

**MACHINE CHECKABLE DESIGN PATTERNS
USING DEPENDENT TYPES AND DOMAIN
SPECIFIC GOAL-ORIENTATED MODELLING
LANGUAGES**

Jan de Muijnck-Hughes

**A Thesis Submitted for the Degree of PhD
at the
University of St Andrews**



2016

**Full metadata for this item is available in
St Andrews Research Repository
at:**

<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:

<http://hdl.handle.net/10023/8968>

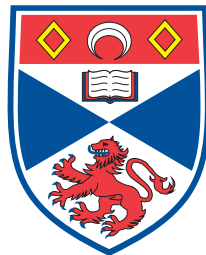
This item is protected by original copyright

**This item is licensed under a
Creative Commons Licence**

Machine Checkable Design Patterns
using Dependent Types and Domain
Specific Goal-Oriented Modelling
Languages

by

Jan de Muijnck-Hughes



University
of
St Andrews

This thesis is submitted to the
UNIVERSITY OF ST ANDREWS
in partial fulfilment for the degree of
DOCTOR OF PHILOSOPHY

submitted on

2015-12-07

Abstract

Goal-Oriented Modelling Languages such as the *Goal Requirements Language* (GRL) have been used to reason about Design Patterns. However, the GRL is a general purpose modelling language that does not support concepts bespoke to the pattern domain. This thesis has investigated how advanced programming language techniques, namely Dependent Types and Domain Specific Languages, can be used to enhance the design and construction of *Domain Specific Modelling languages* (DSMLs), and apply the results to Design Pattern Engineering.

This thesis presents SIF, a DSML for reasoning about design patterns as goal-oriented requirements problems. SIF presents modellers with a modelling language tailored to the pattern domain but leverages the GRL for realisation of the modelling constructs. Dependent types have influenced the design and implementation of SIF to provide correctness guarantees, and have led to the development of NovoGRL a novel extension of the GRL.

A technique for DSML implementation called *Types as (Meta) Modellers* was developed in which the interpretation between a DSML and its host language is implemented directly within the type-system of the DSML. This provides correctness guarantees of DSML model instances during model construction. Models can only be constructed if and only if the DSML's type-system can build a valid representation of the model in the host language.

This thesis also investigated design pattern evaluation, developing PREMES an evaluation framework that uses tailorable testing techniques to provide demonstrable reporting on pattern quality. Linking PREMES with SIF are: FREYJA—an active pattern document schema in which SIF models are embedded within pattern documents; and FRIGG—a tool for interacting with pattern documents.

The proof-of-concept tools in this thesis demonstrate: machine enhanced interactions with design patterns; reproducible automation in the PREMES framework; and machine checking of pattern documents as SIF models. With the tooling and techniques presented, design pattern engineering can become a more rigorous, demonstrable, and machine checkable process.

Candidate's Declaration

I, Jan de Muijnck-Hughes, hereby certify that this thesis, which is approximately 54558 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student in May 2011, and as a candidate for the degree of Doctor of Philosophy in December 2015; the higher study, for which this is a record, was carried out in the University of St Andrews between 2011 and 2015.

Signature of Candidate:

Date: 7th December 2015

Supervisor's Declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Signature of Supervisor:

Date: 7th December 2015

Permission for Publication

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

No embargo on any electronic nor print copy.

Signature of Candidate:

Date: 7th December 2015

Signature of Supervisor:

Date: 7th December 2015

ACKNOWLEDGEMENTS

It has been a long journey to reach this point and it reminds me of some guidance said to me during my undergraduate days. Paraphrasing considerably:

Research should not be seen as a means to an end. It is about providing answers to questions. It shouldn't matter whether the answers you found were the ones you were looking for. What should matter more is that you can show where those answers come from, and that those answers can be reproduced and shown.

Hopefully I have asked the right questions, and the answers are suitably provided. But I could not have done this alone, nor without support.

First and foremost I would like to give thanks to my supervisor Ishbel Duncan, who I managed to trick into accepting me as her PhD student through the offer of *treasure* and *service*. Thank you for giving me the freedom to do my work, acting as a sounding board for explaining concepts and ideas, and being patient. Please accept this thesis as the 'treasure' I promised, and more importantly a sign that I have at last finally got round to doing *some* work.

Next, I would like to thank Edwin Brady, the *man in the pub* who said to me: HC SVNT DRACONES. Thank you for introducing me to the world of Dependent Types, without which this PhD would not be possible, and for inspiring me to consider areas of future research. When it is *Beer o'clock* I'll get the next round in, and promise not to talk too much about work. I still will not play *Go*.

I give thanks to my internal examiner Juliana Küster Filipe Bowles for her support and patience, and suggestions on how to improve the thesis. The final version of this thesis has been improved greatly, and hopefully I have managed to get rid of all the typographical errors. *Obrigado*.

I would also like to thank various people I have interacted with over my years here at the University. Specifically, I would like to thank Özgür Akgün, Chris Schwaab, Franck

Slama, David Castro, Matúš Tejišćák, Victoria Davidson-Mayhew, Simon Dobson, and Philip Hölzenspies. The discussions and conversations we had were much appreciated, helpful, needed, fruitful, and timely. To the Secretaries, Fixit, Systems, and Teaching Fellows past and present, and countless others not mentioned, I thank you too.

Lieve ouders, zussen, en familie. Hartelijk bedankt voor jullie geduld en vertrouwen in mij. Bedankt voor alles. Ik ben nu bijna klaar om één gewone baan te beginnen en carrière te gaan maken. Ik denk dat ik het wel zal redden.

To my father, *Diwedd y gân yw'r geiniog*. Mae'n ddrwg y gân wedi bod yn hir. Diolch yn fawr. Mae wedi bod yn anodd, ond yr wyf yn ei gwneud yn.

Lastly, I want to thank my darling Isabelle who has waited long enough for this day. You have helped me through the strikes and the gutters, the ups and the downs, and when the bear truly ate me. Jij bent m'n lief, m'n leven, m'n alles, het zonnetje in mijn leven. Thank you for everything.

Jan de Muijnck-Hughes
St Andrews
7th December 2015

Doe maar normaal, dan doe je al gek genoeg.

ÞÍŦŦ·RÍÞŦŦ
M H A + P E +
W V T ⊕ A T + T

CONTENTS

Contents	xi
1 Introduction	1
1.1 The Problem with Patterns	2
1.2 Research Hypothesis	4
1.3 Research Approach	5
1.4 Contributions	7
1.5 Research Output	10
1.6 Organisation	11
2 The State of Software Design Pattern Engineering	13
2.1 Software Design Patterns	13
2.2 Pattern Languages	15
2.3 Design Pattern Engineering	16
2.4 Identifying Patterns	17
2.5 Formal Modelling of Patterns	17
2.6 Writing Patterns	19
2.7 Pattern Evaluation	21
2.8 Publishing Patterns	24
2.9 Summary	25
3 Domain Specific Goal-Oriented Modelling	27
3.1 Goal Modelling	27
3.2 The Goal-Requirements Language	28
3.3 Example: Information Secrecy	30
3.4 Domain Specific Languages	33
3.5 Domain Specific Modelling Languages	34

3.6	DSML Creation Techniques	35
3.7	Domain Modelling and the GRL	36
3.8	Summary	37
4	Dependent Types & Well-Typed (Abstract) Interpreters	39
4.1	The ARITH Language	40
4.2	Abstract Syntax	40
4.3	Type Systems	41
4.4	Interpretation Semantics	44
4.5	Dependent Types	45
4.6	Well-Typed Interpreters	50
4.7	Types as (Abstract) Interpreters	54
4.8	Summary	59
5	SIF: A Design Pattern Modelling Language	61
5.1	Overview	61
5.2	A DSML for Patterns	62
5.3	Language Specification	63
5.4	The SIF Evaluator	69
5.5	Case Studies	78
5.6	Discussion	86
5.7	Summary	92
6	Freyja: A Pattern Document Description Schema	95
6.1	Schema Definition	95
6.2	Library Provision	101
6.3	Discussion	102
6.4	Summary	105
7	Frigg: A Utility for Working with Design Patterns.	107
7.1	Overview	107
7.2	Feature Set	108
7.3	Implementation Information	109
7.4	Future Features	110
7.5	Summary	111

8	PREMES: A Pattern Evaluation Framework	113
8.1	Problems with Pattern Evaluation	113
8.2	Approach	114
8.3	Quality Indicators for Patterns	115
8.4	Pattern Report Cards	118
8.5	The PREMES Framework	123
8.6	Evaluation	126
8.7	Discussion	130
8.8	Summary	133
9	Engineering Patterns for Authentication	135
9.1	Overview	135
9.2	The Problem of ‘Authentication’	137
9.3	Addressing Authentication	140
9.4	Model Evaluation	145
9.5	Writing Patterns	147
9.6	Evaluating the Pattern	149
9.7	Pattern Publication	156
9.8	Summary	156
10	NovoGRL: Re-Targeting the GRL for new Domains	159
10.1	Making the GRL a Language	160
10.2	GRL-Derived Goal-Graphs	162
10.3	Building the Goal-Graph Using G^*	170
10.4	The Intermediate Representation: GEXPR	174
10.5	Evaluating Goal Graphs	181
10.6	Modelling the GRL as a DSML	181
10.7	The Paper Modelling Language	190
10.8	Experimental Evaluation	197
10.9	Discussion	198
10.10	Summary	200
11	Types as (Meta) Modellers	201
11.1	Modelling with Differently Shaped Languages	202
11.2	The Paper Planning Modelling Language	203
11.3	Lists of Dependent Types	206

11.4	Working with Interpretation Results	211
11.5	Type Threading	213
11.6	Interpreter for P _{TODO}	215
11.7	Discussion	220
11.8	Summary	223
12	Conclusion	225
12.1	Language-Oriented Design of DSMLs	225
12.2	Better Implemented DSMLs	226
12.3	Machine Checkable Design Patterns	227
12.4	Better Pattern Evaluation and Publication	228
12.5	Linked Concerns in Pattern Engineering	228
12.6	Future Work	229
A	Electronic Appendices	233
B	GRL Forward Evaluation Algorithm	235
B.1	Overview	235
B.2	Calculating Node Satisfaction	236
C	Collecting Dependent Types: Alternative Approaches	241
C.1	Using Wrapper Types	242
C.2	Heterogeneous Vectors	242
C.3	List of Dependent Pairs	242
C.4	Custom Lists	243
	Bibliography	245
	List of Figures	257
	List of Tables	259
	List of Definitions	261
	List of Software	263
	List of Publications	265

INTRODUCTION

The natural world is full of patterns that can be described formally using mathematics. For example the shape of the horn belonging to the *Bighorn Sheep*¹ can be modelled using the Fibonacci sequence, and the spiral of the nautilus shell follows a logarithmic spiral. These patterns have arisen as a direct result of evolutionary design decisions as made by nature. The patterns that we observed in nature are the ones that have stood the test of time, and are proven to have use. When looking at the design of engineered systems, common patterns of design will too naturally arise.

First described in *The Timeless Way of Building* [Ale79], *Design Patterns* are an engineering technique taken from architecture in which well documented solutions are presented for particular problems that occur consistently within a well defined context. Design patterns address the *separation of concerns* between: (a) the conception of a solution for a particular problem; and (b) its application to solve the given problem. Within the domain of software engineering, design patterns are used to present a good solution to a recurrent software engineering problem. It was at OOPSLA '87 where it was first argued that the processes presented by Alexander [Ale79] were also applicable to software engineering, in particular when working with *Object-Oriented* (OO) programs. Beck and Cunningham [BC87] published the first paper that detailed how pattern languages could be created for OO programs. Since the publication of Beck and Cunningham [BC87], pattern-oriented approaches to software engineering

¹Ovis canadensis

have seen consideration, and growth into their own research area [BHS07; Gam+94; Scho3].

Design patterns present a usability enabling construct that allows for complex, hard to grasp, and detailed domain specific solutions to be distilled within a format that makes the solution *easy to use* by non-domain experts. Patterns embody a domain expert's experience in solving these problems and provides a format through which domain knowledge transfer can occur between these experts and non-domain experts who wish to utilise the documented patterns [Gam+94; BC87]. Non-domain experts can become empowered with domain expertise.

1.1 The Problem with Patterns

Although, design patterns present a novel technique for addressing domain knowledge transfer they are themselves not without fault. Patterns are documents that contain a mixture of natural language and formal models. However, the inherent ambiguity of natural language can make interpretations over the meaning of what a pattern portrays even more vague [WCo2]. Such ambiguity can lead to pattern practitioners creating patterns that are not actual patterns. Heyman et al. [Hey+07] presented an evaluation of the then current software design pattern landscape, the evaluation noted that many patterns found were not patterns. Pattern authors had produced patterns that were not considered to be *true* patterns.

For example, Braga et al. [BRD98], presented the INFORMATION SECURITY pattern. This pattern details how 'cryptography' addresses the problem of data confidentiality. However, the presented pattern is not a *true* pattern. First, the pattern presents the conflation of three well-known cryptographic solutions: Symmetric Cryptography, Public Key Cryptography, and Hybrid Schemes. Secondly, the forces associated with addressing the problem do not describe the problem of information secrecy, but describe problems with selecting a cryptographic solution. Ostensibly, a design pattern is the well-described, well-tested, and well-evaluated pairing between a problem and a solution for a given context. The pattern INFORMATION SECURITY is not well-described, not well-tested, nor is it well-evaluated.

Yoshioka et al. [YWM08] detailed pattern engineering as two distinct life-cycles: *Pattern Creation*—concerned with the identification and development of the pattern itself; and *Pattern Application*—concerned with the selection and correct application

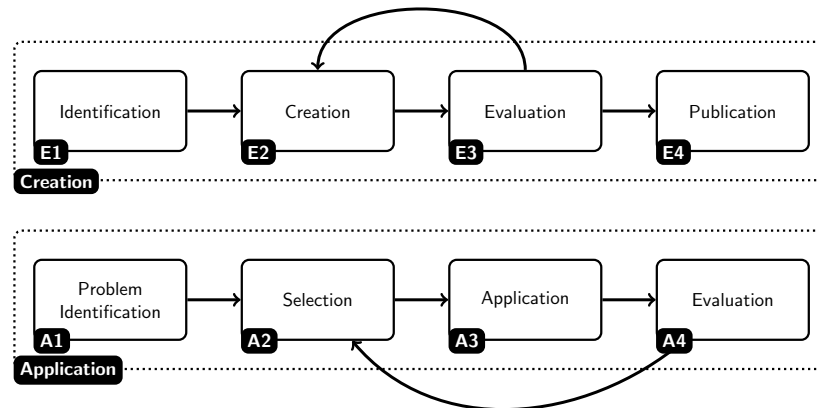


Figure 1.1: Pattern Engineering according to Yoshioka et al. [YWMo8].

of the pattern. Figure 1.1, presents an enumeration of these two stages and details the areas of concern that arise during pattern engineering.

With this notion of *pattern engineering* that guides the development of patterns, how can bad patterns such as the INFORMATION SECURITY pattern not only be detected post creation, but be prevented from being created in the first place? Existing work has already demonstrated how several of the concerns from Figure 1.1 can be addressed.

For instance, there have been several approaches to how patterns can be formally verified. *Goal-Oriented Modelling* (GOM) techniques reason formally on patterns as requirements models [GY01; LHM14; WMo8; MWAo6]. These models are agnostic to how patterns are realised in software and provides a *universal* verifiable means to reason about patterns.

Heyman [Hey13] provides a more formal programming language based approach to reason about software design pattern composition. Here design patterns are abstract software constructs modelled in Alloy. However, these models are not linked to the resulting pattern document itself. Mana et al. [Man+13] proposed the idea of *Computer-Oriented Security Pattern* (CoSP) in which design patterns are described as XML documents, linking UML models to software requirements and providing a machine readable description of a pattern. However, like Heyman [Hey13] they reason about software artefacts only, in this case using UML class diagrams, and do not present a human-readable pattern document. Before CoSP, other XML formats were presented for describing design patterns. For example, Welicki et al. [WLAo5] introduced *Entity Meta-specification Language* (EML) a specification language for

describing patterns and pattern languages, and uses the specification to construct an interactive pattern repository. However, like CoSP and the work in Heyman [Hey13], the resulting solution only reasons on software artefacts and is not applicable to socio-technical systems. Dearden and Finlay [DFo6], investigated the use of Design Patterns in *Human Computer Interaction* (HCI). Within Dearden and Finlay [DFo6], the authors detail work related to machine readable pattern documents for socio-technical systems by Fincher [Fino4], detailing Pattern Language Markup Language an XML based format for describing patterns and pattern languages. Pattern Language Markup Language was extended into *eXtended Pattern Language Markup Language* (xPML) by Kruschitz and Hitz [KH10] and Kruschitz [Kru09]. However, the presented document helps detail a structured document and prohibits for formal descriptions of the pattern itself to be encoded within the document. Further, the resulting pattern documents are themselves evaluated using informal processes such as *Shepherding* [Har99] and *Writer's Workshops* [Gabo2]. These techniques group experienced pattern writers with inexperienced ones and both work together to evaluate and improve the presented patterns.

From the solutions described above, it becomes apparent that there is a disconnect between how patterns are: represented formally; presented as pattern documents; evaluated; and used to address an engineer's existing problems. The formal representations of patterns detailed view patterns as software artefacts and do not take into account a pattern's emergent properties, or that patterns are documents. Nor can these representations be used to model patterns for socio-technical systems. Current pattern evaluation techniques also do not take advantage of more formal descriptions presented by formal methods. There is a noticeable and distinct separation of concerns between how patterns are: represented, evaluated, implemented, and applied. To address these concerns, how design pattern engineering is approached and performed needs to be rethought.

1.2 Research Hypothesis

The hypothesis presented and tested within this thesis is as follows:

Hypothesis. *The disconnected stages within the Design Pattern Engineering lifecycle can be linked through the creation of machine checkable, formally proven, programmable*

design pattern documents.

Existing research has introduced the idea of machine checkable design pattern documents—cf. CoSP from Mana et al. [Man+13], xPML from Dearden and Finlay [DFo6], and EML from Welicki et al. [WLAo6]. These are pattern documents that can be processed and reasoned on using automatic programmable methods. Further, existing research has also shown how formal models for design patterns can be constructed—cf. the use of Alloy by Heyman [Hey13], and GOM as used by Gross and Yu [GYo1].

These existing research contributions only detail how to address several of the stages within design pattern engineering separately. If each of the areas detailed for pattern engineering can be addressed *and* linked then this unified approach could lead to better and natural cohesion between the different areas and ultimately the creation of more robust patterns.

1.3 Research Approach

To provide such machine checkable *and* formally proven design pattern documents, the approach is to embed a formal machine checkable model of a design pattern within the pattern’s own document. With such machine checkable design pattern documents, a document can be used as part of pattern evaluation, publication and application. This section details this approach further.

1.3.1 Domain Specific Modelling of Design Patterns

GOM techniques have been used to reason formally about design patterns and languages [GYo1; LHM14; WMo8; MWAo6]. Here patterns are modelled as requirements models directly in a chosen *Goal Oriented Modelling Language* (GOML), typically that of the *Goal Requirements Language* (GRL). With this approach pattern engineers must first learn the chosen language. This can potentially lead to mistakes in language use especially if the pattern engineer ‘misuses’ concepts from the GOML. Such misuse can originate from a simple error in translating concepts from the pattern domain to that of the GOMLs own domain. To aid in the correct use of GOMLs for the pattern domain, a GOML oriented towards design patterns should be created.

However, rather than construct a new GOML from first principles, it would be prudent to use an existing modelling language. *Domain Specific Modelling Languages* (DSML) are modelling languages tailored specifically for a chosen domain that maps domain specific concepts on-top of an existing (meta) modelling language. With this approach the host language provides an existing implementation of a language and the DSML provides domain specific functionality. Thus, it stands to reason that a Pattern-Oriented DSML can be created that utilises existing concepts from an existing language. The resulting language would be tailored specifically for modelling design patterns, thereby providing pattern engineers with more familiar pattern concepts. Existing use of the GRL for modelling design patterns makes it an ideal candidate meta-modelling language.

1.3.2 Building Better DSMLs using Dependent Types

The creation of DSMLs from a host language is hampered by the visual and graphical nature of modelling languages, and the disconnect between the syntax used to create models, the syntax used to reason about and work with models and implement the domain mappings, and the formal semantics presented to reason about these domain mappings. To bolster the creation of DSMLs and in particular domain specific GOMLs, existing techniques from programming language research can be adapted to provide formal descriptions of the DSMLs for patterns, the chosen meta-modelling language, and the relationship between the two. Language-oriented design can be used to enhance and better DSML design. However, there is still a disconnect between the formalisms and their realisation in code.

Dependent types are a programming language construct in which the types themselves are predicated on some value [Nor09; McBo5; Brao5]. Programming languages that support dependent types provide an environment that not only allows for more precise descriptions of programs to be specified, but also an environment in which formal descriptions of languages can be realised in code [AC99]. With formal descriptions of a domain language and its meta-model implemented in a dependently typed language, the formal description of the relationship between these languages can also be implemented. Brady and Hammond [BHo6] demonstrated how a dependently typed language provides *correctness by construction* guarantees for working with and transforming language constructs.

Dependently typed languages allow for a DSML to be created for patterns such

that compile time and runtime guarantees can be made for, and between, the DSML and its host language. Dependent types *can be* used to bolster the design, specification, and implementation of a GOML for design patterns based on the GRL.

1.3.3 Machine Checkable Design Pattern Documents

With formally verified and machine checkable models for design patterns then what next? Existing work has shown that design pattern documents can be made machine readable using *eXtensible Markup Language* (XML). Given a formally verified and machine checkable model, if such a model can be embedded *née* serialised within a pattern document it can then also be deserialised and the model re-checked. Resulting in active design pattern documents that are machine readable *and* machine checkable. DSMLs can be combined with dependent types and XML to create machine readable and checkable design pattern documents.

With such documents during the evaluation of a design pattern the pattern documents can be ‘machine checked’ to ensure that the original description given during the design phase can be reproduced and is ‘valid’. Further, if the design pattern document was active then transitively so too would the description of the presented solution. If the solution presented can also be made machine readable then such a solution could also aid design pattern application.

1.4 Contributions

The hypothesis presented in §1.2 requires that the different areas in design pattern engineering be linked. §1.3 has outlined an approach that allows for the areas to be linked through active pattern documents that are machine readable and checkable. To achieve this goal the following contributions are presented that collectively support the research hypothesis. Figure 1.2 illustrates how several of these contributions fit into the design pattern engineering process as enumerated by Yoshioka et al. [YWMo8].

1.4.1 Design Pattern Engineering

The first set of contributions are concerned with the modelling and verification of design patterns.

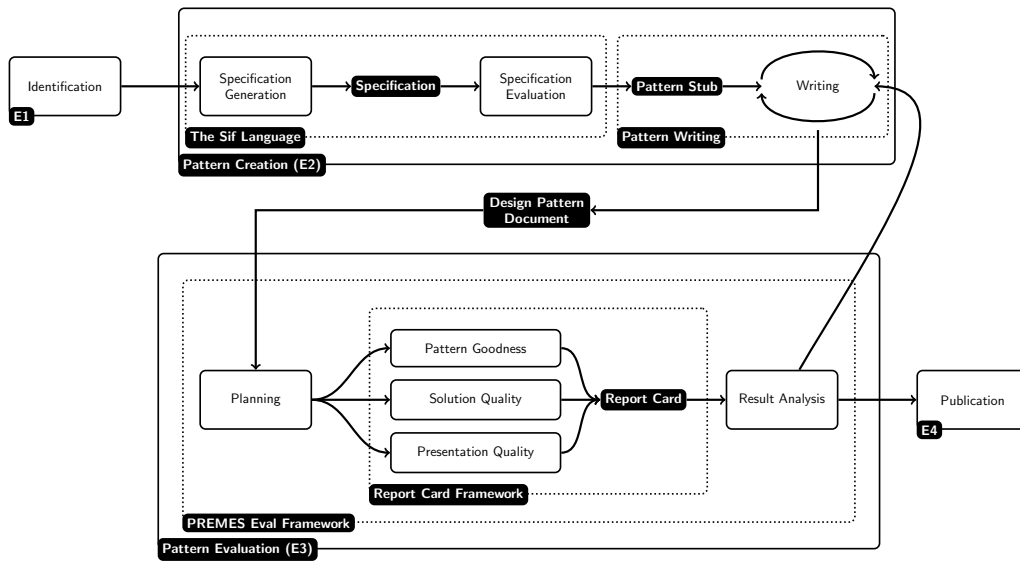


Figure 1.2: The *Pattern Engineering Process* with thesis contributions placed *in situ*.

The Sif Modelling Language SIF is a declarative requirements-based goal-oriented modelling language for prototyping design patterns. Reasoning on design patterns as problem-solution-context pairings. SIF has been designed as a DSML based on the GRL to use the GRL’s existing modelling constructs.

The Sif Evaluator Associated with the modelling language itself, is the reference implementation and an associated model evaluator. The reference implementation is presented as both an EDSL and DSL within Idris.

The Premes Framework Checking design patterns formally is not enough for their verification. The PREMES framework presents a holistic evaluation framework for software design patterns that allows for pattern quality to be made a tailorable, reproducible and measurable assessment within the pattern engineering process.

Freyja An XML schema used to describe active pattern documents, allowing for pattern metadata, a SIF model description, and evaluation metrics to be annotated through out the document itself.

The Frigg Tool FRIGG is a utility for working with pattern documents in the FREYJA format to aid in their evaluation and publication.

1.4.2 Techniques for Engineering DSMLs

The next set of contributions are concerned with *how* DSMLs can be constructed.

Language-Oriented Modelling Language Design Detailed is a language-oriented type-driven approach for the design of declarative modelling languages.

Interpreters can be Model Builders A technique for building DSMLs as a series of well-typed interpreters that detail the language transformations that must occur between the DSML, and the meta-model that represents the host language.

Types as (Meta) Modellers Demonstrates how dependent types are leveraged to embed within implementation of the DSML's type-system the meta-model to which the concepts are being mapped. This technique builds upon existing ideas that combine *abstract interpretation* and dependent types. This technique allows for DSMLs to be created that are structurally and semantically dimorphic.

1.4.3 Re-Imagining the GRL

These next set of contributions are the result of applying the above techniques for DSML engineering to the GRL.

Formal Description A formal language-oriented description for a subset of the GRL is presented.

GRL-as-a-Library The implementation of the GRL has been decoupled from a particular model creation environment allowing for model creation and evaluation to be made accessible as-a-library.

NovoGRL The GRL has been implemented as a series of well-typed interpreters in such a way that the syntax and semantics of GRL models is kept separate. Allowing for DSMLs based upon the GRL to be constructed that are semantically dimorphic, yet structurally isomorphic.

1.4.4 Techniques for Modelling with Abstract Interpretations

The final set of contributions are presented detailing how the *types as (meta) modellers* approach can be applied to, and realised for, structural dimorphic languages.

Data Structure for collecting dependent types DList is an *Algebraic Data Type* (ADT) that facilitates the type-level collection of values contained within the type of a dependent type. This data structure has useful applications for working with the ‘interpretation’ of a value at the type level.

Data Structure for Interpretation Results A technique that requires interpretation results to be represented in a ADT to allow for different/intermediate interpretation results to be represented by the same type.

Type Threading A technique that requires indexing data types by the same value to model explicitly the link between various structural views.

1.5 Research Output

During the course of the thesis, several peer-reviewed papers have been published, of which I was the primary author of, or contributed to the content presented. Here these papers are listed:

- J. de Muijnck-Hughes and I. Duncan. ‘Thinking Towards a Pattern Language for Predicate Based Encryption Crypto-Systems’. In: *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*. 2012, pp. 27–32. DOI: 10.1109/SERE-C.2012.34
- J. de Muijnck-Hughes and I. Duncan. ‘Issues Affecting Security Design Pattern Engineering’. In: *Proceedings of the Second International Workshop on Cyberpatterns*. Oxford Brookes University. July 2013, pp. 54–61
- I. Duncan and J. de Muijnck-Hughes. ‘Security Pattern Evaluation’. In: *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*. Apr. 2014, pp. 428–429. DOI: 10.1109/SOSE.2014.61
- J. de Muijnck-Hughes and I. Duncan. ‘What’s the PREMES behind your Pattern?’ In: *Proceedings of the 22nd Conference on Pattern Languages of Programs (PLoP ’15)*. To appear in the post-conference proceedings. Pittsburgh, PA, USA: ACM, Oct. 2015

Further, all of the research contributions and supporting software developed have been made available as open-source software projects. These outputs have been listed in the

thesis' backmatter, and versions used within the thesis are available in an electronic appendix—Appendix A.

1.6 Organisation

The organisation of the work presented in this thesis is detailed as follows:

Background The first set of chapters provides background material and introductory material to the topics addressed in this thesis. Chapter 2 introduces background material for software design patterns. Chapter 3 details Goal-Oriented Modelling, the GRL, and introduces the idea of DSMLs. Chapter 4 provides an introduction to programming language theory, dependent types, and *Well-Typed (Abstract) Interpreters*.

Machine Checkable Design Patterns These chapters detail the contributions made towards machine checkable design patterns. Chapter 5 introduces the SIF modelling language, detailing its design, implementation, and use to model patterns. Chapter 6 details the FREYJA pattern template and how it supports the description of active pattern documents. Finally, Chapter 7 details the FRIGG tool, a utility for working with design patterns that brings together many of the contributions from this thesis.

Pattern Evaluation The pattern evaluation framework PREMES and pattern report cards are detailed in Chapter 8, together with their use to evaluate existing patterns.

Engineering Patterns for Authentication Chapter 9 presents a case study in using the contributions to engineer patterns for authentication. The engineered patterns presented are: AUTHENTICATION THROUGH SHIBBOLETHS; and AUTHENTICATION THROUGH ID CARDS. The case study demonstrates how the patterns were modelled using SIF, and evaluated using the PREMES framework in conjunction with the supporting utilities FRIGG and FREYJA.

Building Domain Specific Goal Oriented Modelling Languages The final set of chapters provides more technical details surrounding the construction of domain specific goal-oriented modelling languages using dependent types. These chapters should be considered in isolation from design patterns and the

techniques are not bespoke to the pattern domain. Chapter 10 details the formalisation and re-engineering of the GRL to support re-targeting to different domains. How dependent types can be used to provide stronger correctness by construction guarantees between a DSML and a meta-modelling language is detailed in Chapter 11.

Conclusions This thesis concludes in Chapter 12 with a discussion of machine checkable design patterns, type-driven language oriented modelling, and possible directions in which this research can go.

Domain Targeted Roadmaps The work presented in this thesis brings together research from several different areas of Computer Science, namely Design Patterns, Goal Modelling, and Dependent Types & Formal Methods. A reader more or less familiar with certain areas may want to read this thesis differently, and different domain-specific road maps are suggested below.

Area of Interest	Relevant Chapters
Design Patterns	Chapters 2 and 5 to 9
Goal-Modelling	Chapters 3, 5 and 10
Dependent Types & Formal Methods	Chapters 4, 5, 10 and 11

THE STATE OF SOFTWARE DESIGN PATTERN ENGINEERING

First described in *The Timeless Way of Building* [Ale79], *Design Patterns* are an engineering technique taken from architecture in which well documented solutions are presented for particular problems that occur consistently within a well defined context. This chapter introduces software design patterns and provides information on the current *state-of-the-art*.

2.1 Software Design Patterns

Software design patterns are used to present a good solution to a recurrent software engineering problem. Since the publication of Beck and Cunningham [BC87] pattern-oriented approaches to software engineering and language have seen consideration. For example, pattern based approaches have been used to describe common architectures for *Cloud Computing* [Feh+14], interaction patterns for HCI [Sef15], patterns for programming in Java [Sari6] and for *Operational Support Systems* [AG09] to name a few areas. But what exactly is a pattern?

Typically, the approach advocated by Alexander is that a pattern should address the following *core* areas: (a) the *context* in which the pattern is being applied; (b) the *problem* the pattern is solving; (c) the *forces* that drive the choice of solution from the

problem; (d) the *solution* presented by the pattern including the solution's *dynamics* and *structure*; (e) the *resulting context* from application of the pattern; (f) the *relations* with other patterns; and (g) *guidance* for pattern application.

One interesting use of patterns is to document solutions to security problems: *Security Design Patterns*. First introduced in Yoder and Barcalow [YB97] such patterns are used to describe well known security concepts and associated security mechanisms [Sch+06; Scho3; Ferr3]. Pattern-oriented approaches that use security design patterns are increasingly being used to develop secure systems [Fer+11]. Security pattern languages are pattern languages within the security domain.

Use of design patterns allows for complex security concepts and mechanisms to be expressed concisely and explicitly such that non-domain experts can understand them, and consequently use them [DFLo7]. When designing security systems abstract patterns [FWYo8] can be used to abstract over multiple similar patterns that address a common security problem. According to Bunke et al. [BKS11] there are 409 known security design patterns, providing solutions to security problems as diverse as access control, session management, and identity management. These patterns were collected through a systematic literature review of patterns published between the 1997 and 2010. The relevant literature documenting these patterns can be found within Bunke et al. [BKS11].

Not all documented patterns address problems within the same context. Different types of patterns can be used to describe solutions for different areas. Henninger and Corrêa [HCo7] summarised the several different types of patterns that have been described. From this list of pattern types several common types of patterns emerge that can be used to describe software systems.

- **Component Patterns:** Patterns that specify how to achieve some functionality i.e. software design and creation.
- **System Patterns:** Patterns that specify how to combine patterns i.e. architecture and behavioural.
- **Deployment Patterns:** Patterns that specify how to deploy system patterns.
- **Implementation Patterns:** Patterns that specify how to implement/realise some aspect within software/real world setting.

- **Admin Patterns:** Patterns that specify how the system should be administered.
- **Generic Patterns:** Generic Patterns that cannot be described using the other pattern types.

2.2 Pattern Languages

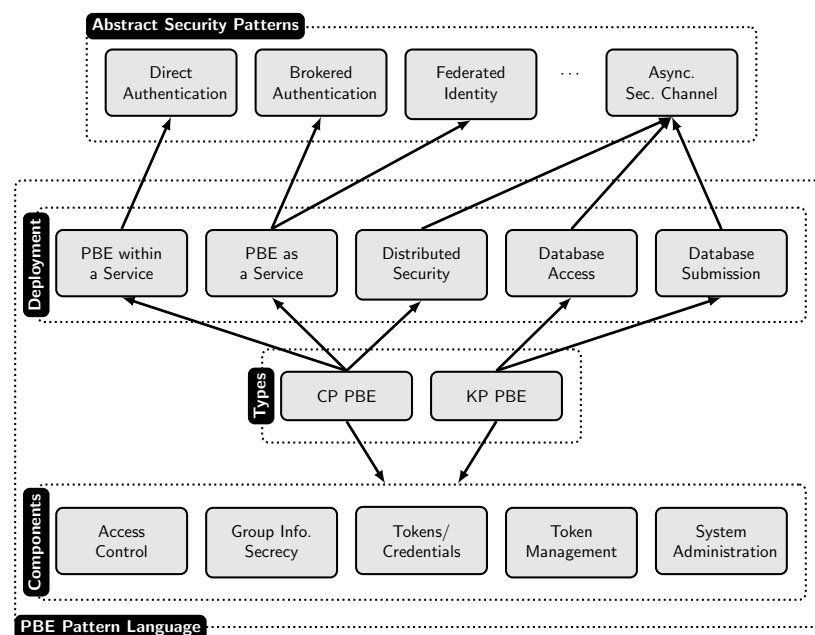


Figure 2.1: Example pattern diagram from de Muijnck-Hughes and Duncan [dMD12] describing a proposed pattern language for Predicate-Based Encryption.

Often recurrent problems may not be solvable with a single pattern; they are too complex and too big. *Pattern Languages* provide a means through which solutions to complex problems can be solved. A pattern language is a network of predefined patterns that define a process for resolving systematically a set of related interdependent software development problems [BHS07]. Pattern languages are themselves described within the pattern format, and are also illustrated as pattern diagrams [Ale+77]. A pattern diagram is a directed graph in which nodes represent patterns and edges the relationships between them. Within these *Pattern Diagrams*, a topological ordering direction of *top-to-bottom* indicates the level of abstraction for contained patterns. Higher patterns represent abstract concepts, and lower patterns concrete notions. This

style of diagram is typically referred as the *Alexandrian* [Ale+77]. An example pattern diagram for a proposed pattern language for Predicate-Based Encryption.

2.3 Design Pattern Engineering

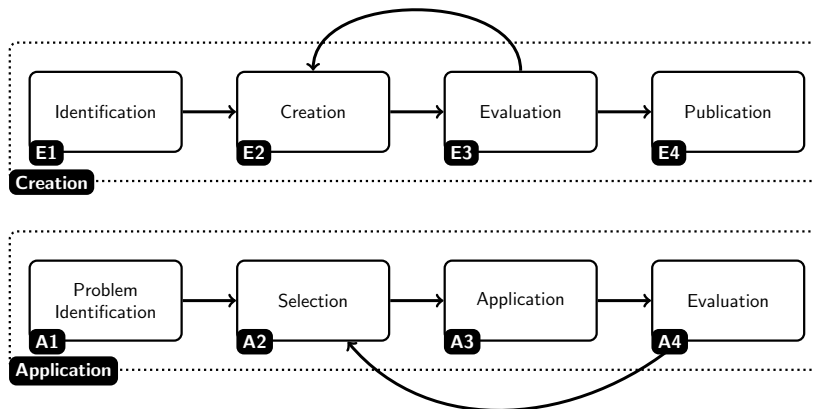


Figure 2.2: Pattern Engineering according to Yoshioka et al. [YWMo8].

The lifecycle of a pattern has two distinct stages. The first, *Pattern Creation*, is concerned with the identification and development of the pattern itself. The second, *Pattern Application*, details the application of said pattern. Yoshioka et al. [YWMo8] enumerated this engineering process into the following set of steps.

- **Creation** Building the Pattern.
 - E1: Finding a recurring problem and its corresponding solution from knowledge and/or experiences of software development.
 - E2: Writing the found pair with forces in a specific pattern format.
 - E3: Reviewing and revising the written pattern.
 - E4: Publishing the revised pattern via some public or private resource.
- **Application** Using the Pattern.
 - A1: Recognizing context and security problems in software development.
 - A2: Selecting software patterns that are thought to be useful for solving the recognized problems.
 - A3: Applying the selected patterns to the target problem.
 - A4: Evaluating the application result.

Figure 2.2 illustrates the connection between these steps. This thesis is concerned primarily with the first stage of pattern engineering: *Creation*. For the remainder of this chapter distinct stages from this process, and related topics, are introduced and detailed further. During consideration of Step E2 *Creation* the topics of formal modelling, and writing of a pattern document will be considered separately.

2.4 Identifying Patterns

Pattern identification is concerned with the identification, and veracity, of the identified problem. A common technique for identifying software patterns is to mine existing software constructs (by hand or automatically) to identify the recurring structures that are used to address some problem [BKS11]. The issue of pattern discovery is an open research question and not the main focus of this thesis.

2.5 Formal Modelling of Patterns

Software design patterns are part software artefact, part design document. There have been various attempts at the formal modelling of the emergent software artefact that arises from a design pattern. This section considers this formal modelling.

2.5.1 Model Checking

Dong et al. has investigated the use of model checking to investigate formalising design patterns. The authors took *Unified Modelling Language* (UML) models, typically used to represent the software constructs, constructed a formal representation of the model to formally verify the resulting formal model's correctness [Don+07]. Following this, the authors then investigated how to combine and verify the composition of the UML models through verification of the composition of the underlying formal models.

Shiroma et al. [Shi+10] took an alternative model-based approach to investigate security patterns. Concentrating on security patterns the authors investigated the dependencies between patterns and modelling these dependencies as model transformations.

One final model checking approach was demonstrated in the PhD thesis of Heyman [Hey13]. The author used the SAT solver Alloy [Jac12] to check design patterns and their requirements, as well as their composition.

2.5.2 Petri Nets

Petri Nets are a formalism used to model the behaviour of software systems [Pet62]. da Silva Júnior et al. [dGM13] considered the use of *Coloured Petri Nets*, an enhancement of Petri Nets to provide more abstraction, to model the structural and behavioural properties of security patterns. Coloured Petri Nets provides a more formal system upon which patterns are analysed when compared to UML. However, the work appears to be constrained to reasoning about patterns for OO languages. How this work would apply to non-OO languages is not clear, nor to patterns that operate within a socio-technical context.

2.5.3 Formal Logic & Ontologies

An alternative approach to modelling design patterns is given by Dietrich and Elgar [DE05]. The authors investigated the use of ontologies to capture patterns. Their approach seeks to model design patterns in a programming language agnostic way.

Bayley and Zhu [BZ10] developed a strict subset of UML to capture design pattern structure and behaviour, and presented a transformation technique to convert the UML models into a formal description based on first-order predicate logic. The resulting formal descriptions were used to reason about patterns, and their composition and transformation.

2.5.4 Goal-Oriented Modelling

Formal approaches to design pattern modelling typically investigate the formal modelling of software constructs. These approaches do not take into account emergent properties such as *quality of documentation*, and *goodness of fit* between the problem and solution. Further, these modelling approaches make an implicit assumption that the underlying software constructs are based on OO design principles. Several existing bodies of work have, to varying degrees, each investigated the applicability of modelling design patterns and pattern languages using Goal-Oriented Modelling techniques. Specifically using the Goal Requirements Language.

The earliest known attempt is in Gross and Yu [GY01] in which the authors present a *Requirements-Driven* approach to design pattern modelling. The authors identify that a limitation in modelling design patterns stems from their heavy use of natural language. To address these limitations, the authors model the forces of a design pattern

as non-functional requirements in a GRL model instance, and apply this model to the design of a system in a secondary model instance. Building on from this work is the work of Mussbacher et al. [MWAo6] in which the authors present similar work. Mussbacher et al. go further than Gross and Yu through illustration over the evaluation of design patterns modelled using the GRL. Weiss and Mouratidis [WMo8] present a means to model pattern languages using the GRL and extract from the graphical model a formal model in Prolog to analyse satisfaction of system requirements. The latest known work to model design patterns is in Li et al. [LHM14]. The authors concentrate on the modelling of security patterns.

Common to all approaches is the need to interpret domain specific concepts from the design pattern domain into concepts known and used by the GRL. In Gross and Yu [GYo1] the interpretation was minimal as the authors took a requirements driven approach to design pattern specification, and thus could use the concepts from the GRL directly. The other approaches presented concepts from the design pattern domain and translated them into GRL concepts. Resulting in the use of the GRL as the host language for modelling DSMLs.

However, the GRL is a requirements language and as such the semantic domain targeted by its syntax is the modelling of socio-technical systems as goal-models. Although design patterns can be used to represent socio-technical systems there is an inherent trioka of problem \times solution \times context w.r.t. the pattern's structure. Although, the problem \times solution aspect can be modelled directly within with the GRL, these present a single model instance. Examining different problem \times solution pairings for a particular context is not practically feasibility within the GRL.

2.6 Writing Patterns

Once the pattern has been identified, the next step is to develop the pattern document, and describe the discovered pattern. When writing patterns advice is available on how to codify and represent the identified pattern constructs. For example, Meszaros and Doble [MD97], Wellhausen and Fießler [WF11] and Harrison [Haro4] have presented three known, and touted, writing guides. These documents provide guidance about the development of the document and advice over naming and decomposition of the ideas to be presented.

