classes and provide standard OO mechanisms such as inheritance.

In addition to allowing for an effective implementation of the strategy pattern, the object-oriented features of VDM++ have other useful benefits, including the ability to add new versions of a strategy that reuse parts of an existing strategy by changing only those parts that must be different. Additionally, object-orientation facilitates modularity and en-

capsulation which, while not essential to develop the model, make it easier to do so.

The use of VDM++ promotes a high-level approach that abstracts away details that are of little importance to harvesting operations. The formal semantics underpinning the VDM language allow us to have confidence in the results and that there are no errors in the language and tool that can "contaminate" the result. Additionally, VDM has features that enable us to describe the properties of the model and its functions, and these properties are constantly checked during model execution. For example, in the model the capacity is expressed as a floating point number, which must always be positive and smaller than 1. VDM invariants allow us to attach such a property to the capacity variable in order to ensure that the model never violates this. While it is a simple example, VDM allows us to express any arbitrary property that can be described in terms of first-order logic.

1.2 Paper structure

The remainder of this paper is structured as follows: in section 2 we present the approach taken in our work: an executable formal model of the harvest operation based on the strategy pattern. Following that, in section 3, we report the results of applying the model to a case study of a real field. The results are then discussed in section 4. We conclude the paper in section 5.

2 Materials and Methods

2.1 Formal Model

The model was developed according to the structure shown in Figure 1.[1] The Execution Engine is responsible for coordinating the simulation and is connected to both the State and the three Strategy classes. The State contains the physical entities involved in the harvest operation. The harvesters are the PUs of the operation. Coverage plans and coordinated service points are developed for the harvesters by the employed strategies. The SUs are tractors with grain wagons whose main objective within the harvest operation is to convey material from the harvesters to the out-of-field storage. The service points coordinate when and where the SUs must meet the PUs in order for material to be passed between the two.

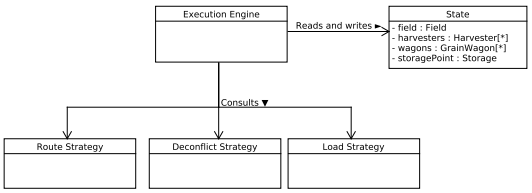


Figure 1: Model structure realised as a UML class diagram.

Both the harvesters and the grain wagons are modelled by their physical parameters such as their working/non-working speed, storage capacity and material offload rate. These parameters are specified in the initialisation of the model. The storage point is the out-of-field storage point where all material from the field must be transported to in order for the harvest operation to be completed. This too is modelled by using its capacity.

The strategy classes define how certain aspects of the harvest operation are executed. In Figure 1 these strategies are represented by the Route Strategy, Deconflict Strategy and Load Strategy classes, respectively.

2.1.1 Route strategy:

A route strategy is responsible for constructing the routes for harvesters. The routes direct the harvester from its location to a point where it will next require a service. A similar approach to the planning of route for harvester was also utilised in [14]. In this way the routes for multiple harvesters can be constructed in a consecutive manner.

As already stated the construction of routes for the harvester and grain wagon are dependent on one an-

[1] The complete formal model is available in the appendix and can be explored with the Overture tool, available from `http://overturetool.org/`.

other, therefore the route strategy must call functions from the loading strategy to ensure that the harvester is able to be serviced at the end of the route. The route strategies are allowed to produce more than one possible route for the harvester, these are later distinguished by the load strategy as appropriate.

Two route strategies have been implemented within the model: Predefined Route strategy and Greedy Route strategy.

The Predefined Route strategy enables the model to execute coverage plans that have been developed externally, provided they are represented as a sequence of rows to harvest. This strategy receives the assignment of a sequence of rows to a harvester as an input. A route is constructed which navigates the harvester along the sequence of rows, inserting service points where they are needed.

The Greedy Route strategy employees a search algorithm on the field to create a short route for the harvester which will end with the harvester being as full as possible and in a position where it can be serviced. An extra constraint is also implemented with the strategy that every row must be harvested in its entirety and that all headland rows must be harvested before work rows can be harvested.

2.1.2 Deconflict strategy:

A deconflict strategy is responsible for determining if a vehicle can move along its route, or calculating new routes if this is not possible. In the Simple Deconflict strategy a vehicle to reroute is chosen non-deterministically.

