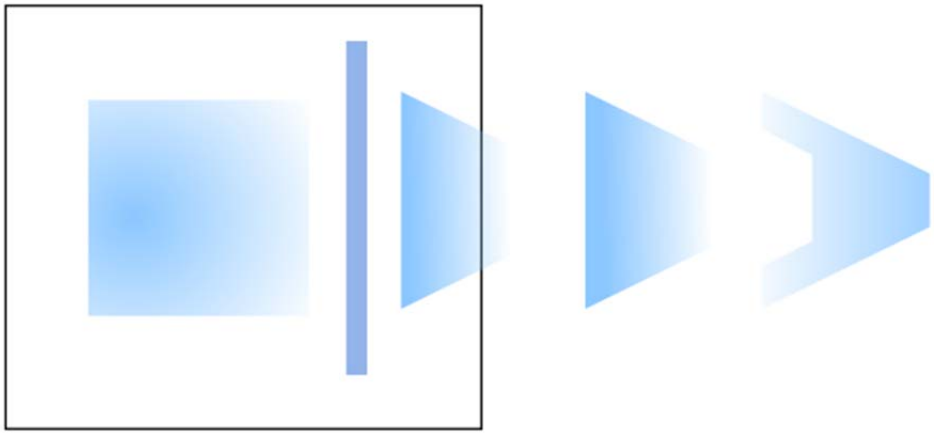
ON EXTENSIBILITY OF **SOFTWARE SYSTEMS**

Electrical and Computer Engineering
Technical Report ECE-TR-19



AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: On Extensibility of Software Systems

Subtitle: PhD Progress Report

Series title and no.: Technical report ECE-TR-19

Author: Luís Diogo Couto
Department of Engineering – Electrical and Computer Engineering, Aarhus University

Internet version: The report is available in electronic format (pdf) at the Department of Engineering website http://www.eng.au.dk.

Publisher: Aarhus University©
URL: http://www.eng.au.dk

Year of publication: 2014 Pages: 24
Editing completed: April 2014

Abstract: This report contains the progress report written as part of the author's PhD qualifying exam. It describes initial work carried out in analyzing and improving the extensibility of software systems, including a detailed case study analyzing the extensibility of the Proof Obligation Generator (POG) of the Overture tool. Additional extension work includes improving the output format of the POG and support for additional logic systems. Future work for the remaining half of the PhD is also discussed, including ways to combine formal modelling and extensibility analysis and also techniques for multi-paradigm extensibility.

Keywords: software engineering and systems

Referee/Supervisor: Peter Gorm Larsen, Joey W Coleman (co-supervisor)

Financial support: EU Project Comprehensive Modelling for Advanced Systems of Systems (COMPASS, grant agreement 287829)

Please cite as: Luis Diogo Couto, 2014. On Extensibility of Software Systems. Department of Engineering, Aarhus University. Denmark. 24 pp. - Technical report ECE-TR-19

Cover image: Luís Diogo Couto

ISSN: 2245-2087

ON EXTENSIBILITY OF SOFTWARE SYSTEMS

Luís Diogo Couto
Aarhus University, Department of Engineering

Abstract

This report contains the progress report written as part of the author's PhD qualifying exam. It describes initial work carried out in analyzing and improving the extensibility of software systems, including a detailed case study analyzing the extensibility of the Proof Obligation Generator (POG) of the Overture tool. Additional extension work includes improving the output format of the POG and support for additional logic systems. Future work for the remaining half of the PhD is also discussed, including ways to combine formal modelling and extensibility analysis and also techniques for multi-paradigm extensibility.

.

Table of Contents

Table of Contents i

List of Figures ii

Chapter 1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Main Theme . . . 1
- 1.3 Context . . . 2
- 1.4 Structure . . . 2

Chapter 2 Background Information 4
- 2.1 Systems of Systems and CML . . . 4
- 2.2 AST and Visitors . . . 4
- 2.3 Proof Obligations and Logics . . . 5
- 2.4 The Symphony Theorem Prover Plug-in . . . 6

Chapter 3 Initial Case Study 7
- 3.1 Introduction . . . 7
- 3.2 The COMPASS Proof Obligation Generator . . . 7
- 3.3 Reuse . . . 7
- 3.4 Conclusion . . . 9

Chapter 4 LFP PO Generation 10
- 4.1 Proof Obligations in McCarthy and Kleene Logics . . . 10
- 4.2 Generating Proof Obligations . . . 13
- 4.3 LPF Extension . . . 15

Chapter 5 AST-based Proof Obligations 17
- 5.1 Original PO Format . . . 17
- 5.2 Converting PO Strings to ASTs . . . 18

Chapter 6 Discussion and Further Work 19

Bibliography 21

A Publications 23

B Courses 24

List of Figures

Fig. 3.1 Sequence diagram representing a COMPASS POG visit 8

Fig. 4.1 Inconsistency Reference for Division by Zero 11
Fig. 4.2 Context Reference for Explicit Function Definition. 14

1

Introduction

1.1 Motivation

Nowadays, new pieces of software are often not built from scratch but rather built on top of existing ones by extending them. These extensions have many forms, from plug-ins that increase the functionalities of a tool to completely new tools that offer a different set of functionalities based around the original, or base, tool.