2.6.1 Patterns *are* Documents

A naïve interpretation of a design pattern commonly seen is that the pattern *is* the presented solution. Instead the *pattern* itself is an abstract concept that is collected and described in a document. Patterns are described using a mixture of natural language descriptions, and formal models. For patterns, these formal models are often presented using one of the modelling languages from UML.

These documents are presented as structured templates that present headings common to many a pattern. Structured headings also allow for pattern authors to be guided over the contents and description of their patterns. Common templates seen include: the *Alexandrian* template that adheres to the *core* pattern areas detailed in §2.1; the *Pattern-Oriented Software Architecture* (POSA) template used by authors of the POSA series—see Buschmann et al. [BHS07] for an example; and the *Patterns 2.0* format used for societal and non-software oriented patterns examples of which are found in Guerra et al. [Gue+14].

However, not all patterns adhere to the known pattern templates. Pattern authors are free to select templates that are more indicative of the pattern being described. This unfortunately harms pattern engineering over how pattern documents are encoded [BKS11]. Which template are authors supposed to follow?

2.6.2 Encoding Design Pattern Documents

§2.5 detailed several approaches to the formal modelling of design patterns. These approaches concentrated on modelling the software artefact from the pattern document. Although this is beneficial to reasoning about patterns in software, not all software design patterns are technical in nature software. Nor do these approaches consider the document itself.

Other work has looked to the encoding of the pattern document encoding design pattern documents as XML documents. Early work into an XML based encoding was by Lucrédio et al. [Luc+03]. The authors describe a tool used to construct an interactive pattern repository for viewing design patterns.

HCI has worked to develop techniques and methodologies for working with pattern documents. Dearden and Finlay [DFo6] presented the state-of-the-art of HCI patterns from 2006. Here the authors detail *Pattern Language Markup Language* (PLML) and its successor xPML [Fino4]. These are XML schema for describing HCI patterns. Of interest here is the *typed* encoding of relationships between other patterns.

A more recent summary of PLML and xPML can be found in Kruschitz and Hitz [KH10] and Kruschitz [Kru09].

An alternative XML encoding for design patterns is that of EML. First introduced in Welicki et al. [WLA05], EML is an XML schema for describing all kinds of patterns and supporting concepts. The work by Welicki et al. [Wel+06] continued in Welicki et al. [WLA06; Wel+06]. Rather than describe textual documents, EML looked to the encoding of the pattern itself (software artefacts) in the XML schema. This is a marked change from the previously discussed schema.

In a similar attempt to Welicki et al. [WLA06], Mana et al. [Man+13] looked to describing security patterns within XML schema. Unlike Welicki et al. the work of Mana et al. concentrated on the encoding of security requirements, software artefacts in the form of UML, and the relations between the two. Further, Mana et al. provided more textual documentation into their schema in comparison to the work of Welicki et al.

Regardless of the presented XML encoding, a trade-off exists when encoding design patterns as an XML schema between: encoding the human readable document; encoding the pattern artefact; and encoding the link between different patterns.

2.7 Pattern Evaluation

Unfortunately, not all patterns created are in fact patterns [WCo2; Hey+07], and pattern evaluation is one of the lesser reported aspects within pattern research. When looking to evaluate a pattern one must determine whether the problem is satisfied by the solution, how ‘tried and testing’ the solution is, and how well documented the pattern is. Further, the evaluation of a pattern must be reproducible, consistent, and allow for fine-grained analysis of the presented pattern. There are several known evaluation practises used within the pattern community, namely: *Peer Review*, *Shepherding* [WF11], and *Writer’s Workshops*. However, these evaluation systems do not provide comprehensive guarantees in *all* these areas. Nor do some evaluation systems provide comprehensive guarantees that the method is reproducible, consistent, nor allow for fine-grained analysis.

One of the difficulties in constructing an evaluation system for software design patterns is that the subject domain covered is heterogeneous. A single general purpose evaluation system cannot be, and should not be specified. For example, Bunke et al.

[BKS11] detail that, among other things, not all patterns adhere to common pattern templates. Any suggested evaluation process must be tailored to the patterns being evaluated. The remaining part of this section on evaluation discusses several known evaluation techniques used.

2.7.1 Shepherding

Shepherding is the recommended pattern evaluation technique for the PLoP series of conferences: PLoP, EuroPLoP, and Viking PLoP. Shepherding is a process in which experienced pattern writers (*shepherds*) are paired with pattern authors—their *sheep*. Harrison [Har99] details a common shepherding process, and provides implementation guidance. The goal of shepherding is to capitalise upon the experienced writer’s knowledge in creating patterns to guide and provide expert commentary on the presented pattern.

However, the shepherding process is a generalised technique that is applicable to all patterns. It does not provide fine-grained nor reproducible guidance over how to measure what constitutes a good pattern from a bad pattern. Nor does the process detail how the solution presented should be evaluated to determine its quality. Further it makes a tacit assumption that the advice of Meszaros and Doble [MD97] was used to drive the writing of the pattern being evaluated. The advice offered by Meszaros and Doble [MD97] tacitly provides a means to ensure good quality patterns by guiding the authors to writing patterns in a good style. Other existing pattern writing guides such as Wellhausen and Fießer [WF11] such Harrison [Haro4] also detail how notions of quality are accounted for during. The Shepherding process provides subjective evaluation over the quality of the pattern itself and not the solution being described.

2.7.2 Writers Workshops

Writer’s Workshops are moderated Socratic discussions involving sets of like patterns. Involved in these workshops are the pattern authors themselves, and a set of peers. Traditionally, during the workshop the pattern author is asked to sit aside from the group and listen to the discussion of their pattern. The group moderator leads the discussions, ensuring that the strengths and weaknesses of the pattern are discussed and that suggestions for improvements are made and noted. Schmidt [Scho6], Coplien

and Woolf [CW97] and Gabriel [Gabo2] presents commonly used guidance used by the pattern community for such workshops.

Remark. At *PLoP 2015* a new style of writer’s workshop was trialled in which the pattern author had more involvement in the discussion. This involvement was to specifically introduce the piece of work, provide a set aims for the feedback, and provide clarification of points raised by participants.

2.7.3 Structured Approaches

Both Shepherding and Writer’s Workshops provide informal approaches to pattern evaluation. There are, however, more structured approaches.

Heyman et al. [Hey+07] presents two methods of evaluation: the first determines if the presented pattern is a pattern; and the second provides an assessment over documentation quality. For the former, Heyman et al. uses a set of simple criteria to investigate the *patternness* of the presented pattern. Quality of documentation is assessed using simple qualitative metrics to assess content quality per expected heading. With each heading being weighted, a final score can be given to determine the overall adherence to documentation quality. However, this approach concentrates on pattern presentation and does not evaluate the pattern in other areas. For example, goodness of the solution to address the problem.

Halkidis et al. [HCS04] performed a qualitative analysis of several security patterns according to: (a) how well the pattern adheres to ten guiding principles for building secure software [VM11]; (b) how well the pattern deters the software developer from building an insecure system; and (c) how the pattern responds to different types of attack. Similar work was also performed by B. H. Cheng et al. [Che+03]. The use of qualitative evaluation criteria as used by Halkidis et al. [HCS04] is inherently problematic. Halkidis et al. [HCS04, Section 4] mention that some of the criteria specified can only be used to assess the patterns implementation and not the pattern itself. For a qualitative analysis of patterns, criteria assessing the patterns and not implementation needs to be specified. The approach by Halkidis et al. is purely for security patterns and concentrates on assessment of the quality of solution and not quality of documentation.

Laverdière et al. [Lav+06] presents a comprehensive set of criteria for security design pattern evaluation using the *Six Sigma* approach. As with the approach taken by Halkidis et al. [HCS04], this technique concentrates on security patterns. However, the guidance established by Laverdière et al. [Lav+06] does provide a more holistic

treatment towards design pattern evaluation. Unfortunately, no implementation guidance, nor results in using this technique were presented.

An underreported aspect of design pattern evaluation is that of usability. Thimthong et al. [TCK13] explore the use of user studies for pattern evaluation. However, use of user studies should be limited as the usefulness of such studies can be ineffective and unhelpful if done improperly [GBo8].

During *PLoP '15*, Xia et al. [Xia+15] introduced a more structured approach to pattern evaluation to bolster writer's workshops. The authors took established evaluation methodologies (checklists and code review) from software engineering and applied them to evaluate design patterns. Checklists are used to represent guiding evaluation criteria for design patterns, and detail what reviewers should look for. Perspectives are used to guide the review from the view-point of an entity involved in the pattern construction and domain of operation.

2.8 Publishing Patterns

Pattern Repositories are collections of patterns bundled together and presented for consumption. Example pattern repositories can be found: online [The13]; within books e.g. Schumacher et al. [Sch+06]; or (and most commonly) within academic literature. Pattern repositories have also been presented as a single PDF document Kienzle et al. [Kie+03].

However, these resources are either: (a) incomplete; (b) cannot be modified; and (c) cannot be used programmatically. The *Common Attack Pattern Enumeration and Classification* (CAPEC) repository [Cor13] is a good example of a pattern repository, presenting a list of patterns describing system weaknesses.

Central to pattern engineering, and also research, is the creation of an easily accessible design pattern repository that can be used by researchers and developers alike. The existence of such a pattern repository would provide pattern researchers with a catalogue through which they can perform pattern related research. This would also benefit pattern developers. For software developers, a centralised repository will facilitate access to a variety of design patterns that they can examine/select for their needs during pattern application.

Remark. Interestingly, how repositories are to be constructed was a topic of discussion at *PLoP '15* [SI15]. The authors led a discussion to foster support for the de-

velopment of an open and collaborative design pattern repository.

2.9 Summary

This chapter has introduced design patterns and their engineering. Although patterns have been around since the mid-eighties the discipline of pattern engineering is rather nascent. There is no comprehensive pattern repository from which patterns can be selected. Formal modelling of patterns tends to concentrate on modelling software architectures and little on modelling patterns themselves.

From a practitioners perspective tooling and practices to support pattern engineering is also lacking. The lack of common consensus of template headings, their encoding in a document markup language, and their representation in code makes working with patterns harder than what it should be. The lack of evaluation seen in published patterns, leaves the efficacy of patterns in doubt. Why would an engineer use a solution that was not evaluated? The evaluation process needs enhancement to not only be reproducible and agnostic to the pattern being written, but also provide detailed evaluation of the presented pattern. Design Patterns are supposed to be well-documented solutions to recurrent problems.

However, one cannot solve design pattern engineering overnight or in a single thesis. Design patterns appear in socio-technical systems and not just in software. One of the more interesting modelling techniques for modelling patterns is that presented by the GRL.

DOMAIN SPECIFIC GOAL-ORIENTED MODELLING

Chapter 10 introduces NovoGRL, a re-engineering of the Goal Requirements Language (GRL) as an Embedded Domain Specific Language within Idris. Chapter 5 introduces SIF a Domain Specific Modelling Language (DSML) that uses NovoGRL as its meta-model. This chapter introduces the reader to Goal-Oriented Modelling (GOM), the Goal Requirements Language, and provides background information on DSML creation.

3.1 Goal Modelling

GOM is commonly used in requirements engineering to reason about socio-technical systems [Mylo6]. Goal models describe: the set of *Goals* associated with the problem, the perceived requirements of a system; the *Tasks* required to address said goals, including technical and non-technical artefacts; and the relationship between goals and tasks. Goals are broken down into smaller goals, and the links between goals and tasks annotated with some satisfaction metric that denotes the intended satisfaction of the goal by a set of tasks. With goal satisfaction, comes the ability to analysis goal models and determine if the given set of tasks can satisfy, and to what degree, a set of goals.

GOMs such as i^* [Yu97], GRL [UTN12], and TROPOS [Bre+04] have been used

to model socio-technical systems as part of the requirements engineering process. These languages allow for a stakeholder's requirements to be modelled and reasoned upon in conjunction with the proposed solution. A natural extension is the modelling of security requirements for software systems [vLamo1], in which goals represent security requirements and tasks software artefacts. Goal models can be constructed to describe software systems as a set of tasks and their relation to the ideal set of security requirements for the proposed software system. The resulting goal model can be analysed to determine whether or not the software system satisfies the security requirements.

3.2 The Goal-Requirements Language

The GRL is a known GOML based on i^* [Yu97] for requirements modelling, and has been specifically designed to aid in reasoning over both functional and non-functional requirements for socio-technical systems. Further, the GRL was incorporated into the standard for *User Requirements Notation* [UTN12].

3.2.1 Definition

Along with i^* , the GRL is a graphical modelling language and requires the visual construction of models. Figure 3.1 details the legend for the GRL. Core to the GRL language is the representation of problems as a goal graph. Nodes (intentional elements) detail constructs in the model, and labelled edges (element links) how the constructs are linked.

Intentional elements can be a goal, soft-goal, task, or a resource. Goals are used to describe functional requirements, and soft-goals non-functional requirements. Tasks are actions that can be used to provide a solution to both goals and soft-goals. Tasks alone cannot be used to satisfy goals, resources are used to specify constructs that are required by both tasks and goals.

Element links are used to describe the structural and intentional relationships between the model constructs. Elements need not be atomic and can be decomposed into sub-elements, as such the GRL supports AND, OR, and XOR element decomposition. Intentional relationships are used to describe the effect that elements have on each other. Contribution links are used to describe the direct impact between elements, and correlation the side-effect that one element's use has on another. Intentional links

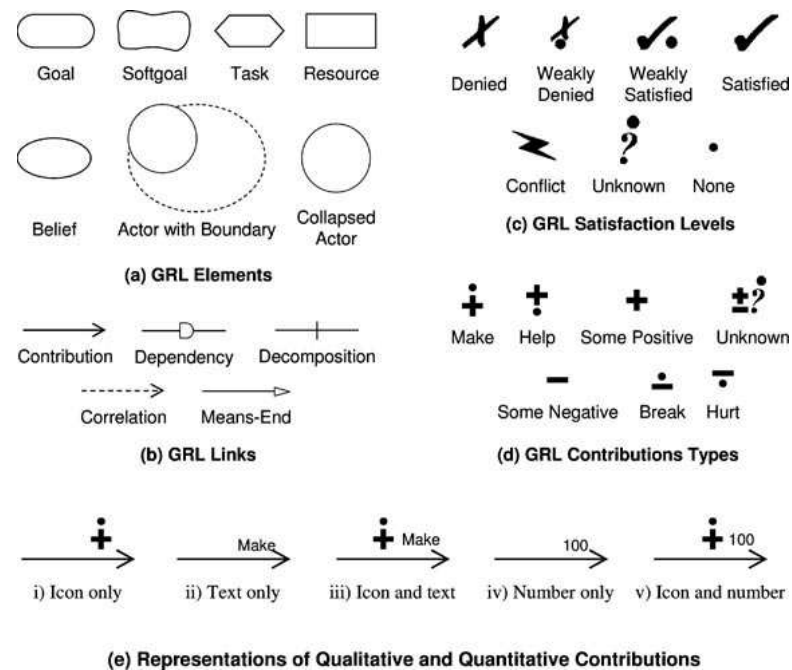


Figure 3.1: Legend for the GRL [Amy+10].

are weighted according to the value of the contribution (or effect) of the link. These contributions can either be qualitative or quantitative.

3.2.2 Model Evaluation

Evaluation algorithms for goal satisfaction in the GRL was described in Amyot et al. [Amy+10] and UTN [UTN12]. Evaluation processes are either qualitative or quantitative in approach, depending how the effect of the intentional elements were described. Further select elements within the model will have an initial satisfaction value. Evaluation strategies are used to propagate the effect of those satisfaction values through-out the model. Giving each element in the model an associated evaluation value. The direction of this propagation is dependent upon the evaluation strategy chosen. These strategies are used to analyse goal models according to known criteria contained within the model—initial satisfaction values. Different evaluation strategies are used to interrogate the goal model:

- **Forward Propagation** Can goals be satisfied given the strategy for all leaf nodes in the model?

- **Backwards Propagation** Given goals that are already satisfied to some degree, what degree of satisfaction must each leaf node have?
- **Hybrid** A mixture of the other two styles.

Depending on the evaluation strategy chosen, different sets of elements within the GRL model instance will be given initial satisfaction values.

Remark. Giorgini et al. [Gio+03] presents a formalisation of the evaluation semantics for generic goal modelling. Interestingly, Amyot and Mussbacher [AM11] mentions that no such formal evaluation semantics exist for the GRL.

3.3 Example: Information Secrecy

To further motivate Goal-Oriented Modelling, and specifically use of the GRL, this section demonstrates the goal-oriented modelling of a simple example. Key to the understanding GOM are the concepts of: *problems*—the goal to be achieved; and *solutions*—the means through which the goal can be satisfied. In this example the problem of *Information Secrecy* will be modelled, and a solution *Symmetric Cryptography* will be presented. The resulting model will be evaluated to see how the solution affects the problem and if the problem is *solved*.

3.3.1 The Problem: Information Secrecy

Information Secrecy is the problem of ensuring the confidentiality of data between two entities Alice and Bob. Such that if Alice sends a message to Bob, only Bob is able to read the message. To effectively model the problem, the goals (or requirements) of the problem need to be identified.

For information secrecy, the root goal is to ensure that the data is kept confidential; the goal of *Data Confidentiality*. However, it is naïve to think that this is the only requirement. A more comprehensive list of requirements include¹

- 1) **Recipient Confidentiality**—data should be viewable by the intended recipient only.
- 2) **Suitable Security Level**—the mechanism should be configurable for different security levels.

¹This list is not exhaustive.

- 3) **Suitable Performance**—the mechanism should not be unnecessarily computational expensive.
- 4) **Comprehensible**—end users must be able to use the solution.
- 5) **Minimal Workflow Disruption**—the resulting solution should have minimal impact on the workflow of the user.
- 6) **Secure Implementation**—the implementation of the solution must be secure.

To consolidate the problem these goals can be grouped under a secondary goal *Information Secrecy* and linked through AND decomposition. Thus, in the goal model these conditions *née* requirements *must be* met for the primary goal to be achieved. Figure 3.2 demonstrates the partial goal model for this problem. Of note, this goal model has a root goal indicative of the final goal that needs to be satisfied. However, such a root goal is not a necessary requirement in producing goal models. Goal models may have several root goals that are not linked, nor subsumed under a higher level goal.

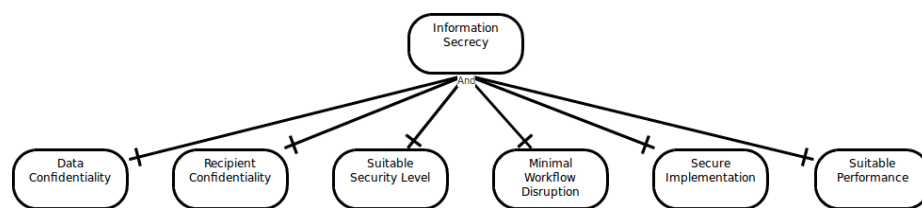


Figure 3.2: An example goal model for the problem of ‘Information Secrecy’.

3.3.2 The Solution: Symmetric Cryptography

Within GOM, the solution is the set of tasks/resources that when brought together will have an effect (positive or negative) on the goals of the problem. Solutions are best thought of as: *the things we need to do to address the problem*. This part of the study examines how symmetric cryptography can be used to address information secrecy. Symmetric cryptography provides developers with mathematically studied techniques for encrypting data under a shared key. Translating into one possible set of tasks for goal modelling, symmetric cryptography can be viewed as the combination of:

- 1) **Mathematical Description**—the operational characteristics common to all symmetric schemes;
- 2) **Symmetric Algorithm**—the specific algorithm used to encrypt;

- 3) **Symmetric Key**—an artefact of the solution used during encryption/decryption;
- 4) **Algorithm Implementation**—the realisation of the algorithm in code.

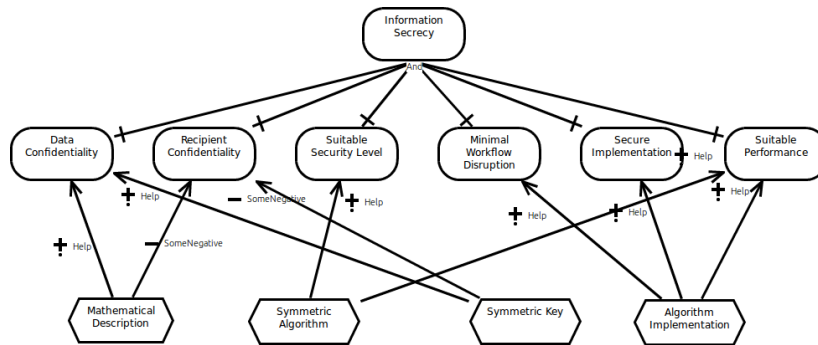


Figure 3.3: An example goal model for the problem of ‘Information Secrecy’ with a solution using ‘Symmetric Cryptography’.

Figure 3.3 details the completed goal model. This model illustrates how the solution affects the problem. These affects are summarised as follows:

- 1) **Mathematical Description**—symmetric cryptography has a positive affect on data confidentiality through virtue of its operation, however, use of a shared key harms recipient confidentiality.
- 2) **Symmetric Key**—mirrors the effects of the mathematical description but w.r.t. the keys, an artefact of the solution.
- 3) **Symmetric Algorithm**—the specific algorithm will allow for suitable security levels to be selected, but also the choice of algorithm will have an affect on the operational performance.
- 4) **Algorithm Implementation**—the realisation of the algorithm will affect the security of implementation, performance of the algorithm, and how it is introduced into an existing workflow.

Further, the effect that these tasks have on the goal are quantified through contribution levels. These levels were presented in Figure 3.1.

3.3.3 Model Evaluation

The final consideration in this study, is model evaluation. Does the presented solution satisfy the given problem? First, a selection of nodes within the model must be initialised with a suitable satisfaction level. This satisfaction level indicates to what

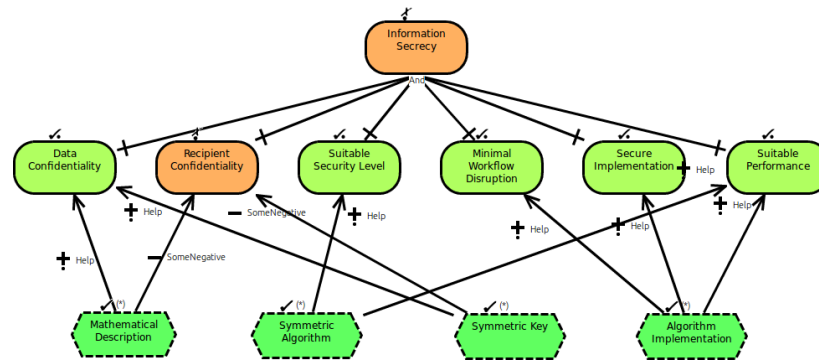


Figure 3.4: Goal model from Figure 3.3 after evaluation.

degree the node has been satisfied. From this, an evaluation algorithm propagates these values across the model. These values were presented in Figure 3.1. For the model under consideration, each of the tasks will be satisfied fully, and the goal nodes left unsatisfied. The model from Figure 3.3 can then be evaluated to determine if Symmetric Cryptography does indeed address the problem of Information Security. Figure 3.4 details the results of the evaluation. These results of the evaluation show that the goal of information security is not achieved as the sub-goal *Recipient Confidentiality* is not achieved.

3.4 Domain Specific Languages

Domain Specific Languages (DSL) are special purpose languages tailored to a specific application domain [Fow10; Ben86]. As DSLs are targeted for a specific application domain their expressiveness is reduced in comparison to general purpose languages. However, the main benefit for using DSL is that of usability. DSLs are constructed to provide domain experts with an environment that is tailored to their domain. Such tailoring allows domain experts to become productive in a familiar environment.

A problem in specifying DSLs is that *all* the functionality required by the domain expert needs to be implemented within the DSL itself. Sometimes this functionality already exists within an existing language. The language itself also needs to be implemented. With DSL construction comes the burden of language and functionality provision. *Embedded Domain Specific Languages* (EDSL) are DSLs that have been embedded within a host language to capitalise upon the host language's functionality and language implementation. The development and use of DSLs and EDSLs have been

seen in many contexts such as programming, document markup, and data querying.

One well-known example of a DSL are macro languages for computing mathematical expressions based upon data contained within a spreadsheet. The language is designed for working with spreadsheets, and not for general purpose programming. Another example is the interface language of the GNU utility `bc`² that provides numeric processing. This language allows users to specify mathematical expressions using well known operators, and not interface with the code implementing `bc` itself. Other well known DSLs include XPath³ and XQuery⁴, DSLs for working with XML.

SHAKESPEARE⁵, is a noteworthy family of EDSLs for working with known web-technologies in Haskell. HTML templating is achieved using HAMLET; CSS specifications with CASSIUS or LUCIUS; and Javascript creation using JULIUS. These EDSLs provide developers with Haskell-oriented constructs to work directly and specifically with the relevant web technologies. Allowing the developer to work in the same environment and use the power of the host language to interact with the presented technologies.

3.5 Domain Specific Modelling Languages

The provision of DSLs is not restricted to programming. DSLs have been seen in other settings, most notably to construct criteria for ‘smart collections’. File browsers and mail applications present users with a means to construct boolean queries. The resulting ‘smart folders’ are then populated with results from the query. A more interesting use of DSLs is related to modelling, specifically in the creation of DSMLs. Generally speaking, DSMLs are the re-use of an existing modelling language for modelling domain specific problems [Fra13].

Outside of modelling generic socio-technical problems, a secondary use of GOMLs is as a host language for the creation of DSMLs [OFK15; LHM14; MWA06; GY01]. UML is a well known family of modelling languages for modelling different aspects of a software system. For instance, Class Diagrams for modelling OO architectures; Component Models for modelling component based architectures; and Activity diagrams for modelling entity interactions. UML provides a generic modelling language and

²<https://www.gnu.org/software/bc/>

³<http://www.w3.org/TR/xpath-31/>

⁴<http://www.w3.org/TR/xquery-30/>

⁵<https://hackage.haskell.org/package/shakespeare>

DSLs have been created to allow for UML modelling using non-standard notation—cf. PLANTUML⁶ & HUTN⁷. UML has also been used as the host language for the creation of Domain Specific Modelling Languages (DSMLs). Two known examples are SYSML [Hauo6] and UMLSEC [Jüro2] for better modelling of software systems and security modelling.

The relationship between host modelling language and domain modelling language differs slightly from that seen with programming language based DSLs. Here the host modelling language (or meta-model) represents a set of semantic and syntactic domain constructs that the domain model either extends or translates their domain knowledge to. Both SYSML and UMLSEC extend UML, and are not embedded within it.

3.6 DSML Creation Techniques

This section details several areas of existing work in DSML construction.

Frank [Fra13] discusses and provides guidance over the design and specification of DSMLs. Detailing for example, selection of host language, requirements elicitation and scrutiny, and evaluation of the resulting language including guidance for graphical notation creation. The framework and techniques presented in this thesis (Chapters 10 and 11) detail the actual acts of DSML creation, and should be seen as a means to implement the phase *development of modelling tool* from guidance presented.

Overbeek et al. [OFK15] details the creation of *GoalML* a DSML for goal modelling within enterprise scenarios. The authors use multi-perspective modelling techniques to provide stakeholders with different views of the scenario being modelling, their GoalML being a single view. The work presented by Overbeek et al. differs from the work presented in this thesis. This thesis is concerned solely with the construction of DSMLs from the host language, Overbeek et al. address the use of goal modelling within the Enterprise domain.

T. Clark and Barn [CB13] provide a *language-driven* approach to DSML creation. Although their target languages are not necessarily goal-oriented the use of formal language design is similar. The authors also look to treat domains as languages providing abstract syntax and semantics, however, they take a different approach in realisation of the language techniques. Rather than using dependent types and the creation of

⁶<http://plantuml.org>

⁷<http://www.omg.org/spec/HUTN/>

interpreters, UML and OCL are used to implement the abstract syntax and semantic mappings to the host language.

3.7 Domain Modelling and the GRL

The GRL has already seen use as a host language for design patterns—see Chapter 2 §2.5.4. With these approaches there is a hard disconnect between the concepts of the design pattern and their interpretation into the constructs of the host language. The domain modelling occurs directly within the host language itself.

When using the GRL as a DSML the resulting model instances will also be a valid GRL instance, however, not all GRL concepts may be used. When using graphical languages, the modeller has to provide a mapping between the syntax and semantics of the host and domain languages. This mapping is provided through a legend, however, the onus is now on the users to ensure correct usage of the host language to model domain concepts.

DSMLs are modelling languages in name only. Although, modellers can utilise existing tooling of the host language to verify the correctness and structure of the resulting model, verification of the DSML is harder to achieve. DSML users themselves have to provide the guarantees over the correctness and structure of their model for their domain in the host language by verifying the correctness of the interpretation of their model into the host language, not to mention correctness of the evaluation semantics.

Moreover, the modeller is free to use the remaining GRL concepts such as correlation links, XOR decomposition or resource and soft goals in their model. How these unused concepts will relate to the modelled domain will not be known, nor is the effect that these untranslated concepts will have when determining goal satisfaction. Further, the translation of concepts from the GRL into the modelled domain must be performed manually by the user.

Finally, visual modelling languages such as GRL are typically presented as standalone programs and their modelling capabilities cannot be used in other programs. For example, the known implementation of the GRL is tied to the Eclipse modelling platform and is provided ‘as is’. The GRL cannot be used ‘as-a-library’ and functionality to work with models outside of model evaluation does not exist.

3.8 Summary

The GRL is a modelling language for reasoning about socio-technical problems. DSMLs are custom modelling languages designed to model a particular domain but use an existing modelling language for semantic (and structural) concepts.

The GRL is one such GOML, however, the GRL is not suitable for acting as the host language for a DSML. The pictorial syntax of the DSML makes formal reasoning about GRL derived constructs a harder, and also distinct, process. With the GRL not only is there a hard disconnect between host and domain language but also the ability to use these models elsewhere is missing.

To address the concerns of designing DSMLs from the GRL, NovoGRL was created. NovoGRL is a language oriented re-engineering of the GRL and is designed as an EDSL within the dependently typed language Idris. Further, NovoGRL also allows for the semantic concepts of the GRL to be re-described for different domains. With the creation of NovoGRL comes the ability to construct DSMLs. NovoGRL was used as the host language for SIF, a DSML for design patterns. SIF has been presented as bespoke tool offering a DSL for modelling design patterns.

DEPENDENT TYPES & WELL-TYPED (ABSTRACT) INTERPRETERS

Programming language theory is the study of how programming languages are created, and provides formal methodologies for reasoning and working with these languages. One of the ideas presented in this thesis is that (programming) language oriented approaches provide for better foundations for the creation and application of both GOMLs, and the use of these languages as a host language for DSMLs.

Programming languages offer a means to encode and represent instructions that a computer can follow. These languages can be modelled formally using a variety of techniques. *Abstract Syntax* to describe language expressions and statements; *type-systems* to govern construction of well-formed expressions; and *semantics* to ensure correct interpretation/execution of expressions. This chapter introduces these concepts through consideration of a simple language ARITH for integer arithmetic and boolean algebra. Part of the material presented in this section is inspired by and adapted from an excellent blog post *Crash Course on Notation in Programming Language Theory* by Siek [Siek].

Further, this chapter also details several styles of construction used within this thesis for the construction of SIF and NovoGRL. Most notably this chapter introduces the reader to dependent types, a programming language construct that allows for software programs to be reasoned on with greater precision.

Note. This chapter provides a comprehensive overview of the technical background required for this thesis, and targeted at those not familiar with programming language design, dependent types, and working at the type level. It is recommended that for those already experienced with programming language design and not with dependent types, should jump to §4.5. For those already experienced with dependent types, you should jump to §4.6.

4.1 The Arith Language

Through-out this chapter the formal description and implementation of a simple language, ARITH will be considered. This language extends the language specified in Siek [Sie12]. The ARITH language allows for the specification of expressions that describe either integer arithmetic or boolean algebra. The supported operators are for:

- **Integer Arithmetic:** Addition, subtraction, multiplication, and division.
- **Boolean Algebra:** Conjunction, disjunction, and negation.

The next section, §4.2, describes how the *syntax* of ARITH is described more formally. §4.3 describes how well-formed ARITH expressions are detailed and reasoned on using types. The remaining sections detail the construction of an interpreter for ARITH and how dependent types provide correctness guarantees towards the interpreters implementation.

4.2 Abstract Syntax

The syntax of programming languages can be described concisely using a formal notation such as *Backus-Naur Form* (BNF). Formal notations define the permissible expressions that are to be found within the language. *Extended-Backus-Naur Form* (eBNF) [ISO96] and *Augmented-Backus-Naur Form* [CO08] are examples of popular variants of BNF.

For ARITH, language expressions include several binary operations on numbers, an unary operation for negation, and parentheses for grouping expressions. An example of a BNF grammar for ARITH is given in Figure 4.1.

Programming Language Theorists are, however, in the business of creating languages and BNF (and its popular variants) is a verbose means to present a language's syntax.

```

1 Expr = Num | Bool
2     | '-' Expr
3     | Expr '+' Expr | Expr '-' Expr
4     | Expr '*' Expr | Expr '/' Expr
5     | Expr '&&' Expr | Expr '||' Expr
6     | '!' Expr
7     | '('? Expr ')'?
8 Num  = [0-9]+
9 Bool = 'true' | 'false'

```

Listing 4.1: The syntax for ARITH described using BNF.

$$e = i \mid b \mid \neg e \mid e + e \mid e - e \mid e / e \mid e * e \mid \neg e \mid e \wedge e \mid e \vee e$$

$$b = \text{True} \mid \text{False}$$

i = place holder for any positive integer

Figure 4.1: The syntax for ARITH.

For programming language theory a particular variant of BNF is used in which the name of the language being defined is replaced by a variable used to range over all the possible values of the language. Figure 4.1 depicts the BNF grammar from Figure 4.1 expressed using more popular notation. The resulting grammar is called *abstract syntax*.

Verbosity of syntax aside, notice how expressions are defined inductively, and that from the syntax alone it appears to be possible to define boolean operations on integers, and integer operations on booleans. The presented abstract syntax only provides a definition of how the language will look, essentially *what we say*, and not if what we say is correct. For the latter, a type system is required.

4.3 Type Systems

Type Systems helps us define a means to *know and reason about language expressions*. *Types* are used in programming languages to differentiate between values, such that the programmer can manipulate these values according to a prescribed set of rules—The Type System. For example, Table 4.2 presents a series of values and their corresponding types.

The set of types for a language are also described using abstract syntax, and is often represented using \mathcal{T} . The ARITH language has two types of constructs: values and

expressions. These form the structure of the language. However, there are two kinds of values and two kinds of expressions: Boolean and Integer. It is these *kinds* that form the type system.

$$\mathcal{T} = \mathcal{B} \mid \mathcal{Z}$$

The type \mathcal{B} , represents boolean values and expressions, and \mathcal{Z} represents integer values and expressions. A type need not be given explicitly for expressions, as the type for an expression will be the type of the objects contained within.

Value	Type	Description
42	Nat	Natural numbers
-273.15	Float	Real numbers
496	Int	Integer numbers
"Cou Cou!"	String	Textual value

Table 4.2: Example pairings of values and their types.

4.3.1 Well-Typed Expressions

Types help us to reason about the correctness of language expressions and ensure that only valid expressions are constructed. The type of an expression is calculated from the types within the expression itself. For example, given the expression $(1 + 2)$. The expression will have type \mathcal{Z} as the result of evaluating the expression is the value 3, which is an integer. Type systems are defined using relations that will allow for the pairing of expressions to types. This relation is also known as *Well-Typed*, and will only contain: *correctly typed expressions paired with their type*. For example:

$$\begin{aligned}((1 + 2), \mathcal{Z}) &\in \text{WellTyped} \\ (\text{True}, \mathcal{B}) &\in \text{WellTyped} \\ (\text{True}, \mathcal{Z}) &\notin \text{WellTyped} \\ (1 + 2 \wedge \text{True}, \mathcal{B}) &\notin \text{WellTyped}\end{aligned}$$

Note we do not have a means (yet) to ensure that only well-typed expressions are constructed. These are typing rules and are introduced in §4.3.3. Before these typing rules can be specified, the idea of *Typing Environments* must first be introduced.

4.3.2 Typing Environments

When working with languages keeping track of what elements in the language have what types is important. For simple languages, such as ARITH, there is no need as there are no variables. The resulting expressions can be clearly disambiguated from each other. In languages with variables, however, simple relations are not enough. *Typing Environments* are used to keep track of local variables and their types so that expressions can be disambiguated by discerning the type of the variable. Traditionally, typing environments are denoted by the Greek letter Γ .

Modelling complete typing environments is not required for ARITH, as the language does not have variables. Typing environments will not be considered. For more information the reader can consult Siek [Siek2]. Regardless, the definition of WellTyped can be improved using the idea of a type-environment by including triples of the form: (Γ, e, \mathcal{T}) . The WellTyped set will contain expressions that have a type \mathcal{T} derived from a local context Γ .

$$(\Gamma, e, \mathcal{T}) \in \text{WellTyped}$$

To save on typing, the short hand $\Gamma \vdash e : \mathcal{T}$ is used. This is read: ‘in a context Γ , e has type \mathcal{T} ’.

4.3.3 Typing Rules

Types and typing environments act as building blocks to help us construct well-typed programs. To construct the set of relations for WellTyped, *Typing Rules* need to be defined that specify how expressions are typed and how types interact when expressions are combined.

What are Typing Rules?

Typing rules are a series of judgements that work in a particular context, with the top line defining the inputs and the bottom line the result. The ARITH language will have the following small set of typing rules. Given that the language has no variables the typing environment will be empty and is represented using the empty set.

Integer Number Arithmetic

$$\text{Numbers } \frac{}{\Gamma \vdash i : \mathbb{Z}} \quad \text{Negation } \frac{\Gamma \vdash e : \mathbb{Z}}{\Gamma \vdash -e : \mathbb{Z}}$$

$$\text{Add} \frac{\Gamma \vdash e_1 : \mathbb{Z} \quad \Gamma \vdash e_2 : \mathbb{Z}}{\Gamma \vdash e_1 + e_2 : \mathbb{Z}} \quad \text{Sub} \frac{\Gamma \vdash e_1 : \mathbb{Z} \quad \Gamma \vdash e_2 : \mathbb{Z}}{\Gamma \vdash e_1 - e_2 : \mathbb{Z}}$$

$$\text{Mult} \frac{\Gamma \vdash e_1 : \mathbb{Z} \quad \Gamma \vdash e_2 : \mathbb{Z}}{\Gamma \vdash e_1 * e_2 : \mathbb{Z}} \quad \text{Div} \frac{\Gamma \vdash e_1 : \mathbb{Z} \quad \Gamma \vdash e_2 : \mathbb{Z}}{\Gamma \vdash e_1 / e_2 : \mathbb{Z}}$$

Boolean Operations

$$\text{Booleans} \frac{}{\Gamma \vdash b : \mathbb{B}} \quad \text{Not} \frac{\Gamma \vdash b : \mathbb{B}}{\Gamma \vdash \neg b : \mathbb{B}}$$

$$\text{And} \frac{\Gamma \vdash e_1 : \mathbb{B} \quad \Gamma \vdash e_2 : \mathbb{B}}{\Gamma \vdash e_1 \wedge e_2 : \mathbb{B}} \quad \text{Or} \frac{\Gamma \vdash e_1 : \mathbb{B} \quad \Gamma \vdash e_2 : \mathbb{B}}{\Gamma \vdash e_1 \vee e_2 : \mathbb{B}}$$

4.4 Interpretation Semantics

So far introduced, is the ability to: (a) model syntax; (b) represent types; and (c) declare well-typed expressions. Generally speaking, semantics provide a means to reason about what expressions in the language *mean*. *Denotational semantics* and *operational semantics* provide a means to reason about what languages do (*denotational*), and how languages are executed—*operational*. For `SIF` this is an interpretation into an instance of `NovoGRL`. For `NovoGRL` this is an interpretation into a graph modelled in `Idris`. The remainder of this section will detail how to provide an interpretation of `ARITH` from the formal notation to the `Idris` programming language. Here the notation $\llbracket e \rrbracket$ will be used to denote the interpretation of an element, that is a transformation from one language to another.

4.4.1 Interpreting Types

Figure 4.2 described how the types in the `ARITH` language are to be interpreted to `Idris` types. Here each type is interpreted directly into its `Idris` equivalent: Integer numbers to `Int`; and booleans to `Bool`.

$$\begin{aligned} \llbracket \mathcal{T} \rrbracket &: \mathcal{T} \rightarrow \text{Type} \\ \llbracket \mathbb{Z} \rrbracket &= \text{Int} \\ \llbracket \mathbb{B} \rrbracket &= \text{Bool} \end{aligned}$$

Figure 4.2: Interpretation and evaluation semantics for the types in `ARITH`.

4.4.2 Evaluating Expressions

Each expression in ARITH will be transformed into their corresponding Idris equivalents, and simultaneously evaluated. Figure 4.3 presents the interpretation and evaluation semantics for ARITH.

$$\begin{aligned}
 \llbracket \mathcal{T} \rrbracket &: \mathcal{T} \rightarrow \llbracket \mathcal{T} \rrbracket \\
 \llbracket i \rrbracket &= i \\
 \llbracket b \rrbracket &= b \\
 \llbracket \neg x \rrbracket &= \text{not } \llbracket x \rrbracket \\
 \llbracket -e \rrbracket &= (-1) * \llbracket e \rrbracket \\
 \llbracket x + y \rrbracket &= \llbracket x \rrbracket + \llbracket y \rrbracket \\
 \llbracket x - y \rrbracket &= \llbracket x \rrbracket - \llbracket y \rrbracket \\
 \llbracket x / y \rrbracket &= \llbracket x \rrbracket \text{ 'div' } \llbracket y \rrbracket \\
 \llbracket x * y \rrbracket &= \llbracket x \rrbracket * \llbracket y \rrbracket \\
 \llbracket x \wedge y \rrbracket &= \llbracket x \rrbracket \&\& \llbracket y \rrbracket \\
 \llbracket x \vee y \rrbracket &= \llbracket x \rrbracket \|\| \llbracket y \rrbracket
 \end{aligned}$$

Figure 4.3: Interpretation and evaluation semantics for ARITH.

Raw values are directly translated into Idris values with type `Int`. Negative numbers are interpreted expressions multiplied by -1 . Finally, the binary operations are mapped directly to their Idris equivalents.

One question that might arise from this section is the computation of the return type for the semantics described in Figure 4.3. Evaluating an ARITH expression will result in a value that has type boolean (\mathcal{B}) or integer— \mathcal{Z} . In many languages being able to change the return type of a function based upon the functions input is not possible. In languages that support full-spectrum dependent types: types can be computed. The next section introduces dependent types further and the many benefits they have.

4.5 Dependent Types

Types are used to distinguish between different values that exist within a programming language. Typically, types are ‘whole’ objects, represent singular concepts, and have a single value. For example: `String`—words; `Int`—whole numbers; and `Person`—a

person. However, types need not be so *whole*, and can contain more information about the values described.

Within several languages polymorphism allows for types to be indexed with other types. A common use of this is the creation of container/collection data types in which the type of the collection is indexed by the type of the element contained within. For example, Table 4.4 illustrates how the Java Generics system supports such descriptions [WNo6]. Here the types are indexed using a secondary type representative of the data contained within. With these polymorphic data structures greater specification of a software program's properties can be given, in conjunction with de-duplication of coding structures for container objects. However, polymorphic classes as seen in Java's collection classes are *only* parameterised using types. Within Haskell, similar polymorphic data structures can be constructed naturally, and also using the language extension of *Generalised Algebraic Data Type* (GADT). However, types can only be indexed using other types. A natural next step is to allow for types to be parameterised by *any value*.