A deconflict strategy is responsible for the infield coordination of the vehicles. It is possible that conflicts can arise when a vehicle may block the path of another vehicle. In this case the deconflict strategy is employed to determine what course of action (such as planning a new route, or waiting for the obstruction to pass) is to be taken.

The Simple Deconflict strategy ensures that two vehicles cannot travel towards each other either along the same row or along two adjacent rows.

2.1.3 Load strategy:

A load strategy is responsible for assisting the route strategy to find a location where the harvester can be serviced and for constructing a route for the grain wagon from its current position to the service point and then to the out of field storage.

This is done through three functions of the load strategy that are called by the route strategy: `isDoneExtendingRoute()`, `isRouteServiceable()`, and `finaliseRoute()`.

`isDoneExtendingRoute()` checks if it is possible to extend a harvesters route. A common reason why it would not be possible to extend a harvester route is if there are no more remaining rows in the field to be harvested, or if the harvester is full.

`finaliseRoute()` modifies a harvesters route to ensure the final position of the harvester is valid. For example if harvesting the full length of the final row of a harvesters route will mean that the harvester will exceed its capacity, the route is modified so that the service point is required partially along the length of the final row.

`isServiceable()` checks that a grain wagon is able to converge on the service point that is required by the harvesters route, for example that there is a previously harvested row adjacent to the service point in which the grain wagon can drive.

Four different versions of the load strategy have been developed in the model. These cover the four basic ways in which harvesters are unloaded during grain harvests.

The Single Point Unload version requires the harvester to transport material directly to the out of field storage point without using a grain wagon. It is important that the harvester must avoid the event of becoming full without a navigable path to the out of field storage. This strategy limits the amount of traffic in the field, which could offer benefits when reducing soil compaction.

The Headland Unload version limits the grain wagon to only travelling in the headland areas of the field. The harvester must avoid becoming full in the middle of the field as a grain wagon would not be able to meet it, therefore service points must be coordinated before the harvester becomes full while it

is turning in the headland area.

The Infield Static Unload version allows the grain wagons to drive in the working areas of the field in order to meet the harvester. Service points are planned for the latest possible moment to ensure that the harvester is full when it passes its load.

The Infield Moving Unload version is similar to the Infield Static Unload strategy, however the harvester and the grain wagon are both moving when the load is being passed. As the machines remain in motion it is imperative that the grain wagon is travelling in the same direction as the harvester when they meet at the service point.

The Route, Load and Deconflict strategies are represented in Figure 1 by their contracts. The various concrete versions of each strategy must conform to this contract. Figure 2 shows how the various load strategies are realised based on the `ILoadStrategy` class that defines the contract. Whenever the model is executed, a concrete strategy of each kind must be provided to the Execution Engine.

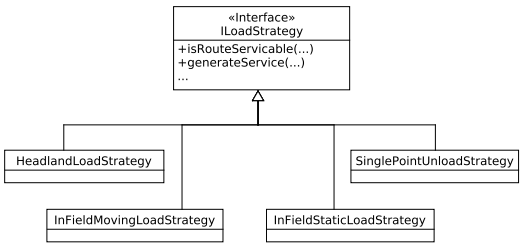


Figure 2: Load strategy hierarchy realised as a UML class diagram.

Not all versions of a strategy can be used in all situations. In order to cope with this, a notion of *strategy feasibility* has been introduced. The strategy feasibility check is implemented as a function in each of the strategies and invoked at the beginning of model execution in order to check if the field meets the requirements of the strategy configuration. The advantage of this approach is that the feasibility of each version of a strategy is encapsulated in that version itself, so the remaining parts of the model need not be aware of its specific details.

The concrete versions of strategies can be used to model different optimisation algorithms and therefore vary in implementation detail as well as the restrictions they impose on the harvest operation.

2.2 Model Execution

In order to execute the model, it is first necessary to configure the harvest operation by loading both the field and the resources, i.e. the State, and also one of each class of strategy to guide the Execution Engine during the simulation. Once this is done, the model is executed and whenever the Execution Engine reaches a point where it needs to make a decision that depends on a strategy, it will consult whatever strategy it has loaded and the output of the strategy will be used to further advance the model. As an example, consider Figure 3. In this figure, the Execution Engine needs to know what vehicles are movable at a given point in time. One particular version of the strategy may allow the harvesters to move because they can offload in the work rows. Another version may not allow the harvesters to move because they can only offload in the headlands and they cannot fully harvest the next work row.[2] In this way, different versions of a strategy lead to different outcomes in the model.