The notion of adding new functionality to software is quite old. In fact, software is often developed incrementally, with new features added over time. But we are interested in a particular kind of extension work, specifically the addition of new, *unplanned*, functionalities.

Being able to add new and unplanned features to software is very valuable as one saves a significant amount of time by reusing much of the existing base functionality (and its implementation).

However, in order for an extension to be successful, the base system must properly support the development of extensions. Extensibility is therefore the attribute that indicates *how well a system supports the development of new functionality.*

1.2 Main Theme

The main theme of this PhD is *software extensibility*, in particular the assessment of a system's extensibility and techniques to extend a system in new ways. Thus, the primary goals of the PhD are:

1: Assessment of Extensibility: take an existing system and evaluate its extensibility. The focus here is on an analysis that can be applied to a system with a minimum of preexisting knowledge.

2: Improvement of Extensibility: just as the previous goal finds problems with a system, this goal finds solutions to such problems. This will consist of techniques to modify the source of a system so that its extensibility is improved.

3: New Extensions: this, somewhat more speculative, goal will look to research extensibility techniques that differ from existing ones such as plug-ins, design patterns, etc.

1.3 Context

This PhD is carried out as a part of the EU FP7 research project Comprehensive Modelling for Advanced Systems of Systems (COMPASS, grant agreement 287829). From the project description[1]:

> The target of our research is the integration of well-founded engineering notations, methods and tools to support developers in building models of [Systems of Systems (SoS)] and analysing the global SoS-level properties of these models. These integrated techniques are intended to allow the comparison of alternative architectures and allocations of responsibilities to constituent subsystems.
>
> It is vital that the techniques we develop are accessible to a wide range of developers, so they must provide different levels of description, starting with a graphical view in a notation such as SysML that is easy for stakeholders to understand. We plan to extend SysML and link it to an underlying formal notation, built from established formalisms, extended with SoS-specific aspects. This level will be accessible to stakeholders trained in the formal notation, which we call the COMPASS Modelling Language (CML).

This PhD connects to COMPASS in two ways. First, an important part of the COMPASS project is tool support and the core of these tools will be developed as extensions of the Overture tool. Overture is an open source, plug-in based tool for Vienna Development Method (VDM) modelling[2]. This tool development work naturally presents opportunities to carry out extensibility research using Overture as a primary research subject.

A second contact point lies in the potential for new areas of new research by looking at the notion of extensibility in a Systems of Systems (SoS) context. There are potentially interesting subjects here such as a definition of SoS extensibility.

1.4 Structure

The remainder of this report is organized in the following manner:

Chapter 2 provides background information on various topics required throughout the report.

Chapter 3 presents the initial work carried out in the PhD: construction of a new Proof Obligation Generator (POG) for the COMPASS project as an extension to the existing Overture POG. This work served as an initial entry point into the area of extensibility and also as a way to gain familiarity with the Overture code base.

[1] taken from http://www.compass-research.eu/

[2] For more information about Overture see http://overturetool.org/

Chapter 4 describes extension work carried out on the Overture POG to provide support for an alternative method of generating proof obligations.

Chapter 5 describes work carried out on the Overture POG to alter its output into a more extension-friendly format.

Chapter 6 discusses the work planned for the second half of the PhD.

Appendix A lists publications written so far as part of the PhD.

Appendix B lists courses taken throughout the PhD.

2

Background Information

This chapter provides background information necessary for the remainder of the report. It will provide a more complete definition for extensibility and its various aspects. It will also present background information on other relevant topics (the Overture/Symphony tools, CML, the POG, etc.) needed for understanding the subsequent chapters of this report.

2.1 Systems of Systems and CML

Systems of Systems (SoS) are the primary focus of the COMPASS project. A SoS [1] is typically a large scale system composed of independent constituents that are themselves systems. Constituent systems often exhibit a very large amount of operational, structural and managerial independence. Constituents sometimes have opposing goals and may collaborate reluctantly to achieve the goals of the larger SoS of which they are a part.