Type	Description
List<String>	List of String Objects.
List<Integer>	List of Integer Objects.
Map<Integer, String>	Mapping between Integers and a String.
Map<Integer, List<Foobar>>	Mapping between Integers and a list of 'Foobar' objects.
Optional<Foobar>	Optional type for an object 'Foobar'.

Table 4.4: Java *Generics* used to describe collections with the type of the elements being described within the type of the collection.

4.5.1 Dependent Types Explained

With dependently typed languages, types are no longer just a 'descriptive label' and can contain more information pertaining to the value being describe. There are also more than parameterised types as provided by GADTs. Within languages that provide full-spectrum dependent types, types are treated as first class language constructs; types can be computed. Allowing for a richer and more expressive type system to be constructed that allows for a program's properties to be specified with greater precision than before. With such information being encoded within the type system itself the type-checker

can now be used to reason about these properties and provide compile time guarantees towards the correctness of software programs. Dependently typed languages such as Idris [Bra13], Agda [Nor09], Epigram [MM04], and Cayenne [Aug98] provide programmers with an unparalleled amount of expressiveness over their software programs.

4.5.2 An Example: ‘Lists with Length’

The power of dependent types can be highlighted through description of the implementation of natural numbers, and their use in defining ‘lists with length’. In these examples, and through-out the remainder of this thesis, the Idris language will be used. For more information about Idris please consult the existing tutorial [Idr15].

Natural Numbers

Mathematically speaking, a natural number is described as any positive integer greater than or equal to zero. This can be encoded using the ‘Peano’ representation [Pea89]. Peano presented a recursive definition for natural numbers that corresponds directly to an ADT in which a natural number is either zero, or the successor of another natural number. This structural representation is useful for the construction of mathematical proofs.

```
data Nat : Type where
  Zero : Nat
  Succ : Nat -> Nat
```

Using this description, the natural numbers 0, 3, and 5 are described as:

- $0 \equiv \text{Zero}$
- $3 \equiv (\text{Succ } (\text{Succ } (\text{Succ } \text{Zero})))$
- $5 \equiv (\text{Succ } (\text{Succ } (\text{Succ } (\text{Succ } (\text{Succ } \text{Zero}))))))$

‘Cons’ Lists

Although `Nat` is not a dependent type, it can be used to construct one. First consider the following definition of a list.

```
data List : (eTy : Type) -> Type where
  Nil  : List eTy
  Cons : eTy -> List eTy -> List eTy
```


The `List` data type is a dependent type parameterised by the type of the elements collected in the list. A list is specified inductively as the empty list (`Nil`), or an element added to the head of another list—`Cons`¹. For example:

```
[5]      ≡ (Cons 5 Nil)
[0,1,1]  ≡ Cons 0 (Cons 1 (Cons 1 (Cons 1 Nil)))
["H", "O", "I"] ≡ Cons "H" (Cons "O" (Cons "I" Nil))
```

Idris supports *syntactic sugar* for more readable list notation in programs. The Idris compiler will convert an expression of the form: `[x, . . . , y]` into a cons-style representation. Any data structure that implements the constructors `Nil` and `(::)` have access to this ‘sugar’.

Vectors

Vectors are comparable to `List` but the type is further indexed by the number of elements contained within the list. The data type `Vect` is defined as:

```
data Vect : (len : Nat) -> (eTy : Type) -> Type where
  Nil  : Vect Zero eTy
  Cons : eTy -> Vect n eTy -> Vect (S n) eTy
```

As the vector is constructed and elements are concatenated to the head of the vector, the length value stored in the type of the vector is incremented by one. With the examples defined earlier, their definition using `Vect` will be:

```
[3]      : Vect 1 Int      A vector of integers with length one.
[0,1,1]  : Vect 3 Int      A vector of integers with length three.
["H", "O", "I"] : Vect 3 String A vector of strings with length three.
```

Concatenation of Lists & Vectors

Representation of a vector’s length using `Nat` ensures that lists of negative length cannot be constructed. Further, with such access to the length of a list in its type more precise descriptions can be used to describe operations on vectors in comparison to lists. For

¹Traditionally, in functional languages based upon ML syntax, the `Cons` constructor is often represented using an infix operator `(::)`. For pedagogical reasons, this chapter uses a traditional constructor. For the remainder of this thesis the infix operator will be used.

example, consider the append operation (`++`) that concatenates two lists. Standard list concatenation is defined as:

```
(++) : List ty -> List ty -> List ty
(++) Nil          ys = ys
(++) (Cons x xs) ys = Cons x (xs ++ xs)
```

The concatenation is achieved by adding each element in the first list (`xs`) to the second list—`ys`. This is the standard definition of list concatenation. The list concatenation function, (`++`), will type check correctly against the type signature. However, the function has been incorrectly implemented; there is a coding mistake. The list that is to be returned is the first list (`xs`) appended to itself. Here the programmer is responsible for ensuring that the implementation of their function acts according to the description contained within the type signature. When using vectors, such problems are mitigated. Compare the following implementation for vector append, with the one above for lists.

```
(++) : Vect a ty -> Vect b ty -> Vect (a+b) ty
(++) Nil          ys = ys
(++) (Cons x xs) ys = Cons x (xs ++ ys)
```

The vector append function details that the resulting vector must not only contain elements of the same type, but the length must be equal to the lengths of the two input vectors combined. If the programmer were to provide an implementation of vector append that returned the first list appended to itself, a type error would be generated. This type error would detail how the resulting length would not be $(a + b)$, thereby reducing the possible set of errors that can be made by the programmer. By reasoning about the length of a vector and the type of elements contained within, stronger more precise specifications can be given towards operations and structures that use vectors.

4.5.3 Why Dependent Types?

Dependent type systems allow for types to depend on values. Use of these values can lead to more precise specifications to be presented, and thereby reducing the possible set of errors that can be encountered. By offering a language that treats types as first-class language objects and also provides full-spectrum dependent types, programmers are given a powerful tool to specify and develop programs. Such power is useful when constructing EDSLs; this is covered in the next section.

```
1 data Arith = Num Int           | Boolean Bool
2           | Neg Arith          | Not Arith
3           | Add Arith Arith    | Sub Arith Arith
4           | Div Arith Arith    | Mul Arith Arith
5           | And Arith Arith    | Or  Arith Arith
```

Listing 4.2: A simple definition of ARITH as an inductive ADT.

4.6 Well-Typed Interpreters

Augustsson and Carlsson [AC99] demonstrated how dependent types are used in the creation of an interpreter for the Typed λ -Calculus [Bar92]. Here, Augustsson and Carlsson modelled the calculus as an EDSL within Cayenne, a dependently typed language [Aug98] of the time. The resulting interpreter is well-typed as it allows for static compile guarantees over the types of expressions. This section takes the ARITH language from earlier in the chapter and constructs a well-typed interpreter for it within Idris, together with a description of how the formalisms are implemented.

4.6.1 Language Definition

Figure 4.1 presented the abstract syntax for ARITH. Within functional languages the expressions for ARITH can be encoded as an inductive ADT. Listing 4.2 presents one such definition.

However, there are problems with this definition when it comes to implementing the typing rules for the language. The types for an EDSL can be modelled within Idris using an enumerated type². For example, the set of types \mathcal{T} in ARITH can be encoded in Idris as follows:

```
data ArithTy = TyNum | TyBool
```

In non-dependently typed functional languages typing rules can be modelled through pattern matching, and the creation of an interpretation function that evaluates the expressions according to their expected types. For example, using the language definition given in Listing 4.2, a partial description of an evaluation function using non-

²More complex types can be defined, but for the purposes of this explanation a simple enumerated type is sufficient.

dependently typed expressions can be described as follows. First, a data type to collect evaluation results is defined:

```
data EvalRes = Err String | BRes Bool | IRes Int
```

Evaluation will result either in an error if the example is ill-typed, a boolean value for evaluating boolean expressions, or an integer for evaluating integer expressions. However, to aid in creation of the evaluation function, `EvalRes` can be turned into a *Functor* and then turned into an *Applicative* data structure. Both functors and applicatives are functional programming techniques that provide for more abstract and generalised programming. This tutorial is now starting to rely upon the need of more complicated programming language techniques to demonstrate show to implement the typing rules.

An alternative approach to implementing the typing rules is to still encapsulate the result in *EvalRes*, but use patterns and pattern matching to pass the expressions around. Listing 4.3 presents such an implementation of the `eval` function.

Within Listing 4.3, `evalNum` and `evalBool` are used to implement the evaluation and error handling for dealing with numerical and boolean expressions respectively, and also where `negate` negates arithmetic expressions. Only a naïve definition for `evalBool` is given, as well as helper functions `doAnd` and `doOr` that implement conjunction and disjunction of boolean terms.

However, this method of implementation for the evaluator for `ARITH` is not the best approach. First, it is rather verbose and requires the creation of custom data type to handle errors and report successes. Second, it requires the creation of functions for each expression, and also the creation of the interpreter. And third, it facilitates the construction of ill-typed expressions that are only detected at run-(née)-evaluation time.

Recall, that dependent types are types that can depend on values. With dependent types, a better implementation of the typing rules involves parameterising the ADT that represents expressions further by the enumerated type representing types in the language—see Listing 4.4. What follows is a *direct embedding* of the typing rules directly within the types of the expressions.

With this representation and construction of the language as an EDSL within Idris, the ability to detect ill-typed expressions now becomes a compile time check.

4. WELL-TYPED (ABSTRACT) INTERPRETERS

```
1 eval : Arith -> EvalRes
2 eval e@(Num x) = evalNum e
3 eval e@(Bool x) = evalBool e
4
5 eval e@(Not x) = evalBool e
6 eval e@(And x y) = evalBool e
7 eval e@(Or x y) = evalBool e
8
9 eval e@(Add x y) = evalNum e
10 eval e@(Sub x y) = evalNum e
11 eval e@(Mul x y) = evalNum e
12 eval e@(Div x y) = evalNum e
13 eval e@(Neg x) = negate x
14
15 evalBool : Arith -> EvalRes
16 evalBool (Boolean b) = BRes b
17 evalBool (And x y) = doAnd (evalBool x) (evalBool y)
18 evalBool (Or x y) = doOr (evalBool x) (evalBool y)
19 evalBool (Not x) = not x
20 evalBool _ = Err "NaB"
21
22 doAnd : EvalRes -> EvalRes -> EvalRes
23 doAnd (BRes x) (BRes y) = BRes $ x && y
24 doAnd _ _ = Err "Not BoolExpr"
25
26 doOr : EvalRes -> EvalRes -> EvalRes
27 doOr (BRes x) (BRes y) = BRes $ x || y
28 doOr _ _ = Err "Not BoolExpr"
```

Listing 4.3: Naïve implementation of an evaluation function for `Arith`.

```
1 data Arith : ArithTy -> Type where
2   Num      : Int -> Arith TyNum
3   Boolean  : Bool -> Arith TyBool
4
5   Neg      : Arith TyNum -> Arith TyNum
6   Add      : Arith TyNum -> Arith TyNum -> Arith TyNum
7   Sub      : Arith TyNum -> Arith TyNum -> Arith TyNum
8   Div      : Arith TyNum -> Arith TyNum -> Arith TyNum
9   Mul      : Arith TyNum -> Arith TyNum -> Arith TyNum
10
11   And      : Arith TyBool -> Arith TyBool -> Arith TyBool
12   Or       : Arith TyBool -> Arith TyBool -> Arith TyBool
13   Not      : Arith TyBool -> Arith TyBool
```

Listing 4.4: Example of using Dependent Types to embed and model typing rules for boolean and integer arithmetic expressions directly in the ADT representing language expressions.

Moreover, less code is required to ensure that expressions are well typed. This guarantees *correctness-by-construction* of language expressions.

The ARITH language does not have variables, and as such the typing environment for Idris can be borrowed completely for modelling the language. If we were to provide variables, then being able to track the types of variables is essential. For more information how to achieve this in dependently typed languages readers are asked to consult for more information: Augustsson and Carlsson [AC99], Brady and Hammond [BH12] and The Idris Community [Idr15].

Remark. An alternative means to model this simple typed Arithmetic language is to introduce functions types in the type to describe operations. This will allow for a more compact and stronger definition. The implementation of the Well-Typed Interpreter in The Idris Community [Idr15], demonstrates this technique.

4.6.2 Implementing The Interpreter

With the implementation of the ARITH language complete, the remainder of this section discusses how to implement the interpreter. Evaluating expressions from ARITH will result in a value that is either a boolean or integer type. An interesting question when constructing the interpreter is: *How to return a value of the correct type?* When exploring earlier how to implement typing rules this was modelled using an ADT. Within dependently typed languages, types can be computed using a function that returns the correct type.

```

1 interpTy : ArithTy -> Type
2 interpTy TyNum   = Int
3 interpTy TyBool  = Boolean

```

Listing 4.5: A Type Interpreter for the Arith language.

Within Idris the interpretation semantics defined in Figure 4.2 can be represented as a function that when given a value returns the appropriate type. This function, `interpTy` is detailed in Listing 4.5. Each type in the model is interpreted directly to its Idris equivalent. This function will be used in the definition of the interpreter to ensure that the result returned has the correct type. Further, notice the similarity between the definition of `interpTy` and the formalism given in Figure 4.2.

The focus now turns to expression evaluation, and how the expressions in ARITH are transformed into their Idris equivalents and simultaneously evaluated. This interpretation was presented in Figure 4.3. Listing 4.6 presents a possible implementation in Idris.

As with the implementation of the type interpreter, note the similarities between the formal representation, and implementation.

```
1 interp : Arith t -> interpTy t
2 interp (Num x)      = x
3 interp (Boolean x) = x
4 interp (Neg x)      = (-1) * (interp x)
5 interp (Not x)      = not (interp x)
6 interp (Add x y)    = (interp x) + (interp y)
7 interp (Sub x y)    = (interp x) + (interp y)
8 interp (Div x y)    = (interp x) `div` (interp y)
9 interp (Mul x y)    = (interp x) * (interp y)
10 interp (And x y)   = (interp x) || (interp y)
11 interp (Or x y)    = (interp x) && (interp y)
```

Listing 4.6: Implementation of the evaluation semantics for ARITH in Idris

A well-typed interpreter for the ARITH language has now been constructed. Dependent types allows for these constructs to be implemented efficiently and concisely, Generally speaking, Idris itself has good support for defining more advanced EDSLs and with concepts not treated here—see Brady and Hammond [BH12]. However, the correctness of the evaluation function has not yet been guaranteed. As with the implementation of list concatenation presented in §4.5.2, there is a mistake in the implementation. Lines 6 & 7 that details the evaluation of addition and subtraction expressions for integer numbers has been evaluated using addition in both cases. Line 7 should subtract the interpretation of x from y . The next section details an approach that can be used to avoid such mistakes.

4.7 Types as (Abstract) Interpreters

As the complexity of a language grows, it becomes increasingly more difficult to reason formally about the language. In other cases the programmer might make a simple mistake during implementation, for example the mistakes seen with list concatenation (§4.5.2) and the interpreter for the ARITH language—§4.6.2. If it can be shown, how-

ever, that expressions in the language can be mapped to expressions from an existing formalism then the formalism can be used to reason about the language itself. If the execution costs of the formalism are lightweight in comparison to the original language then this modelling can be said to be *efficient*. This is a core concept in the technique of *Abstract Interpretation* [JN95].

Abstract interpretation allows for correctness guarantees over a language to be given in direct relation to an easier to model formalism. This technique has found success in a variety of practical settings. One is the analysis of program execution for optimisations by compilers. Another the analysis of program flow for detecting erroneous states, as used by static analysis tools.

This section describes the technique of abstract interpretation and how, using dependent types, abstract interpretations can be used to provide compile-time correctness-by-construction guarantees and run-time checks to a language w.r.t. to a given abstraction.

4.7.1 ‘Casting Out the Nines’

To illustrate the technique of abstract interpretation, Jones and Nielson [JN95] show how arithmetic computations are checked for correctness using the technique known as: ‘Casting out the Nines’. This same illustrative example is repeated here.

Given an arithmetic expression e that evaluates to some value v , the same operations in e are performed. However, for each value in e that is greater than nine, the value is replaced with the sum of its digits until the sum is less than nine. For the expression e to be correct w.r.t. to the value v , the sum modulo nine of the digits of v must be equal to the value obtained through ‘casting out the nines’ for e . Let the calculation to be checked, be defined as follows:

$$123 \times 457 + 76543 \stackrel{?}{=} 132654$$

The result 132654 can be checked by reducing the expression on the left-hand side by ‘casting out the nines’, and calculating the sum modulo nine of the digits on the right-hand side:

$$\begin{aligned}123 \times 457 + 76543 &\stackrel{?}{=} 132654 \\6 \times 16 + 25 &\stackrel{?}{=} 21 \pmod{9} \\6 \times 7 + 7 &\stackrel{?}{=} 3 \\42 + 7 &\stackrel{?}{=} 3 \\6 + 7 &\stackrel{?}{=} 3 \\4 &\stackrel{?}{=} 3\end{aligned}$$

The results of the calculation performed on both the left and right hand sides (four and three) are not equal. Hence, the answer found by the calculation is incorrect. Modelling arithmetic operations, and thus checking the result of, using the cast the nines technique allows for correctness guarantees to be made. As the complexity of the calculations grow, the complexity of the correctness proof does not. One *casts out the nines*.

4.7.2 Working with Abstract Interpretations

A problem, however, with the approach of abstract interpretation is that this modelling and reasoning of a language w.r.t. an interpretation is normally performed external to the implementation of the modelled language. There is a disconnect between the two representations. As types in a dependently typed language can contain any value, it stands to reason that a dependent type can be used to capture at the type level the abstract interpretation of a given language. Brady [Bra05, Chapter 5] demonstrated this approach by modelling operations on GMP integers with an abstract interpretation of the same operations by using natural numbers. Capturing the abstraction at the type level allows for compile time correctness-by-construction guarantees to be given over the presented language; to build the language correctly, the abstract interpretation must also be constructed correctly. Further, by having access to the abstraction at run-time allows for further checks to be made.

The work of Brady [Bra05] on using dependent types as abstract interpreters has been taken further. Preliminary work by Castro et al. [Cas+15] demonstrates how these techniques can be used to reason about structured parallel programs. The work presented in this thesis is an application of the same techniques but used to provide homomorphisms between DSMLs and the GRL.

4.7.3 The Simplified Arith Language

Recall the ARITH language from §4.2. Its complexity in operating over two domains of operation (integer arithmetic and boolean algebra) is not suitable for introducing how an abstract interpretation can be modelled using dependent types. To simplify the example, the support for boolean algebra is dropped, the reformulation will support integer arithmetic only³.

§4.7.1 has already shown how integer arithmetic can be checked using a ‘cast nines’ abstraction. The remainder of this section introduces the interpretation semantics for transforming expressions in ARITH to the ‘cast nines’ representation, and details its implementation using dependent types. The same formal representation for ARITH as presented in §4.2 will be used to represent the simplified ARITH language.

4.7.4 Implementing an (Abstract) Interpreter for Arith

Suppose there is a function `castNine` that sums the digits of a number until the resulting value is less than nine. Figure 4.4 presents interpretation semantics for constructing the ‘cast nine’ abstraction of an expression from ARITH.

$$\begin{aligned}
 \llbracket \text{Arith } e \rrbracket &: \text{Arith} \rightarrow \mathbb{Z} \\
 \llbracket \text{Num } n \rrbracket &= \text{castNine}(n) \\
 \llbracket \text{Neg } n \rrbracket &= (-1) * \text{castNine}(n) \\
 \llbracket \text{Add } x \ y \rrbracket &= \text{castNine}(\llbracket x \rrbracket + \llbracket y \rrbracket) \\
 \llbracket \text{Sub } x \ y \rrbracket &= \text{castNine}(\llbracket x \rrbracket - \llbracket y \rrbracket) \\
 \llbracket \text{Mul } x \ y \rrbracket &= \text{castNine}(\llbracket x \rrbracket * \llbracket y \rrbracket) \\
 \llbracket \text{Div } x \ y \rrbracket &= \text{castNine}(\llbracket x \rrbracket / \llbracket y \rrbracket)
 \end{aligned}$$

Figure 4.4: Interpretation semantics for abstracting ARITH expressions into a ‘Cast Nine’ abstraction.

As with the description of the ‘cast nine’ approach in §4.7.1, each expression in ARITH has the values first converted to their ‘cast nine’ abstraction prior to the operation being performed. The inductive interpretation presented in Figure 4.4 will take any ARITH expression and compute the resulting abstract representation.

³Chapter 11 will detail techniques that provides abstractions for both boolean algebra and integer arithmetic to be modelled.

Traditional Implementation

```
1 convert : Arith ty -> Int
2 convert (Num n ) = castNine n
3 convert (Neg n ) = (-1) * convert n
4 convert (Add x y) = castNine $ (convert x) + (convert y)
5 convert (Sub x y) = castNine $ (convert x) - (convert y)
6 convert (Mul x y) = castNine $ (convert x) * (convert y)
7 convert (Div x y) = castNine $ div (convert x) (convert y)
```

Listing 4.7: Traditional Implementation of Interpretation Semantics from Figure 4.4

Traditionally computing the ‘cast nine’ representation of ARITH expressions required the construction of an evaluation function separate from the standard evaluation function. For example, Listing 4.7 details one such function. In this example `castNine` is a pre-given function used to calculate the ‘cast nine’ representation of a given integer. Notice, that the results of the operations are passed through the `castNine` function to ensure that the abstraction is provided.

When constructing an evaluation function, the `convert` function from Listing 4.7 can be used to convert the expression to its ‘cast nine’ representation. This is shown in Listing 4.8, where `doEval` evaluates `Arith` expressions. The actual result of the evaluation (from `doEval`) is converted using `sumMod9` that sums the digits mod nine of the result. If the resulting values are equal (i.e. the result of calling `convert` and `sumMod9`) then a run-time soundness guarantee can be made over the result of `doEval`.

```
1 eval : Arith ty -> Maybe Int
2 eval expr = let res = doEval expr in
3   if sumMod9 res == convert expr
4     then (Just res)
5     else Nothing
```

Listing 4.8: Evaluation function for `Arith` constructed using traditional techniques.

New Implementation

§4.6 introduced the *Well-Typed Interpreter* style of implementing the ARITH language. As types can be parameterised by more than one value, the data type used to represent ARITH expressions can be further parameterised by their ‘cast nine’ abstraction.

```

1 data Arith : ArithTy -> Int -> Type where
2   Num : (v : Int) -> Arith TyNum (castNine v)
3   Neg : Arith TyNum a -> Arith TyNum (-1 * a)
4   Add : Arith TyNum a
5         -> Arith TyNum b
6         -> Arith TyNum (castNine (a + b))
7   Sub : Arith TyNum a
8         -> Arith TyNum b
9         -> Arith TyNum (castNine (a - b))
10  Mul : Arith TyNum a
11        -> Arith TyNum b
12        -> Arith TyNum (castNine (a * b))
13  Div : Arith TyNum a
14        -> Arith TyNum b
15        -> Arith TyNum (castNine (a `div` b))

```

Listing 4.9: Dependently Typed Implementation of the Language Grammar & Interpretation Semantics from Figure 4.4

With dependent types the interpretation of constructs from `Arith` to their ‘cast nine’ representation now occurs directly within the type of `Arith`. Previously, this was achieved using an external function—see `convert` from Listing 4.7. Using this representation, not only do `ARITH` expressions have to be well-typed, but they must also be valid expressions in the ‘cast nine’ abstraction as well. This connection allows for correctness-by-construction guarantees to be made w.r.t. to a given abstraction. Listing 4.10 illustrates how the abstract representation is made accessible during runtime using Idris’ ability to access implicit type-level values and bring them down to the value level.

```

1 eval : Arith ty a -> Maybe Int
2 eval expr {a} = let res = doEval expr in
3   if sumMod9 res == a
4     then (Just res)
5     else Nothing

```

Listing 4.10: Evaluation function for `Arith` constructed using new techniques.

4.8 Summary

Programming language theory provides a series of techniques useful for defining and working with languages. Formal grammars are used for defining abstract syntax; types

to describe expressions, typing rules to express correct composition of expressions; and semantics to describe a language's interpretation. This section has only covered the basics of programming language theory, and more topics such as dealing with variables have not been covered. Regardless, the knowledge presented in this section is enough to understand how a declarative EDSL can be modelled and constructed within a dependently typed language such as Idris. This is the style of construction that the modelling languages in this thesis are presented.

Using dependently typed languages such as Idris provides programmers with an environment to support compact, efficient, and correct implementations of EDSLs. This was shown with the *Well-Typed Interpreter* for the ARITH language. These techniques are used in the implementation of SIF and NOVOGRL, and the tooling to support pattern document interaction.

The *Types as (Abstract) Interpreter* approach has illustrated how an abstract interpretation for a language can be represented directly within the type of the expressions representing the language. Such modelling allows for compile time correctness-by-construction guarantees to be made, and also runtime checks to be made available. It is using this technique that the SIF language was implemented with NOVOGRL being used as the abstract interpretation. Chapter II discusses this approach in more depth.

SIF: A DESIGN PATTERN MODELLING LANGUAGE

This chapter introduces and details the SIF language and evaluator. Presenting its design (§5.1 to 5.3); important aspects of the evaluator implementation (§5.4); and evaluation of the language to model existing, and new design patterns—§5.5.

5.1 Overview

SIF is a requirements-based goal-oriented DSML for prototyping design patterns, that uses NovoGRL (Chapter 10) as a host language. The language has been designed as a declarative DSML for design patterns that respects the pattern trioka of problem×solution×context. Problem specifications are modelled separately from their potential solutions such that different problem solution pairs can be combined and evaluated to determine how well the given solution satisfies the presented problem. Problems and solutions are parameterised by a domain of operation (i.e. context) such that only problems and solutions indexed by the same domain can be paired.

Specifically, SIF problem specifications are modelled as requirements specifications based upon the FURPS requirements model [Gra92] in which requirements are categorised according to how they relate to the system being modelled. The categories supported by FURPS are: Functional; Usability; Reliability; Performance; or Sup-

portability. Problems are presented using textual descriptions, with associated forces presented as a set of requirements that must be addressed by a solution. Use of FURPS provides a more nuanced requirements model to be provided in comparison to that of the GRL. The GRL only provides goals and soft-goals, representing functional and non-functional requirements.

Solution specifications are not designed per the software artefacts of the presented solution. Not all software design patterns are software based. Rather solution specifications are presented as a set of abstract properties that represent the different aspects of the solution. A concrete link between a problem and solution is provided in the form of traits that describe the advantages, disadvantages, and general aspects of the property and the effect that these traits have on the requirements specified in the problem. Each ‘affect’ that a trait has on a problem is detailed using a qualitative contribution value originating from the GRL. Further, for each trait specified an evaluation value (also taken from the GRL) must be given that describes the level of satisfaction that a modeller has in the existence of said trait.

Alongside the language specification is the SIF evaluator, developed in the dependently typed programming language Idris. This evaluator provides a reference implementation of the SIF language as both a DSML and *Embedded Domain Specific Modelling Language* (EDSML), and facilitates the creation of pattern document stubs in various document formats, including that of FREYJA—Chapter 6.

5.2 A DSML for Patterns

Existing work (Chapter 2 §2.5.4) investigated the use of the GRL to model design patterns and pattern languages. Although, the GRL does provide a rich set of semantics to model socio-technical systems the use of the GRL to model design patterns is nonetheless problematic. The GRL does not support pattern specific concepts in its language, such as problems, solutions, forces, and contexts. This is reasonable as the GRL is designed for modelling socio-technical systems and not design patterns. Nonetheless, how pattern concepts are to be modelled in the GRL is open to debate.

With knowledge of DSMLs creation from Chapter 3 it makes sense to use the GRL as a host language for a DSML that supports the modelling of design patterns but leverages the concepts from the GRL. Here, the GRL is used as a meta-model to which the concepts in SIF are interpreted, and from which the evaluation semantics originate.

Construction of valid GRL models will thus allow for these models to be evaluated using existing techniques developed for the GRL. The support for socio-technical modelling in the GRL makes it a good host language for modelling design patterns. With a DSML approach how pattern concepts are transformed into GRL concepts can be formally specified. Moreover, modellers are presented with constructions and idioms that are bespoke to design patterns.

A secondary concern is practical in nature. The GRL is a visual language that does not support directly DSML creation. Chapter 10 presents a reformulation of the GRL, NovoGRL to allow for creation of DSMLs.

5.3 Language Specification

This section presents the formal language specification for SIF detailing the abstract syntax, type system, and interpretation into a GRL model instance. Chapter 10 introduces and explains the syntax for the formal GRL notation used in this chapter.

5.3.1 Abstract Syntax

Figure 5.1 presents the abstract syntax for SIF together with nominal typing information. The type-system is explained further in §5.3.2. Documentation within SIF is treated as a first class language construct. Many of the core language constructs require that a descriptive title is to be given explicitly, together with an optional textual description. Such first-class treatment ensures that information can be collected and used in the generation of the pattern document stubs. Much like their design, problem specifications are presented as constructs that require a title and description, and a set of requirements. Requirements are given a title and optional description, and different constructors are used to denote to which requirement type the requirement belongs to in the FURPS model. Solution specifications are presented as a list of titled properties that require a set of traits to be given. For each trait in the property, the satisfaction value must be given together with a list of ‘affects’ that detail the effect that the trait has on a requirement. Finally, domains of operation can be constructed that are used within the type-system to ensure that the language constructs used within a single pattern only operate within the same domain.

$$\begin{aligned}
\text{SIF} &= \text{SIF} \mid \varphi \mid p \mid \gamma \mid r \mid s \mid a \mid t \mid l \\
\varphi \in \Phi(\gamma) &= \text{Pattern } t \ d \ p \ s \\
p \in \mathcal{P}(\gamma) &= \text{Problem } t \ d \ \{r_1, \dots, r_n\} \\
r \in \mathcal{R}(\gamma) &= \text{Functional } t \ d \mid \text{Usability } t \ d \\
&\quad \mid \text{Reliability } t \ d \mid \text{Performance } t \ d \\
&\quad \mid \text{Supportability } t \ d \\
s \in \mathcal{S}(\gamma) &= \text{Solution } t \ d \ \{a_1, \dots, a_n\} \\
a \in \mathcal{A}(\gamma) &= \text{Property } t \ d \ \{t_1, \dots, t_n\} \\
t \in \mathcal{E}(\gamma) &= \text{Adv } t \ d \ q \ \{l_1, \dots, l_n\} \mid \text{Dist } t \ d \ q \ \{l_1, \dots, l_n\} \\
&\quad \mid \text{Gen } t \ d \ q \ \{l_1, \dots, l_n\} \\
l \in \mathcal{L}(\gamma) &= \text{Affect } c \ r \ d \\
\gamma \in \mathcal{G} &= \text{Domain } t \ d \\
q \in \mathcal{Q} &= \text{Denied} \mid \text{wDenied} \mid \text{wSatisfied} \mid \text{Satisfied} \\
&\quad \mid \text{Conflict} \mid \text{Unknown} \mid \text{None} \\
c \in \mathcal{C} &= \text{Make} \mid \text{Help} \mid \text{SomePos} \mid \text{Unknown} \\
&\quad \mid \text{SomeNeg} \mid \text{Break} \mid \text{Hurt} \\
t, d \in \text{String} &= \text{String Values}
\end{aligned}$$

Figure 5.1: Abstract Syntax for the SIF modelling language.

5.3.2 Type-System

$$\begin{aligned}
\mathcal{T} &= \mathcal{G} \mid \mathcal{Q} \mid \mathcal{C} \mid \Phi : \mathcal{G} \rightarrow \mathcal{T} \mid \mathcal{P} : \mathcal{G} \rightarrow \mathcal{T} \mid \mathcal{R} : \mathcal{G} \rightarrow \mathcal{T} \\
&\quad \mid \mathcal{S} : \mathcal{G} \rightarrow \mathcal{T} \mid \mathcal{A} : \mathcal{G} \rightarrow \mathcal{T} \mid \mathcal{E} : \mathcal{G} \rightarrow \mathcal{T} \mid \mathcal{L} : \mathcal{G} \rightarrow \mathcal{T}
\end{aligned}$$

Figure 5.2: Types in the SIF modelling language.

Dependent types allow for types to depend on values. The types used to describe pattern constructs are parameterised by the domain of operation that these constructs must exist in. Use of dependent types in this manner allows for modelling constructs to change domain yet remain structurally intact. Further, dependent types allows for type-level guarantees to be made that problems and solutions must be in the same context when being paired. Figure 5.2 presents the types for SIF, these types have the

following semantic meaning.

- \mathcal{G} the type given to a domain of operation.
- \mathcal{Q} the type given to satisfaction values.
- \mathcal{C} the type given to contribution values.
- Φ the dependent type given to all patterns for a given domain.
- \mathcal{P} the dependent type given to all problems for a given domain.
- \mathcal{R} the dependent type given to all requirements for a given domain.
- \mathcal{S} the dependent type given to all solutions for a given domain.
- \mathcal{A} the dependent type given to all properties for a given domain.
- \mathcal{E} the dependent type given to all traits for a given domain.
- \mathcal{L} the dependent type given to all affects for a given domain.

With the set of types given, the typing rules for the language can be presented. With these typing rules *correctness by construction* guarantees can be made with SIF model instances. If a model is well-typed then it will also be structurally correct.

Problems and Requirements

First the typing rules for declaring problems and requirements are presented.

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Functional } t \ d): \mathcal{R}(\gamma)}$$

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Usability } t \ d): \mathcal{R}(\gamma)}$$

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Reliability } t \ d): \mathcal{R}(\gamma)}$$

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Performance } t \ d): \mathcal{R}(\gamma)}$$

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Supportability } t \ d): \mathcal{R}(\gamma)}$$

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash rs:\text{List } (\mathcal{R}(\gamma)) \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Problem } t \ d \ rs): \mathcal{P}(\gamma)}$$

Traits and Affects

Next the rules governing creation of traits and affects are presented.

$$\frac{\Gamma \vdash c:\mathcal{C} \quad \Gamma \vdash r:\mathcal{R}(\gamma) \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Affect } c \ r \ d):\mathcal{L}(\gamma)}$$

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash as:\text{List}(\mathcal{L}(\gamma)) \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Adv } t \ d \ as):\mathcal{E}(\gamma)}$$

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash as:\text{List}(\mathcal{L}(\gamma)) \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Gen } t \ d \ as):\mathcal{E}(\gamma)}$$

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash as:\text{List}(\mathcal{L}(\gamma)) \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Dist } t \ d \ as):\mathcal{E}(\gamma)}$$

Solutions and Properties

The typing rules for solutions and properties are presented as follows.

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash as:\text{List}(\mathcal{E}(\gamma)) \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Property } t \ d \ as):\mathcal{A}(\gamma)}$$

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash as:\text{List}(\mathcal{A}(\gamma)) \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Solution } t \ d \ as):\mathcal{S}(\gamma)}$$

Pattern

Finally the typing rule that governs the creation of patterns is given.

$$\frac{\Gamma \vdash t:\text{String} \quad \Gamma \vdash d:\text{String} \quad \Gamma \vdash p:\mathcal{P}(\gamma) \quad \Gamma \vdash s:\mathcal{S}(\gamma) \quad \Gamma \vdash \gamma:\mathcal{G}}{\Gamma \vdash (\text{Pattern } t \ d \ p \ s):\Phi(\gamma)}$$

5.3.3 Interpretation Semantics

This section details the interpretation semantics for constructing NovoGRL model instances¹ from a SIF model instance. Figure 5.3 presents the interpretation semantics for SIF using set notation to succinctly describe operations on lists. The interpretation is a two stage process: first the problem is interpreted into a goal model; and secondly, the solution is interpreted into a list of declarations that describe the elements of the interpreted solution and their links to the goals and other elements in the model. The

¹As a reminder, the definition of NovoGRL syntax is detailed in Chapter 10.

$$\begin{aligned}
& \llbracket \text{SIF} \rrbracket : \text{SIF} \rightarrow \text{GRL} \\
& \llbracket \text{Pattern } t \text{ d } w \text{ s} \rrbracket = \llbracket w \rrbracket \uplus^* \{x \mid x \leftarrow \llbracket s \rrbracket\} \\
& \llbracket \text{Problem } t \text{ d } \{r_1, \dots, r_n\} \rrbracket = \emptyset \uplus r \uplus^* cs \uplus (r \wedge cs) \\
& \quad \text{where} \\
& \quad r = \text{Goal } t \text{ Unknown} \\
& \quad cs = \{\llbracket r \rrbracket \mid r \leftarrow \{r_1, \dots, r_n\}\} \\
& \llbracket \text{Functional } t \text{ d} \rrbracket = \text{Goal } t \text{ Unknown} \\
& \llbracket \text{Usability } t \text{ d} \rrbracket = \text{Goal } t \text{ Unknown} \\
& \llbracket \text{Reliability } t \text{ d} \rrbracket = \text{Goal } t \text{ Unknown} \\
& \llbracket \text{Performance } t \text{ d} \rrbracket = \text{Goal } t \text{ Unknown} \\
& \llbracket \text{Supportability } t \text{ d} \rrbracket = \text{Goal } t \text{ Unknown} \\
& \llbracket \text{Solution } t \text{ d } \{a_1, \dots, a_n\} \rrbracket = n \cup xs \cup ds \cup (n \wedge xs) \\
& \quad \text{where} \\
& \quad n = \text{Goal } t \text{ Unknown} \\
& \quad (xs, ds) = \{(\bigcup x, \bigcup xs) \mid (x, xs) \leftarrow \llbracket a \rrbracket, a \leftarrow \{a_1, \dots, a_n\}\} \\
& \llbracket \text{Property } t \text{ d } \{a_1, \dots, a_n\} \rrbracket = (n, n \cup xs \cup (\bigcup dds) \cup (n \wedge xs)) \\
& \quad \text{where} \\
& \quad n = \text{Goal } t \text{ Unknown} \\
& \quad (xs, dds) = \{(\bigcup x, \bigcup rs) \mid (x, rs) \leftarrow \llbracket a \rrbracket, a \leftarrow \{a_1, \dots, a_n\}\} \\
& \llbracket \text{Adv } t \text{ d } q \{l_1, \dots, l_n\} \rrbracket = (n, \{n \xrightarrow{c} r \mid (c, r) \leftarrow \llbracket l \rrbracket, l \leftarrow \{l_1, \dots, l_n\}\}) \\
& \quad \text{where} \\
& \quad n = \text{Task } t \text{ q} \\
& \llbracket \text{Dist } t \text{ d } q \{l_1, \dots, l_n\} \rrbracket = (n, \{n \xrightarrow{-c} r \mid (c, r) \leftarrow \llbracket l \rrbracket, l \leftarrow \{l_1, \dots, l_n\}\}) \\
& \quad \text{where} \\
& \quad n = \text{Task } t \text{ q} \\
& \llbracket \text{Gen } t \text{ d } q \{l_1, \dots, l_n\} \rrbracket = (n, \{n \xrightarrow{c} r \mid (c, r) \leftarrow \llbracket l \rrbracket, l \leftarrow \{l_1, \dots, l_n\}\}) \\
& \quad \text{where} \\
& \quad n = \text{Task } t \text{ q} \\
& \llbracket \text{Link } c \text{ r } d \rrbracket = (c, \llbracket r \rrbracket)
\end{aligned}$$

Figure 5.3: Interpretation semantics for converting SIF expressions into GRL constructs.

problem itself becomes the top most goal of the NovoGRL model, with the requirements in the problem specification being interpreted as sub-goals linked using ‘and’ decomposition. Solutions are interpreted into a list of declarations. These declarations specify a secondary goal model that is composed of tasks that represent the constituent components of a solution specification (i.e. the solution itself, properties, traits and affects) arranged hierarchically using AND decomposition links. The resulting NovoGRL model will have a task representing the solution as its top most task with sub tasks representing properties, and so on. The ‘affects’ model elements, represented by the Link constructor are used to link traits directly to requirements in the problem using contribution links with the satisfaction value being specified in the SIF model. The satisfaction values given to traits are also used to provide a default satisfaction value for their GRL interpretation. For traits that represent disadvantages their satisfaction values are inverted to enable negative satisfaction values to be propagated through out the model. For instance, disadvantages that are Satisfied will become Denied, and thus Denied will be propagated. Had the satisfaction value not been inverted then only a positive value would be propagated.

5.3.4 Model Evaluation

The evaluation of a SIF model instance seeks to determine how well the given problem specification is satisfied by the presented solution. That is: *How do the traits specified in a solution’s properties affect the requirements of the problem presented?* With SIF being built upon NovoGRL, the existing evaluation semantics can be leveraged. To evaluate a SIF model instance it is first converted into the NovoGRL model representation, and the resulting model evaluated.

There are several known algorithms for evaluating GRL models—see Chapter 3 §3.2.2. For the structure of the GRL models that represent SIF models a forward evaluation algorithm is required to determine if the goal nodes, representing the problem and requirements, can be satisfied by the task nodes representing the traits. The other known algorithms do not propagate the values in the correct direction. Further, no initial evaluation strategy is required as the model is constructed with all leaf nodes initialised to a default value.

5.4 The Sif Evaluator

This section presents the implementation of the model evaluator for SIF. Detailing specifically its architecture, feature set, and use of dependent types.

5.4.1 Architecture Overview

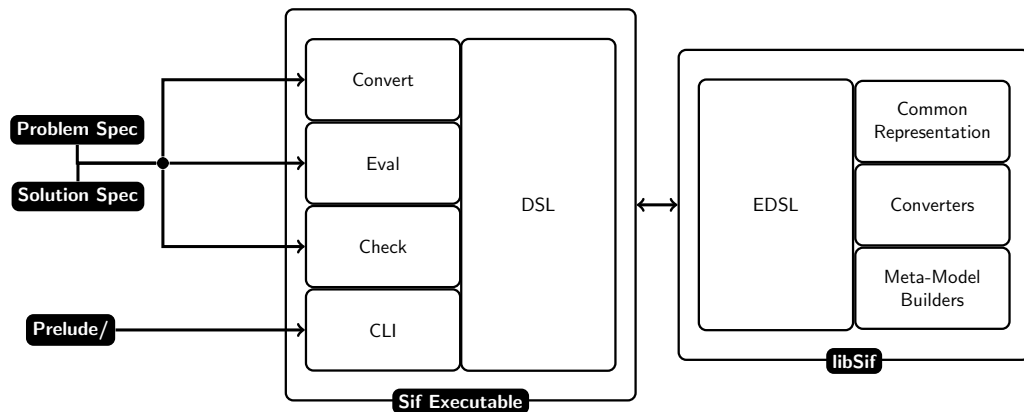


Figure 5.4: Architecture of the SIF Evaluator.

Figure 5.4 illustrates the core architecture for the SIF evaluator. This evaluator presents the reference implementation of the SIF language and has been design for the prototyping of SIF models and the construction of design pattern document stubs. The evaluator has been implemented in Idris and uses dependent types to provide correct-by-construction guarantees (see Chapter 4) and succinct language implementation. Moreover, Idris’ support for *Algebraic Effects* allows for total control over the construction and use of effectful operations [Bra15b]. For example, file IO, state, performance metric gathering, and logging.

The evaluator itself is composed of a core library `libSif` that presents the reference implementation of the language; and an interactive front-end (the SIF executable) that allows for model instances to be constructed using a DSL and for other interactions with SIF models.