One of the key features of the model is the ability to explore strategy combinations and how their interactions affect the performance of the harvest operation. One way to do this is by fixing 2 kinds of strategies and varying the remainder (for example, load strategies) thus investigating how a particular aspect of optimisation affects the overall harvest operation. Conversely, if external restrictions dictate the use of a particular strategy, then the other strategies may be manipulated to find the best solution within the restrictions. For a small number of strategies, testing the different scenarios of interest can be done with manually written tests. However, when the number of scenarios to be tested is large then an automated combinatorial testing feature for VDM can be used

[2] In both of these examples, the route strategy consults the load strategy as part of its calculation of movable vehicles.

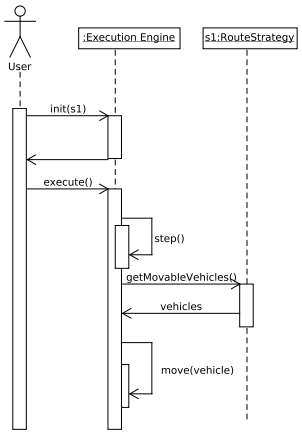


Figure 3: Load strategy hierarchy realised as a UML class diagram.

to concisely specify the various combinations and automatically generate and execute the corresponding tests [12].

2.3 Simulation Visualisation

As part of model execution a log of all the important events in the harvest operation is produced. Logged events include vehicle movement, harvesting of a row, passing load between harvesters and grain wagons, etc. Once execution is complete this log can be inspected in order to get a full understanding of the harvest operation outcome. This log can also be seen as a harvesting plan since it contains detailed instructions of when and where the different vehicles must go.

In order to better understand what occurred during the simulation, the log can also be analysed. However, as manual inspection of the log is difficult a proof-of-concept visualization tool was developed to analyse the log and replay the simulation as shown in Figure 4. The figure shows a representation of the field and movement of vehicles across the field. The status of the rows in the field is updated as the simulation progresses.

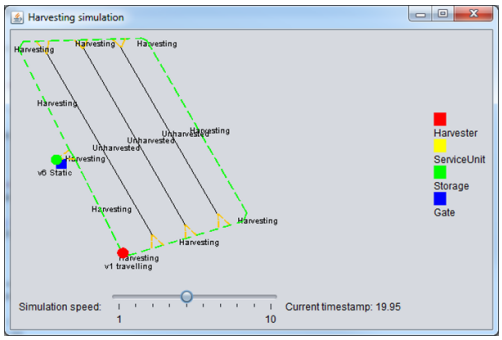


Figure 4: Simulation visualisation.

3 Results

This section demonstrates the approach by reporting results of executing various simulations with the model in order to explore the interactions between all possible combinations of the strategies described in subsection 2.1. Every execution was performed with the same resources and on the same field. The focus is not on changing the parameters of the simulation such as number of harvesters or harvester capacity but in changing the strategy versions used in each simulation.

The simulations were carried out on a representation of a real field located in the vicinity of the Research Center at Foulum, Denmark (56°29'N, 9°35'E).[3] The yield of the field is simulated and is lower for headland rows than for working rows, as is typical in real fields (due to excess soil damage, lower nutrients, etc.). The yield is further constrained such that a complete lap of the field can be made without exceeding the harvester capacity, and no single working row can exceed the capacity of the harvester. The field, partitioned into rows, is shown inFigure 5.

The results of the simulations are summarised in Table 1. Each row in the table represents a particular simulation, indexed by the *Sim.* (Simulation) column. The *Route* and *Load* columns identify the com-

[3]The model has been applied to representations of various other fields, both real and invented. However, these results are not reported here as the focus of this paper is on strategy interaction and not field analysis.