One of the main outputs of the COMPASS project is the COMPASS Modelling Language (CML), a language developed purposefully for the modelling and analysis of SoS. Broadly speaking, a CML model consists of a collections of types, values, functions, classes, channels and processes. CML processes contain state and operations specified in the VDM and communicate with the exterior via Communicating Sequential Processes (CSP)-style synchronization.

The COMPASS project is also developing a tool – the Symphony tool – to support CML. Symphony is an extension of Overture that provides a complete new set of functionalities based on the Overture base. Symphony reuses large portion of the Overture User Interface (UI) code (which in turn relies on Eclipse) but also of certain core Overture modules such as the interpreter, type checker and the POG.

2.2 AST and Visitors

There are two important programming concepts that the Overture and Symphony implementations rely on: an Abstract Syntax Tree (AST) and the visitor design pattern [2].

The AST is responsible for providing a more abstract view of a VDM or CML model. It is an in-memory representation that can be analysed and manipulated to achieve the

goals of various plug-ins. The CML AST is an extension of the VDM one and both are created with a tool called ASTCreator. Both ASTs are very large and complex data structures and their analysis is nontrivial.

To perform an analysis (or any other kind of operation) on one of the ASTs developers are encouraged to make heavy use of the visitor pattern. This pattern allows one to separate a piece of functionality from the data it operates on. The Overture implementation of visitor pattern makes extensive use of inheritance. In Overture, a visitor consists fundamentally of a series of `caseX(...)` methods, where `X` corresponds to the the specific AST node the method is to be applied to. Matters of traversing the AST and method dispatching are handled automatically by the existing visitor superclasses. Developers are free to focus their implementation efforts on the actual functionality to be added.

Because of the heavy use of the visitor pattern in the Overture architecture, most plug-ins are built around visitors as well. Let us consider the particular case of the POG in more detail. The POG, as its name implies, is responsible for the generation of Proof Obligations (POs). These are logical assertions that, once discharged, ensure the internal consistency of a VDM model. The Overture POG is built on two sets of classes: visitors and proof obligations.

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 a number different from 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 wish to generate, the visitor hierarchy reflects the CML AST. There are 4 kinds of visitors, each responsible for a subset of AST nodes (POGProcessVisitor is responsible for traversing processes, etc.). At any point during execution, a visitor of any type may be needed, so every visitor has a pointer to its parent visitor to enable re-dispatching.

2.3 Proof Obligations and Logics

Type checking is statically undecidable in VDM [3]. VDM specifications can be generally divided into three sets: in one group we have correct or “good” specifications; in the other group we have incorrect or “bad” specifications; and between these two groups, we have undecidable specifications.

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

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 [8], the same as the interpreters for the executable subsets of VDM [9, 10]. However, the formal semantics of VDM [11] uses Logic of Partial Functions (LPF) [12, 13, 14] which is based on the Kleene semantics with associativity 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 different.

2.4 The Symphony Theorem Prover Plug-in

The POG generates the obligations, but the task of discharging them is usually handled by some kind of proof-assistant tool. In the COMPASS project, this is done with the Symphony theorem prover plug-in.

This plug-in consists of two components. The first is a mechanisation of the semantics of CML in the Isabelle [15] theorem prover as a series of theories. This mechanisation is essentially a deep embedding of CML in Isabelle. The second component is a visitor that creates an Isabelle version of a CML model mostly by means of syntactical transformation. For a given CML model, the theorem prover plug-in will generate corresponding theory files so that assertions about the model (including POs) may be written and discharged.

The theorem prover plug-in is also responsible for providing an interface with the Isabelle theorem prover that allows the automated submission of models and POs to Isabelle, then the collection and interpretation of the results.

3

Initial Case Study

This chapter presents the initial extensibility work carried around the Overture and Symphony POGs. This work was originally reported in [16] and presented at the 11th Overture workshop. The paper has been summarised here.

3.1 Introduction

As CML and Symphony are based upon VDM and Overture, the AST generated by the COMPASS parser is extended from the Overture AST. This reuse of the AST allows one to also reuse various plug-ins of Overture such as the type checker, interpreter and POG. Because CML is based partly on VDM, many of the PO from Overture must also be generated in Symphony. As such, we will reuse the existing Overture POG as much as possible, while extending it to deal with the new CML AST.

3.2 The COMPASS Proof Obligation Generator

Much like Overture, the COMPASS POG is built as a series of visitors. Its primary visitor simply traverses a model's AST and dispatches it to relevant sub-visitors that treat each type of node. This is shown in the UML sequence diagram in Figure 3.1. These visitors must reuse their Overture counterparts.