Within `libSif` the ‘core’ language is implemented as a EDSL using dependent types in the style of the *Well-Typed Interpreter*—see Chapter 4. With the resulting implementation of the EDSL and DSL mirroring the abstract syntax presented in Figure 5.1. This approach not only provides correct-by-construction guarantees over language construction but also provides further correctness guarantees to be between

SIF and NovoGRL. The language has been implemented agnostic to the meta-model it is interpreted to, and allows for different meta-models to be used. This is explained more in §5.4.3 & §5.4.6. The language implementation also presents a common interface for working with SIF model instances regardless of underlying meta-model. Allowing, for example, evaluation functionality to be kept consistent across meta-model implementations and for consistent serialisation of SIF model instances into different publication formats—§5.4.5.

The ‘as-a-library’ construction style of the evaluator also allows the use of the language both as tool to be used by modellers, and as-a-library for the inclusion of programmatic based modelling of design patterns. This was used in the implementation of the FRIGG utility in Chapter 7. For this thesis, only the former aspect was explored in more depth. In fact the evaluator was constructed using SIF-as-a-library.

5.4.2 The Sif DSL & Executable

The SIF executable is a command line program used to check, and allow interaction with, SIF specification files. Figure 5.5 presents the grammar for the SIF DSL using eBNF syntax [ISO96]. The grammar is presented sans declarations for the Haskell style code comments supported by the language. During model evaluation the grammar helps construct an Abstract Syntax Tree that is then used to build SIF model instances. Chapter 9 presents example model instances specified using the SIF DSL. The SIF DSL allows for problem and solutions to be defined in separate files. Problems are specified together with the contexts in which the problem exist, together with the problem’s requirements. Corresponding to the `Problem` expression in the language. Each language construct in the problem file is associated with an identifier used in the solution file. Solution files contain all the information required to construct the `Solution` expression, and provide an abstract for the resulting pattern. The solution file uses the identifiers specified in the problem file to refer to specific requirements, contexts, and the problem itself. The interpreter for the DSL uses standard compiler implementation techniques to link identifiers with expressions, detect uninitialised identifiers, and to construct an environment that stores identifiers and their expressions. As specification files are external to the evaluator, an externally defined *Prelude* is presented—see Appendix A. This prelude contains lists of problems and solutions with specific pairings specified using a YAML² configuration file. When an interactive session using the SIF

²<http://yaml.org/>

```

<file> = 'sif', (<pFile> | <sFile>)

<pFile> = 'problem', <probDecl>, <context>*, <requirement>+
<probDecl> = <doc>?, <id>, '<->', 'Problem', <quotedStr>
<context> = <doc>?, <id>, '<->', 'Context', <quotedStr>
<requirement> = <doc>?, <id>, '<->', <reqType>, <quotedStr>
<reqType> = 'Functional' | 'Usability' | 'Reliability'
           | 'Performance' | 'Supportability'

<sFile> = 'solution', <pattDesc>, <solution>
<pattDesc> = 'Description', (<literalStr> | <quotedStr>)
<solution> = <doc>?, 'Solution', <id>,
           'solves', <id>, 'in', <id>, '{', <prop>+, '}'
<prop> = <doc>?, 'Property', <quotedStr>, '{', <trait>+, '}'
<trait> = <doc>?, <traitType>, <quotedStr>, 'is',
        <sValue>, '{', <affect>+, '}'
<traitType> = 'Advantage' | 'Disadvantage' | 'General'
<affect> = 'Affects', '{', <affectBody>, ('', <affectBody>)*, '}'
<affectBody> = <cValue>, <id>, ('by', <quotedStr>)?
<cValue> = 'Makes' | 'Helps' | 'SomePos'
           | 'Unknown'
           | 'SomeNeg' | 'Hurts' | 'Breaks'
<sValue> = 'Satisfied' | 'WeakSatis'
           | 'Unknown'
           | 'WeakDen' | 'Denied'

<doc> = ('>', <anyText>)+
<quotedStr> = '"', <anyText>, '"'
<literalStr> = '""', <anyText>, '""'
<id> = [A-Za-z]+
<anyText> = Any text

```

Figure 5.5: eBNF grammar for the SIF Domain Specific Language

evaluator is started, the pairings presented in the YAML file are automatically loaded into the evaluator for ‘interaction’. Further, the evaluator executable is modal in its operation, with four modes of operations defined:

- **Evaluation:** Evaluation of problem-solution pairings and reporting of the satisfaction values for each node in the resulting GRL model instance.
- **Syntax Checking:** Syntax checking (i.e. *linting*) externally defined files.
- **Convert:** A pattern document stub generator. Generation of pattern document stubs from SIF models, output formats include several well-known mark-up formats and the FREYJA encoding. The generator uses the models own documentation to populate fields.
- **Command-Line Interface:** A command line interface to for interacting with the distributed set of problem solution pairings, and accessing the modes of operation. The CLI facilitates: viewing and evaluation of existing patterns; and evaluation of externally defined problem-solution file pairings.

5.4.3 Abstract Patterns

A novel aspect of the evaluator is the implementation of the EDSL itself, and representation of the language expressions. The general construction of the expression language is derived from the *Well-Typed Interpreter*—see Chapter 4—and is bolstered using the *Types as (Meta) Modellers* approach detailed in Chapter 11.

However, with this style of construction an interpreter must be constructed for each meta-model that the domain language is to be paired with. Recent work has investigated the use of dependent types to aid in facilitating cross-language compilation of functional languages [Bra15a]. Of interest in the work of Brady [Bra15a] is the construction and support for interfacing with foreign libraries and support for IO operations. Here dependent types are used to present generic IO operations with types parameterised by the foreign function interface for the target language. Dependent types allow programmers to write generic code that can be tailored for the correct backend during compilation by the compiler itself. The techniques presented in Brady [Bra15a] are used in the construction of the evaluator to provide abstract pattern representations for each language expression. This is the ABSTRACT FACTORY pattern in action.

Listing 5.1 presents the `SifExpr` data type that represents SIF language expressions.

```

1 data SifTy = TyAFFECTS | TyTRAIT | TyPROPERTY | TyREQ
2           | TySOLUTION | TyPROBLEM | TyPATTERN
3
4 data SifDomain = MkDomain String (Maybe String)
5
6 data SifExpr : (ty    : SifTy)
7               -> (d    : SifDomain)
8               -> (impl : SifTy -> SifDomain -> Type)
9               -> Type where
10      MkExpr : SifRepAPI impl => impl ty d -> SifExpr ty d impl

```

Listing 5.1: The core data types used to provide an abstract representation of SIF expressions.

This data type is parameterised by three data structures central to the language implementation, and is a wrapper around a concrete expression instance that is generically represented. The first is an enumerated type `SifTy` representing the types in the SIF language. This follows the use of enumerated types to provide a ‘meta-type-system’ in the *Well-Typed Interpreter*. Second, is a simple data type `SifDomain` representing the domain of operation that indexes expressions within SIF. A domain is represented as having a title and possible description. The final parameter is an abstract description of the concrete expression representation allowing for different representations (and thus meta-models) to be explored for representing SIF model instances. The implementation of the representation must also be parameterised using `SifTy` and `SifDomain`. These type-level parameters are presented to language expressions in the constructor.

Listing 5.2 presents the `SifRepAPI` interface, this interface ensures that operations on language expressions are applicable to all representations, The `SifRepAPI` interface declares and constrains the permissible operations on language representations to a fixed set. These operations allow for inspection of language expressions and their contents, and also the execution of the evaluation operation itself. This interface also constrains the permissible representations of language expressions to be of type `impl`, the same parameter declared in the type. `SifExpr` is also used to declare an implementation that respects this interface. Further, the values in the type of the parameter in the constructor are also used to populate the type of the resulting `SifExpr` type. These data types allow for highly generic language expressions to be constructed. However, to do so, factories need to be defined that allow for implementation agnostic expressions to be generated such that only expressions parameterised by the same values can be used together.

Note. Language expressions in the abstract syntax for SIF (§5.3.1) are parameterised by their domain of operations. Here language expressions are also parameterised by their domain of operation. This provides compile time and runtime guarantees that only language expressions within the same domain can be combined.

```

1  interface SifRepAPI (impl : SifTy -> SifDomain -> Type) where
2    getTitle   : impl ty d -> {auto prf : HasMData ty} -> String
3    getDesc    : impl ty d -> Maybe String
4    getRTy     : impl TyREQ      d -> RTy
5    getTTy     : impl TyTRAIT    d -> TTy
6    getSValue  : impl TyTRAIT    d -> SValue
7    getCValue  : impl TyAFFECTS  d -> CValue
8
9    getProblem : impl TyPATTERN d -> impl TyPROBLEM d
10   getSolution : impl TyPATTERN d -> impl TySOLUTION d
11
12   getReqs      : impl TyPROBLEM d -> List (impl TyREQ      d)
13   getProperties : impl TySOLUTION d -> List (impl TyPROPERTY d)
14   getTraits    : impl TyPROPERTY d -> List (impl TyTRAIT    d)
15   getAffects   : impl TyTRAIT    d -> List (impl TyAFFECTS  d)
16   getReq       : impl TyAFFECTS  d -> impl TyREQ d
17
18   evalPattern  : impl TyPATTERN d -> Sif.EvalResult
19   fetchMetaModel : impl TyPATTERN d -> MetaModel
20
21   getDomain : impl ty d -> SifDomain
22   getDomain {d} _ = d

```

Listing 5.2: The SifRepAPI interface that defines operations common to all model representations.

5.4.4 Abstract Pattern Factories

Within the implementation of the SIF evaluator, a dependent record `SifBuilder` captures a set of factories for building valid SIF expressions that are all tied to the same implementation and domain. `SifBuilder` is indexed by the underlying language representation used. These factories are functions stored within the record³. The use of these factories are analogous to the use of constructors in constructing data type instances. Dependent types are used to *thread* instances of `SifExpr` in the type

³In a functional language functions are first class.

```

1 record SifBuilder (impl : SifTy -> SifDomain -> Type) where
2 ...
3   buildReq : (d : SifDomain)
4             -> RTy
5             -> String
6             -> Maybe String
7             -> SifExpr TyREQ d impl
8
9   buildProblem : (d : SifDomain)
10                -> String
11                -> Maybe String
12                -> List (SifExpr TyREQ d impl)
13                -> SifExpr TyPROBLEM d impl
14
15  buildSolution : (d : SifDomain)
16                 -> String
17                 -> Maybe String
18                 -> List (SifExpr TyPROPERTY d impl)
19                 -> SifExpr TySOLUTION d impl
20
21  buildPattern : (d : SifDomain)
22                -> String
23                -> Maybe String
24                -> SifExpr TyPROBLEM d impl
25                -> SifExpr TySOLUTION d impl
26                -> SifExpr TyPATTERN d impl
27 ...

```

Listing 5.3: Partial Declaration of the SifBuilder Record.

signature to ensure that the expressions are in the same domain, and also that the typing rules specified in §5.3.2 are adhered to. Listing 5.3 presents a partial declaration (for illustrative purposes) of the `SifBuilder` record. When presenting a new language implementation, only an instance of `SifBuilder` needs to be defined for that language. The rest of the evaluator implementation works with `SifExpr`.

Further, the expression factories in a `SifBuilder` instance are not used directly. An API has been defined to provide more usable type synonyms, and a set of functions to abstract over accessing and calling the functions stored in a `SifBuilder` instance. For example, the actual expression factory presented for constructing patterns is defined in Listing 5.4, together with the expanded type-synonyms.

As with the use of dependent types in implementing typing rules, here the values in the function's type signature are used to ensure correct use. When building a pattern, the presented problem and solution must both be parameterised over the same domain,

```

1 mkPattern : SifBuilder impl
2           -> (d : SifDomain)
3           -> String
4           -> Maybe String
5           -> PROBLEM impl d — SifExpr TyPROBLEM d impl
6           -> SOLUTION impl d — SifExpr TySOLUTION d impl
7           -> PATTERN impl d — SifExpr TyPATTERN d impl
8 mkPattern impl d t desc p s = (buildPattern impl) d t desc p s

```

Listing 5.4: An example expression factory for pairing a problem and solution in SIF.

and have been constructed using the same underlying representation. This can be seen in Listing 5.4, in which the *values* `d` and `impl` are propagated through out the type signature. When this function is called the parameter’s types must all have the same value. These values have been ‘threaded’ through the signature. This *threading* of values is a common pattern when using dependent types⁴.

5.4.5 Pattern Document Stub Generation

Recall that a mode of operation for SIF is that of document stub generation. The ability to convert documents to other well-known formats comes from a document conversion library (inspired by a similar well-known tool called PANDOC⁵) developed as part of the thesis. This tool is available online [dMH15c]. Documentation in SIF is a first-class language construct and accessible for use by the EDSL implementation. It is this documentation that provides the text stubs used to fill in sections of the generated document.

An interesting use of dependent types is the construction of a generic conversion function to convert pattern documents to a supported output format. Listing 5.5 details the dependent function. Here a *well-typed* interpreter has been constructed to allow for a single function to convert the model to multiple named formats, returning a data type specific for the output selected.

5.4.6 Meta-Model Representation

The final novel aspect of the evaluator is how the SIF language’s meta-model is represented and linked to the language constructs. SIF language constructs are ‘parameterised’

⁴See Chapter II §II.5 on *Type Threading* for more information

⁵<http://www.pandoc.org>

```

1 data SifOutFormat = ORG | LATEX | CMARK | XML | STRING | FREYJA
2
3 convTy : SifOutFormat -> Type
4 convTy LATEX = String
5 convTy CMARK = String
6 convTy ORG   = String
7 convTy XML   = XMLDoc
8 convTy STRING = String
9 convTy FREYJA = XMLDoc
10
11 convTo : (fmt : SifOutFormat)
12         -> PATTERN impl d
13         -> (convTy fmt)
14 convTo FREYJA p = Freyja.toXML (toFreyja p)
15 convTo XML     p = toXML p
16 convTo EDDA   p = toEdda p
17 convTo ORG    p = org (toEdda p)
18 convTo LATEX  p = latex (toEdda p)
19 convTo CMARK  p = markdown (toEdda p)
20 convTo STRING p = toString p

```

Listing 5.5: A *well-typed* interpreter for converting patterns in SIF to supported formats.

by the language’s meta-model. This style of construction allows for different meta-models to be constructed and used to construct SIF model instances. This construct was detailed in §5.4.4. During the construction of the evaluator two `SifBuilder` factories were constructed to represent two different construction styles for the interpretation semantics detailed in §5.3.3.

The first style was a classical *direct interpretation* from a SIF expression to the corresponding expression from NovoGRL. This translation occurred during model evaluation, and was presented to the evaluator *just-in-time* for use. Although this approach works, during the construction of the interpreter, there is a disconnect between representing the domain language expressions and their interpretation. The validity of the underlying meta-model (NovoGRL model instance) must be manually checked for correctness post SIF model construction. The *time of creation* of the SIF model thus differs from the *time of check* of the correctness of the underlying meta-model. This leaves scope for possibly incorrect NovoGRL models to be constructed from SIF models, and for this incorrectness to be discovered post creation and during use.

The second style of implementation used the *types as (meta) modellers* approach detailed in Chapter 4 §4.7. Use of this approach allows for better construction guarantees than the direct interpretation. Unlike the direct interpretation valid SIF constructs can

only be constructed if valid constructs in the meta-model are also constructed. Here the *time of creation* of the SIF model is the same as the *time of check* of the underlying meta-model. More information of this style of construction, and the differences between these two techniques, is discussed further in Chapter 11.

5.5 Case Studies

This chapter has introduced the SIF modelling language, its design, and construction. To investigate the use of SIF for modelling design patterns, the SIF was used to model several design patterns both existing and new. Table 5.3 summarizes the patterns considered/created during evaluation. This section details several, but not all, of these patterns. In particular, this section discusses notable aspects of their modelling, construction, and evaluation. The resulting SIF model files are presented as part of the SIF prelude, distributed along side the evaluator's implementation [dMH15g], and are also provided in Appendix A.

5.5.1 Modelling the Policy Enforcement Points Pattern

Policy Enforcement Points (PEP) are fixed points in a software system that governs access to objects by a subject. When a subject requests access to the object, the enforcement point checks the access policies detailing the rights of the subject, and how the object must be accessed. Zhou et al. [ZZP02] describes such a pattern for PEPs.

Modelling this pattern in SIF resulted in a problem definition for *Policy Enforcement*, and a solution explicitly for Policy Enforcement Points. Interestingly, during the construction of the problem specification it was noted that Zhou et al. [ZZP02] do not explicitly state in their list of forces that the solution should explicitly enforce policies. Forces are supposed to detail aspects of the problem that needs addressing. For PEPs, *policy enforcement* should be a listed force.

When listing the consequences in using the solution several more unlisted forces that are problem specific were also listed. These were: *Universality*—application to a variety of deployment scenarios; and *Understandability*—ability of the developer to implement the solution. This forces should instead be listed as forces that drive the selection of the solution. This further highlights problems in creating patterns good patterns.

Problem	Solution	Context
Abstract Data Types ¹	Simple Factory	Default
	OO Factories	Default
	Abstract OO Factories	Default
Information Secrecy ²	Symmetric Crypto	Default
	Asymmetric Crypto	Default
	Hybrid Crypto	Default
Authentication ³	Shibboleths	Socio
	ID Cards	SocioTech
	Digital Certificates	Tech
Access Control ⁴	Generic Door Lock	Physical
	Keyed Door Lock	Physical
	Coded Door Lock	Physical
Token-Based Auth. ³	ID-Cards	SocioTech
Policy Enforcement ⁵	Policy Enforcement Points	Default

¹ Taken from [Gam+94; Fre+04]. ² New/Inspired by [BRD98]. ³ New patterns for the thesis. ⁴ Taken/adapted from [WF11]. ⁵ Taken from [ZZP02].

Table 5.3: Summary of the patterns modelled using SIF during evaluation.

5.5.2 Modelling Factory Patterns

Gamma et al. [Gam+94] is a seminal book that presented several patterns for engineering systems in OO languages. One of the patterns presented was the **FACTORY PATTERN**, detailing how data can be represented and created separately from its use. Freeman et al. [Fre+04] also detail the pattern together with several variants. The complete list of considered patterns being **SIMPLE FACTORY**, **OO FACTORIES**, and **ABSTRACT OO FACTORIES**. The second study investigated modelling the factory patterns from Freeman et al. [Fre+04] and Gamma et al. [Gam+94] to investigate how different solutions can address the same problem.

Problem

The modelling of these patterns first required identification of the problem common to all factory patterns. This is the problem of *Abstract Data Types*. That is, how to work with data structures without knowledge of their concrete representation. Table 5.5

presents a set of six requirements for this problem, that were identified through analysis of existing literature.

	Type [†]	Requirement	Description
1	F	Language Agnostic	The solution should limit the use of language specific features.
2	S	Minimal Dependencies	The resulting code should have minimal dependencies.
3	F	Flexible Replacement	There should be an ability to replace the method of creation easily.
4	F	Closed for Modification	The creation and use of data instances should be closed for modification.
5	F	Agnostic Generation of Objects	The solution must allow for the generation of data structure instances without implementation of data structures.
6	F	Separate Data Structures	The solution must provide a means to separate representation and knowledge of the underlying implementation.

[†] For brevity the requirement types are listed by first letter only.

Table 5.5: Requirements for the problem of *Abstract Data Types*.

Context

For these patterns it was decided not to model the contexts in which the problem and solutions exist and the default context was chosen.

Solutions

The descriptions of the chosen patterns from Freeman et al. [Fre+04] were used to model the solutions presented below. Table 5.7 presents the results of their evaluation.

Simple Factory The first solution presented is that for the *Simple Factory* pattern that requires programmers to program against interfaces. These interfaces describe the permissible operations on an abstract data structure. During development, programmers implement concrete implementations of the data structures that are interacted

	Simple Factory	OO Factory	Abstract OO Factory
1	None	Denied	Denied
2	wSatisfied	Satisfied	wDenied
3	wDenied	wSatisfied	wSatisfied
4	Satisfied	Satisfied	Satisfied
5	Satisfied	wSatisfied	wSatisfied
6	Satisfied	Satisfied	wSatisfied
Overall	wDenied	Denied	Denied

Table 5.7: Evaluation results for the factory patterns modelled in SIF.

with solely through the interface. Restriction of the interactions with the data structure through the interface allows for different concrete implementations to be considered without affecting code that uses the underlying implementation. The notions of interfaces, concrete products, factories, and clients were used as the core properties of the solution. A fifth property was presented to represent the idea that not all languages support interface programming. It is this last property and the idea that for each data structure an explicit factory must be created that causes the solution to become weakly denied.

OO Factories The second solution represents the *Factory Method* pattern. This pattern extends the simple factory pattern conceptually and provides an interface for creating data but lets the subclasses decide which class to instantiate. A factory method allows a class defer instantiation to subclasses. As with the previous pattern similar core notions were used to define the properties. However, with OO factories a more diverse range were created to reflect the entire pattern. This pattern evaluates to denied due to the use of explicit OO concepts that are a core part of the solution.

Abstract OO Factories The final solution presented is a refinement of the *Factory Method* pattern. Providing an interface for creating *families* of related or dependent data structures without specifying their concrete implementations. Table 5.7 shows that a similar result is seen during evaluation as with the *Factory Method* pattern. However, there are noticeable differences. For example, as this solution describes the creation of a family of products, this will have a negative affect on minimal dependencies in the resulting solution.

5.5.3 Modelling Information Secrecy

This case study investigated modelling solutions to information secrecy using cryptographic solutions. This was inspired from initial readings of the INFORMATION SECRECY pattern, first detailed in Braga et al. [BRD98]. This reading viewed a set of patterns that were *just wrong*. For a demonstrable evaluation showing the *badness* of the Braga et al. patterns see Chapter 8 §8.6 in which the report cards for the patterns are detailed. The goals of the study were to investigate how to model:: (a) new patterns based upon existing patterns; and (b) a set of patterns that solve the same problem.

Problem

	Type	Requirement	Description
1	R	Secure Implementation	The implementation should be implemented securely.
2	U	Minimal Workflow Disruption	The mechanism should not impact on normal operations.
3	U	Comprehensible by Non-Experts	Mechanism should be usable by non-experts.
4	P	Suitable performance	Protecting data should not take forever.
5	P	Suitable Security Level	Different levels of security should be permitted.
6	F	Recipient Confidentiality	The data should be viewable only by the intended recipient
7	F	Data Confidentiality	Core principle that ensures that the data is kept confidential.

¹ For brevity the requirement types are listed by first letter only.

Table 5.9: Requirements for the problem of Information Secrecy.

One of the noted problems with the treatment of information secrecy by Braga et al. [BRD98] is that the problem description describes problems associated with the use of cryptography and not the problem of information secrecy itself. Before any cryptographic solutions can be modelled, a set of solution agnostic requirements must be designed. Table 5.9 presents these new requirements.

Solutions

With the set of requirements defined, a set of cryptographic solutions can be modelled. A second problem with the INFORMATION SECRECY as presented by Braga et al. [BRD98] is that the design presents an agnostic cryptographic solution that does not respect the known different types of ciphers available: *Symmetric & Asymmetric*. Nor their known combination as a part of a *Hybrid* scheme. To respect these usage patterns, three solutions were modelled:

- **Symmetric Cryptography:** use of a shared key to encrypt and decrypt data using the same key.
- **Asymmetric Cryptography:** use of key pairs for distinct encrypting and decryption of data.
- **Hybrid:** use of symmetric cipher to encrypt payloads, and use of asymmetric cipher to encrypt the symmetric key used.

For all the solutions modelled, they were described according to properties that detail: (a) the mathematical constructs that support the solution; and (b) the operational characteristics of the solution. Table 5.11 presents the results of model evaluation.

Symmetric Cryptography Symmetric ciphers have been developed to provide fast operations over large data. To achieve such fast operations the mathematics involved requires that the operations are symmetric in nature and that the same key is used. Further, different ciphers have different modes of operation that affects the behaviour of the cipher used. Ultimately, although the mathematics behind symmetric cryptography is sound incorrect usage can cause problems.

Asymmetric Cryptography Asymmetric cryptographic schemes define a set of three functions that when used together allow for data to be kept confidential without the need for a shared key. A key theme in asymmetric crypto is the idea of separate encryption and decryption key pairs. Data is encrypted under an encryption key such that only the decryption key that is paired with the original encryption key can decrypt the data. Although, asymmetric ciphers provide better functional operation, they are known to be slow when dealing with large data. Further, with this solution incorrect parameter choices will affect the security of the data.

Hybrid Cryptography Hybrid schemes present a combined use of asymmetric and symmetric encryption schemes to provide an efficient solution to information security. Symmetric schemes are used to provide efficient encryption of data, regardless of data size. The symmetric key is itself encrypted using an asymmetric scheme to provide the functionality associated with asymmetric schemes. The efficiency of asymmetric schemes are achieved as symmetric keys are small. This solution to information secrecy was modelled by combining, adapting, and extending the model descriptions from the previous two solutions.

Model Evaluation

Req.	Sym Crypto.	Asym Crypto.	Hybrid Crypto.
1	wSatisfied	wSatisfied	wSatisfied
2	Satisfied	wDenied	wSatisfied
3	wSatisfied	wDenied	wSatisfied
4	wSatisfied	wDenied	wSatisfied
5	wSatisfied	wSatisfied	wSatisfied
6	Denied	Satisfied	wSatisfied
7	Denied	wSatisfied	wSatisfied
Overall	Denied	wDenied	wSatisfied

Table 5.11: Evaluation results for the Information Secrecy patterns modelled in SIF.

Table 5.11 presents the results of running the evaluator against the three different solutions. The results are interesting for a variety of reasons. Notice the mixture of satisfaction values between the symmetric, asymmetric and hybrid solutions. This is indicative of the strengths and weaknesses of both solutions in addressing the problem.

Even more interesting are the values obtained from the hybrid solution. Indicating how when combined, the two cryptographic solutions satisfy the problem of information secrecy better than they do individually. This aspect also highlights a limitation of SIF in that it does not model pattern languages and prohibits for multiple solutions to be combined to address a single problem. This aspect is further indicated in the actual satisfaction values themselves for the requirements. For example, none of the solutions out-rightly address the problem of information secrecy. Specifically, the requirements used to denote data confidentiality has been evaluated to Denied

for symmetric solution. Although, the models present technical solutions, the problem they are addressing cannot be solved using the technical descriptions alone. For cryptography to be used successfully, a whole host of problems and solutions must be addressed first. This would require a pattern language to be modelled. How this is achieved is left for future work.

5.5.4 Modelling Authentication

The tutorial from Chapter 9 demonstrates how SIF can model various authentication patterns, detailing the construction of AUTHENTICATION THROUGH SHIBBOLETHS and AUTHENTICATION THROUGH ID CARDS. The next, and final, set of case studies discussed in this section details the remaining patterns constructed. Brown and Fernandez [BF99] introduces the AUTHENTICATOR pattern, a technical solution to the problem of authentication with a remote service. Erber et al. [ESP07], Fernández and Sinibaldi [FS03], Fernández and Warriar [FW03], Fernández [Fero7], Ajaj and Fernández [AF10], Morrison and Fernández [MFO6] and Weiss [Weio6] presented similar authentication related patterns. Looking at authentication, this case study specifically investigated both the modelling of problems, and modelling of problems and solutions for different contexts.

Problems & Contexts of Authentication

Chapter 9 §9.2 discusses the problem of authentication and the contexts in which it operates. Authentication is a problem that occurs across many different contexts ranging from a purely social context, to a purely technical one. Although not supported by the formal description of the language, the evaluator itself supports the specification of problems and their contexts of operations. How, different problems and their contexts are formally modelled in SIF are left for future work and is discussed further in §5.6.

One difficulty found in modelling the problem of authentication (see Chapter 9 §9.2) is that the resulting problem is too generic and does not take into account other ‘problems of authentication’. For example, authentication problems can be categorised further by the ‘proof’ used to attest authenticity. An authentication mechanism will either be: (a) *Something you have*—a physical token, for example an ID Card; (b) *Something you know*—a passphrase, for example a Shibboleth; or (c) *Something you are*—a biometric marker, for example gait or finger print. These additional traits will affect

the set of requirements used to describe the problem. For problems to be accurately modelled for these authentication mechanism the problem detailed needs to be extended in at least three ways: One for each of the types of ‘proof’. Within SIF this requires a complete and separate problem file to be specified for each of the three types of authentication: *Token Authentication*; *Passphrase*; and *Biometric Authentication*. Modelling problems in SIF that are extensions of existing problems is too verbose.

Solutions to Authentication

The problems of efficient modelling files was also found when modelling solutions to authentication. Three solutions were modelled, two of which were discussed in Chapter 9. The third and final was the modelling of digital certificates for purely technical authentication.

Further, a solution to *Token Authentication* was explored. For the pattern of `TOKEN AUTHENTICATION THROUGH ID CARDS` it was interesting to note the similarities between the solution file created for the pattern: `AUTHENTICATION THROUGH ID CARDS`. The resulting files only differ w.r.t. to the difference in references found between the problem specifications representing authentication and token authentication. This result further highlights the verbosity in modelling with SIF.

5.6 Discussion

This section presents a discussion over the language and its limitations.

5.6.1 Accuracy & Efficacy

§5.5 presented and discussed the results of using SIF to model several design patterns. By virtue of being able to model both existing (`ABSTRACT FACTORY`) and new (`AUTHENTICATION THROUGH SHIBBOLETHS`) design patterns, the ability of SIF to model design patterns is self evident. However a question remains over the accuracy and efficacy of the SIF language.

When compared with the meta-modelling language `NOVOGRL`, the scope of modelling with SIF possesses inherent limitations. For one, problem specifications are limited to listing sets of requirements only, how requirements interact with each other cannot be modelled. Modelling the effect that a requirement has on other requirements

would allow for more precision w.r.t. to the modelling of the problem itself. NovoGRL allows for contributions and correlations to be modelled between requirements. Incorporating this into the SIF language would greatly improve the language's accuracy.

Another aspect for consideration is the modelling of properties. These properties have a single type: `Property`, however, not all properties of a solution are the same. A property might refer to an abstract concept, or pure software solution, or even an administrative aspect. Identification of the different types of property will be useful to allow for greater accuracy in pattern representation. The type of a solution trait had an affect on the construction of the NovoGRL model instance. How different types of properties may affect the traits contained within them and to the underlying NovoGRL model instance could also affect accuracy of modelling. Further, SIF cannot model the effect that properties within a solution have on each other. Modelling these relationships would allow for a more accurate model to be produced.

NovoGRL can model satisfaction values and contribution values either qualitatively or quantitatively. Qualitative values being represented by a fixed set of enumerated values, and quantitative values bounded to the interval $(0, 100)$. SIF has been designed to use the qualitative definitions of these values, presenting modellers with more descriptive values. However, a side affect of using these values is that the values are *too* discrete. They do not offer as fine-a-grained capture of the contribution that traits have on requirements, nor the final satisfaction value. This was seen during the evaluation when modelling the `AUTHENTICATION THROUGH SHIBBOLETHS` pattern. When the satisfaction value of the trait *Act of C-R* was changed from `Satisfied` to `wSatisfied` the satisfaction value for several of the requirements were altered to `Unknown`.

A final accuracy aspect to consider is the role of 'disadvantages'. Several GOMLs have extensions designed to model security-related concepts [MG07; MMZ07]. The NovoGRL has no such extensions. Although, resarch has considered the goal-oriented modelling of security requirements [Gio+05; MMZ10], the focus of this thesis was to improve upon existing work in modelling design patterns using the GRL. Here disadvantages are a means to use existing GRL concepts to model negative traits within a design pattern. Traits are generic modelling constructs. SIF is a general purpose modelling language for design patterns. Being able to reason precisely on security requirements was not considered in the initial design of SIF. Future considerations will be to investigate how to enrich the concepts (security related and others) in SIF to model design patterns more accurately.

5.6.2 Performance

The focus on this research was primarily on the correct construction of a modelling language. Producing a language with efficient evaluation was not a prime concern. Regardless, it was noted that during evaluation of the case studies detailed in §5.5, the performance remained constant regardless of whether the direct or indirect representation was used. However, the modelling files constructed were not large, and *could be* considered realistic in size and typical for what one would expect to see. Future work will be to investigate the performance of the evaluator when operating on larger models.

5.6.3 Alternative Meta-Modelling Language

The SIF language has been designed to use NOVOGRL as its meta-modelling language. Theoretically (and practically), SIF could be re-targeted for different meta-modelling languages. One could even bypass the need for using NOVOGRL as the meta-modelling language and directly interface with the evaluation mechanisms of the NOVOGRL. The use of the ABSTRACT FACTORY pattern allows for code that uses the SIF modelling language to be closed for modification w.r.t. to the language implementation. Different modelling language representations can be swapped out without affecting the rest of the evaluator implementation. Which meta-modelling language SIF should be re-targeted to is left open for further research. Other languages to consider are i^* [Yu97], and TROPOS [Bre+04].

5.6.4 Choice of Implementation Language

SIF has been implemented in Idris, a general purpose dependently typed functional language. From an engineering perspective, the choice of language is important and affects how a program is developed.

Why Idris? Although Idris is not the only dependently typed language in existence it is, however, the most practical and has been designed from conception to promote general purpose programming with dependent types. Idris is a language for ‘Real World Programming’. Another popular language, Agda, has better support for reasoning with dependent types but is not as proficient when providing support for constructing

interactive programs [Nor09]. Other dependently typed languages are not as feature complete nor mature for real-world programming.

Why Functional and not Object-Oriented? The rich and expressive type systems in dependently typed languages allows for more precise guarantees to be made about programs—cf. `Vect` and `List`. This was shown in Chapter 4 when introducing dependent types, and *Well-Typed (Abstract) Interpreters*. However, Idris is a functional language. The experimental evaluation presented in §5.5 noted that when modelling patterns, the resulting specifications exhibited characteristics that could be better modelled using an OO language such as JAVA. For example, modelling both generic authentication problems and specific problems, could be better explained with inheritance. Similarly, the combining of solutions to address a single problem could be realised using object composition. Unfortunately, provision of this in Idris, would require bespoke implementations of these constructs purely for the SIF language. In an OO language one gets inheritance (or object composition) for free, but at the loss of the rich and expressive type system provided by a language that supports dependent types. Choice of an OO language would mean loosening of the construction guarantees afforded by the *Well-Typed (Abstract) Interpreters* approach of EDSL construction. Lost too would be that ability to index language expressions by their domain and underlying representation. An interesting question would be: *Why can we not have both styles?* SCALA⁶ is a multi-paradigm OO language that targets the *Java Virtual Machine* and has limited support for dependent types. However, SCALA’s support for dependent types is known to be limited in comparison to other dependently typed languages. Future research considerations may be to investigate the provision of a general purpose dependently typed OO language.

5.6.5 Problems & Solutions and Multiple Domains

The type-system for SIF requires that the type associated with model constructs are index by their domain of operation. This restricts the specification of these expressions to be under a single domain. However, the relationship between constructs and a domain are not necessarily one-to-one. For instance, a problem will exist in multiple domains; the relationship is one-to-many. Take, for example, the modelling of patterns for authentication in Chapter 9. The problem of authentication can be found across

⁶<http://www.scala-lang.org>

several domains, and the two presented solutions each exist in domains that are distinct from each other. With solutions, it can be argued that a specific solution will only exist in a single domain. For example, Shibboleths are a societal-based solution and do not exist in the other presented domains. Fortunately, the formal description of problems in SIF can be extended to provide this check. For example:

$$\frac{\frac{ys:List\ \mathcal{G}}{p:\mathcal{P}(ys)} \quad \frac{y:\mathcal{G}}{s:\mathcal{S}(y)} \quad y \in ys \quad t:String \quad d:String}{(Pattern\ t\ d\ p\ s) : \Phi(\gamma)}}$$

Here, the pattern will be built if the domain of the solution is in the list of domains associated with the problem. However, a problem arises when looking to index requirements. It is not clear if the requirements should also be indexed by the same list of domains as the problem, or should it be allowed that requirements be indexed by a subset of the specified domains?

These changes will have an affect on the complexity of the implementation, and might result in loss of the functionality afforded by the abstract pattern implementation. The types become more complex. However, this is more of an engineering issue and the checks are nonetheless decidable. Although, the implementation of the EDSL follows the formalism exactly, the DSL does not. The implementation of the DSL (as witnessed in Chapter 9) provides for a problem to be associated with multiple domains. When constructing patterns from externally specified files, the specification's AST is inspected and only patterns can be constructed if the context specified by the solution is found in the list of contexts specified with the problem.

5.6.6 Hard vs. Soft Requirements

The design of SIF allows for different requirements to be given a type from the FURPS requirements model. Allowing for greater accuracy when modelling requirements when compared to the GRL. However, the GRL allows for requirements to be labelled either as: 'hard'—a requirement with well defined terms for satisfaction; or 'soft'—whether the conditions for satisfaction are not so well defined. This difference recognises that not all requirements are well defined. The current interpretation semantics (§5.3.3) for SIF translate the requirements into *hard* goals only. How SIF requirements map to *soft* goals was not considered. This affects the accuracy of the underlying goal model w.r.t. to the requirement 'hardness'.

An improvement to the SIF language would be to further parameterise the type for requirements \mathcal{R} with a type to describe whether the requirement is: soft; or hard. Take for example, the following amendment to SIF's abstract syntax and type system:

$$\begin{aligned} k \in \mathcal{K} &= \text{Hard} \mid \text{Soft} \\ r \in \mathcal{R}(\gamma, \text{Hard}) &= \text{HFunctional } t \ d \mid \dots \\ r \in \mathcal{R}(\gamma, \text{Soft}) &= \text{SFunctional } t \ d \mid \dots \\ \mathcal{T} &= \dots \mid \mathcal{K} \mid \mathcal{R} : \mathcal{G} \rightarrow \mathcal{K} \rightarrow \mathcal{T} \mid \dots \end{aligned}$$

The interpretation semantics can be extended to provide more accurate interpretation of requirements. For example, functional requirements would be interpreted as follows.

$$\begin{aligned} \llbracket \text{HFunctional } t \ d \rrbracket &= \text{Goal } t \ \text{Unknown} \\ \llbracket \text{SFunctional } t \ d \rrbracket &= \text{SGoal } t \ \text{Unknown} \end{aligned}$$

More interestingly with this amendment to SIF, the type-system has gained more expressiveness and precision. The extra information within the type for requirements can now be leveraged to reason more precisely about requirements in SIF. For example, the number of hard and soft requirements in the problem specification, or possibly how traits affect requirements. Further, if SIF were to be expanded to allow for decomposition of requirements, typing rules can be constructed to use this new information to reason more precisely on the decomposition. For instance, restrict decomposition of hard requirements into other hard requirements only.

The described reasoning is only between different hard and soft requirements. An even better improvement would be to further parameterise the type for requirements by the kind of requirement being represented. For example:

$$\begin{aligned} k \in \mathcal{K} &= \text{Hard} \mid \text{Soft} \\ f \in \mathcal{F} &= \text{TyFunc} \mid \text{TyUsab} \mid \text{TyReli} \mid \text{TyPerf} \mid \text{TySupp} \\ r \in \mathcal{R}(\gamma, k, f) &= \text{Requirement } t \ d \\ \mathcal{T} &= \dots \mid \mathcal{F} \mid \mathcal{K} \mid \mathcal{R} : \mathcal{G} \rightarrow \mathcal{K} \rightarrow \mathcal{F} \rightarrow \mathcal{T} \mid \dots \end{aligned}$$

However, although the suggested improvements appear straightforward their implementation is not. SIF abstracts away the underlying meta-model—§5.4.3 Listing 5.1 & Listing 5.2. The suggested improvements increases the engineering challenge somewhat. When the suggested improvements are viewed in conjunction with §5.6.5, the engineering challenge becomes even more challenging. These extensions to SIF are left for futurework.

5.6.7 Modelling Pattern Languages

An aspect of SIF that has not been considered so far is its inability to model pattern languages. SIF only models single patterns. During evaluation it was noted that there were results that would be better modelled as a pattern language rather than a single pattern. For instance, the pattern INFORMATION SECURITY USING HYBRID ENCRYPTION was more a combination of the patterns INFORMATION SECURITY USING SYMMETRIC CRYPTOGRAPHY and INFORMATION SECURITY USING ASYMMETRIC CRYPTOGRAPHY. This is not surprising given that in practice asymmetric encryption is not used on its own and is almost always used as part of a hybrid encryption scheme, or through an integrated encryption scheme. The repetition of properties and traits seen amongst the different solutions only enforces this.

An interesting extension to SIF would be to investigate the modelling of pattern languages fully. Such a language could either provide a deep or shallow modelling of patterns. A shallow modelling would only look to exploit and model the relationships between many problem solution pairings. A deep modelling would be to look inside the problems and solutions to explore the relationship between the different problem requirements and solution properties. Further, it would be useful to explore how the different types of patterns (presented in Chapter 2) would affect this modelling.

5.7 Summary

Chapter 2 §2.5.4 detailed how design patterns can be modelled using the GRL. SIF provides a DSML to allow patterns to be reasoned on using the GRL but with domain specific constructs. Future work will be to improve the accuracy of the language when modelling design patterns.

Use of dependent types allows for the succinct implementation of a verified interpreter for the language w.r.t. to the meta-modelling language in the style of the

Well-Typed Interpreter. Chapter 3 §3.7 noted the difficulty in constructing DSMLs from the GRL. To address, and explore this, two representational backends to the evaluator were constructed. The implementation of the SIF language constructs are parameterised over the underlying meta-model representation. When combined with use of the FACTORY METHOD pattern, and implementation techniques of Higher-Order Functions and Dependent Records, the evaluator allows for the creation of SIF model instances using different model representations. Resulting in a *Direct* and *Abstract* implementation of the interpreter from SIF to the GRL. The latter representation uses dependent types to model the interpretation between the two directly within the type of SIF's implementation. This approach provides further guarantees towards the correctness of the SIF language itself and when interpreting the model to the GRL, and is described in more detail in Chapter 11. Modellers using the evaluator can choose between a variety of modelling representations should they wish.

FREYJA: A PATTERN DOCUMENT DESCRIPTION SCHEMA

This chapter presents an active document schema for describing pattern documents, agnostic to their representation in code or for publication. Named FREYJA, the schema is presented using implementation agnostic tooling that facilitates their ability to be machine readable. The presented schema allows for a SIF model and evaluation scores to be embedded as metadata directly within the document itself. The resulting documents are active and contain more information than the written text. With this extra information tooling can be constructed to provide programmable interactions with the document itself. This chapter presents the schema describing its structure, provision as a software library, and discussion over its use.

6.1 Schema Definition

Representationally speaking a design pattern is *just* a structured document following a predefined set headings—the pattern template. Such document structures are naturally described and represented as XML schema. FREYJA is a bespoke pattern template and XML schema developed for this thesis. This section documents the schema definition, and details the structure of the pattern template. Not all details for the schema are given, a full listing is provided in Appendix A and available in the code repository [dMH15d].

The schema is presented using the Relax NG Compact notation [CCMo3].

Note. In the definitions that follow: `desc` is a simple element for providing descriptions; `score` presents an enumerated attribute denoting an evaluation result; and `id` is an attribute for storing identifiers.