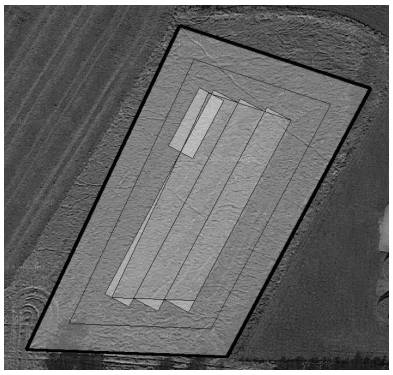


Figure 5: Agro Park field.

bination of strategies used in each particular simulation (the same deconflict strategy – Simple Deconflict – is used for all simulations). The *Op. Time* (Operational Time) column reports the duration of the harvest operation in seconds and serves as an indication of how well a combination of strategies performs. Finally the *Exec. Time* (Execution Time) column reports the actual, physical time in seconds it takes to execute the simulation.

The simulation was executed using a Java 7 code-generated version of the model on a Fujitsu LIFEBOOK U772 laptop with a 1.7GHz Intel Core i5 processor and 8Gb of memmory running a Windows 7 Professional Edition operating system.

4 Discussion

Table 1 shows that for the field subject to analysis, for most of the unloading strategies the *Greedy Route* strategy produces a better solution, than the *Predefined Route* strategy as indicated by the operational time. This is due to the harvesters route used as an input for the *Predefined Route* strategy being developed as a coverage plan that ignores the coordination of the service units. As the *Greedy Route* strategy was able to enquire the constraints of the unloading strategy while developing the harvesters route, the final solution is more integrated and allows for more efficient operations. This indicates that it may be advantageous to use optimisation approaches that consider both harvesters and service units when developing routes.

The *Infield Moving Unloading* strategy offers the best operational times for both of the routing strategies. This unloading strategy is likely to offer the best solution as it allows the harvester to be completely full when it offloads and does not require the harvester to stop. It is also worth noting that the model allows this hypothesis that the solution seems to be independent of route strategy choice to be confirmed. As further route strategies are added, it will be possible to continue checking if this still holds.

In terms of actual execution times, most combinations yield similar results for *Greedy* and *Predefined* strategies. The exception is for the *Single Point Unload* strategy, where the *Greedy* version has a significantly higher execution time. This is mostly due to the fact that many more routes have to be computed for this particular combination, which makes it significantly slower than its *Predefined Route* counterpart.

5 Conclusions

In this paper, it has been shown how optimisation algorithms for different aspects of the harvest operation encoded as strategies can be combined. This was achieved using a combination of the strategy pattern and formal modelling and simulation. The model can be executed with different strategy combinations, yielding harvest times that can be used to compare the combinations. More detailed analysis is also enabled by analysing a log file that is generated for each execution, and which contains all major events for that particular harvest.

The execution of the model has been demonstrated on a representation of a real field and a comparison for the field under analysis has been made based on the results for 8 strategy combinations.

These results can be generalised to other kinds of problems where there is a need to combine and compare multiple algorithms for the same operation, but where there is a significant amount of data and computation required in order to produce meaningful results.

Sim.	Route	Load	Op. Time [s]	Exec. Time [s]
1	Greedy	Headlands	425.558	12.619
2	Predefined	Headlands	497.38	13.417
3	Greedy	In Field Static	420.694	12.319
4	Predefined	In Field Static	463.484	13.912
5	Greedy	In Field Moving	410.298	7.056
6	Predefined	In Field Moving	446.854	7.25
7	Greedy	Single Point	679.498	26.977
8	Predefined	Single Point	623.347	4.421

Table 1: Results summary.

Looking forward, the work presented in this paper can be taken further by moving the harvester control to a distributed setting. The current version of the model assumes a global view of the harvest operation and directly controls the resources involved in it. In the future, we will take a distributed view of the harvest operation, where the resources involved must exchange information and coordinate with each other. To carry out this work we will take the model further by changing to a dialect of VDM called VDM-RT that extends VDM++ with support for modelling of distributed systems.

Acknowledgements

The work described in this paper was partially carried out in the context of the Danish High Technology Foundation research project Off-line and on-line logistics planning of harvesting processes. We would like to thank all our colleagues on the project for their valuable contributions and feedback, particularly Peter Gorm Larsen, Claus Grn Srensen, Dionysis Bochtis and Morten Bilde.

References

[1] DD Bochtis and CG Sørensen. The vehicle routing problem in field logistics part i. *Biosystems Engineering*, 104(4):447–457, 2009.