A particularly important aspect of the POG's behavior is dealing with unsupported nodes. Whenever one of the sub-visitors encounters a node that it cannot handle (for example, the expression visitor encounters an operation node), the unsupported node is passed back to the main visitor so that it may re-dispatch it to the correct one.

3.3 Reuse

The main reuse goal for the Symphony POG was to directly utilise the Overture POG to generate all Proof Obligations from VDM constructs directly. However, this was complicated by the handling of CML nodes. Because the CML AST adds new nodes of existing types, the Symphony visitors cannot simply pass the AST to their Overture counterparts.

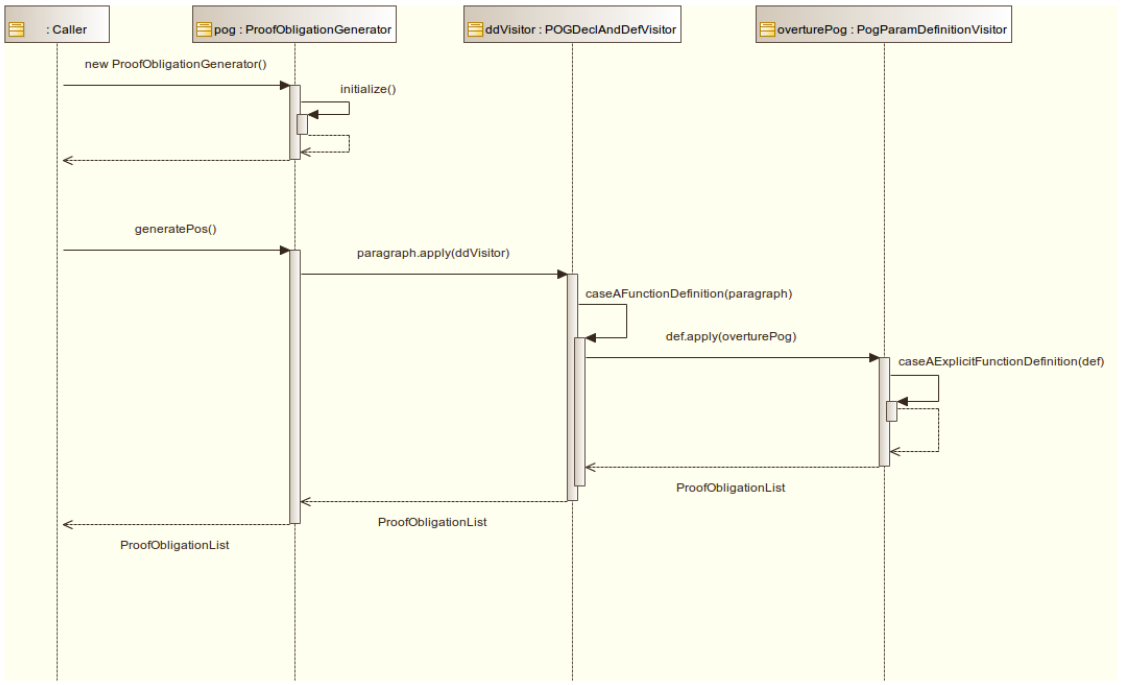


Figure 3.1: Sequence diagram representing a COMPASS POG visit

They must also handle the new nodes and that leads to the main problem when extending the POG.

The Symphony POG loses control of the flow of execution when it passes an AST node to Overture. This node will be a VDM node from the Overture AST as there are the nodes for which we wish to reuse the Overture POG. However, the VDM node may have CML nodes as its children and the Overture POG is incapable of dealing with them. This is essentially an issue of inheritance. The Symphony POG visitors cannot subclass their Overture counterparts since they must subclass the generic AST visitor classes. This issue could potentially be addressed with multiple inheritance but that is not possible in Java (the implementation language of Overture) and might have led to other issues.

To address this issue, each Symphony visitor reuses its Overture counterpart via composition instead of subclassing. Furthermore, because the only way to cope with CML nodes is with the Symphony POG, we had to alter the existing Overture POG to enable its visitors to release control flow back to Symphony as soon as they processed the node that was submitted. This was done with a 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. After this change, calls to the Overture visitors simply consist of unpacking the node, generating any relevant proof obligations and applying the Symphony visitors to any sub-nodes.