6.1.1 Pattern Template

Here the root element of the document is presented and provides the outline of the pattern template used. Each pattern is described using an identifier `id` encoded as an attribute, followed by a list of sectional elements. Presented below is the definition for the root pattern element, this definition also highlights the order of the sectional elements.

```
pattern = element pattern {  
  attribute id { text }  
  , name, summary, metadata, context, problem  
  , solution, evidence, studies, relations?  
}
```

Each of these sectional elements are used to document different aspects of the pattern, and have been given descriptive names. As with existing pattern templates salient details over the problem, solution, and context are described. This template also requires explicit mention of evidence detailing the existence/usefulness of the solution against the problem, together with example case studies documenting solution application. The remainder of this section will detail these elements.

6.1.2 Problem Specifications

The first major sectional element describes the problem being solved. Problems are titled descriptions with a list of requirements describing elements of the problem that must be addressed by the solution. Traditionally, pattern templates also present a list of *forces*. Here problem's are instead described using a requirements-oriented approach. The problem element has the following definition:

```
problem = element problem {  
  score, name, desc, requirements  
}
```

The `requirements` element collects the list of requirements associated with the problem. The FURPS requirements model has been used to model the different permissible types of requirements. The definition for the `requirements` element and each of the requirement types are presented below.

```
requirements = element requirements {
  ( element functional      { id, score, name, desc }
  | element usability       { id, score, name, desc }
  | element reliability     { id, score, name, desc }
  | element performance    { id, score, name, desc }
  | element supportability { id, score, name, desc }
  )+
}
```

Each requirement type supports the same structure, an identifier `id` (presented as an attribute), a name and a description. A secondary attribute `score` documents how well the requirement is graded w.r.t. to suitability. This is linked to the PREMES evaluation framework presented in Chapter 8.

6.1.3 Solution Descriptions

Classic pattern templates present solutions rather loosely using natural language, and with emphasis given towards the solution's structure and dynamics. Often these textual descriptions are presented with accompanying UML models. For the SIF modelling language, an alternative approach to solution modelling was presented—see Chapter 5. Solutions are described as solutions to requirements problems, and with this approach solutions are presented as a set of properties that detail aspects of the solution and how said aspects affect the requirements of the problem. For the document schema presented these notions are incorporated with existing solution descriptions.

```
solution = element solution {
  name, desc, models, properties
}
```

The remainder of this section details the sub elements detailing models and properties, and traits and affects.

Models

Solutions to problems have a structure, comprised of possible components, and a set of dynamics describing the interaction between components within the structure. The `models` element captures the different structures and dynamics associated with the solution, and has the following definition:

```
models = element models {  
  ( element structure { score, name, desc, model }  
  | element dynamic   { score, name, desc, model }  
  )+  
}
```

Models are either structural or dynamic, and are given a name, description, and a model element that allows for a non-XML based description of the model to be given. The form that `model` takes is defined agnostic to intended language used. Such agnosticism allows for different modelling languages to be represented that are not necessarily bespoke to software architecture as UML is geared towards. For example, the contents of `model` could also contain a *Business Process Modelling* (BPM) language model that details a solution for more socio-technical oriented processes [KLL09].

Properties

Presented alongside the models are the solutions properties. Properties describe aspects of the solution that affect the problem, describing how the problem is addressed and in what manner. As with the definition in SIF, properties in FREYJA are given a title, description and many traits. The definition is:

```
properties = element properties {  
  element property { score, name, desc, traits }+  
}
```

Traits

The definition for traits follow their SIF definition as well, describing the effect that the property has on the problem being addressed. Their definition is:

```
traits = element traits  
( element advantage {score, sValue, name, desc, as}
```

```

| element disadvantage {score, sValue, name, desc, as}
| element general      {score, sValue, name, desc, as}
)+

```

From the SIF definition, a trait's satisfaction value is stored within an attribute. The modeller provides a name and textual description for the trait, and then describes how the trait affects the solution using affects.

Affects

The final set of elements to be described are the relations between traits and the problem requirements they affect.

```

as = element affects {
  element affect { cValue, id, text }+
}

```

For each effect, the direction is encoded by referencing the identification number of the requirement as an attribute. The contribution value is encoded as an attribute and sourced from SIF. Optionally, a description for the effect can be given to provide more information.

6.1.4 Case Studies

Although the evidence element provides a section in which authors can show the existence of the pattern, the effect of pattern application should also be described. Classical pattern templates document this effect using sectional elements for establishing first, the *context* in which the problem existed, and second the *resolved context* that arose post pattern application. The template presented in this chapter takes an alternative approach.

```

studies = element studies {
  element study {
    score
    , name
    , element before { text }
    , element after  { text }
  }+
}

```

The `context` element documents the physical context in which the problem occurs with the problem being described in the `problem` element. The effect that solution application has on the problem is already documented in the `solution` element. To further illustrate the effect of pattern application, the `studies` element is introduced to allow for the description of case studies to be given. These studies consist of a simple title, together with elements used to detail the effect *before* and *after* pattern application.

6.1.5 Relations

The last major sectional element in the pattern document lists the relations that the described pattern has towards other named patterns. Pattern languages typically codify the relation between different patterns using a singular linked relationship: Patterns are *linked to* other patterns. However, existing work in the modelling of patterns for HCI identified several other naturally occurring links between patterns. Dearden and Finlay [DFo6] describe that the relationship between patterns is reminiscent of the relationship between objects within UML class diagrams. Patterns can: *extend* other patterns; *use* other patterns, *implement* other patterns; and are *associated* with other patterns.

```
relations = element relations {  
  ( element specialises { score, patternID, text }  
  | element requires   { score, patternID, text }  
  | element linked     { score, patternID, text }  
  | element implements { score, patternID, text }  
  )+  
}
```

With these different kinds of links, the final set of elements in the pattern schema describe the links, and kind of links, that the presented pattern has with other named patterns. The different relations are contained within the `relations` element, and are described using the tag names: `specialises` for extending; `requires` for aggregation; `implements` for realisation; and `linked` for association. For each different relation element the `element` value can be used to provide descriptive text detailing the relationship, and an attribute within the element to capture the identifier of the other pattern that this pattern is linked to.

6.1.6 Metadata

Alongside the sectional elements of the pattern template, a metadata element is presented. The metadata element documents modification times, the list of authors and auditors, alternative names for the pattern, and categorisation descriptors. The definition is:

```

metadata = element metadata {
  element aliases    { element alias { text }+ }
  , element tags      { element tag   { text }+ }
  , element created   { xsd:date   }
  , element modified  { xsd:date   }
  , element evaluated { xsd:date   }
  , element authors   { element author { text }+ }
  , element auditors  { element auditor { text }+ }
}

```

6.2 Library Provision

By design XML schema are technology agnostic, and describes the structure of a document. The schema presented in this chapter *describes* the structure of FREYJA encoded pattern documents. To promote the use of the schema in tooling, and to promote machine-readable design patterns, a software library has also been developed. This library is available online [dMH15d] and has been developed using Idris, the same language used to develop SIF. The main functionality provided by the library is two fold.

First, the library facilitates the serialisation of pattern documents from an XML encoding to custom data structures (records) and *vice versa*. The resulting data structures, facilitating programmatic interaction with pattern documents.

Second, this library supports the transformation of pattern documents to other document markup formats such as L^AT_EX, CommonMark, and Org-Mode. Such functionality being provided by the EDDA software library, a document transformation library written for Idris [dMH15c]. With access to document markup conversion tools, the library allows for the encoded pattern documents to be transformed into other formats for publication in a pattern repository.

The FREYJA library is used as part of the SIF evaluator (Chapter 5) to construct pattern document stubs from a SIF model, and by the FRIGG tool (Chapter 7) for

machine lead interaction with pattern documents.

6.3 Discussion

This discussion looks at the choice of headings in the pattern template, the idea of active documents, alternative representations, and sufficiency of the active components.

6.3.1 Chosen Headings and their Semantics

The classic *Alexandrian* pattern template presents one possible set of default headings to use when constructing pattern documents. During the creation of SIF, a natural set of alternative headings emerged for three of the sectional elements used to describe the problem, the solution, and their domain of operation. For the scope of this thesis the precise set of headings chosen should not be seen as a major contribution. The template chosen is just one example of how to encode pattern templates in a machine readable form.

When compared to the classical template the problem and forces sections have been merged and replaced with a single section for problems and listed together with a set of requirements. This collects and isolates the problem description from the rest of the pattern document, and enforces the writer to view the forces for the pattern solely in conjunction with the problem and not the solution.

The context section is used as was intended with the classical template such that it describes the domain of operation in which the pattern is to be applied.

Solutions within FREYJA combine the classical headings of solution description and the architectural details, and extend them with more structured headings. The exact set of headings and their decomposition provides a reflection over how a solution can be described. The architecture of the solution will need to be described using some formal notation, for example UML or BPM, together with ontological details provided alongside. Such ontological information and how it relates to the formal models is described using the properties subsections. These sections allow for properties (emergent or otherwise) to be detailed clearly. Further, within each of these subsections, the traits (good, bad, or neutral) can also be described together with links to the precise requirement from the problem that is affected. Thus, facilitating detailed descriptions over the effect that a solution has on the pattern and allowing pattern designers greater detail in describing their solutions.

Two headings used in FREYJA that are without counterparts in the classical pattern template are the `evidence` and `studies` sections. These sections allow pattern authors space to document evidence of the present solution in action, and case studies to document solution application.

6.3.2 Active Pattern Documents

Active Documents is a term most associated with *reproducible research*, and details: (a) how authors can construct a document that encodes more information over the content; and (b) provides readers with a document that can be interrogated and manipulated [Del+12; SD11]. Active documents *are* interactive documents. Such documents are implemented using a combination of a markup languages that supports the embedding of metadata within the document itself, and tooling to interact with the document. A common triple seen is that of: *Org-Mode*¹ for the markup; *Emacs*² for interaction; and *Org-Babel*³ to provide activation of the document's active components. Other known coupling of technologies to provide active documents include Mathematica notebooks⁴ for interactive math notebooks, their Python/Jupyter⁵ variants, and \LaTeX when working with Sweave and R.

With FREYJA comes the ability to describe active pattern documents, and construct an infrastructure to support working with such documents. Use of XML allows for machine readable pattern documents to be presented, and for extra information concerning the pattern to be encoded within the document. Other markup languages were considered for provision of active pattern documents. Popular data serialisation languages such as JSON⁶ and YAML⁷ are data-centric and not document-centric. Document oriented markup languages such as COMMONMARK⁸ are not expressive enough and require use of extension elements that are not supported in all implementations, or are not deployment agnostic as seen with *Org-Mode* and its reliance on Emacs. For this reason, an XML based format was chosen it is both: language and tooling agnostic.

¹<http://orgmode.org/>

²<https://www.gnu.org/software/emacs/>

³<http://orgmode.org/worg/org-contrib/babel/>

⁴<https://www.wolfram.com/technology/nb/>

⁵<http://jupyter.org/>

⁶<http://www.json.org>

⁷<http://www.yaml.org>

⁸<http://commonmark.org/>

Document templates, through schema definitions, are well-defined and also presented agnostic to the language of tool construction.

For FREYJA the extra information relates to the SIF model representation of the pattern, and evaluation data originating from PREMES. Further, notice that the definition of models within the FREYJA schema is left undefined. Model instances can be inserted directly within the document, extracted using the tooling and reasoned about directly.

6.3.3 Alternate Document Representations

The FREYJA schema is not the first XML oriented pattern document schema to be proposed. The serialisation of documents is important as it allows for programmatic interaction to occur with pattern documents. Chapter 2 §2.6.2, documented several known existing encodings. From the encodings presented, a common theme emerges of using either: an XML based encoding; direct representation within a database; or no representation at all. XML allows for programming language agnostic tooling to be developed with the core schema being documented outwith the influence of any one language.

The FREYJA schema is document-centric and facilitates the representation of pattern documents. This is a core theme within the thesis that a design pattern *is a* document. Modelling of the complete document promotes a holistic view of a design pattern. Other pattern representations focus on the modelling of the solution detailed within the pattern itself, or focus on representing pattern languages.

6.3.4 Sufficiently Active?

FREYJA is a schema for active pattern documents. The schema encodes the following active components within the document: (a) *a SIF model*—the formal model of the pattern; (b) *partial evaluation scores*—stemming from the evaluation process; (c) *architecture models*—UML or some other modelling notation; and (d) *sectional divisions*—the individual component's of the document. However, are the resulting documents sufficiently active? Are there any other aspects of the document that can be interacted with?

CoSP investigated the use of combining the emergent properties associated with the described solutions to the system models [Man+13]. The author's developed an

XML schema to formally link these properties with artefacts, however, the resulting document is not a *true* pattern document. CoSPs sacrifice the textual expressiveness for more precise descriptions. The schema proposed in this chapter does not provide a similar link as seen with the CoSP format. An area of improvement comes from working out how to link the formal system's models to the described properties and traits.

Other schema such as PLML, xPML, and EML, also take into account the relations between different patterns. FREYJA provides shallow linkage between patterns to be described. Provision of a deep modelling between individual solutions & problems, indexed by their domain of application, would allow for deeper reasoning about pattern languages to occur. FREYJA prohibits the efficient encoding of common elements found between patterns, nor allows for them to be reasoned upon.

6.4 Summary

This chapter presents a crucial aspect in the quest for better engineered design patterns. The FREYJA document schema provides the interchange format for serialising and representing pattern documents outside of software and is literally the *rug that ties the room together*. Further, FREYJA has active components and facilitates active machine readable pattern documents to be created.

With machine readable documents comes the provision of machine based interaction. Tooling can be constructed to interact with these documents, modify them, and transform them. Partial application of these ideas has already been presented in the form of the FREYJA library itself and its provision of custom data structures, serialisation support, and transformation to different document formats. Although, the library is implemented in Idris, the schema itself is programming language agnostic and alternative interaction libraries can be constructed for other languages.

However, one must not be quick to think that FREYJA is *the* pattern document and *the* template to be used. Existing work in pattern document representation has presented alternative takes that must be considered. For example, the CoSP schema from Mana et al. [Man+13] highlights how a closer bond can be constructed between the models representing the solution and solution properties. Future work can look towards replicating this bond between solution models and patterns.

FRIGG: A UTILITY FOR WORKING WITH DESIGN PATTERNS.

This chapter introduces and discusses a *proof-of-concept* tool FRIGG that builds upon several of the other contributions in this thesis, providing programmatic based interactions with design pattern documents. This chapter provides an overview of the tool, how it relates to the other contributions in this thesis, current limitations, and possible future directions. Chapter 9 demonstrates use of the tool during the evaluation process.

7.1 Overview

Existing work on interacting with design patterns was detailed in Chapter 2 §2.6.2 in which Lucrédio et al. [Luc+03] and Welicki et al. [WLA06] documented solutions for interactive viewing platforms and knowledge management of pattern repositories. The authors also explore how their systems can aid in the pattern application process using tooling that combines code generation and UML modelling. Rather than concentrate on providing a system to solely view patterns and provide code generation from solution descriptions, the design of FRIGG was motivated to investigate improvements to the evaluation process *and* interactions with pattern documents. FRIGG illustrates how active pattern documents that are both machine readable and checkable can be leveraged within a tool.

7.2 Feature Set

FRIGG has been designed as a modal command-line application to work with a single pattern document. The different modes of operation supported are:

- **Document Conversion:** Pattern documents can be converted from the XML based representation to other document markup formats as supported by the FREYJA library.
- **Readability Metric Calculation:** Pattern documents are analysed to produce a readability metric using several known algorithms. These precise algorithms used align with those discussed as part of the PREMES framework to grade *Pattern Legibility* in Chapter 8 §8.4.3.
- **Weighted Template Adherence:** FRIGG accepts both weightings and grading schemes used as part of the PREMES framework as input. These weights and grades are used to calculate the *Weighted Template Adherence* for each specified heading in the weightings file, accessing the scores stored as metadata within the pattern document itself.
- **Sif Model Checking:** FRIGG uses the SIF library to extract and evaluate the embedded SIF model.
- **Command-Line Interface (CLI):** User-led interaction with pattern documents comes in the form of a CLI. This interface provides access to the other described modes, together with fine-grained section based viewing of the loaded pattern and the ability to run XPath queries over the document.

With the documented feature set presented, FRIGG is a tool that can be used at different stages of the pattern engineering process. During the creation phase of pattern engineering, FRIGG can aid in the evaluation and publication of patterns. For pattern publication, the tool supports translation of patterns documents into alternative publication formats for different media sources. For pattern evaluation, the tool can extract, view, and calculate (automatically) evaluation data from the document directly. For instance, calculating various metrics used in parts of the PREMES evaluation framework. Thus, allowing for parts of the framework to be automated. Lastly, the tool can automatically extract and re-verify, the embedded SIF model representation.

Further, patterns can be viewed and interacted with. This can be beneficial during the application phase of pattern engineering for viewing patterns and interrogating their contents. The use of XPath and section based views also facilitates fine-grained viewing and querying of the document’s content.

7.3 Implementation Information

A pure Idris implementation was chosen to provide re-use of the modelling library from SIF, and to further explore working with dependent types. The choice in using Idris presents a restrictive engineering aspect in the implementation of FRIGG. Idris’ package ecosystem is not mature. When the research was being undertaken Idris was in its infancy and with little library support. This limits the possible feature set, any and all library support would have to be constructed *by hand*. Many of the dependencies created are listed and mentioned in the theses backmatter. Like the implementation of SIF algebraic effects were used to provide total control over effectful operations—see Chapter 5 §5.4.

An alternative approach would have been to use a language with a more comprehensive and maturer ecosystem to provide the user experience. However, this would have required extension of the SIF evaluator to work outside of Idris. This would necessitate facilitation of inter-process communication between an interaction layer and a SIF evaluator process, an ‘IDE Protocol’—cf. Mehnert and Christiansen [MC14]. While possible the approach was too complex for the projects needs. The pure Idris implementation was chosen for simplicity. An alternative approach might be to use a foreign function interface, however, there is no support for interacting with Idris libraries outside of Idris.

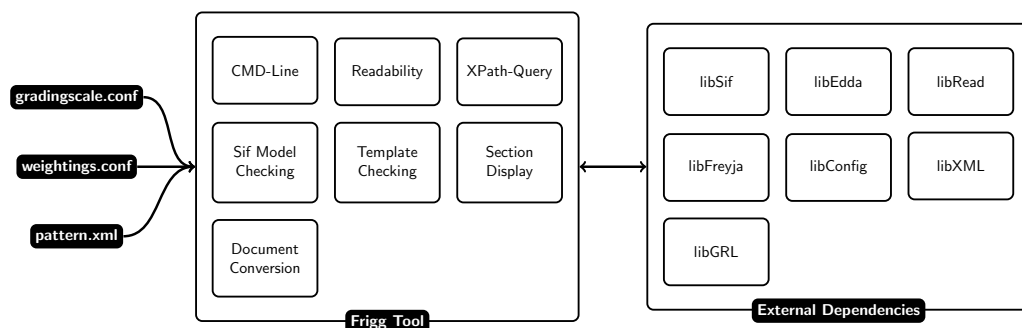


Figure 7.1: Feature-Set and dependency overview for FRIGG.

Figure 7.1 summarises the overall architecture of the FRIGG tool and the dependencies that it relies upon. The application itself is lightweight and brings together functionality from several existing libraries developed as part of the thesis itself. For example, both the core libraries arising from the implementation of both SIF, FREYJA, and NOVOGRL are used by FRIGG to support the evaluation of pattern models, the serialisation of pattern documents, and their conversion to other formats. The dependencies are enumerated as follows:

- **libSif** The core library of the SIF language, described in Chapter 5 [dMH15g].
- **libFreyja** Serialisation and parsing support for FREYJA encoded design patterns [dMH15d].
- **libGRL** Used by SIF to provide GRL modelling capabilities [dMH15e].
- **libRead** A library implementing several known readability metrics [dMH15f].
- **libEdda** A document modelling and processing tool providing a common document representation format and conversion tools [dMH15c].
- **libConfig** A library to support parsing of configuration files [dMH15a].
- **libXML** A library to support modelling and processing of XML documents, together with querying using XPath [dMH15i].

7.4 Future Features

There are several areas of improvement surrounding the engineering of the utility to provide a more richer feature set. For example, allowing multiple patterns to be ‘loaded’, better quality of output for results from possible queries, and support for other pattern templates. However, from a research perspective other areas for improvement relate to the creation of active patterns.

UML Metrics FREYJA presents an active document specification, and as part of that specification is support for the presentation of UML models. An interesting direction to take with FRIGG is not only the verification and validation of the presented SIF models, but also of the UML models inside. Part of the PREMES framework includes addressing the complexity of the presented coding solution. Having a design

pattern utility that presents UML complexity results automatically would be beneficial. However that would require support for a UML modelling library that allows support for metric collection as well as reasoning about different types of UML model¹.

More Automatic Evaluation Chapter 9 §9.6.2 demonstrated how the SIF evaluation results were used, in part, to calculate the *Weighted Solution Satisfaction* grade for the design patterns. The FRIGG utility can be extended to incorporate the automatic evaluation of this metric.

7.5 Summary

FRIGG is a *proof-of-concept* tool for interacting with design pattern documents. The functionality presented by the utility is limited, and most of the functionality presented by FRIGG is not new. Regardless, FRIGG demonstrates how the concepts and tooling in this thesis can be used to enhance how patterns are interacted with. For instance, facilitating for formal models embedded within the pattern document itself to be extracted, inspected, and verified automatically. Further, this tool shows how portions of the PREMES framework can be supported through programmatic interaction with the document itself. Chapter 9 documents how FRIGG helps with pattern engineering by demonstrating its use to evaluation patterns for authentication.

¹An early research direction involved the modelling of UML models in Idris. However, the resulting library and directions were dropped. The resulting library is available online [dMH15h].

PREMES: A PATTERN EVALUATION FRAMEWORK

Patterns are not *just* abstract concepts, they are also living documents. Although, the SIF modelling language can reason about patterns as requirements models, SIF cannot be used to provide assurances towards the quality of the pattern document itself. This thesis also presents PREMES: A holistic evaluation framework for addressing pattern quality. This chapter introduces the PREMES evaluation framework; PATTERN REPORT-CARDS—the reporting mechanism used within the framework; and provides a discussion about the framework’s limitations by using the framework to evaluate known patterns.

Note. The work presented in this chapter was originally presented at PLoP ’15 in de Muijnck-Hughes and Duncan [dMD15].

8.1 Problems with Pattern Evaluation

To promote better evaluation practices, pattern evaluation must be seen in the same regards as the evaluation of software systems or engineering results. Design pattern evaluation must be reproducible, must be consistent, and must allow for a fine-grained analysis of the presented pattern. Subjectivity of the assessor should also be managed

and reduced. Without these traits the evaluation process will be *laissez-faire*.

One of the difficulties in constructing an evaluation system for design patterns is that the subject domain covered by patterns is heterogeneous. Stipulating that patterns are to be evaluated using a single generic system is dangerous. The generality of such a system may not provide suitable treatment across important domain-specific aspects of the pattern. Further, stipulating that all patterns regardless of domain should follow the same pattern template is also too restrictive for similar reasons. Both Heyman et al. [Hey+07] and Bunke et al. [BKS11] noted that their research was hampered in part but the pattern template heterogeneity. A single general purpose evaluation system cannot be, and should not be specified. The differences between different types of patterns must not only be recognised but also celebrated. Any suggested evaluation process must be tailored to the pattern being evaluated. A question that naturally arises is:

Can a holistic approach be taken to: (a) construct a pattern evaluation system that tests for the quality of a pattern; (b) determine if the pattern is a solution to a problem, is tried and tested, and well documented; and; and (c) tailor the evaluation to the pattern(s) being evaluated?

8.2 Approach

The approach presented in this thesis is to establish a series of quality indicators drawn from existing literature that, when tested for, provide a sign over pattern quality. With these *indicators*, a mixture of qualitative and quantitative evaluation techniques are used to construct an evaluation system that determines how well the indicator has been satisfied. Use of quantitative and qualitative metrics allows for the quality of a pattern to be measured allowing for problem areas relating to pattern quality to be highlighted during the engineering process, and thus resolved before publication. With such metrics, patterns can then be graded according to their quality w.r.t. to the different indicators.

However, provision of summative feedback prohibits the provision of formative feedback over why a pattern fails. When the summative feedback is analysed, this lack of detail can lead to confusion over why a certain grade was allowed. Even more so this can damage the transparency of the evaluation process. Evaluation results should be enhanced by provision of formative feedback as well. Pattern Grades, combined with formative feedback results in PATTERN REPORT-CARDS.

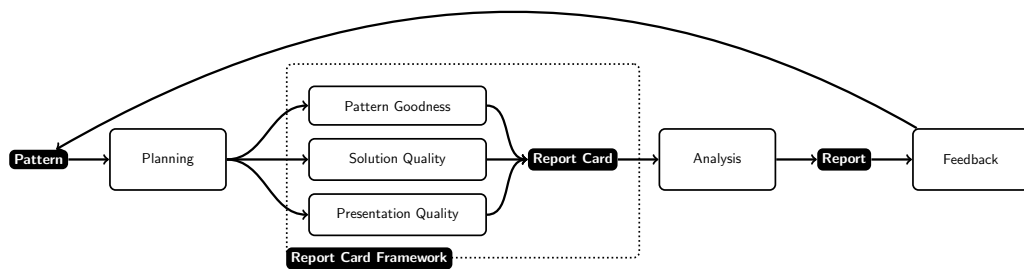


Figure 8.1: Schematic overview of the stages and indicators of the PREMES evaluation framework.

Different patterns will have different requirements for their assessment. Remember patterns do not exist in homogeneous domains. The *Plan-Do-Check-Act* (PDCA) management cycle is a standardised technique used for establishing bespoke evaluation procedures. Specification of such a management system for pattern evaluation allows for the establishment and execution of an evaluation system that can also be tailored to the pattern being evaluated. This management system can also be used to provide guidance not only over how best feedback can be given to pattern writers, but also guidance over how to execute the evaluation process.

With this combined framework (management system, evaluation system, and indicators) comes a reproducible means to evaluate patterns, and through provision of a tailored approach allows the quality to be assessed at a fine-grained level. Together this is the PREMES framework. Figure 8.1 presents a schematic overview of the entire framework.

8.3 Quality Indicators for Patterns

When evaluating patterns the pattern document and its contents must inform the evaluation criteria used. Laverdière et al. [Lav+06] presented a set of desirable properties for design patterns and nominal measures of their quality. These properties are taken and combined with other existing bodies of work on pattern evaluation [WF11; Haro4; BKS11; MD97; Hey13]. From the presented corpus a set of quality indicators that, if tested for, can be used to determine the quality of the presented pattern. If one analyses the informal textual definition of a design pattern it can be argued that design patterns are *supposed* to be: well-documented; tried & tested; and solutions to problems. The

quality indicators are grouped around these three notions.

8.3.1 Quality of the Pattern Presented

Research performed by Bunke et al. [BKS11], Heyman et al. [Hey+07] and Winn and Calder [WCo2] have identified that a proportion of the presented patterns fail at being patterns. Design patterns are supposed to be *solutions to problems*. But what exactly is a design pattern supposed to be? Attempts at providing more formal definitions do exist (see Chapter 2 §2.5) however, these definitions view design patterns as software artefacts and do not necessarily take into account emergent properties of the pattern such as documentation quality, and solution goodness. The first set of indicators determine if the presented pattern is a pattern, and address in part the core areas of: *context*, *problem*, *forces*, and *solution*.

Definition 1 (Pattern Coherency). *The pattern presented must present a solution to a recurrent problem for a particular context. The presented solution should match the level of abstraction of the presented problem (its forces), and also the context in which the problem exists.*

Definition 2 (Pattern Atomicity). *The pattern presented must be an entity from which other sub-patterns cannot be extracted. A single pattern should present a single discrete problem; the problem should not be composed of multiple problems. Related problems should be referenced in the pattern document, and a group of linked patterns should be presented as a pattern language.*

Definition 3 (Problem Independence). *The problem described together with its forces should not be influenced in description and construction by the presented solution. The problem should be independent from any presented solution.*

8.3.2 Quality of Solution Presented

Design patterns are supposed to be tried & tested solutions. Many pattern documents and guides stress that a design pattern is the successful resolution of a series of problem forces by a series of actions within a particular context [WF11]. These next set of quality indicators are concerned with the quality of the presented solution, and address in part the core areas of: *solution*, and *resulting context*.

Definition 4 (Solution Appropriateness). *The solution presented should be a solution for the presented problem. The presented solution should address the described problem in its entirety for the given context, and not present a general solution that is applicable to other problems.*

Definition 5 (Solution Complexity). *The presented solution should not be overly complex, and should not be difficult to apply. Solution complexity will impact upon the applicability of the pattern and how well it can be deployed to address the presented problem.*

Definition 6 (Solution Effectiveness). *The pattern document should provide evidence of the solution's effectiveness and robustness for addressing the presented problem. Such evidence can be used to determine the quality of the solution and how well the problem is addressed.*

8.3.3 Quality of Pattern Presentation

The final set of indicators are related to pattern presentation. Design patterns are supposed to be *well-documented*. Poorly presented ideas will be poorly received by readers. As design patterns are ostensibly used for domain knowledge transfer (from domain expert to non-domain experts) the quality of the presented pattern document should also be assessed. Harrison [Haro4], Heyman et al. [Hey+07], Yoshioka et al. [YWMo8] and Wellhausen and Fießer [WF11] present work in evaluating a pattern document's presentation and content. This final set of indicators provides treatment at the documentation level of all areas of concern within the pattern.

Definition 7 (Pattern Structure). *Pattern templates are used to provide a common structure for describing like patterns. A coherent structure provides better presentation of the topics. Measuring the quality of adherence to a known template can be used to indicate good structure. Further the chosen template should be suitable for the presented pattern.*

Definition 8 (Pattern Legibility). *The language used in the pattern documentation should convey clearly to the target audience the ideas being described. Use of overly complex language, or too simple a use of language, can hamper the reader's ability to comprehend the presented material.*

Definition 9 (Presentation Accessibility). *The problem, solution, and ideas should*

be presented in a way that promotes accessibility and does not hamper the readers ability to familiarise themselves with the concepts. Terminology, and concepts should be explained clearly and presented appropriately.

8.4 Pattern Report Cards

This section presents PATTERN REPORT CARDS, an evaluation system that can be used to test against the described indicators from §8.3. This system took inspiration from educational report cards that detail how well a student is performing. For each of the three quality areas several qualitative and quantitative techniques are used to gauge the pattern's quality. These techniques are used to create data points that can be used to track the quality of the pattern during the engineering process. Further, the assessment process has been designed to present pattern writers and auditors with a repeatable means to determine pattern quality, and indicate areas of improvement. The remainder of this section details the techniques used, and how they can indicate pattern quality.

8.4.1 Quality of Pattern Presented

The first part of the report card grades the pattern according to pattern quality using qualitative measurements. Grading schemes have been presented that allow for each of the quality indicators to be assessed. These schemes have been designed such that better grades indicate patterns that have the correct form, and lesser grades degradation of said form. The following list of tables presents The grading schemes for each of the pattern quality indicators:

- Table 8.2 presents the grading scheme for *Pattern Coherency*.
- Table 8.4 presents the grading scheme for *Pattern Atomicity*.
- Table 8.6 presents the grading scheme for *Problem Independence*.

8.4.2 Quality of Solution Presented

The next section of the report card determines solution quality. For this next set of grades quantitative values are obtained primarily from qualitative measurements. For each indicator of solution quality a different measurement and transformation is presented. Unlike the previous grading section, the formulations and values presented

-
- A** The pattern presents a well defined problem that is recurrent, describes a solution for that problem in a particular context, and both problem and solution are at the same level of abstraction.
-
- B** The pattern represents a reasonably defined problem that is recurrent, the solution addresses most of the problem presented for a particular context, and the levels of abstraction for the solution and problem are the same.
-
- C** The pattern presents an ill-defined problem that is somewhat recurrent, the solution addresses a substantial portion of the presented problem for a particular context, and the levels of abstraction for the solution and problem is similar.
-
- D** The pattern presents an ill-defined but not recurrent problem, the solution only addresses part of the problem specified for a particular context, and the problem and solution have similar yet differing levels of abstraction.
-
- E** The pattern does not present a well defined nor recurrent problem, the solution does not address the problem for any context, and the problem and solution have differing levels of abstraction.
-

Table 8.2: Grade descriptor for the *Pattern Coherency* indicator.

-
- A** The pattern is sufficiently constrained and cannot be decomposed into smaller patterns.
-
- B** The pattern presented is suitably constrained and aspects of the pattern could be turned into other patterns.
-
- C** The pattern presented is not constrained to a single problem and aspects should be decomposed into other patterns.
-
- D** The pattern presented addresses several unrelated problems and should be decomposed into several smaller patterns.
-
- E** The pattern presented is in fact a pattern language and addresses many inter-related problems, and should be decomposed into smaller patterns.
-

Table 8.4: Grade descriptor for the *Pattern Atomicity* indicator.

A	The problem presented is independent of the presented solution, and the problem forces are not indicative of the solution being proposed.
B	The problem presented is not influenced by the presented solution, and the problem forces bear some resemblance to the issues affecting the solution.
C	The problem presented is influenced somewhat by the presented solution, and the problem forces resemble issues affecting the use of the solution.
D	The problem presented is influenced by the presented solution, and the problem forces are descriptive of issues affecting the solution.
E	The problem presented is directly influenced by the presented solution, and the problem forces explicitly describe problems affecting the solution and not the problem.

Table 8.6: Grade descriptor for the *Problem Independence* indicator.

are not generic to all patterns and will differ per pattern. Where appropriate suitable descriptions are provided.

Solution Appropriateness

The first grading scheme is for solution appropriateness. Such appropriateness can be made quantifiable by establishing a metric sourced from qualitative values. To calculate this metric, one must first identify the problem forces, and assign a weighting (percentage) to indicate the importance of each force. Secondly, a grading scheme is defined to grade how well the solution addresses each of the problem forces in turn. This grading scheme must also be represented by a scalar number. For each force, multiplying each of the resulting grades by the force's weighting, a metric for solution appropriateness namely the *weighted solution satisfaction*, can be calculated.

Definition 10 (Weighted Solution Satisfaction). *Given a set of forces $\mathcal{F} = \{f_0, \dots, f_n\}$. Let $\mathcal{W} = \{w_1, \dots, w_n\}$ be a set of weightings for each $f \in \mathcal{F}$ such that $\sum_{i=0}^n w_i = 100$. Let $\mathcal{G} = \{1, 2, \dots, m\}$ be a bounded range of integer values that represents a grading scale. Let $\mathcal{E} = \{e_1, \dots, e_n\}$, $e_i \in \mathcal{G}$ be a set of evaluation values for each $f \in \mathcal{F}$. The weighted solution satisfaction for a pattern is calculated as follows:*

$$\sum_{i=0}^n w_i \times e_i$$

Solution Complexity

The second grading scheme is related to solution complexity. Within software design pattern literature, a solution's structure and dynamics are often modelled using UML. When presented with UML models the solution's complexity can be inferred by calculating model complexity. For UML Class Models, complexity metrics have been described [YWG04; MLo5; Mar98; MGP03]. However, metrics for other UML models such as deployment, component, and message sequence are not so well developed. For coding-oriented implementation patterns, metrics for code quality can also be constructed [LW12], and used to infer solution complexity. Regardless of UML model or provided code when determining the complexity of the given solution, a set of metrics for solution complexity can be generated providing quantitative measures for the report card. For patterns that do not provide software artefacts a qualitative grading scheme can be presented instead. Table 8.8 presents one such example.

Complex	The solution is too complex with a structure that contains too many modules that have too many relations. Further, the interactions between the components are too many for the interactions described.
Adequate	The solution presented has a structure and set of dynamics that are suitable for addressing the problem presented.
Simple	The solution presented has a structure and set of dynamics that are too simplistic for the problem being addressed. This solution does not capture enough detail for the problem presented.

Table 8.8: Sample grading scheme for *Solution Complexity* indicator.

Solution Effectiveness

The final aspect of solution quality is that of solution effectiveness. How effective the solution address the problem can aid in determining pattern quality. Whereas complexity can be quantitatively measured, effectiveness is a qualitative measurement that can be performed through walk-throughs with especial regard to both normal and abnormal usage. Some quantitative measures can come from taking the pattern requirements and generating test cases to apply to the pattern design. Guidance from requirements engineering evaluation practices will advise this process with especial regard to the interface, inputs and outputs.

Naïvely, a simple solution is to look for evidence of metrics and evaluation criteria with the presented pattern document. With such a naïve measure the grading scheme (shown in Table 8.10) can be employed to judge solution effectiveness.

8.4.3 Quality of Pattern Presentation

The final set of grading schemes presented are for pattern presentation. Here the schemes presented are a mixture of quantitative values constructed from qualitative measurements, and qualitative grading. Pattern presentation can be assessed according to: adherence to known templates; use of representational aides; and quality of writing style used.

Pattern Structure

Heyman et al. [Hey+07] presented a methodology for assessing the quality of pattern documentation according to how well a pattern adheres to a given template: *Weighted Adherence to a Pattern Template*. This indicator can be used in the report card to determine the adherence a given pattern has towards a specified template. Each heading in a given pattern template is associated with a weighting indicating the importance of each heading within the template. During evaluation each heading is graded to indicate the quality of provision. An adherence metric can then be calculated through summation of the scores for each element. The higher the score the greater the adherence to the template. This is formalised as:

-
- | | |
|----------|--|
| A | The pattern presents ample evidence that the solution presented is effective. |
| B | The pattern presents sufficient evidence that the solution presented is effective but some aspects of the solutions effectiveness are not described. |
| C | The pattern provides links to evidence that the presented solution is effective in addressing the problem. |
| D | The pattern alludes to the effectiveness of the solution but does not categorically present evidence attesting to the fact. |
| E | The pattern does not present any evidence that the presented solution is effective in addressing the problem. |
-

Table 8.10: Sample grading scheme for the *Solution Effectiveness* indicator.

Definition 11 (Weighted Adherence to Pattern Template). *Given a pattern template \mathcal{T} . Let $\mathcal{W} = \{w_1, \dots, w_n\}$ be a set of weightings for each $t \in \mathcal{T}$ such that $\sum_{i=0}^n w_i = 100$. Let $\mathcal{G} = \{1, 2, \dots, m\}$ be a bounded range of integer values that represents a grading scale. Let $\mathcal{E} = \{e_1, \dots, e_n\}$, $e_i \in \mathcal{G}$ be a set of evaluation values for each $t \in \mathcal{T}$. The weighted adherence to a pattern templates is calculated as:*

$$\sum_{i=0}^n w_i \times e_i$$

Pattern Legibility

Pattern document legibility can be assessed using existing readability metrics such as Flesch-Kincaid, Coleman, and FOG [Kin+75]. In readability metrics low scores represent use of simplified language constructs, and higher scores represent more complex language. Often such readability metrics are interpreted according to the American Educational Grade Level to allow easy interpretation of the result. These metrics are used to indicate how advanced, or simplified, the language used will be. Allowing for the actual reading level of the document to be determined from the intended reading level.

Presentation Accessibility

Table 8.12 presents the final grading scheme for assessing pattern accessibility. Accessibility can be determined by judging the pattern according to the terminology used and clarity of the descriptions. This grading schemes requires that the existence and suitability of additional presentation attributes be assessed together with the clarity of presentation. Examples of additional presentation attributes can include the existence of, for example, UML models, diagrams, references to existing work and usage, and code examples. This grading scheme differs from that of the readability metrics by looking not at the language used but how the concepts are presented.

8.5 The PREMES Framework

The previous section introduced *Pattern Report Cards* and how they are used to grade patterns. Several of the grading schemes presented are not generic to all patterns and need to be tailored prior to use. The PREMES approach uses the PDCA cycle (§8.2) to

A	The pattern is presented in an accessible manner using clear language. The concepts and terminology used are explained appropriately.
B	The pattern is presented in an accessible manner but does not use clear language. The concepts and terminology are explained using unclear language.
C	The pattern is presented using clear language but is nonetheless inaccessible. The use of terminology is given but poorly explained and presented unsatisfactorily.
D	The pattern is presented using unclear language, makes use of terminology that is poorly explained, and presented unsatisfactorily.
E	The pattern is presented using unclear and inaccessible language. Terminology used is unfamiliar and unclear to the reader. The pattern is presented poorly.

Table 8.12: Grading scheme used for the *Presentation Accessibility* indicator.

manage the execution of the evaluation process and manage the introduction of the required tailoring. This section describes the activities required for each of the four stages, and the rationale behind each stage where appropriate.

Note. The description of PREMES has been given under the assumption that the pattern is to be reviewed by a group of *auditors* who are separate from the pattern author. This is to aid in documenting the framework’s execution.

8.5.1 The Planning Stage

The first stage is a preparatory planning stage that requires for the scope and extent of the evaluation to be established. The required activities involved are:

- 1) Identification of the patterns that are to be evaluated.
- 2) Agreement on the weightings to be used in the Report-Card process.
- 3) Identification of the Pattern Template used and weightings for the headings.
- 4) Agreement on the readability metric used for analysing language style.
- 5) Agreement on marking criteria for each of the qualitative grading schemes.
- 6) Agreement on number of iterations that the cycle will go through.
- 7) Agreement on how to collate results from different report cards.
- 8) Agreement on how results are to be reported.

This stage explicitly ensures that there is a consensus for how the evaluation is to be conducted, and how each aspect of the evaluation is to be performed. This will

included identifying how the evaluation is to be tailored for the particular set of patterns presented. Of note is the agreement on how results are to be reported, allowing those auditing the patterns to decide precisely what results the author will receive. For example, provision of free flow formative feedback alongside the summative grading.

8.5.2 Grading the Pattern

The second stage documents the execution of the pattern report card process. The required activities are:

- 1) Grade the Quality of Pattern Presented.
- 2) Grade the Quality of Solution Presented.
- 3) Grade the Quality of Pattern Presentation.
- 4) Creation of the Pattern Report Card.
- 5) Detail any formative feedback.

8.5.3 Analysing the Results

The third stage stipulates: the collection of the results from each of the report cards; collation of the results into a single report card; and identification of actionable items. The required activities involved are:

- 1) Discussion of the report cards' results.
- 2) Collation of the results according to the agreed upon scheme.
- 3) Identification of actionable items and recommendations for change.
- 4) Ordering of actionable items in order of precedence.
- 5) Creation of a report detailing the results of the report card together with formative feedback.

The report created must provide a summary of the grading schemes utilised in the evaluation. This stage provides an agreed upon analysis of the results and present of a coherent set of prioritised actionable items. These items can be produced alongside a single document that contains a single report card for each pattern and more formative textual feedback.

8.5.4 Results Reporting

The final stage of the process requires: the reporting of the results; improvement of the pattern according to recommendations; and re-evaluation of the changed pattern.

The required activities involved are:

- 1) Submission of the report card and recommendations.
- 2) Changes based upon the recommendations.
- 3) Submission of the amended pattern for review.
- 4) Reevaluation of the pattern.

8.6 Evaluation

To help determine the efficacy of the PREMES approach, several existing patterns were evaluated and report cards generated. These patterns provide a diverse set of pattern types, and each use different pattern templates.

Guerra et al. [Gue+14] provides behavioural patterns that support developer workflows; Priebe et al. [Pri+04] details abstract patterns for software-based authorisation management; and Braga et al. [BRD98] provides patterns for information security. Table 8.13 lists the specific patterns chosen for evaluation, together with their grades and summative scores. This section discusses the results of the evaluation and how the report cards were generated.