[2] Jan F. Broenink, John Fitzgerald, Carl Gamble, Claire Ingram, Angelika Mader, Jelena Marincic, Yunyun Ni, Ken Pierce, and Xiaochen Zhang. Methodological guidelines 3. Technical report, The DESTECS Project (INFSO-ICT-248134), October 2012.

[3] Gareth Edwards, Martin P Christiansen, Dionysis D Bochtis, and Claus G Sørensen. A test platform for planned field operations using lego mindstorms nxt. *Robotics*, 2(4):203–216, 2013.

[4] Gareth Edwards, Martin Andreas Falk Jensen, and Dionysis D Bochtis. Coverage planning for capacitated field operations under spatial variability. *International Journal of Sustainable Agricultural Management and Informatics*, 1(2):120–129, 2015.

[5] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[6] John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014.

[7] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.

[8] IA Hameed, DD Bochtis, CG Sørensen, Allan Leck Jensen, and Rene Larsen. Optimized driving direction based on a three-dimensional field representation. *Computers and electronics in agriculture*, 91:145–153, 2013.

[9] Martin Andreas Falk Jensen. *Operations planning for agricultural machinery under capacity constraints*. PhD thesis, Aarhus University, 2014.

[10] Martin Andreas Falk Jensen, Dionysis Bochtis, Claus Grøn Sørensen, Morten Rufus Blas, and Kasper Lundberg Lykkegaard. In-field and inter-field path planning for agricultural transport units. *Computers & Industrial Engineering*, 63(4):1054–1061, 2012.

[11] Jian Jin and Lie Tang. Optimal coverage path planning for arable farming on 2d surfaces. *Transactions of the ASABE*, 53(1):283, 2010.

[12] Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle. Combinatorial Testing for VDM. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, pages 278–285, Washington, DC, USA, September 2010. IEEE Computer Society. ISBN 978-0-7695-4153-2.

[13] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall International, 1988.

[14] Timo Oksanen and Arto Visala. Coverage path planning algorithms for agricultural field machines. *Journal of Field Robotics*, 26(8):651–668, 2009.

[15] Stephan Scheuren, Stefan Stiene, Ronny Hartanto, Joachim Hertzberg, and Max Reinecke. Spatio-temporally constrained planning for cooperative vehicles in a harvesting scenario. *KI-Künstliche Intelligenz*, 27(4):341–346, 2013.

[16] Mark Spekken and Sytze de Bruin. Optimized routing on agricultural fields by minimizing maneuvering and servicing time. *Precision agriculture*, 14(2):224–244, 2013.

[17] Jeff Tullberg. Tillage, traffic and sustainabilitya challenge for istro. *Soil and Tillage Research*, 111(1):26–32, 2010.

[18] Rodrigo S Zandonadi. *Computational Tools for Improving Route Planning in Agricultural Field Operations*. PhD thesis, University of Kentucky, 2012.

Bibliography

[1] Apache Velocity homepage. `http://velocity.apache.org/`. Accessed: August 2015.

[2] The ASTCreator tool homepage. `http://overturetool.org/astcreator/`. Accessed: 2015-08-21.

[3] Eclipse Equinox homepage. `http://www.eclipse.org/equinox/`. Accessed: September 2015.

[4] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[5] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[7] P Baudin, Jean-Christophe Filliâtre, Claude Marché, and B Monate. ACSL: ANSI/ISO C Specification Language. *CEA LIST and . . .* , 2008.

[8] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994.

[9] E Börger and R F Stärk. *Abstract State Machines: A Method for High-level System Design and Analysis ; with 19 Tables*. Springer Science & Business Media, 2003.

[10] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970.

[11] J.H. Cheng. *A Logic for Partial Functions*. PhD thesis, Department of Computer Science, University of Manchester, 1986.

[12] J.H. Cheng and C.B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. Woodcock, editors, *Proceedings of the Third Refinement Workshop*. Springer-Verlag, 1990.

[13] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[14] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2001.

[15] Paul Clements and Linda Northrop. Software product lines: practices and patterns. 2002.

[16] Joey W Coleman, Anders Kaels Malmos, Peter Gorm Larsen, Jan Peleska, Ralph Hains, Zoe Andrews, Richard Payne, Simon Foster, Alvaro Miyazawa, Cristiano Bertolini, and André Didier. COMPASS Tool Vision for a System of Systems Collaborative

Development Environment. In *Proceedings of the 7th International Conference on System of System Engineering, IEEE SoSE 2012*, pages 451–456, July 2012.