3.4 Conclusion

Overall, reuse of the Overture POG was possible and quite powerful. However, the task was not easy and there were several issues complicating the extension of the Overture POG. We also benefited greatly from being able to alter existing Overture code. The visitor context swaps (particularly, returning from Overture back to Symphony) were very challenging and without changes to the existing code, it would have been impossible to implement the Symphony POG with proper reuse. It is clear that more work must be done to improve the extensibility of Overture. Areas for improvement include the excessively complex class hierarchy, lack of documentation and the excessive use of the `switch` construct and static calls between modules.

4

LFP PO Generation

This chapter presents a new extension to the Overture POG: the generation of POs according to an alternative semantic treatment. This work was originally reported in a paper targeting ABZ 2014 [17]. The paper has been summarised here.

4.1 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 kinds of obligation, each of which checks for a particular kind of inconsistency [18]. 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, its definedness predicate is written $\delta(E)$ [19]. The compilation of a reference document for all undefined elements is ongoing in Overture [18] and an example of it can be seen in Figure 4.1.

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.

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: $\delta(\texttt{e/x}) \iff \texttt{x <> 0}$

Figure 4.1: Inconsistency Reference for Division by Zero

This PO-driven approach also influences reasoning about the POG itself, which is frequently done in terms of missing or incorrect obligations. 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.

The current version of the Overture POG (henceforth referred to as the McCarthy POG) is implemented according to McCarthy's 3-valued logic [8] 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 4.1, taken from [20], yields two separate POs, as shown in Listing 4.2.

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

Listing 4.1: A VDM function with two sources of inconsistency.

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

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

Listing 4.2: Two non-zero proof obligtations.

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 4.2 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 4.3. 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.

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

Listing 4.3: A VDM function with guards on its sources of inconsistency.

The function in Listing 4.3 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 the McCarthy logics 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 4.1. In the McCarthy POG, this yielded two obligations (see Listing 4.2), 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 4.4.

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

Listing 4.4: LPF proof obligation.

For LPF, there is only one obligation that is somewhat more complex than both of the McCarthy POs. 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 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 4.4. 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 a direct mapping to the 2 McCarthy obligations shown in Listing 4.2 but the other clauses are new.

It is worth noting that, in a McCarthy setting, the evaluation of the PO shown in Listing 4.4 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 4.5.

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

Listing 4.5: A VDM function with one source of inconsistency.

Now there is only one source of inconsistency and the McCarthy POG generates just one obligation, shown in Listing 4.6 but the same problem as before remains: the obligation cannot be discharged.

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

Listing 4.6: McCarthy Non-zero proof obligation for revised function.

However, the LPF POG has a more interesting output, shown in Listing 4.7. It is not possible to prove x<>0, but **true** can certainly (and trivially) be proven.

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

Listing 4.7: LPF Proof Obligation for revised function.

The example function in Listing 4.5 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.

4.2 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 contexts. 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 4.2. Note that handling of context extraction at top-level (e.g. for functions) is exactly the same for McCarthy and LPF POGs.

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 then it may be rerred to in the post condition, therefore "**`exists`** `r : T(r) &`", where `r` stands for the result value, should also be prepended.

Figure 4.2: Context Reference for Explicit Function Definition.

4.3 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:

- **`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:

$$
\begin{array}{l}
pogMcCarthy : Exp \times Ctxt \to PO\text{-set} \\
pogMcCarthy\,(e, c) \triangleq \\
\quad \text{cases } e : \\
\quad\quad \text{mk\_}\,(l, \mathrm{OR}, r) \to \\
\quad\quad\quad \text{let } lpos = pogMcCarthy\,(l, c), \\
\quad\quad\quad\quad newc = makeImpliesCtxt\,(c, l) \text{ in} \\
\quad\quad\quad lpos \cup pogMcCarthy\,(r, newc), \\
\quad\quad uex \to pogUnaryExp\,(uex, c), \\
\quad\quad \text{others} \to \{\} \\
\quad \text{end};
\end{array}
$$

In an LPF setting there is a need to use a different approach. The treatment of individual expressions and generation of definedness predicates (the *pogUnaryExp* function) does not change as the sources of inconsistency remain 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):

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

$pogLPF\,(e, c) \triangleq$
  cases e :
    mk_$(l, \mathrm{OR}, r) \to$
        let $popreds = pogLPF\,(l, c) \cup pogLPF\,(r, c)$ in
        $makeOrLPF\,(l, r, popreds, c)$,
    $uex \to$
        $pogUnaryExp\,(uex, c)$,
    others $\to \{\}$
  end;

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.[1] Essentially this is the construction of the POs in a disjunctive normal form.