8.6.1 Methodology

The tooling presented in this thesis (FREYJA & FRIGG) can aid in the evaluation of software patterns presented. However, the behavioural patterns do not have structure and explicit dynamics. Each of the patterns presented were encoded using the org-mode syntax to provide syntactically similar pattern files. These files were analysed by a simple tool written in Idris to automatically calculate the readability scores for each document using the same tooling as exposed by the FRIGG tool. The different pattern templates were given a different set of weightings, and each set of identifiable forces in each pattern were given equal weighting. A lightweight representation in code of a pattern report card was also implemented in Idris, facilitating dynamic calculation of the *Weighted Solution Appropriateness*, and *Weighted Adherence to Pattern Template* values, as well as collation and printing of the results for each evaluation.

For assessment of solution complexity, the grading scheme detailed in §8.4.2 will be used with an extra grade descriptor of ‘incomplete’ to represent incomplete information. Assessment of solution effectiveness shall use a six point grading scale, represented by the set {A, B, C, D, E, F}. Each point on the scale describes a different quality of

provision of evidence of the solutions effectiveness. Pattern legibility is determined by the *Kincaid* readability metric [Kin+75]. Appendix A presents the data files, utility software, and raw results of the evaluation.

8.6.2 Results

Table 8.13 summaries the results of the evaluation. The remainder of this section discusses the results of the evaluation around the three areas of the report card: Pattern Quality, Solution Quality, and Presentation Quality. An detailed statistical analysis of the result was not performed due to the low sample size. For larger evaluation sets, use of statistical models could result in more analysis between the effect of the different indicators and pattern quality.

Quality of Pattern

The quality of presented patterns were mixed. No overall group of patterns did particularly well. The patterns from Priebe et al. [Pri+04] and Braga et al. [BRD98] scored low in comparison with the other patterns. If one looks at the problems and associated forces from both sets of patterns, the forces and solutions are more indicative of the solutions being presented than for the problem itself. Thus, explaining the low scores.

Also evident in all of the evaluations was the patterns lack of coherency. Although many of the presented patterns had the right level of abstraction between the solution, context, and problem, these three concepts were ill-defined.

Of note were the good scores for pattern atomicity. Here most pattern's achieved a grade B or higher. Not many authors proposed patterns that should have been modelled as pattern languages.

Quality of Solution

Similar to the results for pattern quality, the results for solution quality were also mixed. All patterns lacked clear evidence over their effectiveness, with the complexity of the solution hard to determine. Solution appropriateness was also hard to determine when the forces were not explicitly given in the pattern. Of which the patterns from Priebe et al. [Pri+04] are a good example, they all scored 0.0 when calculating the *Weighted Solution Satisfaction* metric.

Source	Pattern	Quality Indicators ¹									
		Pattern			Solution			Presentation			
		I1	I2	I3	I4	I5	I6	I7	I8	I9	
Guerra et al. [Gue+14]	CHOOSE YOUR WEAPON	C	E	E	0.5	incomplete	D	50.0	9.14	B	
	UNDERSTAND CLASS ROLE IN ARCHITECTURE	C	B	B	0.25	incomplete	D	50.0	11.72	B	
	FUNCTIONALITY LIST	B	B	B	0.5	adequate	D	68.75	8.10	B	
	KNOW YOUR NEIGHBOURHOOD	B	A	E	0.75	adequate	D	63.75	10.71	B	
Priebe et al. [Pri+04]	AUTHORISATION	C	B	C	0.0	adequate	D	52.0	9.64	B	
	SESSION	C	D	B	0.0	adequate	D	36.0	8.30	C	
	RBAC	C	B	E	0.0	adequate	D	36.0	8.46	C	
	MBAC	C	B	E	0.0	adequate	D	36.0	10.02	C	
Braga et al. [BRD98]	SESSION MBAC	C	B	E	0.0	adequate	D	36.0	9.486	C	
	INFORMATION SECURITY	B	E	E	0.25	adequate	D	41.68	6.83	C	
	MESSAGE INTEGRITY	B	B	E	0.41	adequate	D	41.68	7.05	C	
	MESSAGE AUTHENTICATION	B	B	B	0.5	adequate	D	50.01	7.00	C	
	SENDER AUTHENTICATION	B	C	E	0.25	adequate	D	50.01	7.44	C	

¹ For brevity the indicators are given in order of presentation from §8.3.

Table 8.13: Summative pattern report cards for several existing patterns.

When the solution was poorly presented, this hampered determining the solution's complexity, and effectiveness. Notice the low grades for solution complexity and effectiveness, where no one pattern scored higher than *adequate*. This is expected as the insufficient description did not present all the facts. For example, the efficiency of the solution is important for the INFORMATION SECURITY problem. Solutions to algorithm selection nor key length selection, for example, are not alluded to at all in the pattern. Similarly, analysis of abstract patterns such as those presented by Priebe et al. [Pri+04] makes determining solution quality harder. There was incomplete information over the presented solution, and only UML class diagrams could be used to judge the complexity of the presented solutions.

8.6.3 Quality of Presentation

The presentation quality for each pattern was also mixed. The patterns from Priebe et al. [Pri+04] and Braga et al. [BRD98] exhibited poor adherence to their specified templates. With only a third of the patterns scoring higher than 50.00. This was a result of lack of information given.

The legibility indicators showed a degree of consistency within each of the individual sets of patterns presented. Comparing between sets, it was interesting to see that the Braga et al. [BRD98] patterns had the lowest and smallest interval: [6.84, 7.44] of size 0.6. Whereas, Priebe et al. [Pri+04] had an interval of [8.3, 10.02] with size 1.72. Guerra et al. [Gue+14] had the highest and widest interval of [8.10, 11.72] with a difference in grades of 3.62. Comparing the intervals to the corresponding American grade level, as indicated by the Kincaid readability metric, the difference in grades varied. The Guerra et al. [Gue+14] patterns differed by at most four grade levels; two grade levels for the Priebe et al. [Pri+04] patterns; and one for the Braga et al. [BRD98] patterns. It is interesting to note how one set of patterns varies when compared to another. If one looks at the wording used in UNDERSTAND CLASS ROLE IN ARCHITECTURE, the pattern with the highest readability score, it uses more complex words, when compared to the lowest scored pattern of INFORMATION SECURITY. While this may have an effect of the resulting scores, the INFORMATION SECURITY pattern uses fewer words overall.

Finally, the accessibility indicator shows that many of the provided patterns were written with some assumed knowledge and that not all concepts were explained. This raises a question over how much assumed knowledge should the pattern author expect

of their reader, and how much should be presented in the documented or linked as a reference.

8.7 Discussion

The PREMES framework has coalesced into a single solution various techniques and methodologies from various different fields such as requirements engineering, software design, testing, and modelling. The resulting solution evaluates patterns using a mixture of metrics gathered from quantitative and qualitative data sources. Use of the iterative process presented, and gathered metrics, allows for pattern deficiencies to be identified and tracked during development. Allowing for weak pattern areas to be identified and addressed. This section discusses the proposed framework according to: its scope; style of feedback; measurement techniques; and areas for improvement.

8.7.1 Scope of Evaluation

§8.3 introduced the indicators used to identify quality. A question naturally arises over how complete these indicators are in determining pattern quality.

An aspect not explicitly mentioned, nor tested for, is that of usability. *How usable is the pattern document by non-domain experts?* Usability can be used to identify how accessible the pattern document is, and also ease of pattern application. For assessment of pattern usability such testing would require user studies. Thimthong et al. [TCK13] have explored this area. However, use of user studies should be limited as the usefulness of such studies can be ineffective and unhelpful if done improperly [GBo8]. Nonetheless, the presented quality indicators, and the tests for those indicators, in fact test for important aspects relating to usability. For example, accessibility of the language used to introduce the concepts, the known and perceived complexity of the solution presented, and how well the auditor believes the concepts are made accessible.

A secondary problem in determining the scope of the analysis is that most of the report card system is left purposely undefined. It is up to those auditing the pattern to determine precisely what is involved in the analysis. For the quantitative portions of the analysis such concerns can be minimised through specification of approved analysis techniques whose scope and limitations are known. For the qualitative measurements, more guidance can be produced to guide in selection of a qualitative value.

Several of the indicators use information presented in the pattern as testing criteria. If this information is badly presented this will affect the quality of the resulting evaluation. This problem is addressed as part of the evaluation process. Having multiple rounds of evaluation facilitates consideration of this information and ensure that the information is sufficiently presented prior to the evaluation process.

8.7.2 Evaluation of Pattern Languages

Pattern language quality is just as important as individual pattern quality. The combined use of several patterns together may affect the quality of the solution being presented to address the larger problem being tackled. However, the PREMES framework evaluates patterns in isolation, and does not take into account related patterns, nor the evaluation of pattern languages. The PREMES approach is limited in effectiveness when used against pattern languages. Future work will be to investigate if the approach can be extended to include pattern language evaluation.

8.7.3 Formative vs Summative Feedback

When using Pattern Report Cards in isolation the feedback is summative and prohibits further explanation of why grades were awarded. When combined with the management process more formative feedback can be given alongside the report card, allowing explicit mention of a pattern's deficiencies. More so, during the checking phase (§8.5.3) there are no restrictions on the style and detail of the reports created. For example, given the presented metrics and a sufficiently large data set more complex statistical analyses could be performed on the data to determine further meaning.

8.7.4 Qualitative or Quantitative

The evaluation techniques proposed use a mixture of analyses: pure quantitative analysis; quantitative analysis derived from qualitative measurement; and pure qualitative analysis. These techniques allow for the pattern quality to be measured, and also made reproducible. However, such a mixture of evaluation techniques, especially the use of qualitative measurements, increases the difficulty in ensuring consistent reproducible evaluation scoring. Pure qualitative measurement is known for being highly subjective and open to interpretation, with quantitative scoring based on qualitative measurement less so. The requirement of subjective evaluation could have a detrimental effect

on the quality of analysis: *one person's junk is another person's treasure*. In its current state *Pattern Report Cards* offers too high a degree of subjectivity in its evaluation with too many of the grading schemes being purely qualitative. Future work will be to investigate how these subjective aspects of the evaluation can be made more objective or the subjectivity cancelled out. There are several approaches that can be considered:

Better Guidance A naïve first approach is to produce more guidance for each of the grading schemes presented. This guidance would provide a more detailed description over what a pattern would look like if it was to be awarded a specific grade. This will facilitate use of the grading scheme in a more effective manner.

More Quantitative from Qualitative There will always be a subjective aspect relating to pattern quality if qualitative techniques are employed. A second more practical approach is to reduce the size of the qualitative measurements being taken. This will reduce the effect that an auditor's subjectivity will have on the grading. This is achieved by transforming the purely qualitative measurements into a quantitative grade calculated from qualitative measurements. Each of the quality indicators can be broken down into individual attributes that can be measured qualitatively and these values used to construct a quantitative grade. This is the approach taken for weighted adherence to pattern template.

Psychometric Questionnaires A third and final approach is to fully embrace and acknowledge the existence of subjectivity in quality evaluations. With this, psychometric testing techniques such as Likert and Guttman Scales, can be used to determine the auditor's attitude towards pattern quality using these techniques. Such scales were used by Thimthong et al. [TCK13] in their usability studies. In particular Likert Scales measure a subject's response according to their level of agreement or disagreement. When applied to patterns, a psychometric test can be devised to determine the auditor's agreement level over the quality of the presented pattern. Key to the use these techniques is the need to minimise the inherent biases present within the subject's own response such that a true picture of the subject's attitude is measured. This can be used to control and minimise the inherent bias as presented by the auditor.

8.7.5 Need for an Overall Grade

Pattern Report Cards do not provide an overall grade for the pattern being evaluated. Although the individual grading schemes provide a detailed assessment over how well a pattern performed, the lack of an overall grade may trouble the pattern writer over how good their presented pattern is. This raises a secondary question of: *Can differing levels of quality be established given the disparate set of grading tools used for each quality indicator?* To aid in the creation of an overall grade, the disparate set of grades needs to be interpreted to provide a single value that is indicative of pattern quality. That is, a grade conversion and collation algorithm needs to be created. Given the second approach to increasing objectivity in the evaluation given in §8.7.4, a better approach may be to harmonise the grading schemes such that the same reporting scale is used. Each indicator would be divided into smaller weighted attributes that are used to calculate a quality value for the indicator using the same reporting scale. The indicators can also be assigned a weighting (denoting importance) such that a weighted average can be constructed from the grades presented. From this differing levels of quality could be established based on the possible range of the final value. This is left as future work.

8.8 Summary

Building on top of existing academic work for pattern evaluation, PREMES is an holistic evaluation mechanism for design patterns. At the heart of this framework are PATTERN REPORT CARDS. These report cards assess a pattern according to a set of quality indicators. How the pattern is graded can be tailored per pattern and the management cycle makes this tailoring an explicit process. PATTERN REPORT CARDS presents a measurable and reproducible evaluation technique for design patterns, that allows problem areas of a pattern to be highlighted during evaluation.

However, PATTERN REPORT CARDS are not perfect, and still suffer, to a degree, from subjective scoring. Further, the more quantifiable measurements taken (i.e. *Weighted Solution Appropriateness*) are dependent upon certain aspects of the pattern being readily identifiable. Priebe et al. [Pri+04] presented a set of patterns that had no identifiable set of forces. Lack of such information will harm the quality of the evaluation reporting. However, it can be argued that this will become self-evident during the evaluation, and can be reported back to the writer for clarification.

ENGINEERING PATTERNS FOR AUTHENTICATION

Chapter 2 §2.3 introduced the engineering process for software design patterns as described by Yoshioka et al. [YWM08]. This thesis presents tooling and techniques that enhances the pattern engineering process to make it more robust and demonstrable. Figure 9.1 illustrates the connection between the presented technologies and the workflow associated with pattern development. This chapter illustrates the thesis contributions further by providing a tutorial that details how patterns for the problem of authentication can be *engineered*. Presented are patterns for: AUTHENTICATION THROUGH SHIBBOLETHS; and AUTHENTICATION THROUGH ID CARDS

Note. The scope of this chapter is to highlight how the contributions can enhance design pattern engineering. The patterns presented are illustrative in nature, and their veracity not the main concern.

9.1 Overview

The AUTHENTICATOR pattern [BF99] presents a technical solution to the problem of authentication with a remote service. This is not the only authentication related pattern seen in literature. Erber et al. [ESP07], Fernández and Sinibaldi [FS03], Fernández

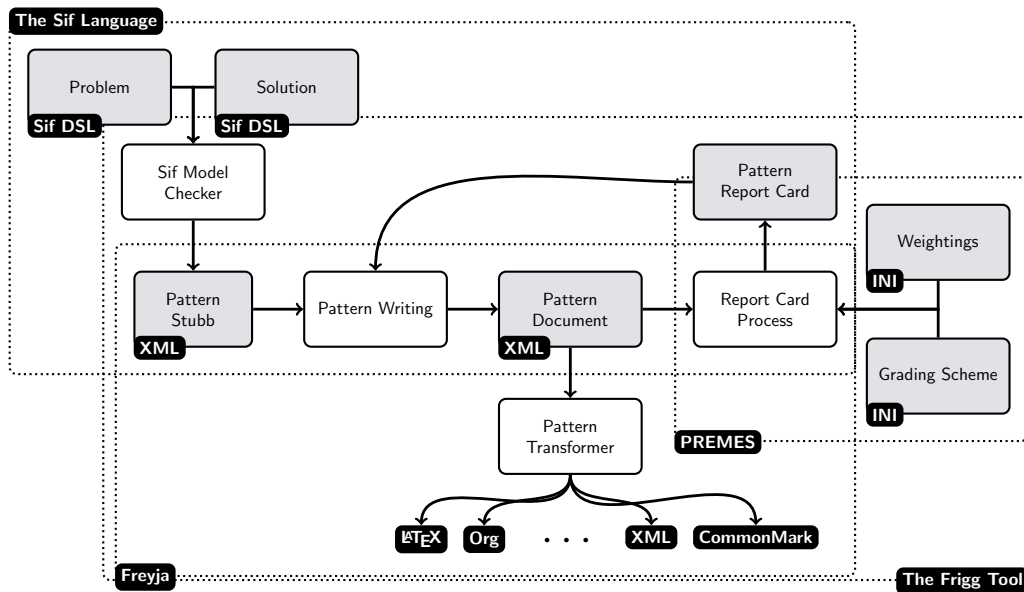


Figure 9.1: Tools and technologies presented in this thesis and their placement in the pattern engineering process.

and Warriar [FW03], Fernández [Fero7], Ajaj and Fernández [AF10], Morrison and Fernández [MF06] and Weiss [Weio6] each present a series of similar patterns for authentication. These patterns differ in technologies used and domains of operation.

This tutorial describes how the SIF modelling language (Chapter 5) can be used to produce a generic requirements-oriented problem specification for the ‘Authentication Problem’. The resulting specification will be agnostic not only to the context in which authentication appears but agnostic to any proposed solution. Using the created problem specification, two solutions for authentication will be specified. The first, *Shibboleths*, details how authentication can operate *sans* technology. The second, *ID-Cards*, looks at authentication that requires use of technology. The evaluation mechanism for SIF will be used to determine how well each of the presented solutions satisfies the problem. SIF can then be used to generate document stubs that adhere to the pattern template FREYJA—Chapter 6. Following the creation of the pattern documents, the resulting patterns are evaluated using the PREMES framework (Chapter 8) helped by the FRIGG utility—Chapter 7—allowing for potential deficiencies to be identified. The FRIGG tool allows for various aspects of the evaluation process to be automated, and for the pattern document to be transformed into other formats for publication.

9.2 The Problem of ‘Authentication’

The AUTHENTICATOR pattern [BF99] presents a technical solution to the problem of authentication with a remote service. This section details how a SIF problem specification for this problem can be constructed. Problems in SIF are modelled as sets of requirements that are indexed by the domain of operations in which the problem occurs. The full specification is presented at the end of this section in Listing 9.3, and Chapter 5 §5.4.2 details the SIF DSL used.

9.2.1 Problem Declaration

The problem of authentication can be summarised as:

Given two entities Alice and Bob, how can Bob authenticate with Alice such that Alice knows that Bob is who he says he is.

Authentication mechanisms are used to provide assurances over the identity of an entity; or provenance of an object. When modelling problems with SIF we must first declare the existence of the problem and then provide the problem’s requirements. Listing 9.1 provides the initial declaration of the authentication problem. SIF only allows for a single problem to be specified per file. The declaration on Line 1 informs the SIF evaluator that a problem specification is being declared. When declaring problems within a specification, the type of specification needs to be specified before the problem instance itself can be given. Line 5 provides the declaration of the problem itself. The left arrow (<-) declares and initialises the symbolic representation of the problem. Right of the arrow is the type declaration **Problem**, and right of the type declaration is the title associated with the symbol. This notation is used throughout the problem declaration to assign values to symbols.

```

1 sif problem
2
3 > How to authenticate entities such that assurances
4 > can be made of their identity.
5 authentication <- Problem "Authentication"
```

Listing 9.1: Initial problem declaration for modelling the problem of *Authentication* in SIF.

Preceding the problem declaration is the corresponding textual description for the problem presented as SIF documentation. Model documentation is indicated using ‘Bird’ notation i.e. the ‘>’ operator.

9.2.2 Contexts of Operation

With the problem itself declared, the domains of operation can now be defined. Existing literature has already provided a corpus of patterns that illustrate the various contexts in which authentication can occur [ESP07; FSo3; FW03; Fero7; AF10; MFo6; Weio6]. These identified patterns illustrate that the problem of authentication exists in a myriad of contexts. For this tutorial five different contexts were identified:

- **Local Technical:** How to perform authentication locally between entities without human assistance. For example, *Firmware Signing* is a means for devices to authenticate software locally on a machine.
- **Remote Technical:** How to perform authentication remotely between entities without human intervention. For example, *Certificate-Based SSH Login* is an automatic authentication mechanism to allow two devices to communicate.
- **Local Socio-Technical:** How to perform authentication locally between entities that require human involvement. For example, password based device authentication.
- **Remote Socio-Technical:** How to perform remote authentication between entities that require human involvement. For example, device pairing requires human involvement to enter a ‘pairing code’ to allow two devices to pair with one another.
- **Societal:** How to perform authentication between entities that does not require technology. For example, shibboleth’s and code phrases are challenge-response mechanisms used to allow agents to authenticate with each other without the need for technology.

Listing 9.2 shows how these contexts are declared in the specification file. The syntax required for context declaration follows that for problem declarations. However, the type differs and specifies that the resulting object is of type **Context**. For each context

```

1 socio          <- Context "Non-Technical"
2 sociotechLocal <- Context "Local Socio-Technical"
3 sociotechRemote <- Context "Remote Socio-Technical"
4 techLocal      <- Context "Local Technical"
5 techRemote     <- Context "Remote Technical"

```

Listing 9.2: Modelling of contexts in which authentication can take place in SIF.

defined, an identifier is required and an explanatory title. Each context can be documented, if the modeller wishes to provide more information. During model evaluation, contexts are automatically assigned to the presented problem when presented with a solution for a specified context.

9.2.3 Requirements

Next comes specification of requirements detailing aspects of the problem that the solution must address. For the problem of authentication, one possible set of requirements can be:

- **Proof of Authenticity:** A proof is required that will attest to the authenticity of an entity.
- **Enrolment** Solutions must have a means to enrol entities into the authentication procedure and provide them with proof of identity.
- **Authentication Step:** Solutions must have an authentication step in which the proof of an entity's identity will be tested.
- **Changeable:** The means of authentication must be changeable and re-enrolment possible.
- **Consistent:** The act of authentication must be consistent and each authentication act, if repeated, must have the same outcome.
- **Timely:** Processing authentication requests must be timely. Further, the duration of the complete authentication process must be timely in its duration.
- **Lockout:** Entities that repeatedly authenticate incorrectly should be denied the right to authenticate for some predefined period of time.
- **Effortless:** The authentication process should be effortless to perform.

```
1 sif problem
2
3 > How to authenticate entities such that assurances
4 > can be made of their identity.
5 authentication <- Problem "Authentication"
6
7 socio <- Context "Non-Technical"
8 sociotechLocal <- Context "Local Socio-Technical"
9 sociotechRemote <- Context "Remote Socio-Technical"
10 techLocal <- Context "Local Technical"
11 techRemote <- Context "Remote Technical"
12
13 enrollment <- Functional "Enrollment"
14 proof <- Functional "Proof of Authenticity"
15 authStep <- Functional "Authentication Mechanism."
16 changeable <- Functional "Changeable."
17 consistent <- Reliability "Authentication happens."
18 timely <- Performance "Timely Authentication."
19 lockout <- Functional "Limited access attempts."
20 effortless <- Usability "Effortless Authentication"
```

Listing 9.3: SIF Problem specification for the problem of *Authentication*.

Within SIF, types for requirements come from the FURPS requirements model [Gra92]. This model categorises requirements as being related to one of the following concepts: Functional, Usability, Reliability, Performance, or Supportability. Allowing each requirement to be given a type that better describes the requirement being detailed. For example, the ‘consistent’ requirement is a reliability requirement rather than a purely functional one. Listing 9.3 lists the complete problem specification and also details the requirements for the problem together with their requirement types. For brevity, documentation for each requirement has not been given. With the problem specification given, the two *potential* solutions can now be investigated.

9.3 Addressing Authentication

The patterns presented by Hashizume et al. [HFH09], Hashizume and Fernández [HF10], Braga et al. [BRD98], Cuevas et al. [Cue+09] and Cuevas et al. [Cue+10] detail technological solutions to authentication. Common to these patterns, and other non-pattern described solutions, is the idea that authentication requires some form of enrolment step, a proof of authenticity, and the actual act of verifying the proof.

This section will detail the construction of two solutions for authentication for two

different contexts: *Shibboleths*—for the societal context; and *ID Cards*—for the socio-technical context. These will result in SIF models for the patterns: AUTHENTICATION THROUGH SHIBBOLETHS; and AUTHENTICATION THROUGH ID CARDS. Evaluation of these models is detailed in §9.6. For the sake of brevity, a more rigorous treatment and discussion of ‘design rationale’ is given for modelling the Shibboleth solution only.

9.3.1 Using Shibboleths

Technically, speaking a shibboleth is a tried and tested societal authentication mechanism used to determine whether a person belongs to a certain social group. Shibboleths are phrases whose true pronunciation will only be known to a specified social group, and have been used throughout history in times of war to test persons of interest. Some well known examples from the Second World War are: *Schrijver van Scheveningen*—used by the Dutch Resistance; *Visser van Vlissingen*—another phrase used by Dutch Resistance; and *Lollapalooza*—used by US Soldiers in the Pacific Theatre. These phrases were chosen due to the difficulty in pronunciation that non-native speakers and the enemy had. For example, in Dutch the syllable ‘Sch’ has a very distinctive sound that only a fluent/native Dutch speaker will be able to reproduce. In the Pacific Theatre most speakers of an East Asian language will have difficulty in pronouncing the letter ‘L’. Upon encountering a person who claims to be from a certain social group, the person can be challenged to pronounce the group’s shibboleth. If the person cannot pronounce the shibboleth correctly it can be argued that the person is not an authentic member of the social group, and should be ‘dealt with’.

To model shibboleths in SIF, the properties of the solution must first be identified together with their *traits*. For this example two properties were identified representing the ‘proof’ and ‘authentication procedure’. These properties are explained below, together with how they are translated into a SIF solution specification. Listing 9.4 presents the resulting SIF model.

Modeling the Property ‘Trapdoor Pronunciation of a Known Phrase’

The first property models that a shibboleth is a difficult phrase to pronounce unless you are aware of its *true* pronunciation. Properties are titled declarations that contains a set of traits.

Property "Trapdoor Pronunciation of a Known Phrase" { ... }

For this property two traits can be identified. The first trait details the inherent ‘proof’ aspect of shibboleths, depending on the culture and phrased used, knowledge of how to pronounce the phrase may be more wide spread than originally thought. This can be translated into SIF as:

```
Trait "Known to a select few" is WeakSatis {  
  Affects { SomePos proof, SomePos authStep }}
```

Each trait will contain an **Affects** section that lists a set of affects as a pairing between the impact of the trait on a requirement from the problem specification. The impact values are sourced from the GRL.

Shibboleths are phrases that only a few should know how to pronounce. However, use of a well-known phrase increases the likelihood that it will be learned by those outside the social group: *Impostors*. Thus, the trait is ‘weakly satisfied’ due to the weakness associated with the possibility of a poorly chosen well-known phrase. Further, the trait will have minimal impact upon the *proof* and *authStep* requirements identified in Listing 9.3 because of this weakness.

The second trait details the *testing* and *proof* aspects of shibboleths when used for authentication. The ability to pronounce the phrase is both the *proof*, and also the *act of authentication*. The resulting SIF representation for this property is:

```
Trait "Pronunciation is the Proof" is Satisfied {  
  Affects { Makes proof, Helps authStep }}
```

When challenged, the ability to pronounce the shibboleth is the *proof* that the person is authentic. The impact upon the requirements for authentication are that: (a) the proof requirement *is made* (i.e. is satisfied) through existence; and (b) this ‘proof’ helps towards but does not satisfy the authentication step requirement—*authStep*. Thus, this aspect is ‘satisfied’ as shibboleths as proofs is a sound idea; shibboleths have been used as authentication mechanisms. For the authentication step to be satisfied shibboleths need to be part of an organised challenge response mechanism. The focus of the next property.

Modelling the Property ‘Shibboleths are a Challenge-Response Mechanism’

Knowledge of a shibboleth on its own does not constitute an authentication mechanism. The chosen phrase must be used in a challenge-response context. This property describes the use of shibboleths for authentication. As such identified traits will impact the requirements dealing with the operation and implementation of an authentication mechanism. The property is modelled as:

Property "Shibboleths are a C-R Mechanism" {...}

For this property, two traits were identified. The first recognises that *Challenge Response is an Authentication Mechanism*. Challenge-response mechanisms are a known authentication procedure in which upon request, an entity must provide proof of authenticity. The act of challenge-response is also a repeatable procedure and this reproducibility can help but not guarantee consistency of authentication; the complexity of the procedure ultimately determines its consistency. Related is the idea of a lockout. Detection of an incorrect response is deterministic, however, it is not clear how many attempts a person will have in reproducing the shibboleth before being locked out. Translating this trait into SIF gives:

Trait "Challenge Response is Authentication" is Satisfied {
Affects { **Makes** authStep, **Helps** consistent
, **Helps** timely, **Helps** lockout}}

The next trait describes how entities learn and use shibboleths: *Learning and Using Shibboleths*. To participate in the authentication procedure, entities must first learn the shibboleth. New phrases can be acquired, and their difficulty will impact the ability of an entity to learn the phrase.

Trait "Learning and Using Shibboleths" is Satisfied {
Affects { **Makes** enrollment, **SomePos** consistent
, **Helps** changeable, **SomePos** effortless}}

The trait ‘Learning and Using Shibboleths’ satisfies the enrollment requirement by virtue of the fact that entities must learn the shibboleth. However, entities may forget or make a mistake pronouncing the shibboleth when questioned. This ultimately will affect the strength of impact on the consistent requirement: It helps but does not make the requirement.


```
1 sif solution
2
3 Solution "Shibboleths" solves authentication in socio {
4
5 Property "Trapdoor Pronunciation of a Known Phrase" {
6   Trait "Known to a select few" is WeakSatis {
7     Affects { SomePos proof, SomePos authStep}}
8
9   Trait "Pronunciation is the Proof" is Satisfied {
10    Affects { Makes proof, Helps authStep}}
11
12 Property "Shibboleths are a C-R Mechanism" {
13   Trait "Learning and Using Shibboleths" is Satisfied {
14     Affects { Helps enrollment, SomePos consistent
15               , Helps changable, SomePos effortless}}
16
17   Trait "Act of C-R" is Satisfied {
18     Affects { Makes authStep, Helps consistent
19               , Helps timely, Helps lockout}}}}
```

Listing 9.4: Complete solution specification for using ‘Shibboleths’ for authentication.

Complete Specification

With the properties and traits of the solution modelled. The complete specification can be brought together. The complete specification is given in Listing 9.4, collecting the descriptions detailed earlier in this section.

A solution specification, like a problem file, begins with a declaration that is used by the evaluator to recognise solution files. This is Line 1. Line 3, is the actual solution declaration. Each solution is given a title and the identifier of the problem it solves together with the identifier of the context in which the problem and solution operate.

The evaluation of this model is discussed in §9.6. The next part of this tutorial will detail an alternative solution for authentication using identification cards that are used in a socio-technical context rather than a purely social context.

9.3.2 Using Identification Cards

Shibboleths are a purely societal solution to authentication. The next solution to authentication relies on both human interaction and technology to address problems of authentication. *ID Cards* are ‘issued’ documents that attest to the identity of the holder. Many organisations and governmental bodies issue members with identification

documents. Students at a university are given student cards, and employees in an organisation are given staff cards. Upon enrolment with the issuing body, entities have their identity asserted and an *Identity Card* is produced that represents the assurance that the governing body attests to the identity of the entity in question. The cards issued follow a standardised design unique to the governing body, and will display salient details about the card holder.

Many RFID cards are also ‘active’ and respond to interactions with technology. For instance, RFID-enabled Smart Cards have been used as the basis for ID Card technology. The *burdon of proof* in ID Card systems, is the knowledge that only the issuing body can reasonably attest and physically construct the cards. After enrolment entities that possesses these cards can have their identities attested by others, often by technological means. However, cards are physical tokens and can be lost, stolen, forged, or hacked. These are known disadvantages of ID Cards.

From the description given so far a set of properties and traits of those properties can be identified. Listing 9.5 presents the full listing of the SIF model solution for *ID Cards*, together with supporting documentation illustrating how a more comprehensive SIF solution file can look. Not all traits are neutral in their affect. For example, as ID Cards can be stolen or hacked, these traits have been marked as being disadvantageous and as such have a negative affect on the solution. Regardless of whether marking traits as being positive (**Advantage**), neutral (**Trait**), or negative (**Disadvantage**), the satisfaction value details the veracity of the trait. How a disadvantageous trait affects the evaluation of the model is detailed in Chapter 5.

9.4 Model Evaluation

With the modelling files for the two patterns complete, the next stage in the engineering process is to evaluate the patterns. Evaluation will take a problem solution pairing and determine how well each of the problem requirements are satisfied by the presented solution. Table 9.2 summarises the results of running the SIF evaluator on both models. The results show how both the presented *solutions* weakly satisfy the problem of authentication. However, different levels of satisfaction are presented for each requirement. Both shibboleths and ID Cards satisfy the requirements of *Proof of Authenticity* and *Authentication Mechanism*, yet ID Cards presents more satisfied requirements. This is due to the effect that the technology property has on the requirements.

```

1  sif solution
2
3  Description ""ID Cards are physical tokens handed out by
4  a governing authority and are used by people to verify
5  their identity.""
6
7  > ID Cards are physical tokens handed out by a governing
8  > authority. Users must enroll with the body to be issued
9  > with an ID Cards.
10 Solution "ID Cards" solves authentication in sociotech {
11
12 > The ID Cards is a token carried by people.
13 Property "ID Cards is a Token" {
14 > Only authorised bodies can construct cards.
15 Trait "Assignment of Cards" is Satisfied {
16 Affects { Makes enrollment, Makes proof }}
17
18 > Tokens can be stolen/lost.
19 Disadvantage "Stolen/Lost Tokens" is WeakSatis {
20 Affects { Hurts consistent }}
21
22 > Specific tokens can have a validity period.
23 Advantage "Validity Period" is Satisfied {
24 Affects { Helps consistent, Helps changeable }}}
25
26 > Technology is Used.
27 Property "Technology" {
28 > Cards can be made electronic.
29 Trait "Machine Readable" is Satisfied {
30 Affects { Makes consistent, Makes timely
31           , Makes lockout, Makes effortless
32
33 > Technology is not always right.
34 Disadvantage "Hackable" is WeakSatis {
35 Affects { Breaks consistent, Breaks timely, Breaks lockout}}}
36
37 > ID Cards are a standardised means of identification that are
38 > applied across an organisation and administered under a
39 > single governing body's remit.
40 Property "Governing Body Issued" {
41 Trait "Standardised form of Identification" is Satisfied {
42 Affects { Makes enrollment, SomePos changeable
43           , SomeNeg effortless }}
44
45 Trait "Standarised Mechanism" is Satisfied {
46 Affects { Makes authStep }}}}
```

Listing 9.5: Complete solution specification for using 'ID Cards' for authentication.

Requirement	Shibboleths	ID Cards
Enrolment	wSatisfied	Satisfied
Proof of Authenticity	Satisfied	Satisfied
Authentication Mechanism	Satisfied	Satisfied
Changeable	wSatisfied	wSatisfied
Authentication happens	wSatisfied	Satisfied
Timely Authentication	wSatisfied	Satisfied
Limited access attempts	wSatisfied	Satisfied
Effortless Authentication	wSatisfied	wSatisfied
Authentication	wSatisfied	wSatisfied

Table 9.2: SIF evaluation results for AUTHENTICATION THROUGH SHIBBOLETHS & AUTHENTICATION THROUGH ID CARDS.

When working with the evaluator, only three artefacts are presented. A *single* solution file, together with the two solution files. Further, arbitrary pairings of files cannot be presented to the evaluator. Had the presented files not matched in the problem and solution and context pairing, the evaluator would have refused to run.

Note. The two models presented are for solutions in different contexts and that, as with any modelling, the strength of the results relies on the detail expressed within the model.

9.5 Writing Patterns

The next stage in the pattern engineering process is the construction of pattern documents that describe the patterns in more detail. As well as the generation of several known flavours of markdown languages, the SIF evaluator also facilitates the construction of FREYJA encoded pattern stubs. The FREYJA pattern template can be used for creating active document specifications—see Chapter 6. When stubs are generated using SIF, the evaluator will use the documentation in the SIF model files to fill in as much of the document as possible. This leaves the pattern writer to literally *fill in the gaps*.

For example, the AUTHENTICATION THROUGH SHIBBOLETHS was modelled in the previous section. Listing 9.6 gives the XML fragment corresponding to the problem

```
...
<problem>
  <name>Authentication</name>
  <description>How to authenticate entities such that
    assurances can be made of
their identity.</description>
  <requirements>
    <usability id="8">
      <name>Effortless Authentication</name>
      <description>To Be Added</description>
    </usability>
    <functional id="7">
      <name>Able to set limited access attempts</name>
      <description>To Be Added</description>
    </functional>
    <performance id="6">
      <name>Timely Authentication</name>
      <description>To Be Added.</description>
    </performance>
    <reliability id="5">
      <name>Authentication happens</name>
      <description>To Be Added</description>
    </reliability>
    <functional id="4">
      <name> Changable </name>
      <description>To Be Added</description>
    </functional>
    <functional id="3">
      <name>Authentication Mechanism</name>
      <description>To Be Added</description>
    </functional>
    <functional id="2">
      <name>Proof of Authenticity</name>
      <description>To Be Added</description>
    </functional>
    <functional id="1">
      <name>Enrolment /name>
      <description>To Be Added</description>
    </functional>
  </requirements>
</problem>
...
```

Listing 9.6: The problem description from the FREYJA stub generated from the SIF model for the pattern: AUTHENTICATION THROUGH SHIBBOLETHS

of authentication as described in Listing 9.3. Note how the name and documentation from the SIF model have been serialised into the XML encoding, with a *key phrase* of `To Be Added` to denote the missing content.

Listing 9.7 provides a similar partial snippet used as part of the solution description. Listing 9.4 presents the corresponding SIF model. Patterns are more than just the abstract concepts described in the SIF model. The FREYJA encoding provides pattern writers with an explicit means to provide more information over the structure and dynamics of the presented solution using the XML tag `model`. For example, Listing 9.8 presents a possible UML component model to depict how one possible structural description can be given in the XML file. The textual UML notation from PLANTUML¹ was used to give a more human readable serialisation of the component model. Other representations are permissible.

Although, XML is a machine-readable serialisation format, the resulting XML encoded files can be hard to read. The FRIGG tool presented in this thesis allows, among other features, a means to provide more human-understandable views of a FREYJA encoded file. Listing 9.9 details how the problem description encoded in XML (Listing 9.6) can be viewed using the `Org-Mode` markdown syntax.

Appendix A presents the finalised written forms for both `AUTHENTICATION THROUGH SHIBBOLETHS` and `AUTHENTICATION THROUGH ID CARDS`. The next section describes how the resulting patterns are evaluated for quality. To illustrate how the evaluation detects good and bad qualitative aspects of a pattern, the `AUTHENTICATION THROUGH ID CARDS` pattern was written in a purposefully bad style.

9.6 Evaluating the Pattern

The next stage in the engineering process is pattern evaluation. To evaluate patterns the PREMES evaluation framework was developed—see Chapter 8. *Pattern Report Cards* is a tailorable evaluation system that allows for a generic evaluation framework to be tailored for a group of like patterns. This framework is based upon the *Plan-Do-Check-Act* management cycle. The FRIGG tool (Chapter 7) has been designed for working with pattern documents to aid in report card generation. This section will not explicitly report all aspects of the evaluation process but concentrate on reporting how the framework was used to conduct the evaluation itself.

¹<http://www.plantuml.org>

```
1 <solution>
2   <name> Shibboleths </name>
3   <description>To Be Added</description>
4   <models>
5     <dynamic modelTy="unknown">
6       <name> Example Dynamic </name>
7       <description> To Be Added</description>
8       <model><![CDATA[
9         "MODEL inserted here"
10      ]]></model>
11     </dynamic>
12     <structure modelTy="unknown">
13       <name>Example Structure</name>
14       <description>To Be Added</description>
15       <model><![CDATA[
16         "MODEL inserted here"
17      ]]></model>
18     </structure>
19   </models>
20   <properties>
21     <property>
22       <name> Shibboleths are a C-R Mechanism </name>
23       <description>To Be Added</description>
24       <traits>
25         <general svalue="SATISFIED">
26           <name> Act of C-R </name>
27           <description> The C-R mechanism is the authentication step
28             .</description>
29           <affects>
30             <affect cvalue="HELPS" id="7">clearly defined process,
31               but confusion over which phrase used.</affect>
32             <affect cvalue="HELPS" id="6">parties will now the
33               result immediately.</affect>
34             <affect cvalue="HELPS" id="5">predefined phrases are
35               learned.</affect>
36             <affect cvalue="MAKES" id="3">very nature the C-R step
37               is the authentication step.</affect>
38           </affects>
39         </general>
40       </traits>
41     </property>
42   </properties>
43 </solution>
```

Listing 9.7: An extract from the solution description from the FREYJA stub generated from the SIF model for the pattern: AUTHENTICATION THROUGH SHIBBOLETHS.

```

1 <structure modelTy="uml-component">
2   <name> Example Structure </name>
3   <model><![CDATA[
4     component Subject
5     component Authenticator
6     component Enroller
7
8     interface "Enrollement" as enrol
9     interface "Authenticate" as auth
10
11     Authenticator - auth : provides
12     Enroller      - enrol : provides
13
14     Subject ..> enrol : requests
15     Subject ..> auth  : requests
16   ]]></model>
17   <description> Within the shibboleth setup there are three main
18     components. The subject requesting access. The
19     authenticator that performs the authenticity check. An
20     enrolment component that enrols a subject into the system.
21   </description>
22 </structure>

```

Listing 9.8: Example model given in PLANTUML notation depicting the structure of the solution from the AUTHENTICATION THROUGH SHIBBOLETHS.

9.6.1 The Planning Stage

For analysing the two patterns the following decisions were made.

Quality of Pattern Presented For these patterns, the first part of the evaluation does not require decisions to be made before execution of the evaluation process.

Quality of Solution Presented To calculate the *weighted solution satisfaction* metric (see Chapter 8 §8.3.2) each requirement shall be treated equally and given the same weighting i.e. each of the eight requirements has a weighting of 0.125. The grading scheme used is based upon the satisfaction values from the GRL and presents an example mapping of the qualitative satisfaction values to quantified values. The mappings are given in Table 9.4. The SIF evaluation data will be used to assign a grade to each of the requirements. For assessment of solution complexity, the grading scheme detailed in Chapter 8 §8.4.2 will be used with an extra grade descriptor of ‘incomplete’. Assessment of solution effectiveness shall be from a naïve assessment that looks for evidence of effectiveness.


```
1 frigg> :display problem
2 * Problem: Authentication
3 How to authenticate entities such that assurances can be made
4   of
5 their identity.
6 ** Requirements
7 ** FUNC: Able to set limited access attempts
8 Entities that repeatedly try to authenticate with incorrect
9   data should be
10  locked out of the system for some predefined period of time.
11 ** FUNC: Changable
12 The authentication step must be changable and re-enrollment
13   possible.
14 ** FUNC: Authentication Mechanism
15 Solutions must consist of an authentication step.
16 ** FUNC: Proof of Authenticity
17 A proof is required that will attest to the authenticity of
18   an entity.
19 ** FUNC: Enrolment
20 Solutions must have a means to enroll entities into the
21   authentication
22   procedure.
23 ** USAB: Effortless Authentication
24 The authentication process should not require undue effort to
25   do.
26 ** RELI: Authentication happens
27 Authentication must be consistent and that entities with up-
28   to-date
29 authentication details should authentication.
30 ** PERF: Timely Authentication
31 Processing authentication requests must not take forever, and
32   be timely
33 in their duration.
```

Listing 9.9: Using FRIGG to provide a more human-understandable view of the authentication problem encoded in Listing 9.6.

Quality of Pattern Presentation For calculating the weighted adherence to a pattern template it was decided that the classic Alexandrian template, detailed in Chapter 2 §2.6.1, will be used. For this evaluation, the heading named *Implementation* is replaced with *Evidence*, and an additional heading *Case Studies* added. Each heading will be treated equally, thus for the eight headings in the template the weightings will be 0.125. The grading scheme used will be the same one used to calculate *weighted solution satisfaction*. Pattern Legibility shall be assessed using the Flesch-Kincaid readability metric [Kin+75].