[17] Bernard Coulange. *Software Reuse*. Springer Science & Business Media, 2012.

[18] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA+ Proofs. In *18th International Symposium On Formal Methods-FM 2012*, volume 7436, pages 147–154. Springer, 2012.

[19] Luís Diogo Couto. Introducing the Overture Architecture Guide. In *Proceedings of the 12th Overture Workshop*, June 2014.

[20] Luís Diogo Couto. On Extensibility of Software Systems. Technical report, Department of Engineering – Electrical and Computer Engineering, Aarhus University, April 2014.

[P21] Luís Diogo Couto, Nick Battle, and Peter Gorm Larsen. LPF-Aware Proof Obligation Generation in VDM/Overture. In *5th International ABZ Conference*, May 2016. To be submitted.

[P22] Luís Diogo Couto, Simon Foster, and Richard Payne. Towards Verification of Constituent Systems through Automated Proof. In *Workshop on Engineering Dependable Systems of Systems (EDSoS)*, May 2014.

[P23] Luís Diogo Couto, Peter Gorm Larsen, Miran Hasanagić, Georgios Kanakis, Kenneth Lausdahl, and Peter W V Tran-Jørgensen. Towards Enabling Overture as a Platform for Formal Notation IDEs. In *Proceedings of the 2nd Workshop on F-IDE*, June 2015.

[P24] Luís Diogo Couto and Richard Payne. The COMPASS Proof Obligation Generator: A test case of Overture Extensibility. In *Proceedings of the 11th Overture Workshop*, 2013.

[P25] Luís Diogo Couto and Peter W V Tran-Jørgensen. Extending the Overture code generator towards Isabelle syntax. In *Proceedings of the 13th Overture Workshop*, June 2015.

[P26] Luís Diogo Couto, Peter W V Tran-Jørgensen, Joey W. Coleman Coleman, and Kenneth Lausdahl. Migrating to an Extensible Architecture for Abstract Syntax Trees. In *12th Working IEEE / IFIP Conference on Software Architecture*, May 2015.

[P27] Luís Diogo Couto, Peter W V Tran-Jørgensen, and Gareth Edwards. Combining Harvesting Operation Optimisations using Strategy-based Simulation. *Computers and Electronics in Agriculture*, 2016. To be submitted.

[P28] Luís Diogo Couto, Peter W V Tran-Jørgensen, and Kenneth Lausdahl. Principles for Reuse in Formal Language Tools. In *31st ACM Symposium on Applied Computing*, April 2016.

[29] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[30] Robert C. Daley and Jack B. Dennis. Virtual Memory, Processes, and Sharing in MULTICS. *Commun. ACM*, 11(5):306–312, May 1968.

[31] Marcel Dausend and Alexander Raschke. Introducing Aspect–Oriented Specification for Abstract State Machines. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 174–187. Springer, 2014.

[32] Marcel Dausend, Michael Stegmaier, and Alexander Raschke. Debugging Abstract State Machine Specifications: An Extension of CoreASM. In *Proceedings of the Posters and Tool Demo Session, iFM 2012 & ABZ 2012*, Pisa, Italy, 2012.

[33] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77(1-2):71–104, 2007.

[34] Roozbeh Farahbod, Vincenzo Gervasi, Uwe Glässer, and George Ma. CoreASM plug-in architecture. In *Rigorous Methods for Software Construction and Analysis*, pages 147–169. Springer, 2009.

[35] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[36] Simon Foster and Richard J Payne. Theorem Proving Support - Developers Manual. Technical report, COMPASS Deliverable, D33.2b, September 2013.

[37] Simon Foster, Frank Zeyda, and Jim Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *Unifying Theories of Programming*, pages 21–41. Springer, 2015.

[38] William B Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE transactions on Software Engineering*, (7):529–536, 2005.

[39] Etienne M Gagnon and Laurie J Hendren. SableCC, an Object-Oriented Compiler Framework. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '98, pages 140–154, Washington, DC, USA, 1998. IEEE Computer Society.

[40] E Gamma, R Helm, R Johnson, and R Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.

[41] Erich Gamma and Kent Beck. *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley Professional, 2004.

[42] Michael Gordon, Robin Milner, and Christopher P Wadsworth. Edinburgh LCF. *Lecture Notes in Computer Science*, 78, 1979.