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

$makeOrLPF\,(l, r, preds, c) \triangleq$
  let $lf = makeEquals\,(l, \mathsf{true})$,
      $rf = makeEquals\,(r, \mathsf{true})$,
      $alld = chain\,(preds, \mathrm{AND})$ in
  let $dnf =$ mk_$(lf, \mathrm{OR},$ mk_$(rf, \mathrm{OR}, alld))$ in
  $addScope\,(dnf, c)$;

$chain : Exp\text{-set} \times BOP \to Exp$

$chain\,(s, op) \triangleq$
  if card $s = 1$
  then let $e \in s$ in
      e
  else let $e \in s$ in
      mk_$(e, op, chain\,(s \setminus \{e\}, op))$

The implementation side of this work, though done only for a small subset of POs as proof of concept, was straightforward which speaks positively about the extensibility of Overture. However, there is an issue of how the two versions of the POG may coexist. The LPF modifications, if applied directly to the source code, will remove the existing (McCarthy) version of the POG's functionality. As such, one needs a mechanism that allows the implementation of the LPF functionality side-by-side with the McCarthy one. Furthermore, this mechanism must preserve the extensibility of the POG by allowing further alternative versions of the POG to be added. A possible avenue of research would be a technique for achieving this kind of extension.

[1] the *addScope* function simply adds quantifiers over any variables introduced — the scoping context functionality is retained.

5

AST-based Proof Obligations

This chapter introduces the work done on turning the proof obligations from Strings to ASTs. The work reported here has not yet been published anywhere though it is expected that it will be used in a paper on extensibility, currently being prepared for the International Journal on Software Tools for Technology Transfer.

5.1 Original PO Format

In the Overture POG, a PO is a memory object with its class corresponding to whatever type of obligation it represents. The most relevant part of said obligation class is the predicate of the PO itself. There is other information in the class but most of it is for implementation assistance or for UI purposes. The PO itself can safely be reduced to a logic predicate representing the obligation itself and some auxiliary data. For example, consider a `NonZeroObligation` and its logic predicate as shown in Listing 5.1

```
Non-Zero Obligation:
(forall x:int &
x <> 0)
```

Listing 5.1: PO predicate for `NonZeroObligtation`.

The representation of the predicate is the most important part of the representation of POs. The original version of the Overture POG produced POs only for manual inspection. As such, no particular care was given to the format for representing predicates and the main criteria for choice of format was ease of implementation and efficiency of source code. The choice therefore fell upon plain strings that directly represented the predicate as it was be shown to users of the tool.

The string format was adequate for representing POs for manual inspection. However, as part of COMPASS project, a theorem prover plug-in was implemented for discharging obligations. These obligations are generated by the POG. This made the string representations much less adequate. Strings in general are not very adequate for further processing since they typically require a parse step before any further processing.

In the concrete case of the POs and the theorem prover, the processing step required was to convert them from CML syntax to Isabelle syntax. While this conversion was

relatively straightforward, there are enough complexities involved that it could not be done with simple string replacements.

Another important task of the theorem prover plug-in was the conversion of a CML model into Isabelle. This task was implemented with a series of visitors over the CML AST. This, combined with the need to convert the POs into Isabelle syntax pointed to necessary changes to the POG: the PO predicates would need to be represented with the CML AST instead of with strings.

5.2 Converting PO Strings to ASTs

The conversion from Strings to ASTs was possible because the CML (and VDM) AST follows the language grammar quite closely and the PO predicates themselves were expressed as VDM or CML predicates (depending on the version of the POG). This meant that no changes would have to be made to the POs but only to the variables representing the predicates.

No changes were made to the behavior of the POG (expressed in the visitor classes) since, to the outside, the proof obligation classes did not change. The conversion work involved changing the type of the PO predicates from string to `INode` (the default type of an AST member) and drastically altering the constructor code of each obligation. Because all the necessary information for constructing the predicate was already passed to the obligation, the changes simply focused on how that information was used in constructing the predicate. This made the changes have very low impact since 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.

To better illustrate the scope of these changes we present volume and complexity metrics for both versions of the POG in Table 5.1. The metrics clearly show the AST-based POs make the POG significantly larger (though not much more complex on average). However, these changes are all worth the cost since they enabled us to directly use the theorem prover visitors when converting the POs to Isabelle thus achieving the fundamental step for integrating the POG and the theorem prover without the need for any dedicated code.

	Volume (LoC)	**Avg. Complexity (McCabe)**
String POG	6063	2.348
AST POG	17642	2.122

Table 5.1: Source code metrics for both versions of the Overture POG.

6

Discussion and Further Work

This chapter describes the remaining work to be carried out in the PhD work, as a continuation of what has already been achieved as well some more speculative goals whose work has not yet begun.