Denied	Conflict	wDenied	Unknown	wSatisfied	Satisfied
0.0	0.0	0.25	0.5	0.75	1.0

Table 9.4: Mappings from GRL satisfaction values to quantitative values used for pattern evaluation.

9.6.2 Grading the Pattern

Chapter 8 details how the qualitative indicators are to be scored, together with the various grade descriptors used. This section details how the FRIGG tool can be used to aid in the grading process. Formative comments will not be reported in this tutorial. Table 9.6 presents the resulting report cards for both patterns evaluated.

Indicator	Shibboleths	ID Cards
Coherency Grade	A	A
Atomcity Grade	B	B
Problem Independence	A	A
Solution Appropriateness	0.81	0.94
Solution Complexity	Adequate	Incomplete
Solution Effectiveness	D	B
Pattern Structure	162.5	37.5
Pattern Legibility	7.71	6.36
Presentation Accessibility	B	D

Table 9.6: Report Cards Grades for the AUTHENTICATION THROUGH SHIBBOLETHS and AUTHENTICATION THROUGH ID CARDS patterns.

```
frigg> :evaluate sif
WEAKSATIS ==> Requirement: Timely Authentication
WEAKSATIS ==> Requirement: Authentication happens
WEAKSATIS ==> Requirement: Effortless Authentication
WEAKSATIS ==> Requirement: Enrollment
SATISFIED ==> Requirement: Proof of Authenticity
SATISFIED ==> Requirement: Authentication Mechanism
WEAKSATIS ==> Requirement: Changeable
WEAKSATIS ==> Requirement: Able to set limited access
    attempts
WEAKSATIS ==> Problem: Authentication
```

Listing 9.10: Re-Calculating the SIF Evaluation Result for the AUTHENTICATION THROUGH SHIBBOLETHS pattern.

Solution Appropriateness The satisfaction results for each pattern were presented earlier in Table 9.2. These results are not stored in the FREYJA encoding. The purpose of the SIF evaluator is to calculate these values. However, the FRIGG tool knows about SIF models and can recover the model from a FREYJA encoded pattern document. At the moment FRIGG can be used to display the results of the SIF evaluator. Listing 9.10 presents an example listing illustrating this for AUTHENTICATION THROUGH SHIBBOLETHS. These values are then used in-conjunction with the value mappings in Table 9.4 to calculate the scores. Although, the calculations must be performed by hand, FRIGG can be extended to do the calculations automatically.

Pattern Structure The FRIGG tool can also be used to calculate the *Weighted-Template Adherence* metric automatically from the pattern document. Each heading/tag in the document can be given a qualitatively described score using the score attributed of the specification format. For example, <problem score="A">. Feeding the tool with a set of grade mappings that detail the template structure and weightings, FRIGG can automatically calculate the resulting value for template adherence. Listing 9.11 presents an example listing illustrating this for AUTHENTICATION THROUGH SHIBBOLETHS.

```
frigg> :evaluate template
162.5
```

Listing 9.11: Calculating Weighted-Template Adherence for the AUTHENTICATION THROUGH SHIBBOLETHS using FRIGG.

Pattern Legibility Readability metrics can be calculated automatically. FRIGG has support for calculating the values for a variety of known readability metrics. Listing 9.12 demonstrates how this is achieved using the FRIGG tool, and from which the required readability metric can be selected.

```
frigg> :evaluate read
(FLESCH, 59.54554252199415)
(ARI, 7.248867302052787)
(KINCAID, 6.354149560117303)
(COLEMAN, 11.21700268817204)
(FOG, 4.996480938416423)
(SMOG, 6.404542420195273)
```

Listing 9.12: Calculating the readability scores for AUTHENTICATION THROUGH ID CARDS using FRIGG.

Remaining Descriptors The remaining aspects require subjective assessment, and cannot be automated. The resulting grades for the remaining indicators were chosen based upon the descriptions given in the grade descriptors from Chapter 8.

9.6.3 Analysing the Results

With the report cards constructed and summarised, the next stage is to analyse the results. Recall, that the pattern AUTHENTICATION THROUGH SHIBBOLETHS was written in a good style, and the pattern AUTHENTICATION THROUGH ID CARDS in a bad style. This is evident in the report-card, where for the indicator *Pattern Structure* Shibboleths scored higher. This does not mean that Shibboleths are better. Analysis of the results for *Solution Effectiveness* and score for *Solution Appropriateness* show that *ID Cards* score higher than Shibboleths and are ‘better’ than shibboleths in these areas.

Further for both patterns, the report card also identifies that the pattern’s problem, solution, and context are coherent. Such coherency could be the result of modelling the pattern using SIF, allowing for patterns to be described in a manner more conducive for respecting the problem×solution×context triple. However, this thesis does not provide evidence to support this conclusion.

The report card can be used to produce a report providing detailed formative feedback on the deficiencies presented in the pattern along side the already reported

summative feedback. The indicators in the report card can be used to situate the feedback better. Further, the presented report can be concluded with a set of prioritised, actionable, items that provides advice on which areas are required for improvement.

9.6.4 Results Reporting

With the evaluation complete, the next stage is reporting and analysis of the results document. With such a document comes precise feedback on the patterns deficiencies and successes, allowing for improvements to the pattern to be targeted. With successive iterations of the evaluation process, previous results documents can be compared to track the improvement of the pattern.

9.7 Pattern Publication

The final stage in Software Design Pattern Engineering is that of pattern publication. §9.5 already demonstrated how FRIGG can be used to view FREYJA encoded pattern documents, an example of which was provided in Listing 9.6. FRIGG can also be used to convert the templates into other document-markup formats, for example \LaTeX , Markdown, and Org-Mode. Appendix A contains the result of converting the patterns in this chapter to the different supported formats. Through automatic generation of the pattern document to multiple documentation formats comes ease of distribution of the pattern to different media.

9.8 Summary

This chapter presented a tutorial demonstrating how the various contributions in this thesis are used for pattern engineering. Specifically, the contributions were used to *engineer* two example patterns for the problem of authentication: AUTHENTICATION USING SHIBBOLETHS and AUTHENTICATION USING ID CARDS. SIF was used during the creation phase to guide development of and machine check a requirements model describing the pattern. The problem description was shared between the two models. FREYJA facilitated the encoding of the pattern document and requirements model in a machine readable and checkable format. The resulting pattern document and its encoded information was used by FRIGG during the evaluation process. Here

FRIGG was used to machine read and check the documents. Combination of these tools can lead to an enhanced creation phase for pattern engineering.

NovoGRL: RE-TARGETING THE GRL FOR NEW DOMAINS

The GRL is an existing GOML that presents a visual modelling language—Chapter 3 §3.2. However, when using a such a language with visual language constructs, a more formal language-driven treatment for DSML creation is harder to produce. This arises due to the disconnect between the notation used for the DSML, and its formalisation and transformation into a host language model instance.

This chapter introduces NovoGRL a language oriented re-engineering of a subset of the GRL. Specifically, this framework facilitates the provisioning of new semantic overlays allowing for the language to be re-targeted for new modelling domains. This framework is used for the construction of DSMLs that are structurally equivalent to the GRL but provide different semantic constructs.

In addition to the description of NovoGRL, two DSMLs are presented as case studies to illustrate how the original GRL can be re-targeted for different domains. The DSMLs presented are: (a) a replication of the GRL itself; and (b) a language used for modelling academic paper writing. NovoGRL and the languages described in this chapter are available online [dMH15e]. Further, the DSML that replicates the GRL was used as the meta-modelling language to which SIF model instances are interpreted to.

10.1 Making the GRL a Language

Goal Graphs are a formal model for describing goal models as directed graphs where each node represents a goal, and the edges the links between each goal [Gio+03]. Giorgini et al. [Gio+03] provide a formal definition for these goal-graphs.

Definition 12 (Goal Graph). *Formally a Goal-Graph is a directed graph described as a pairing $\langle \mathcal{G}, \mathcal{R} \rangle$, where \mathcal{G} is a set of goals, and \mathcal{R} is a set of goal relations over \mathcal{G} . Given the relation $(\{g_1, \dots, g_n\}) \xrightarrow{r} g$ in \mathcal{R} where $g_i \in \mathcal{G}$. Nodes to the left of the relation are described as source goals, and the goal on the right is the target goal.*

Using these goal-graphs, Amyot et al. [Amy+10] derived a series of evaluation techniques to calculate model satisfaction.

The GRL is a graphical modelling language using visual cues to illustrate concepts, and fixates upon the idea that modellers will use a visual means to both construct, view, and reason about their *goal-graphs*. However, these visual cues and language constructs are harder to reason about formally and also mechanically [MCo1]. Furthermore, when using visual modelling languages, the creation of a DSML must be performed through adhoc methods unless the modelling language's tooling explicitly supports DSML creation. When working with GOMLs, the language syntax and type-system are presented visually and their representation in code buried alongside the evaluation semantics in the tooling.

To address these issues one has to first think of the graphical notation used as an alternative view of the model. Programming languages are comprised of: an abstract syntax to describe language expressions; a type-system to ensure the creation of well-formed expressions; and semantics to ensure the correct execution of expressions. These notions are linked together in the tooling provided for the language. Chapter 4 introduced the idea of the *Well-Typed Interpreter* and how a DSL can be modelled as an EDSL within Idris, providing static compile-time *correctness by construction* guarantees over the program and language implementation. This approach illustrated how the language's syntax, typing rules, and interpretation can be formally modelled and implemented within the same system. Further, the Idris language has built in support for tactic style theorem proving and auto implicit argument construction. Use of these abilities further enables compile time checks to be performed for properties of the language that enhances the *correctness by construction* guarantees made. With a language-based approach to DSML construction, the GRL can be re-envisaged as a

declarative language (in the style of the *Well Typed Interpreter* [AC99]) from which *goal-graphs* are constructed.

The interest in this thesis, however, is not with the (re)construction of the GRL *per se* but rather its use as a meta-modelling language for constructing DSMLs, and DSMLs primarily for reasoning about design patterns. Work by Brady and Hammond [BHo6] has shown how a verified staged interpreter can be written in a dependently typed language to translate concepts between domains (i.e. languages) *and* provide *correctness by construction* guarantees for these translations. Such an approach allows for the structural equivalence between the two languages to be shown more formally, and more importantly enforced during implementation. This approach can be combined with the *Well-Typed Interpreter* approach to allow for a more formal description of the re-domaining of the GRL to occur if the interpretation between domains can be specified.

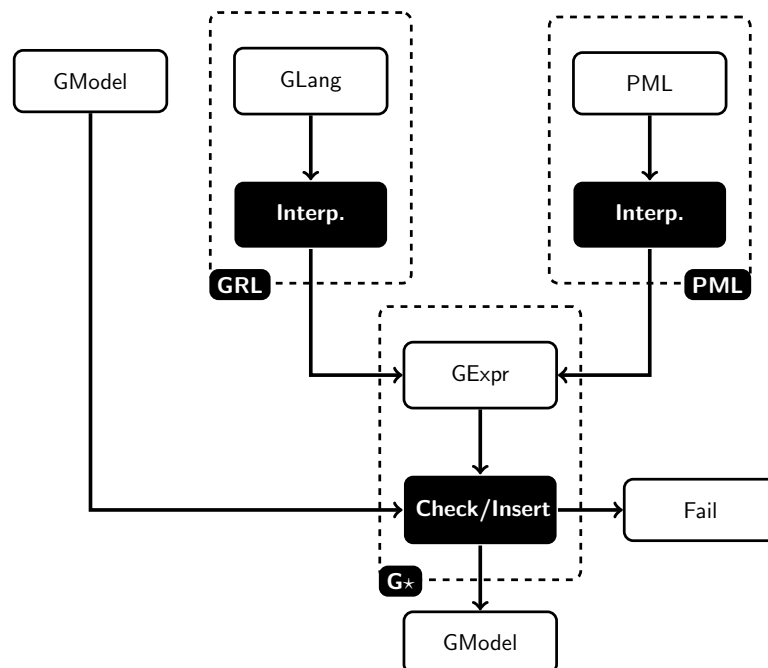


Figure 10.1: Schematic illustrating how DSML relate to, and are interpreted into, No-vogRL goal models.

However, this approach requires that for every DSML constructed, a bespoke interpreter must also be constructed. To reduce the effort in constructing multiple DSMLs, a two-stage interpreter is used. At different stages of interpretation different

properties on the resulting intermediate forms can be checked. Figure 10.1 illustrates this two stage process.

DSMLs are first translated into an intermediate representation that captures language constructs common to all DSMLs: GEXPR. This transformation is achieved using transformation functions unique to the DSML. The resulting intermediate representation provides the instructions for constructing the GRL goal-graph using an internal modelling language: G*. A goal-graph constructed using G* is evaluated for satisfaction using known techniques [Amy+10]. Interpreting down to a common representation, allows for the construction of goal-graphs to be reasoned about regardless of the DSMLs original language constructs.

Here DSMLs are defined as a variation of the GRL that uses structurally similar language constructs to the GRL but with semantic constructs unique to the domain of interest. The languages are structurally isomorphic, but semantically dimorphic.. Rather than modelling and constructing a separate evaluator for the DSML, interpretation semantics can be presented that map how the constructs from the DSML are translated into those from the GRL.

Note. For this initial attempt a subset of the GRL was chosen to be implemented. The framework does not support the GRL concepts of actors, and dependencies between two actors. The evaluation strategy supported is qualitative, quantitative evaluations are not supported.

10.2 GRL-Derived Goal-Graphs

NovoGRL is concerned with the construction of GRL-derived goal-graphs. This section presents a top-down formal definition of the goal-graphs that underpin all GRL models within this framework. Further, this section makes explicit several language design decisions made in the GRL model, and w.r.t. to the implementation of the goal-graphs. How goal-graphs are constructed is detailed in §10.3, and how *correct by construction* guarantees are made in §10.4. The next section details the language G* that enables the construction of the goal-graphs as described in this section.

The *User Requirements Notation* [UTN12] details the official GRL language specification in which the GRL is presented as a UML meta-model. This meta-model effectively details the syntactical structure of GRL models only. There is little detail over correctness properties w.r.t. to model instances. Although the GRL authors detail that

their modelling tool supports user-defined semantic rules [Amy+10, § 2], important semantic rules common to all GRL models are not documented sufficiently. For example, it is not clear whether introduction of a cyclic dependency using decomposition links should be a valid, however, specified in the same document is the restriction that intentional links cannot affect resource nodes. López et al. [LFM11] identified a similar problem in the i^* goal-modelling language. During implementation of NovoGRL several of these implicit assumptions were required to be made explicit.

10.2.1 Definitions

§10.1 already noted that goal models are directed graphs in which goals are nodes, and their relations encoded as vertices. For GRL instances this is illustrated implicitly in the UML meta-model that details the GRL syntax. However, Definition 12 is not suitable as it does not encompass the language details specific to the GRL.

GRL Goal Graph

First the goal-graph definition is given, this definition resembles Definition 12 but is less compact. The subsequent definitions will detail more of the required definitions.

Definition 13 (GRL Goal-Graph). *Let $gs = \{g_1, \dots, g_n\}$ be a set of goals, where $g_i \in \mathcal{N}$, and $ls = \{l_1, \dots, l_m\}$ be a set of relations operating over the nodes in gs and where $l_i \in \mathcal{L}$. A GRL goal-graph M is defined as the tuple $\langle gs, ls \rangle$ of goals gs and relations ls . We let \mathcal{M} denote the domain given to all GRL model instances.*

The goal-graph is implemented as a graph data structure implemented in the adjacency list style using a Binary Search Tree¹. The definition of `GModel` is given as a type alias for a graph with vertices of type `GoalNode`, and edges of type `GoalEdge`.

```
GModel : Type
GModel = Graph (GoalNode) (GoalEdge)
```

Goal Nodes

Goal nodes are defined as a tuple that collects the type of node in the graph, a satisfaction value, and a unique identifier.

¹de Muijnck-Hughes [dMH15b] details the precise implementation.

Definition 14 (Goal Nodes). *Given a node type $e \in \mathcal{N}'$, a title t , and a satisfaction value $q \in \mathcal{Q}$. A goal node g is defined as the tuple $g = \langle e, t, q \rangle$. We let \mathcal{N} represent the domain given to all nodes.*

Goal nodes are implemented as a record that details a node's type e , a title t , a possible satisfaction value q , and whether or not the node is linked to other nodes with structural links through storing of the link type as the s .

```
record GoalNode where
  constructor GNode
  getNodeTy      : GElemTy
  getNodeTitle   : String
  getSValue      : Maybe SValue
  getStructTy    : Maybe GStructTy
```

This implementation differs from the formal definition. By definition, node decomposition stipulates that a node can have only one type of decomposition. This is an emergent property of the graph data structure, thus representation of the decomposition type in the record itself facilitates a compact and efficient means to determine if the node has a decomposition value.

Goal Edges

Within the GRL there are two types of links between goals: structural and intentional. These can be represented directly in the graph as directed edges between two nodes.

Definition 15 (Intentional Link). *Let $x, y \in \mathcal{N}$ be two nodes. Given a contribution value $c \in \mathcal{C}$ that indicates the effect that x has on y , an intentional relation between x and y is defined as the relation: $x \circ_i^c y$. Where $i \in \{\text{contrib}, \text{correl}\}$ indicates the type of intentional link: contribution or correlation.*

Note. The GRL syntax declares an additional intentional relation of *means-end* between a resource nodes and non-resource nodes. These relations are special cases of contribution links, and thus need not need be considered in the formal model.

Definition 16 (Structural Links). *Let $x \in \mathcal{N}$ be a goal, and $ys = \{y_1, \dots, y_n\} \subset \mathcal{N}$ be the sub-goals that a node x decomposes into. Given a decomposition type $d \in \{\wedge, \vee, \oplus\}$, a decomposition relation between x and ys is defined as: $x \circ_d ys$*

With the goal-graph, edges are defined as either a structural or intentional link.

Definition 17 (Goal Edges). *Let $\mathcal{L} = \{l_1, \dots, l_n\}$ be the set of all goal edges, where each $l_i \in \mathcal{L}$ is defined over goals from \mathcal{N} , and is either an intentional link or a structural link.*

Edges are implemented as an ADT with constructors for an intentional link together with its contribution value, or a structural link.

```
data GoalEdge : Type where
  Contribution : CValue -> GoalEdge
  Correlation   : CValue -> GoalEdge
  Decomp       : GoalEdge
```

Values and Weights

These final set of definitions detail miscellaneous values and weights used within the goal-graph representation.

Definition 18 (Node Types). *Let the types representing the different kinds of nodes in a goal-graph be:*

$$\mathcal{N}' = \{\text{GOALty}, \text{SOFTty}, \text{TASKty}, \text{RESty}\}$$

Definition 19 (Contribution Values). *Let the set of allowed contribution values be:*

$$\mathcal{C} = \{\text{Make}, \text{Help}, \text{SomePos}, \text{Unknown}, \text{SomeNeg}, \text{Break}, \text{Hurt}\}$$

Definition 20 (Satisfaction Values). *Let the set of allowed satisfaction values be:*

$$\mathcal{Q} = \{\text{Denied}, \text{wDenied}, \text{wSatisfied}, \text{Satisfied}, \text{Conflict}, \text{Unknown}, \text{None}\}$$

Definition 21 (Decomposition Link Types). *The decomposition types of ‘and’, ‘or’, and ‘exclusive or’ are defined by the set:*

$$\{\wedge, \vee, \oplus\}$$

Definition 22 (Intentional Link Types). *The intentional link types of contribution and correlation are defined by the set:*

$$\{\text{contrib}, \text{correl}\}$$

These set definitions are implemented as enumerated ADTs.

```

data GElemTy  = GOALty | SOFTty | TASKty | RESTy
data GStructTy = ANDty  | XORty  | IORTy
data GIntentTy = IMPACTSty | AFFECTSty
data CValue   = MAKES    | HELPS    | SOMEPOS
                | UNKNOWN | SOMENEG | BREAK  | HURTS
data SValue   = DENIED   | WEAKDEN | WEAKSATIS
                | SATISFIED | CONFLICT | UNKNOWN
                | NONE     | UNDECIDED

```

10.2.2 Correctness of Edges

Several correctness properties associated with goal-graphs are now presented. These correctness properties allow for the structure and composition of goal-graphs to be reasoned about.

Well Formed Relations

This first set of properties detail the construction of relations when they are considered in isolation from a goal-graph. Valid intentional links are relations that declare that a goal x can affect other nodes aside from resource nodes and themselves. The formal definition is:

Definition 23 (Well-Formed Intentional Link). *Let $l = x \circ_i^c y$ be an intentional link, where $x, y \in \mathcal{N}$, $c \in \mathcal{C}$, and $i \in \{\text{contrib}, \text{correl}\}$. l is a valid intentional link if:*

$$\text{wellFormedIntent}(l) = \begin{cases} \text{True} & x \neq y \wedge \neg \text{isResource}(y) \\ \text{False} & x = y \vee \text{isResource}(y) \end{cases}$$

Valid decomposition links are relations that declare that a goal x can be linked to a set of unique nodes.

Definition 24 (Well-Formed Decomposition Link). *Let $s = x \circ_d \{y_1, \dots, y_n\}$ be a structural link where $x, y_i \in \mathcal{N}$, and $d \in \{\wedge, \vee, \oplus\}$. s is a valid decomposition link if:*

$$\text{wellFormedDecomp}(s) = \begin{cases} \text{True} & x \notin y_s \wedge \forall a, b \in y_s, a \neq b \\ \text{False} & x \in y_s \vee \exists a, b \in y_s, s = b \end{cases}$$

Valid Relations

We now define properties to describe the validity of a single relation w.r.t. to a goal-graph. Relations are valid in a goal-graph, if they are both well-formed and the nodes within the relation exist within the graph.

Definition 25 (Valid Intentional Link). *Given a goal-graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{N}$ and $ls = \{l_1, \dots, l_n\}, l_i \in \mathcal{L}$. Let $i \in ls = x \circ_{ty}^c y$ be an intentional link in M , where $c \in \mathcal{C}$, and $ty \in \{\text{contrib}, \text{correl}\}$. The intentional link i is valid in M if:*

$$\text{validIntent}(M, i) = \begin{cases} \text{True} & \text{wellFormedIntent}(i) \wedge x, y \in gs \\ \text{False} & \neg \text{wellFormedIntent}(i) \vee y \notin gs \vee x \notin gs \end{cases}$$

Definition 26 (Valid Decomposition Link). *Given a goal-graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{N}$ and $ls = \{l_1, \dots, l_n\}, l_i \in \mathcal{L}$. Let $s \in ls = x \circ_{ty}$ $\{y_1, \dots, y_n\}$ be a structural link in M , where $ty \in \{\wedge, \vee, \oplus\}$. The structural link s is valid in M if:*

$$\text{validDecomp}(M, s) = \begin{cases} \text{True} & \text{wellFormedDecomp}(s) \wedge x \in gs \wedge y_s \subset gs \\ \text{False} & \neg \text{wellFormedDecomp}(s) \vee x \notin gs \vee \exists y \in y_s, y \notin gs \end{cases}$$

10.2.3 Goal-Graph Properties

Next the correctness properties of goal-graph instances are considered. These properties are used in §10.2.4 to define goal-graph correctness.

Goal Uniqueness

Goal uniqueness is a property to ensure that each goal within the graph is unique.

Definition 27 (Goal Uniqueness). *Given a goal-graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{N}$ and $ls = \{l_1, \dots, l_n\}, l_i \in \mathcal{L}$. M has goal uniqueness if:*

$$\text{uniqueGoals}(M) = \begin{cases} \text{True} & \forall x, y \in gs, x \neq y \\ \text{False} & \exists x, y \in gs, x = y \end{cases}$$

Valid Intentional Link

This property defines the correctness of all intentional links respective to a given goal in the graph. This property is an extension to Definition 25 in which the set of intentional

links from a given goal g must all be valid, and that there can only be a single edge between g and any other given node.

Definition 28 (Valid Goal Intentional Link). *Given a goal-graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{N}$ and $ls = \{l_1, \dots, l_n\}, l_i \in \mathcal{L}$. Let $g \in gs$ be a goal node in M , and let $is = \{g \circ_{ty_1}^{c_1} y_1, \dots, g \circ_{ty_n}^{c_n} y_n\} \subset ls$ where $ty_i \in \{contrib, correl\}$ and $c_i \in \mathcal{C}$. is is the set of intentional relations that goal node g affects either as contribution or as a correlation. Let $ys = \{y_1, \dots, y_n\}$ be the set of target nodes from is that g affects. The goal node g has valid intentional links if:*

$$\text{validGoalIntent}(M, g) = \begin{cases} \text{True} & \forall i \in is, \text{validIntent}(M, i) \wedge \forall x, y \in ys, x \neq y \\ \text{False} & \exists i \in is, \neg \text{validIntent}(M, i) \vee \exists x, y \in ys, x = y \end{cases}$$

Strongly Valid Intentional Links

An optional and stronger correctness property for the intentional relations associated with a goal g in M is when the set is contains no duplicate edges. Formally this is given as:

Definition 29 (Strongly Valid Goal Intentional Link). *Given a goal-graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{N}$ and $ls = \{l_1, \dots, l_n\}, l_i \in \mathcal{L}$. Let $g \in gs$ be a goal node in M , and let $is = \{g \circ_{ty_1}^{c_1} y_1, \dots, g \circ_{ty_n}^{c_n} y_n\} \subset ls$ where $ty_i \in \{contrib, correl\}$ and $c \in \mathcal{C}$. If $\text{validGoalIntent}(M, g) = \text{True}$ then let $cs = \{c_1, \dots, c_n\}$ be the set of contribution values from is . The goal node g has strong valid intentional links if:*

$$\text{stronglyValidIntent}(M, g) = \begin{cases} \text{True} & \text{validIntent}(M, g) \wedge \forall x, y \in cs, x \neq y \\ \text{False} & \neg \text{validIntent}(M, g) \vee \exists x, y \in cs, x = y \end{cases}$$

Valid Node Decomposition

This next property defines the correctness of structural links in the goal-graph. Core to this property is the idea that nodes can be decomposed into other nodes, and that a node cannot be decomposed into itself neither directly nor through a descendant. That is, the structure of the decomposition constructs a sub-graph that is also a tree.

Valid structural relations are those that contain no cycles and each link emanating from a parent node to a child node must each have the same decomposition type. More formally:

Definition 30 (Valid Goal Decomposition). *Given a goal-graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{G}$ and $ls = \{l_1, \dots, l_n\}, l_i \in \mathcal{L}$. Let $g \in gs$ be a goal node in M , and let $ss \subset ls = \{g \circ_{ty_1} ys_1, \dots, g \circ_{ty_m} ys_m\}$ be the set of decomposition relations associated with g that have decomposition type ty_i . Further we let $ts = \{ty_1, \dots, ty_n\}$ be the set of decomposition types from ss . The goal node g has valid goal decomposition if:*

$$\text{validGoalDecomp}(M, g) = \begin{cases} \text{True} & \forall s \in ss, \text{validDecomp}(s) \wedge \\ & \forall ty_a, ty_b \in ts, ty_a = ty_b \wedge \\ & ys \subset gs \\ \text{False} & \exists s \in ss, \neg \text{validDecomp}(s) \vee \\ & \exists ty_a, ty_b \in ts, ty_a \neq ty_b \vee \\ & \exists y \in ys, y \notin gs \end{cases}$$

Strongly Valid Node Decomposition

An optional though stronger correctness property for structural relations is that every single structural relation declared must also be unique. Formally, this is given as:

Definition 31 (Strongly Valid Node Decomposition). *Given a goal-graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{N}$ and $ls = \{l_1, \dots, l_n\}, l_i \in \mathcal{L}$. Let $g \in gs$ be a goal node in M , and let $ss = \{g \circ_{ty_1} ys_1, \dots, g \circ_{ty_m} ys_m\} \subset ls$ be the set of decomposition relations associated with g . If $\text{validGoalDecomp}(M, g) = \text{True}$ then let $s' = g \circ_{ty} ys, ys = \bigcup_{i=1}^m ys_i$ be the structural relation formed by unifying each set of decomposition relations ys_i from ss . The node g has strong valid node decomposition if:*

$$\text{stronglyValidGoalDecomp}(M, g) = \begin{cases} \text{True} & \forall a, b \in ys, a \neq b \\ \text{False} & \exists a, b \in ys, a = b \end{cases}$$

Valid Structural Span

With these notions of valid node decomposition, the structural property that ensures hierarchical correctness of goal decomposition from the root parent through to its decedents can be defined:

Definition 32 (Structural Span). *Given a goal-graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{G}$ and $ls = \{l_1, \dots, l_n\}, l_i \in \mathcal{L}$. The structural span $\text{sspan}(M, g)$*

of a node $g \in gs$ is defined as the sub-graph g' that emerges by traversing all child nodes along structural edges in ls where g is the root until a goal node with no structural edges is encountered. If g has no structural edges then g does not have a structural span.

Definition 33 (Valid Structural Span). *The structural span $s\text{span}(g)$ is a valid span if the resulting span is also a tree.*

$$\text{validSSpan}(M, g) = \begin{cases} \text{True} & \text{isTree}(s\text{span}(M, g)) \\ \text{False} & \neg \text{isTree}(s\text{span}(M, g)) \end{cases}$$

10.2.4 Goal-Graph Correctness

Goal-Graph correctness is defined using the definitions and properties given in this section. A goal-graph M is valid if all the relations in M are valid and well-formed, all the goals are unique, and all structural spans in M are also valid structural spans.

Definition 34 (Goal-Graph Correctness). *Given a goal-graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}$, $g_i \in \mathcal{N}$ and $ls = \{l_1, \dots, l_n\}$, $l_i \in \mathcal{L}$. Let $is \subset gs$ be a subset of goals that have intentional links, and $ss \subset gs$ be a subset of goals that have structural links. M is a correct goal-graph if*

$$\text{validGraph}(M) = \begin{cases} \text{True} & \forall i \in is, \text{validGoalIntent}(M, i) \wedge \\ & \forall s \in ss, \text{validGoalDecomp}(M, s) \wedge \\ & \forall s \in ss, \text{validSpan}(M, s) \wedge \\ & \text{uniqueGoals}(M) \\ \text{False} & \exists i \in is, \neg \text{validGoalIntent}(M, i) \vee \\ & \exists s \in ss, \neg \text{validGoalDecomp}(M, s) \vee \\ & \exists s \in ss, \neg \text{validSpan}(M, s) \vee \\ & \neg \text{uniqueGoals}(M) \end{cases}$$

10.3 Building the Goal-Graph Using G^\star

This section presents the formal definitions and implementation details for the modeling language G^\star used for goal-graph construction. This section begins with the formal language definition for G^\star , building on the formal definition of goal-graphs given in Definition 10.2.1. Second, the type-system used to govern construction of models from

the declarations is given. Finally, interpretation semantics are given that detail how the G★ modelling language can construct goal-graphs, together with properties detailing how the construction of correct goal-graphs is achieved.

The purpose of G★ is to perform the insertion of valid goals and relations into the model. The next language GEXPR reasons about the validity of candidate goals and relations such that only valid terms are inserted. Hence, within this language definition the implementation and typing of relations are left semi-abstract and are considered in §10.4.

10.3.1 Abstract Syntax

The modelling language G★ has been designed as a declarative EDSL. Figure 10.2 presents the language definition for G★, together with nominal typing information.

$$\begin{aligned}
 \text{G★} &= \text{G★} \mid \text{x} \mid \text{l} \mid \text{m} & (10.1) \\
 &\mid \text{m} \uplus_{\mathcal{N}} \text{x} \mid \text{m} \uplus_{\mathcal{I}} \text{i} \mid \text{m} \uplus_{\mathcal{S}} \text{s} & (10.2) \\
 &\mid \text{m} \uplus_{\mathcal{N}}^* \{\text{x}_1, \dots, \text{x}_n\} \mid \text{m} \uplus_{\mathcal{I}}^* \{\text{i}_1, \dots, \text{i}_n\} \mid \text{m} \uplus_{\mathcal{S}}^* \{\text{s}_1, \dots, \text{s}_n\} & (10.3) \\
 \text{m} \in \mathcal{M} &= \text{Model} \{\text{x}_0, \dots, \text{x}_n\} \{\text{l}_0, \dots, \text{l}_m\} & (10.4) \\
 \text{x}, \text{y} \in \mathcal{N} &= \langle \text{e}, \text{t}, \text{q}, \text{sTy} \rangle & (10.5) \\
 \text{i} \in \mathcal{J} &= \text{x} \circ_{\text{contrib}}^{\text{c}} \text{y} \mid \text{x} \circ_{\text{correl}}^{\text{c}} \text{y} & (10.6) \\
 \text{s} \in \mathcal{S} &= \text{x} \circ_{\wedge} \{\text{y}_1, \dots, \text{y}_n\} \mid \text{x} \circ_{\vee} \{\text{y}_1, \dots, \text{y}_n\} \mid \text{x} \circ_{\oplus} \{\text{y}_1, \dots, \text{y}_n\} & (10.7) \\
 \text{q} \in \mathcal{Q} &= \text{Denied} \mid \text{wDenied} \mid \text{wSatisfied} \mid \text{Satisfied} \\
 &\mid \text{Conflict} \mid \text{Unknown} \mid \text{None} & (10.8) \\
 \text{c} \in \mathcal{C} &= \text{Make} \mid \text{Help} \mid \text{SomePos} \mid \text{Unknown} \\
 &\mid \text{SomeNeg} \mid \text{Break} \mid \text{Hurt} & (10.9) \\
 \text{e} \in \mathcal{N}' &= \text{TyGoal} \mid \text{TySoft} \mid \text{TyTask} \mid \text{TyRes} & (10.10) \\
 \text{sTy} \in \mathcal{S}' &= \wedge \mid \vee \mid \oplus & (10.11)
 \end{aligned}$$

Figure 10.2: Language definition for G★.

The core declarations for G★ are presented on Lines 10.1 10.2 & 10.3. These declarations allow for the declaration of nodes, edges, a goal-graph, and insertion functions. With the precise constructors for goal-graphs and nodes on Lines 10.4 & 10.5. Intentional links (Line 10.6) are binary, linking a source goal (x) to a target goal (y), allowing

for the contribution value (Line 10.9) to be specified. Structural links (Line 10.7) are one-to-many linking a single source goal (x) to many source goals— $\{y_1, \dots, y_n\}$.

$G\star$ also provides several operators for modification of the goal-graph: \uplus for insertion of a single expression; and \uplus^* for many. Note, many of the language constructs for goal-graph representation are either adapted or taken directly from the definitions given in §10.2.

10.3.2 Type System

This section presents the types and typing rules for $G\star$. As a declarative EDSL constructed in Idris, we can leverage Idris' existing mechanisms to manage the typing environments for the language. The typing rules thus need not be given a context or typing environment. Further, the typing rules for creation of satisfaction values, contribution values, node types and decomposition types are not given for brevity.

The Types

The set of types specific to $G\star$ are:

$$\mathcal{T} = \mathcal{M} \mid \mathcal{N} \mid \mathcal{S} \mid \mathcal{J} \mid \mathcal{C} \mid \mathcal{Q} \mid \mathcal{N}' \mid \mathcal{S}'$$

with semantic meaning of: \mathcal{M} —goal-graphs; \mathcal{N} —nodes; \mathcal{S} —structural declarations; \mathcal{J} —intentional declarations; \mathcal{C} —contribution values; \mathcal{Q} —satisfaction values; \mathcal{N}' —the type for all node type indicators; \mathcal{S}' —the type for all decomposition type indicators.

Rules for Language Declarations

$$\frac{e:\mathcal{N}' \quad t:\text{String} \quad q:\mathcal{Q} \quad s:\mathcal{S}'}{\langle e, t, q, s \rangle:\mathcal{N}} \text{Element}$$

$$\frac{x:\mathcal{N} \quad y:\mathcal{N} \quad c:\mathcal{C}}{x \circ_{\text{contrib}}^c y:\mathcal{J}} \text{Contribution} \quad \frac{x:\mathcal{N} \quad y:\mathcal{N} \quad c:\mathcal{C}}{x \circ_{\text{correl}}^c y:\mathcal{J}} \text{Correlation}$$

$$\frac{x:\mathcal{N} \quad ys:\text{List}(\mathcal{N})}{x \circ_{\wedge} ys:\mathcal{S}} \text{AND} \quad \frac{x:\mathcal{N} \quad ys:\text{List}(\mathcal{N})}{x \circ_{\vee} ys:\mathcal{S}} \text{IOR}$$

$$\frac{x:\mathcal{N} \quad ys:\text{List}(\mathcal{N})}{x \circ_{\oplus} ys:\mathcal{S}} \text{XOR}$$

Rules for Declaration Insertion

Single

$$\frac{m:\mathcal{M} \quad n:\mathcal{N}}{m \uplus_{\mathcal{N}} n:\mathcal{M}} \text{Element}$$

$$\frac{m:\mathcal{M} \quad i:\mathcal{J}}{m \uplus_{\mathcal{I}} i:\mathcal{M}} \text{Intentional} \quad \frac{m:\mathcal{M} \quad s:\mathcal{S}}{m \uplus_{\mathcal{S}} s:\mathcal{M}} \text{Structural}$$

Many

$$\frac{m:\mathcal{M} \quad ns:\text{List}(\mathcal{N})}{m \uplus_{\mathcal{N}}^* ns:\mathcal{M}} \text{Element}$$

$$\frac{m:\mathcal{M} \quad is:\text{List}(\mathcal{J})}{m \uplus_{\mathcal{I}}^* is:\mathcal{M}} \text{Intentional} \quad \frac{m:\mathcal{M} \quad ss:\text{List}(\mathcal{S})}{m \uplus_{\mathcal{S}}^* ss:\mathcal{M}} \text{Structural}$$

These typing rules play several roles within the construction of goal-graphs. The first is to ensure that the construction of the two different relation types can be distinguished. The second, to ensure that correct graph construct (node or edge) are passed to the correct insertion functions—see §10.4.6.

10.3.3 Interpretation Semantics

Given the various insertion operations and link definitions defined for G^* , a set of interpretation semantics are described in Figure 10.3 to denote how these operations build up the goal-graph. Figure 10.3 does not present all the interpretations. Transformations between contribution values, satisfaction values, and node type are not given. These transformations are one-to-one mappings between the syntax given in Figure 10.2 and enumerated types given in §10.2.1.

The construction of goal-graphs requires that each relation and goal are first transformed into `GoalEdge` and `GNode`. Each relation is turned into a labelled edge that details the source, destination, and label, with decomposition edges being transformed into a set of edges. Edges are inserted into the graph using the `insertLink` function that inserts the edge into the graph. Nodes use the `insertNode` function. The insert many operation (\uplus^*) is interpreted as the interpretation of multiple insertion operations.

Note. The insertion of intentional links swaps the source and target goal nodes during construction of the labelled edges. This is to ensure that during construction

$$\begin{aligned}
\llbracket G\star \rrbracket &: G\star \rightarrow \text{Goal Graph} \\
\llbracket \langle e, t, q, sTy \rangle \rrbracket &= \text{GNode } \llbracket e \rrbracket \llbracket t \rrbracket \llbracket q \rrbracket \llbracket s \rrbracket & (10.12) \\
\llbracket x \circ_{\text{contrib}}^c y \rrbracket &= \langle \llbracket y \rrbracket, \llbracket x \rrbracket, \text{Contrib } \llbracket c \rrbracket \rangle & (10.13) \\
\llbracket x \circ_{\text{correl}}^c y \rrbracket &= \langle \llbracket y \rrbracket, \llbracket x \rrbracket, \text{Correl } \llbracket c \rrbracket \rangle & (10.14) \\
\llbracket x \circ_{\wedge} \{y_1, \dots, y_n\} \rrbracket &= \{ \langle \llbracket x \rrbracket, \llbracket y \rrbracket, \text{Decomp} \rangle \mid y \leftarrow \{y_1, \dots, y_n\} \} & (10.15) \\
\llbracket x \circ_{\vee} \{y_1, \dots, y_n\} \rrbracket &= \{ \langle \llbracket x \rrbracket, \llbracket y \rrbracket, \text{Decomp} \rangle \mid y \leftarrow \{y_1, \dots, y_n\} \} & (10.16) \\
\llbracket x \circ_{\oplus} \{y_1, \dots, y_n\} \rrbracket &= \{ \langle \llbracket x \rrbracket, \llbracket y \rrbracket, \text{Decomp} \rangle \mid y \leftarrow \{y_1, \dots, y_n\} \} & (10.17) \\
\llbracket m \uplus_N x \rrbracket &= \text{insertNode } \llbracket x \rrbracket m & (10.18) \\
\llbracket m \uplus_N^* xs \rrbracket &= \{ \llbracket m \uplus_N x \rrbracket \mid x \leftarrow \{x_1, \dots, x_n\} \} & (10.19) \\
\llbracket m \uplus_I i \rrbracket &= \text{insertLink } \llbracket i \rrbracket m & (10.20) \\
\llbracket m \uplus_I^* is \rrbracket &= \{ \llbracket m \uplus_I i \rrbracket \mid i \leftarrow \{i_1, \dots, i_n\} \} & (10.21) \\
\llbracket m \uplus_S ss \rrbracket &= \{ \text{insertLink } \llbracket s \rrbracket m \mid s \leftarrow \llbracket ss \rrbracket \} & (10.22) \\
\llbracket m \uplus_S^* sss \rrbracket &= \{ \llbracket m \uplus_S ss \rrbracket \mid ss \leftarrow \{ss_1, \dots, ss_n\} \} & (10.23)
\end{aligned}$$

Figure 10.3: Interpretation semantics for $G\star$

of the forward evaluation algorithms for the GRL graph traversal can be performed top to bottom from root goal node to leaf goal node.

10.4 The Intermediate Representation: GExpr

The previous section introduced $G\star$, the declarative language for the construction of GRL goal-graphs. This section introduces GEXPR, the intermediate language that captures expressions common to all GRL-derived DSMLs. It is with this language that structural and correctness properties of the goal-graph (as given in §10.2) are tested for.

10.4.1 Language Syntax

Like $G\star$, GEXPR has been designed as a declarative EDSL within Idris. While, $G\star$ has been designed for goal-graph construction, the primary purpose of GEXPR is to facilitate the reasoning about all goal-graphs during construction. Various constructs from $G\star$ have been brought forward and are reused in the definition of GEXPR. The language definition for GEXPR is given in Figure 10.4.

$$\begin{aligned} \text{GEXPR} &= \text{GEXPR} \mid e \mid m \uplus e \mid m \uplus^* \{e_1, \dots, e_n\} & (10.24) \\ e \in \mathcal{G} : \mathcal{J} \rightarrow \mathcal{T} &= x \mid i \mid s & (10.25) \\ x, y \in \mathcal{G}(\text{E}) &= \text{Element} \text{ q s} & (10.26) \\ i \in \mathcal{G}(\text{I}) &= \text{Impacts} \text{ x c y} \mid \text{Affects} \text{ x c y} & (10.27) \\ s \in \mathcal{G}(\text{S}) &= \text{And} \text{ x } \{y_1, \dots, y_n\} \mid \text{OR} \text{ x } \{y_1, \dots, y_n\} & (10.28) \\ &\quad \mid \text{XOR} \text{ x } \{y_1, \dots, y_n\} \\ \mathcal{J} &= \text{E} \mid \text{I} \mid \text{S} & (10.29) \\ m \in \mathcal{M} &= \text{Model definition taken from } G\star & (10.30) \\ q \in \mathcal{Q} &= \text{Satisfaction values taken from } G\star & (10.31) \\ c \in \mathcal{C} &= \text{Contribution values taken from } G\star & (10.32) \\ n \in \mathcal{N}' &= \text{Node types taken from } G\star & (10.33) \end{aligned}$$

Figure 10.4: Language definition for GEXPR.