[43] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer Berlin Heidelberg, 1995.

[44] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[45] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[46] Tony Hoare. *Communication Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.

[47] Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, April 1998.

[48] Claire Ingram, Richard Payne, John Fitzgerald, and Luís Diogo Couto. Model-based Engineering of Emergence in a Collaborative SoS: SysML & Formalism. In *Proceedings of INCOSE*, USA, 2015.

[49] Claire Ingram, Richard Payne, Simon Perry, Jon Holt, Finn Overgaard Hansen, and Luís Diogo Couto. Modelling patterns for systems of systems architectures. In *Systems Conference (SysCon), 2014 8th Annual IEEE*, pages 146–153. IEEE, 2014.

[50] ISO. *ISO/IEC 25010:2011,Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. ISO, 2011.

[51] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Heyward Street, Cambridge, MA02142, USA, revised edition, February 2012. ISBN-10: 0262017156.

[52] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.

[53] C. B. Jones and C. A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[54] Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981.

[55] Cliff B. Jones. Specification and Design of (Parallel) Programs. In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP, 1983.

[56] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.

[57] Peter W V Jørgensen, Luís Diogo Couto, and Morten Larsen. A Code Generation Platform for VDM. In *Proceedings of the 12th Overture Workshop*, June 2014.

[58] S.C. Kleene. *Introduction to Mathematics*. North Holland, 1952.

[59] Philippe B Kruchten. The 4+ 1 view model of architecture. *Software, IEEE*, 12(6):42–50, 1995.

[60] Charles W Krueger. Software Reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

[61] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[62] Jan Łukasiewicz. O logice trójwartościowej. *Ruch Filozoficzny*, pages 169–171, 1920. Translated as (On three-valued logic) in Polish Logic 1920–39, S. McCall (ed.), Oxford U.P., 1967.

[63] Andrian Marcus, Jonathan Maletic, et al. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 107–114. IEEE, 2001.

[64] Paolo Masci, Luís Diogo Couto, Peter Gorm Larsen, and Paul Curzon. Integrating the PVSio-web modelling and prototyping environment with Overture. In *Proceedings of the 13th Overture Workshop*, June 2015.

[65] J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Western Joint Computer Conference*, 1961.

[66] Stephan Merz and Hernán Vanzetto. Automatic verification of TLA+ proof obligations with SMT solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 289–303. Springer, 2012.

[67] Stephan Merz and Hernán Vanzetto. Harnessing SMT Solvers for TLA+ Proofs. In *12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)*, volume 53. EASST, 2012.

[68] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[69] Robin Milner, Lockwood Morris, and Malcolm Newey. A logic for computable functions with reflexive and polymorphic types. In *Proceedings of Conference on Proving and Improving Programs*, 1975.

[70] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[71] Jacob Porsborg Nielsen and Jens Kielsgaard Hansen. Development of an Overture/VDM++ Tool Set for Eclipse. Master's thesis, Technical University of Denmark, Informatics and Mathematical Modelling, August 2005.

[72] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[73] Daniel Novais and Eduardo Pessoa. Translating from VDM To Alloy. Technical report, Department of Informatics – University of Minho, 2015.

[74] O Owe. An Approach to Program Reasoning Based on a First Order Logic for Partial Functions. Technical Report 89. Technical report, Institute of Informatics, University of Oslo, 1985.

[75] D L Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Trans. Softw. Eng.*, 5(2):128–138, March 1979.

[76] Lawrence C Paulson. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In Renate A Schmidt, Stephan Schulz, and Boris Konev, editors, *Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010*, volume 9 of *EPiC Series*, pages 1–10, 2010.

[77] Gordon D Plotkin. Types and partial functions. *Post-Graduate Lecture Notes, Department of Computer Science, University of Edinburgh*, 1985.

[78] P van der Spek, N Plat, and C Pronk. Syntax Error Repair for a Java-based Parser Generator. *SIGPLAN Not.*, 40(4):47–50, 2005.

[79] J Woodcock, A Cavalcanti, J Fitzgerald, P Larsen, A Miyazawa, and S Perry. Features of CML: a Formal Modelling Language for Systems of Systems. In *Proceedings of the 7th International Conference on System of System Engineering*. IEEE, July 2012.