The work carried out so far has covered the first two themes of the PhD: assessment and improvement of extensibility. The Symphony POG work provided an initial assessment and improvement of Overture extensibility whereas the work on AST POs provided another way to improve Overture extensibility. The LPF on the other hand pointed to a possible future improvement of extensibility.

The initial POG case study covered the first two themes of the PhD as it allowed us to perform both an assessment and an improvement of Overture extensibility. It also provides an opportunity for follow-up work. We have successfully extended the POG and during that work managed to both identify an extensibility problem an solve it. The next step from this should be to generalize the problem and its solution. We will characterize the problem in general and provide ways to identify it. We must also make the solution more general. These two techniques can then be combined into a guideline for extensibility. We will then look to show how the POG case study was an application of said guideline.

Further guidelines can be researched in a similar manner thus allowing us to continue covering the themes of extensibility assessment and improvement. Two examples stem from other work already carried out in the PhD.

The conversion of POs from string to AST format identifies a clear problem in that the output format of the POG is unfit for further processing. On the other, the conversion to ASTs provides a possible solution to this problem. While the AST POs do not constitute an extension by themselves, they will enable further extensions of the POG and its integration into a tool chain of generation and discharge of obligations.

The modifications to the POG to support LPF PO generation can also be generalised as an extension technique. In this case, we are changing the output of a complex functionality but would like to do so in a way that preserves the original output version and allows further versions to be added alongside. Support for this kind of side-by-side extensions was very poor in Overture therefore an guidelines can be created to better deal with this situation.

Finally, the third theme (new extensions) remains mostly unexplored. As such, we propose the following avenues of research to identify new possibilities for extensibility:

SoS extensibility: One possibility for new extensibility techniques is to combine them with the topic of SoS. This is particularly appropriate due to the PhD's association with the COMPASS. Possible topics of research include investigating what is the meaning of SoS extensibility; what are possible techniques for evaluating the extensibility of an SoS; carry out an extension of an SoS (possibly one of the COMPASS case studies); and looking at the connection between constituent systems in an SoS and its extensibility.

Extensibility modelling: Another possibility is to move away from source code extensibility and combine it with formal modelling. There are various possibilities for this combination. An immediate one is to lift the notions of software extensibility to modelling notations. A notation such as VDM++ already supports object-orientation so many existing extensibility techniques should be applicable. However, a more interesting approach to the topic would be to combine the concepts of system modelling and system extensibility. We can assess the extensibility of the model and perhaps extract information on the extensibility of the system. Another alternative is to model a system with the explicit goal of assessing its extensibility. Thus, we can use models to explore various possible system architectures and analyze any extensibility trade-offs between them.

Multi-paradigm extensibility: There are two potential approaches to this area of research. The first is to utilize a multi-paradigm programming language to combine extension techniques from multiple paradigms in the same code base. The Symphony tool makes some use of the Scala programming language to implement the integration between the POG and the theorem prover and therefore may prove a useful case study for such techniques. A second possibility is looking into ways of utilising the extensibility techniques of one paradigm while extending source code written in a language from another paradigm. The intuition here is that some extensibility problems can best be tackled with a particular technique that might not necessarily be available in the chosen programming language so it would be beneficial if one could find ways of expressing said technique in another paradigm.

Bibliography

[1] M. W. Maier, "Architecting Principles for Systems-of-Systems," *Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.

[2] R. 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.

[3] H. Bruun, F. Damm, and B. S. Hansen, "An approach to the static semantics of VDM-SL," in *VDM '91: Formal Software Development Methods*, pp. 220–253, VDM Europe, Springer-Verlag, October 1991.

[4] B. K. Aichernig and P. G. Larsen, "A Proof Obligation Generator for VDM-SL," in *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)* (J. S. Fitzgerald, C. B. Jones, and P. Lucas, eds.), vol. 1313 of *Lecture Notes in Computer Science*, pp. 338–357, Springer-Verlag, September 1997. ISBN 3-540-63533-5.

[5] A. Ribeiro and P. G. Larsen, "Proof Obligation Generation and Discharging for Recursive Definitions in VDM," in *The 12th International Conference on Formal Engineering Methods (ICFEM 2010)* (J. Song and Huibiao, eds.), Springer-Verlag, November 2010.

[6] J. Fitzgerald, P. G. Larsen, and S. Sahara, "VDMTools: Advances in Support for Formal Modeling in VDM," *ACM Sigplan Notices*, vol. 43, pp. 3–11, February 2008.

[7] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The Overture Initiative – Integrating Tools for VDM," *SIGSOFT Softw. Eng. Notes*, vol. 35, pp. 1–6, January 2010.

[8] J. McCarthy, "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.

[9] P. G. Larsen and P. B. Lassen, "An Executable Subset of Meta-IV with Loose Specification," in *VDM '91: Formal Software Development Methods*, VDM Europe, Springer-Verlag, March 1991.

[10] K. Lausdahl, P. G. Larsen, and N. Battle, "A Deterministic Interpreter Simulating A Distributed real time system using VDM," in *Proceedings of the 13th international conference on Formal methods and software engineering* (S. Qin and Z. Qiu, eds.), vol. 6991 of *Lecture Notes in Computer Science*, (Berlin, Heidelberg), pp. 179–194, Springer-Verlag, October 2011. ISBN 978-3-642-24558-9.

[11] ISO, "Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language," December 1996.

[12] J. Cheng, *A Logic for Partial Functions*. PhD thesis, Department of Computer Science, University of Manchester, 1986. UMCS-86-7-1.

[13] C. B. Jones and K. Middelburg, "A typed logic of partial functions reconstructed classically," Tech. Rep. 89, Department of Philosophy, Utrecht University, April 1993.

[14] J. S. Fitzgerald and C. B. Jones, "The connection between two ways of reasoning about partial functions," *Inf. Process. Lett.*, vol. 107, pp. 128–132, July 2008.

[15] L. C. Paulson, "Isabelle: The next seven hundred theorem provers," in *Proceedings of the 9th International Conference on Automated Deduction* (E. Lusk and R. Overbeek, eds.), (Argonne, Illinois), pp. 772–773, Springer Verlag LNCS 310, 1988. System abstract.

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

[17] L. D. Couto, N. Battle, and P. G. Larsen, "LPF-Aware Proof Obligation Generation in VDM/Overture (submitted)," in *Submitted to ABZ 2014*, Lecture Notes in Computer Science, Springer-Verlag, June 2014.

[18] L. D. Couto, P. G. Larsen, and N. Battle, "Overture Proof Obligations Reference Guide (in preparation)," Tech. Rep. TR-008, The Overture Project, www.overturetool.org.

[19] C. Jones, *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

[20] C. Jones, M. Lovert, and J. Steggles, "A Semantic Analysis of Logics That Cope with Partial Terms," in *Abstract State Machines, Alloy, B, VDM, and Z* (J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, eds.), vol. 7316 of *Lecture Notes in Computer Science*, pp. 252–265, Springer Berlin Heidelberg, 2012.

Publications

- Main Publications
  - 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* Department of Engineering, Aarhus Univerisity, 2013.
  - Luís Diogo Couto and Nick Batte and Peter Gorm Larsen. "LPF-aware Proof Obligation Generation in Overture/VDM" submitted to "*4th International ABZ 2014 conference*, June, 2014
  - Luís Diogo Couto and Simon Foster and Richard Payne. "Towards Certification of Constituent Systems through Automated Proof" to be submitted to *Workshop on Engineering Dependable Systems of Systems*, May, 2014
  - Luís Diogo Couto and Richard Payne and Joey Coleman. "On the Extensibility of Formal Methods Tools" to be submitted to *International Journal on Software Tools for Technology Transfer*
- Additional Publications
  - Claire Ingram and Richard Payne and Simon Perry and Jon Holt and Finn Overgaard Hansen and Luis Diogo Couto. "Modelling Patterns for Systems of Systems Architectures" accepted in *8th Annual IEEE International Systems Conference*, March-April, 2014
  - Luís Diogo Couto and Jeremy Jacob and Klaus Kristensen. "Characteristics of VDM- and CSP-style Modelling in CML" submitted to *4th International ABZ 2014 conference*, June, 2014

Courses

Course	ECTS
CSP Study Group	1
Science Teaching 1	3
Science Teaching 2	2
Modelling of Mission-Critical Systems	5
The World of Research	2
Compilation	10
Paradigms of Programming *(Q3 2014)*	5

Table B.1: Courses taken

The above table lists only 28 ECTS worth of courses. For the remaining 2 points, the plan is to carry out a study group on the semantics of programming languages, preferably based around a relevant programming language for the PhD such a Scala or Python. The second, fall back, option is to take a final transferable skills course worth 2 credits such as Scientific Writing and Communication or Software Engineering Journal Club or a Summer School.

Luis Diogo Couto, On Extensibility of Software Systems, 2014

Department of Engineering
Aarhus University
Edison, Finlandsgade 22

8200 Aarhus N
Denmark

Tel.: +45 4189 3000