Within GEXPR, EDSL declarations have been unified under a single type that allows for a unified set of expressions to be given for declaration insertion. Whereas the syntax of $G\star$ is for the construction of the goal-graph, the syntax of GEXPR is more language oriented and introduces the declarative language that modellers build upon.

10.4.2 Types

GEXPR introduces two new types to NovoGRL.

$\mathcal{G} : \mathcal{J} \rightarrow \mathcal{T}$ the dependent type for expressions within GEXPR. GEXPR is implemented in the style of the *Well-Typed Interpreter* such that its expressions have been implemented as an inductive ADT that is parameterised by \mathcal{J} the meta-type for the different expressions. Listing 10.1 presents the implementation of GEXPR.

\mathcal{J} the meta-type for expressions within GEXPR. This type has been implemented as an enumerated type in which the constructors represent the types $\mathcal{G}(\text{E})$, $\mathcal{G}(\text{I})$, and $\mathcal{G}(\text{S})$ from GEXPR.

```
data GTy = ELEM | INTENT | STRUCT
```



```

1 data GExpr : GTy -> Type where
2   Elem   : GElemTy -> String -> Maybe SValue
3           -> GExpr ELEM
4   ILink  : GIntentTy -> CValue
5           -> GExpr ELEM -> GExpr ELEM
6           -> GExpr INTENT
7   SLink  : GStructTy -> GExpr ELEM
8           -> List (GExpr ELEM) -> GExpr STRUCT

```

Listing 10.1: Implementation of GExpr

10.4.3 Typing Rules

Parameterising GExpr with GTy allows for typing rules for expressions to be specified and reasoned about directly at the type level. Ensuring that only well typed expressions can be constructed; this is a compile time check. The Elem constructor represents elements and stores the type of the element, its title, and a possible satisfaction value. ILink represents intentional links, storing the type of intentional link, the value of the contribution, and the elements in the link. Notice that for the ILink constructor, the two GExpr expressions must be of type GExpr ELEM. Only elements can be expressed in a link. Similarly, SLink represents structural links, but allows for a single element to be linked to multiple elements in a single expression. As with G*, GEXPR is also constructed as an EDSL allowing for the host system to take care of the typing environment and context.

Element Declaration

$$\frac{n:\mathcal{N}' \quad t:\text{String} \quad q:\mathcal{Q} \quad s:\mathcal{S}'}{\text{Elem } n \ t \ q \ s : \mathcal{G}(E)} \text{Element}$$

Intentional Declarations

$$\frac{x:\mathcal{G}(E) \quad c:\mathcal{C} \quad y:\mathcal{G}(E)}{\text{Impacts } x \ c \ y : \mathcal{G}(I)} \text{Contrib.} \quad \frac{x:\mathcal{G}(E) \quad c:\mathcal{C} \quad y:\mathcal{G}(E)}{\text{Affects } x \ c \ y : \mathcal{G}(I)} \text{Correl.}$$

Structural Declarations

$$\frac{x:\mathcal{G}(E) \quad ys:\text{List}(\mathcal{G}(E))}{\text{And } x \ ys : \mathcal{G}(S)} \quad \frac{x:\mathcal{G}(E) \quad ys:\text{List}(\mathcal{G}(E))}{\text{OR } x \ ys : \mathcal{G}(S)}$$

$$\frac{x:\mathcal{G}(E) \quad ys:\text{List}(\mathcal{G}(E))}{\text{XOR } x \ ys : \mathcal{G}(S)}$$

Single Declaration Insertion

$$\frac{m:\mathcal{M} \quad x:\mathcal{G}(E)}{m \uplus x:\mathcal{M}} \text{Elements} \quad \frac{m:\mathcal{M} \quad x:\mathcal{G}(I)}{m \uplus x:\mathcal{M}} \text{Intentional}$$

$$\frac{m:\mathcal{M} \quad x:\mathcal{G}(S)}{m \uplus x:\mathcal{M}} \text{Structural}$$

Many Declaration Insertion

$$\frac{m:\mathcal{M} \quad xs:\text{List}(\mathcal{G}(E))}{m \uplus^* xs:\mathcal{M}} \text{Element} \quad \frac{m:\mathcal{M} \quad is:\text{List}(\mathcal{G}(I))}{m \uplus^* is:\mathcal{M}} \text{Intentional}$$

$$\frac{m:\mathcal{M} \quad ss:\text{List}(\mathcal{G}(S))}{m \uplus^* ss:\mathcal{M}} \text{Structural}$$

10.4.4 Interpretation Semantics

Dependently typed languages allow for types to be computed. With this functionality, the multiple insertion operations in G^* can be coalesced into a single pair of operations: \uplus , and \uplus^* . Figure 10.5 presents the interpretation semantics for insertion using the coalesced operations.

$$\begin{aligned} \llbracket GEXPR \rrbracket &: GEXPR \rightarrow G^* \\ \llbracket \text{Element } qs \rrbracket &= \langle e, t, q, sTy \rangle \\ \llbracket \text{Impacts } x \text{ c } y \rrbracket &= \llbracket x \rrbracket \circ_{\text{contrib}}^c \llbracket y \rrbracket \\ \llbracket \text{Affects } x \text{ c } y \rrbracket &= \llbracket x \rrbracket \circ_{\text{correl}}^c \llbracket y \rrbracket \\ \llbracket \text{And } x \{y_1, \dots, y_n\} \rrbracket &= \llbracket x \rrbracket \circ_{\wedge} \{ \llbracket y_1 \rrbracket, \dots, \llbracket y_n \rrbracket \} \\ \llbracket \text{OR } x \{y_1, \dots, y_n\} \rrbracket &= \llbracket x \rrbracket \circ_{\vee} \{ \llbracket y_1 \rrbracket, \dots, \llbracket y_n \rrbracket \} \\ \llbracket \text{XOR } x \{y_1, \dots, y_n\} \rrbracket &= \llbracket x \rrbracket \circ_{\oplus} \{ \llbracket y_1 \rrbracket, \dots, \llbracket y_n \rrbracket \} \\ \llbracket m \uplus x \rrbracket &= \begin{cases} (x:\mathcal{G}(E)) & m \uplus_N \llbracket x \rrbracket \\ (x:\mathcal{G}(I)) & m \uplus_I \llbracket x \rrbracket \\ (x:\mathcal{G}(S)) & m \uplus_S \llbracket x \rrbracket \end{cases} \\ \llbracket m \uplus^* \{x_1, \dots, x_n\} \rrbracket &= \begin{cases} (x:\mathcal{G}(E)) & m \uplus_N^* \{ \llbracket x_1 \rrbracket, \dots, \llbracket x_n \rrbracket \} \\ (x:\mathcal{G}(I)) & m \uplus_I^* \{ \llbracket x_1 \rrbracket, \dots, \llbracket x_n \rrbracket \} \\ (x:\mathcal{G}(S)) & m \uplus_S^* \{ \llbracket x_1 \rrbracket, \dots, \llbracket x_n \rrbracket \} \end{cases} \end{aligned}$$

Figure 10.5: Interpretation semantics for GEXPR.

10.4.5 Property Checking

§10.2 detailed goal-graph correctness. To ensure that only valid models can be constructed using the GEXPR language the insertion of any expression e into a model m must ensure that the property $\text{validGraph}(m)$ holds post insertion of e . For elements this requires that all nodes in the model m are unique.

Definition 35 (Valid Element Insertion). *Given a goal-graph $M = \langle gs, rs \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{N}$ and $rs = \{r_1, \dots, r_n\}, r_i \in \mathcal{L}$. Let g be a goal to be inserted into M , g can only be inserted if the properties $\text{uniqueGoals}(M)$ and $\text{validGraph}(M)$ holds post insertion of g .*

For intentional links this requires that the intentional link presented is a valid intentional link for M .

Definition 36 (Valid Intentional Link Insertion). *Given a goal-graph $M = \langle gs, rs \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{N}$ and $rs = \{r_1, \dots, r_n\}, r_i \in \mathcal{L}$. Let $l = x \circ_i^c y$ be an intention link that is to be inserted into M , where $x, y \in \mathcal{N}, c \in \mathcal{C}, i \in \mathcal{J}$. The link l can only be inserted into M if $\text{validIntent}(M, i) = \text{True}$, and $\text{validGraph}(M)$ holds post insertion of l .*

For structural links this requires that the structural link presented is a valid structural link for M

Definition 37 (Valid Decomposition Link Insertion). *Given a goal-graph $M = \langle gs, rs \rangle$ where $gs = \{g_1, \dots, g_n\}, g_i \in \mathcal{N}$ and $rs = \{r_1, \dots, r_n\}, r_i \in \mathcal{L}$. Let $s = x \circ_s \{y_1, \dots, y_n\}$ be a decomposition link that is to be inserted into M , where $x, y_i \in \mathcal{N}$ and $s \in \mathcal{S}$. The link s can only be inserted into M if $\text{validDecomp}(M, s) = \text{True}$, and $\text{validGraph}(M)$ holds post insertion of s .*

These properties are essential when looking to construct goal-graphs using declarative languages and can be checked for when updating the model using these constructs.

10.4.6 Implementation Details

GEXPR has been design as the intermediate representation common to all GRL-oriented DSMLs. To ensure that all languages can be converted correctly to $G\star$ instances, the GRL interface has been constructed. This interface forces language designers to specify how the expressions from the DSML are to be translated into GEXPR expressions. The interface is defined as follows:

```

interface GRL (a : GTy -> Type) where
  mkGoal    : a ELEM    -> GExpr ELEM
  mkIntent  : a INTENT  -> GExpr INTENT
  mkStruct  : a STRUCT  -> GExpr STRUCT

```

With the use of this interface comes a restriction that all DSMLs must also be parameterised using GTy.

Converting GExpr to G \star

The conversion of GEXPR expressions to G \star expressions is achieved using the `convExpr` function that is polymorphic in its return type. This function is an amalgamation of the interpretation from a GEXPR expression to a G \star expression (see Figure 10.5), and G \star to the underlying graph-node representation—see Figure 10.3.

```

convExpr : {ty:GTy} -> GExpr ty -> interpTy ty
convExpr (Elem eTy t s) = GNode eTy t s Nothing
convExpr (ILink IMPACTSty cTy _ _) = Contribution cTy
convExpr (ILink AFFECTSty cTy _ _) = Correlation cTy
convExpr (SLink _ _ _)             = Decomp

```

The function `convExpr` takes in a GExpr as a parameter and returns the converted element. The return type is computed using the `interpTy` function that is detailed below. As types are first class language constructs in Idris the return type of `convExpr` is calculated during use.

```

interpTy : GTy -> Type
interpTy ELEM    = GoalNode
interpTy INTENT  = GoalEdge
interpTy STRUCT  = GoalEdge

```

Choosing the correct Insertion Function

Insertion of multiple (Ψ^*) and single (Ψ) elements into the goal-graph are governed using the following functions.

```

insert : GRL expr => expr ty -> GModel -> GModel
insert {ty=ELEM}  d m = insertElem  (mkElem  d) m
insert {ty=INTENT} d m = insertIntent (mkIntent d) m
insert {ty=STRUCT} d m = insertStruct (mkStruct d) m

```

```
insertMany : GRL expr => List (expr ty)
            -> GModel -> GModel
insertMany ds model = foldl (flip . insert) model ds
```

Both functions abstract over the intermediate representation and accepts expressions that conform to the GRL interface, an existing model, and returns an updated model with the new expression inserted. The interface allows for the DSML expression to be interpreted into an GEXPR instance. It is within the `insert` function that the domain translation of expressions from new domain to the GRL domain takes place using the `mkElem`, `mkIntent`, and `mkStruct` functions.

The `insert` function is exposed to DSML users to allow for a unified means through which goal-graphs can be constructed. However, `insert` is right associative and does not lead to an easy to use interface for building models. This is addressed through provision of a left associative infix operator (`\=`) that allows users to build up models from the left.

```
infixl 4 \=
(\=) : GRL expr => {ty : GTy} -> (m : GModel)
      -> (d : expr ty) -> GModel
(\=) model decl = insert decl model
```

Property checks, and insertion, for the resulting goal-graph do not occur directly in the insertion functions. The actual insertion of expressions into the graph are performed by the specialised functions listed below. The semantics for insertion were detailed in Figure 10.3.

```
insertIntent : GExpr INTENT -> GModel -> GModel
insertStruct : GExpr STRUCT -> GModel -> GModel
insertElem   : GExpr ELEM   -> GModel -> GModel
```

These functions will perform run time checks to decide if inclusion of the expression will invalidate any structural or correctness properties according to the GRL standard. The properties are detailed in Definitions 35 to 37, and the *User Requirements Notation* [UTN₁₂]. If the expression results in an invalid GRL model, the program will terminate.

10.5 Evaluating Goal Graphs

This section details model evaluation. Standard evaluation algorithms can be used to evaluate a model for satisfaction [Amy+10]. These algorithms require that the given GRL model instance is initialised according to a predefined strategy. A strategy is a list of element and satisfaction value pairings. In NovoGRL only the forward propagation algorithm as described in Amyot et al. [Amy+10] and Appendix B was implemented. The algorithm was implemented as a depth first graph traversal algorithm. It has been made accessible to modellers in the framework using the following function:

```
evalModel : GModel -> Maybe Strategy -> List GoalNode
```

This function will initialise the model with a given strategy, if a strategy was given, and perform the evaluation returning a list of nodes in the graph and their resulting satisfaction value. This function will return an empty list if the model is incorrectly initialised using the strategy. Within G^* , strategies are defined as an association list of GoalNode SValue pairings.

```
Strategy : Type
Strategy = List (GoalNode, SValue)
```

To construct strategies for DSML languages, a buildStrategy function is provided that converts elements from the DSML to GoalNodes It is specified as:

```
buildStrategy : GRL expr =>
  List (expr ELEM, SValue) -> Strategy
```

Although there are known evaluation algorithms for the GRL, there are no known formal evaluation semantics [AM11]. TROPOS and i^* are cousin and parent to GRL, and these languages have been provided with formal evaluation semantics [Gio+03] based upon goal-graphs and a more restricted semantics for evaluation. Future work will be to take the formal description of GEXPR and construct a set of evaluation semantics in the same style as presented by Giorgini et al. [Gio+03].

10.6 Modelling the GRL as a DSML

The previous sections described the implementation and design of NovoGRL for DSML construction. This section illustrates how this framework can be used to provide

an implementation of the GRL itself.

10.6.1 Language Definition

$$\begin{aligned} \text{GRL} &= \text{GRL} \mid e \mid m \uplus e \mid m \uplus^* \{e_1, \dots, e_n\} & (10.34) \\ e \in \mathcal{G}' : \mathcal{T} &\rightarrow \star = x \mid i \mid s & (10.35) \\ x, y \in \mathcal{G}'(\mathcal{E}) &= \text{Goal } t \ q \mid \text{SGoal } t \ q \mid \text{Task } t \ q \mid \text{Res } t \ q & (10.36) \\ i \in \mathcal{G}'(\mathcal{I}) &= x \xrightarrow{c} y \mid x \xrightarrow{-c} y & (10.37) \\ s \in \mathcal{G}'(\mathcal{S}) &= x \wedge \{y_1, \dots, y_n\} \mid x \vee \{y_1, \dots, y_n\} \mid & (10.38) \\ & \quad x \oplus \{y_1, \dots, y_n\} & (10.39) \\ \mathcal{J} &= \text{Meta type taken from } \text{GEXPR} & (10.40) \\ m \in \mathcal{M} &= \text{Model definition taken from } \text{G}\star & (10.41) \\ q \in \mathcal{Q} &= \text{Satisfaction values taken from } \text{G}\star & (10.42) \\ c \in \mathcal{C} &= \text{Contribution values taken from } \text{G}\star & (10.43) \end{aligned}$$

Figure 10.6: Language definition for the GRL.

Figure 10.6 presents the language definition for the GRL. Recall that in the framework expressions are a dependent type indexed by an element from \mathcal{J} . These expressions are then transformed into instances of GEXPR through instructions specified in an implementation of the GRL interface. As with the definition of GEXPR , several constructs from $\text{G}\star$ have been brought forward in the definition, and the language is declarative in nature. That is, Goal graphs, definitions for contribution and satisfaction values are reused from $\text{G}\star$; insertion functions have been lifted from GEXPR ; and a model specification is comprised of a series of expressions that either declare elements and links, or perform insertion of these structures into the goal-graph. Lines 10.36, 10.37, & 10.39 detail the portions of the language specific to the GRL domain: Elements, Intention Links, and Structural Links.

Table 10.2 illustrates how the abstract syntax specified in Figure 10.6 maps to both the original graphical notation (given in Figure 3.1), and Idris implementation in Listing 10.2. The pictorial language constructs presented are a derivation of the original constructs presented, in an effort to promote clarity in pictorial representation. The changes made relate to the node style for soft goals and how evaluation values are

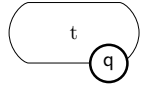
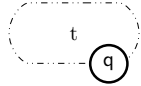
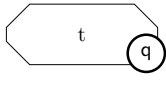
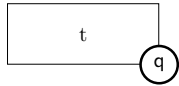
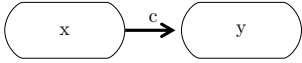
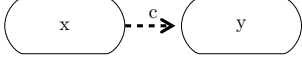
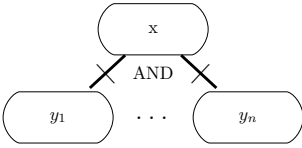
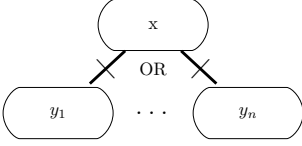
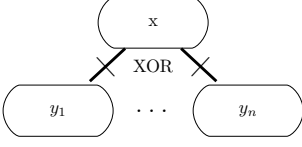
Construct	Abstract	Idris	Pictorial
Goal	$\text{Goal } t \ q$	<code>MkGoal t q</code>	
Soft Goal	$\text{SGoal } t \ q$	<code>MkSGoal t q</code>	
Task	$\text{Task } t \ q$	<code>MkTask t q</code>	
Resource	$\text{Res } t \ q$	<code>MkRes t q</code>	
Contrib.	$x \xrightarrow{c} y$	<code>MkImpact c x y</code>	
Correl.	$x \dashrightarrow^c y$	<code>MkAffects c x y</code>	
And	$x \wedge \{y_1, \dots, y_n\}$	<code>MkAnd x ys</code>	
Or	$x \vee \{y_1, \dots, y_n\}$	<code>MkIor x ys</code>	
XOR	$x \oplus \{y_1, \dots, y_n\}$	<code>MkXor x ys</code>	

Table 10.2: Various representations of the GRL: pictorial, abstract syntax, and Idris.

presented. The insertion operations are not comparable between syntaxes as the act of insertion in the graphical representation is emergent functionality.

10.6.2 Type System

Within this implementation of the GRL a single new dependent type has been introduced: \mathcal{G}' that is indexed by the meta-type \mathcal{J} from GEXPR . The typing rules for declarations in GRL are presented below:

Elements

$$\frac{t:\text{String} \quad q:\mathcal{Q}}{\text{Goal } t \ q:\mathcal{G}'(\text{E})} \text{Goal} \quad \frac{t:\text{String} \quad q:\mathcal{Q}}{\text{SGoal } t \ q:\mathcal{G}'(\text{E})} \text{Soft Goal}$$

$$\frac{t:\text{String} \quad q:\mathcal{Q}}{\text{Task } t \ q:\mathcal{G}'(\text{E})} \text{Task} \quad \frac{t:\text{String} \quad q:\mathcal{Q}}{\text{Res } t \ q:\mathcal{G}'(\text{E})} \text{Resource}$$

Intentional Links

$$\frac{x:\mathcal{G}'(\text{E}) \quad c:\mathcal{C} \quad y:\mathcal{G}'(\text{E})}{x \xrightarrow{c} y:\mathcal{G}'(\text{I})} \quad \frac{x:\mathcal{G}'(\text{E}) \quad c:\mathcal{C} \quad y:\mathcal{G}'(\text{E})}{x \dashrightarrow^c y:\mathcal{G}'(\text{I})}$$

Structural Links

$$\frac{x:\mathcal{G}'(\text{E}) \quad ys:\text{List } (\mathcal{G}'(\text{E}))}{x \wedge ys:\mathcal{G}'(\text{S})} \quad \frac{x:\mathcal{G}'(\text{E}) \quad ys:\text{List } (\mathcal{G}'(\text{E}))}{x \vee ys:\mathcal{G}'(\text{S})}$$

$$\frac{x:\mathcal{G}'(\text{E}) \quad ys:\text{List } (\mathcal{G}'(\text{E}))}{x \oplus ys:\mathcal{G}'(\text{S})}$$

Typing rules for declaration insertion need not be given. The language constructs in this implementation of the GRL are first interpreted into constructions from GEXPR from which the insertion of declarations is performed. §10.6.3 details how the constructs bespoke to this GRL implementation are transformed into constructs from the GEXPR .

Idris Implementation

The Idris implementation of the GRL has been parameterised solely over GTy . In the specification of GLang each of the element types for the GRL has been given its own constructor, that allows for a title and possible initial satisfaction value to be specified. Intentional links that provide contribution and correlation links between elements also

have their own constructors. Finally, each of the different structural decomposition links can be specified using their own constructors.

```

1 data GLang : GTy -> Type where
2   MkGoal : String -> Maybe SValue -> GLang ELEM
3   MkSoft  : String -> Maybe SValue -> GLang ELEM
4   MkTask  : String -> Maybe SValue -> GLang ELEM
5   MkRes   : String -> Maybe SValue -> GLang ELEM
6
7   MkImpacts : CValue -> GLang ELEM
8             -> GLang ELEM -> GLang INTENT
9   MkEffects : CValue -> GLang ELEM
10            -> GLang ELEM -> GLang INTENT
11
12  MkAnd : GLang ELEM -> List (GLang ELEM)
13        -> GLang STRUCT
14  MkXor : GLang ELEM -> List (GLang ELEM)
15        -> GLang STRUCT
16  MkIor : GLang ELEM -> List (GLang ELEM)
17        -> GLang STRUCT

```

Listing 10.2: Definition of the Algebraic Data Type that represents GRL expressions.

10.6.3 Interpretation Semantics

To complete the creation of the GRL in NovoGRL, interpretation instructions must be given to describe how GRL expressions are converted into expressions from GEXPR. This is a simple one-to-one transformation between the languages, as detailed in Figure 10.7 and implemented in Listing 10.3.

With the specification of how the GRL implementation is to be interpreted into expressions from GEXPR comes the ability of NovoGRL to construct GRL derived goal-graphs using these declarations. Although, this example is not indicative over how the GRL can be re-targeted to a new domain, it nonetheless illustrates how this framework can be used to construct a DSML.

10.6.4 Modelling Academic Paper Writing

Using this DSML, model instances using the GRL syntax can be constructed and evaluated using NovoGRL. To illustrate the use of the GRL in this setting, consider the

$$\begin{aligned}
\llbracket \mathcal{G}'(x) \rrbracket &: \mathcal{G}'(x) \rightarrow \mathcal{G}(x) \\
\llbracket \text{Goal } t \ q \rrbracket &= \text{Elem GOALty } t \ q \\
\llbracket \text{SGoal } t \ q \rrbracket &= \text{Elem SGOALty } t \ q \\
\llbracket \text{Task } t \ q \rrbracket &= \text{Elem TASKty } t \ q \\
\llbracket \text{Res } t \ q \rrbracket &= \text{Elem RESTy } t \ q \\
\llbracket x \xrightarrow{c} y \rrbracket &= \text{Impacts } \llbracket x \rrbracket \ c \ \llbracket y \rrbracket \\
\llbracket x \overset{c}{\dashrightarrow} y \rrbracket &= \text{Affects } \llbracket x \rrbracket \ c \ \llbracket y \rrbracket \\
\llbracket x \wedge \{y_1, \dots, y_n\} \rrbracket &= \text{And } \llbracket x \rrbracket \ \{\llbracket y_1 \rrbracket, \dots, \llbracket y_n \rrbracket\} \\
\llbracket x \vee \{y_1, \dots, y_n\} \rrbracket &= \text{OR } \llbracket x \rrbracket \ \{\llbracket y_1 \rrbracket, \dots, \llbracket y_n \rrbracket\} \\
\llbracket x \oplus \{y_1, \dots, y_n\} \rrbracket &= \text{XOR } \llbracket x \rrbracket \ \{\llbracket y_1 \rrbracket, \dots, \llbracket y_n \rrbracket\}
\end{aligned}$$

Figure 10.7: Interpretation semantics for converting GRL expressions into GEXPR expressions.

```

1 GRL GLang where
2   mkGoal (MkGoal s e) = Elem GOALty s e
3   mkGoal (MkSoft s e) = Elem SOFTty s e
4   mkGoal (MkTask s e) = Elem TASKty s e
5   mkGoal (MkRes s e) = Elem RESTy s e
6
7   mkIntent (MkImpacts c a b) =
8     ILink IMPACTSty c (mkGoal a) (mkGoal b)
9   mkIntent (MkEffects c a b) =
10    ILink AFFECTSty c (mkGoal a) (mkGoal b)
11
12   mkStruct (MkAnd a bs) =
13     SLink ANDty (mkGoal a) (map (mkGoal) bs)
14   mkStruct (MkXor a bs) =
15     SLink XORty (mkGoal a) (map (mkGoal) bs)
16   mkStruct (MkIor a bs) =
17     SLink IORty (mkGoal a) (map (mkGoal) bs)

```

Listing 10.3: Interpreting GRL expressions into GEXPR

goal-oriented modelling of academic paper writing using the GRL. The requirements for this DSML stipulate:

The root goal of the graph is the paper itself. Each paper will have a single abstract and bibliography, and multiple sections. These are the goals of the model. Tasks performed during the creation process are related to the writing and reviewing of the components. For each sub-goal, a single review and writing task must be modelled.

The components of a paper are modelled as goals, and actions for writing and reviewing modelled as tasks. Components can be divided into sub-components through ‘and’ decomposition, and actions affect tasks and sub tasks through weighted GRL intention links.

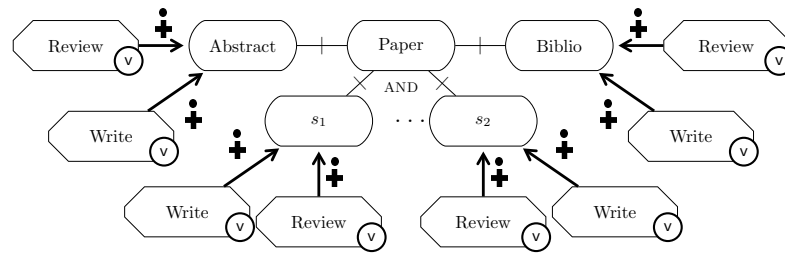


Figure 10.8: Generic instantiation of a GRL model for academic paper writing.

Use of the implemented evaluation mechanism will allow for a modelled TODO list to be checked for satisfiability when presented with an initial strategy to represent the state of each of the presented tasks. Figure 10.8 presents a sample GRL model, illustrating the resulting generic model for all paper modelling instances. A specific instance of this example for a paper that follows the classical scientific structure of IMRAD is given in Figure 10.9.

To implement this model each of the pictorial language constructs needs to be represented using the GRL DSML. As the language is an EDSL, the model can be constructed in several stages. The first stage (zero) requires declaration of the goal elements as variables within the Idris language. For brevity they are not listed below, a complete listing of these variables is given in Appendix A. Listing 10.4 details the next stage in which these elements are inserted into an empty model.

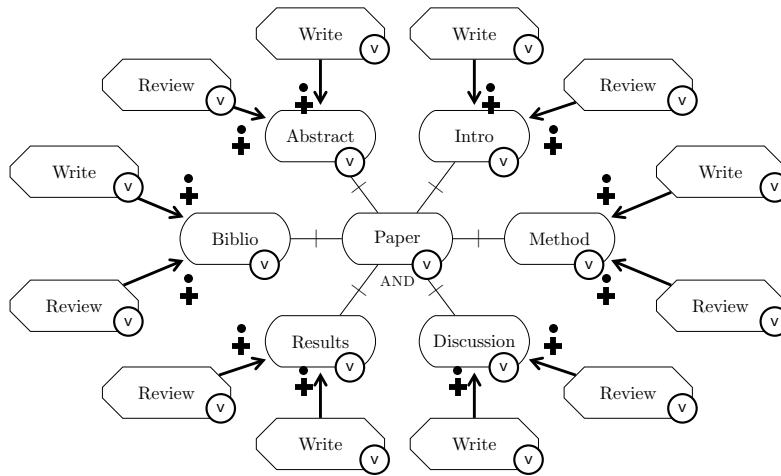


Figure 10.9: Model instance of a GRL model for an academic paper.

```

1 modelElements : GModel
2 modelElements = emptyModel
3   \= paper
4   \= abst   \= wabs   \= rabs   \= bib   \= wbib   \= rbib
5   \= intro  \= wIntro \= rIntro \= meth  \= wMeth \= rMeth
6   \= res    \= wRes   \= rRes   \= disc  \= wDis  \= rDis

```

Listing 10.4: Insertion of elements into a GRL Model to represent academic paper writing.

Listing 10.5 details the second stage that requires the creation and insertion of the structural links that detail the paper's structure.

```

1 modelStructure : GModel
2 modelStructure = modelElements
3   \= (And paper [bib, abst, intro, meth, res, disc])

```

Listing 10.5: Insertion of structural information into a GRL model to represent academic paper writing.

Finally, the intentional links that link the tasks of reviewing and writing to the individual sections are created and inserted. Listing 10.6 illustrates this last stage and shows the creation of the model for the paper.

With the constructed model, different strategies can be created to track the paper

```

1 imradPaper : GModel
2 imradPaper = modelStructure
3   \=(MkImpacts MAKES wabs  abst) \=(MkImpacts MAKES rabs  abst)
4   \=(MkImpacts MAKES wbib  bib ) \=(MkImpacts MAKES rbib  bib )
5   \=(MkImpacts MAKES wMeth meth) \=(MkImpacts MAKES rMeth meth)
6   \=(MkImpacts MAKES wRes  res ) \=(MkImpacts MAKES rRes  res )
7   \=(MkImpacts MAKES wDis  disc) \=(MkImpacts MAKES rDis  disc)
8   \=(MkImpacts MAKES rIntro intro)
9   \=(MkImpacts MAKES wIntro intro)

```

Listing 10.6: Insertion of contribution links into a model for academic paper writing.

writing process. For example the initial state of the paper can be represented as follows:

```

myProgress : Strategy
myProgress = [(wabs, DENIED), (rabs, DENIED)'
              (wbib, DENIED), (rbib, DENIED)'
              (wIntro, DENIED), (rIntro, DENIED)'
              (wMeth, DENIED), (rMeth, DENIED)'
              (wRes, DENIED), (rRes, DENIED)'
              (wDis, DENIED), (rDis, DENIED)]

```

and evaluation of the model as:

```

res : List (GNode, SValue)
res = eval imradPaper myProgress

```

Where `res` will return a list of the nodes in the model and their satisfaction values. For this example, all nodes will be denied, and the paper not satisfied. As the tasks are completed, the strategy can be updated with satisfaction values that are indicative of the paper's state. These strategies can be used to evaluate the model and keep track of the writing process.

10.6.5 Cleaning up the Syntax

Using the GRL in its current form will result in type signatures and link declarations that are verbose, and can lead to unwieldy model specifications and descriptions in which implementation details are exposed unnecessarily to the user. Take the declaration of the type for elements in the model and the constructor for the various relations. Elements are given the type `GLang ELEM`. This is not descriptive and provides no detail, aside from the GRL constructs, of what is being modelled. The constructor

```

1 syntax [a] "==" [b] "|" [c] = MkImpacts c a b
2 syntax [a] "~>" [b] "|" [c] = MkEffects c a b
3 syntax [a] "&=" [bs] = MkAnd a bs
4 syntax [a] "X=" [bs] = MkXor a bs
5 syntax [a] "|=" [bs] = MkIor a bs

```

Listing 10.7: Syntax extensions for prettifying GRL implementations in Idris

MkImpacts MAKES wabs abst is also not intuitive w.r.t. to which element is producing the effect.

This verbosity can be reduced, and more clarity given, using Idris' support for custom syntax declarations, and type aliasing. Each element can be given its own type alias, and links between elements can be give custom syntax that is more readable. Further, Idris' support for syntax definitions can be used to introduce syntax that mirrors the abstract syntax used in the formal definition. Listing 10.7 presents several suitable syntax declarations. With these syntax overlays, the intentional relations between elements in the model can be represented. Listing 10.8 presents the results of using these syntax overlays.

```

1 imradPaper : GModel
2 imradPaper = modelStructure
3 \= (wabs ==> abst | MAKES) \= (wbib ==> bib | MAKES)
4 \= (wIntro ==> intro | MAKES) \= (wMeth ==> meth | MAKES)
5 \= (wRes ==> res | MAKES) \= (wDis ==> disc | MAKES)
6 \= (rabs ==> abst | MAKES) \= (rbib ==> bib | MAKES)
7 \= (rIntro ==> intro | MAKES) \= (rMeth ==> meth | MAKES)
8 \= (rRes ==> res | MAKES) \= (rDis ==> disc | MAKES)

```

Listing 10.8: Syntax Extensions applied to the code from Listing 10.6.

10.7 The Paper Modelling Language

Although, the model of academic paper writing presented in §10.6.4 is a valid GRL instance, not all of the GRL concepts are used. Here the modeller is free to use the remaining GRL concepts such as correlation links, XOR decomposition or resource and soft goals in their model. How they relate to the domain of paper writing is not known, nor is the effect that these untranslated concepts will have when determining goal satisfaction. Further, the translation of concepts from the GRL into the modelled

domain must be performed manually by the user. This is indicative of the problem in producing DSMLs from the GRL.

When using this technique it is the modellers themselves that have to ensure not only the correctness of their model for their domain in the host language, but also the correctness of the interpretation of their model into the host language. These problems can be addressed using NovoGRL to create a new DSMLs that is structurally equivalent to the GRL but uses domain specific terminology and constructs. The resulting DSMLs are new semantic overlays for the GRL. To illustrate this re-domaining, this section presents a DSML based on the GRL, the *Paper Modelling Language* (PML), that re-targets the GRL for the domain of academic paper writing.

10.7.1 Language Design

Figure 10.10 presents the abstract syntax for PML. Recall the GRL model instance for paper writing in Figure 10.9 with the generic construction in Figure 10.8. A distinct usage pattern is present when using the GRL for modelling papers. The root goal of the model represents the paper, and has ‘and’ structural decomposition to represent the elements of the paper. Intentional links are made between each review and writing task element to a paper element with a contribution value of MAKES.

Thus in the re-domaining of the GRL the node types are replaced with nodes that represent directly, the paper, paper elements, and the tasks of writing and reviewing each component. The intentional links will also be reduced to a single unlabelled edge that allows for a writing or reviewing node to be linked to a component node only. Similarly, the structural links will be reduced to a single ‘and’ decomposition that allows only the root element to be decomposed into the paper components. Figure 10.10 presents the resulting abstract syntax for the PML. The restrictions in PML on the creation of structural and intentional links between the nodes are enforced using the type system.

10.7.2 Type System

The PML is a simply-typed language in which each node and relation declaration are typed. The list of types are as follows:

$$\mathcal{T} = \mathcal{P} \mid \mathcal{A} \mid \mathcal{B} \mid \mathcal{S} \mid \mathcal{W} \mid \mathcal{R} \mid \mathcal{J} \mid \mathcal{D}$$

$$\text{PML} = \text{PML} \mid e \mid s \mid i \mid m \uplus d \mid m \uplus^* \{d_1, \dots, d_n\} \quad (10.44)$$

$$e = \text{Paper } t \mid \text{Abstract} \mid \text{Biblio} \mid \text{Section } t \mid \text{Write } t \mid \text{Review } t \quad (10.45)$$

$$i = e \rightarrow e \quad (10.46)$$

$$s = e \wedge e \quad (10.47)$$

$$m \in \mathcal{M} = \text{Model definition taken from } G^* \quad (10.48)$$

Figure 10.10: Abstract syntax for language declarations in the PML.

with semantic meaning of: \mathcal{P} —papers; \mathcal{A} —abstracts; \mathcal{B} —bibliographies; \mathcal{S} —sections; \mathcal{W} —writing tasks; \mathcal{R} —reviewing tasks; \mathcal{J} —task assignment declarations; \mathcal{D} —paper building declarations.

The typing rules to ensure well-formed construction of language constructs from PML are as follows.

Element Construction

$$\frac{}{\text{Abstract} : \mathcal{A}} \text{The Abstract} \quad \frac{t : \text{String}}{\text{Write } t : \mathcal{W}} \text{Writing} \quad \frac{t : \text{String}}{\text{Review } t : \mathcal{R}} \text{Reviewing}$$

$$\frac{t : \text{String}}{\text{Paper } t : \mathcal{P}} \text{The Paper} \quad \frac{t : \text{String}}{\text{Section } t : \mathcal{S}} \text{A Section}$$

Intentional Relations Correct construction of the intentional relation that associates the tasks of writing and reviewing each component of a paper is governed by the following typing rules.

$$\frac{x : \mathcal{W} \quad y : \mathcal{A}}{x \rightarrow y : \mathcal{J}} \text{Writing Abstract} \quad \frac{x : \mathcal{R} \quad y : \mathcal{A}}{x \rightarrow y : \mathcal{J}} \text{Reviewing Abstract}$$

$$\frac{x : \mathcal{W} \quad y : \mathcal{B}}{x \rightarrow y : \mathcal{J}} \text{Writing Biblio} \quad \frac{x : \mathcal{R} \quad y : \mathcal{B}}{x \rightarrow y : \mathcal{J}} \text{Reviewing Biblio}$$

$$\frac{x : \mathcal{W} \quad y : \mathcal{S}}{x \rightarrow y : \mathcal{J}} \text{Writing Section} \quad \frac{x : \mathcal{R} \quad y : \mathcal{S}}{x \rightarrow y : \mathcal{J}} \text{Reviewing Section}$$

Structural Relations Ensuring that only paper nodes can be decomposed using an ‘and’ relation is governed by the following rules:

$$\frac{x : \mathcal{P} \quad y : \mathcal{A}}{x \wedge y : \mathcal{S}} \text{Adding the Abstract} \quad \frac{x : \mathcal{P} \quad y : \mathcal{B}}{x \wedge y : \mathcal{S}} \text{Adding the Bibliography}$$

$$\frac{x:\mathcal{P} \quad y:\mathcal{S}}{x \wedge y:\mathcal{S}} \text{ Adding a section}$$

10.7.3 Interpretation Semantics

Figure 10.11 presents the interpretation semantics for converting language constructs from PML to GEXPR as detailed in §10.7.1.

$$\begin{aligned} \llbracket \text{PML} \rrbracket &: \text{PML} \rightarrow \mathcal{G}(x) \\ \llbracket \text{Paper } t \rrbracket &= \text{Elem GOALty } t \text{ Unknown} \\ \llbracket \text{Abstract} \rrbracket &= \text{Elem GOALty "Abstract" Unknown} \\ \llbracket \text{Biblio} \rrbracket &= \text{Elem GOALty "Bibliography" Unknown} \\ \llbracket \text{Section } t \rrbracket &= \text{Elem GOALty } t \text{ Unknown} \\ \llbracket \text{Write } t \rrbracket &= \text{Elem TASKty } t \text{ Unknown} \\ \llbracket \text{Review } t \rrbracket &= \text{Elem TASKty } t \text{ Unknown} \\ \llbracket x \rightarrow y \rrbracket &= \text{Impacts } \llbracket x \rrbracket \text{ Make } \llbracket y \rrbracket \\ \llbracket x \wedge y \rrbracket &= \text{And } \llbracket x \rrbracket \llbracket \{y\} \rrbracket \end{aligned}$$

Figure 10.11: Interpretation semantics for converting PML expressions into GEXPR expressions.

Paper and section elements are converted directly into goal nodes with no initial satisfaction value. The Bibliography and Abstract are also converted into goals but are populated with a default title. Reviewing and writing tasks are converted into tasks with no initial satisfaction value. Assignment of paper elements to a paper gets converted into an ‘and’ decomposition link, with assignment of tasks into an intentional link with a default contribution value of MAKES.

10.7.4 Implementation Details

To facilitate integration with the NovoGRL, PML must be indexed by the meta-type \mathcal{T} to instruct the framework how the language constructs are to be interpreted. However, dependent types can be used further to provide an efficient and compact implementation of PML through further parameterisation of the type given to PML language constructs. The type system described in §10.7.2 is implemented through parameterising the type of PML with a secondary type: PTy.

```
data PTy = ElemTy ETy | SLinkTy | ALinkTy
```

PTy represents the types for PML. The element types are represented through the construct ElemTy which is indexed by the type ETy that represents the type of each node.

```
data ETy = PaperTy | SecTy | AuthTy
         | RevTy   | BibTy | AbsTy
```

Representing the types in this manner facilitates the compact representation of the typing judgements in the style of the *Well-Typed Interpreter*. With the type-system described in §10.7.2, Listing 10.9 presents an implementation of the PML.

```
1 data PML : PTy -> GTy -> Type where
2   MkPaper  : String -> PML (ElemTy PaperTy) ELEM
3   MkSect   : String -> PML (ElemTy SecTy)   ELEM
4   MkBib    : PML (ElemTy BibTy) ELEM
5   MkAbs    : PML (ElemTy AbsTy) ELEM
6   MkAuth   : String -> PML (ElemTy AuthTy) ELEM
7   MkRev    : String -> PML (ElemTy RevTy)  ELEM
8
9   AddElem  : PML (ElemTy PaperTy) ELEM
10             -> PML (ElemTy x) ELEM
11             -> {auto prf : ValidPElem x}
12             -> PML SLinkTy STRUCT
13
14   AddAction : PML (ElemTy x) ELEM
15             -> PML (ElemTy y) ELEM
16             -> {auto prf : ValidAction x y}
17             -> PML ALinkTy INTENT
```

Listing 10.9: PML implemented as a parameterised algebraic data type in Idris.

The PML data structure, allows for the creation of paper and section elements that only have a name as an argument. As bibliographies and abstracts have well-known names their constructors have no arguments. The reviewing (MkRev) and writing (MkAuth) elements can be given a description. No initial satisfaction value is required as this can be set using a strategy.

Of interest here is the use of the Idris proof search mechanism to restrict the specification of an action between two elements, and assignment of an element to a paper. The types ValidPElem and ValidAction are used as predicates to ensure that only valid elements can be assigned to a paper node, and that writing and reviewing

processes can only be assigned to section, abstract, and bibliography elements. The values within the `ElemTy` constructor are used to populate the predicates, accordingly. These predicates implement the extra domain specific typing rules that determine valid expressions in this language, and are evaluated at compile time. For `ValidPElem` the following predicate was defined:

```
data ValidPElem : ETy -> Type where
  SP : ValidPElem SecTy
  BP : ValidPElem BibTy
  AP : ValidPElem AbsTy
```

Listing 10.10 presents the instance of the GRL typeclass for PML that implements the interpretation semantics from §10.11.

```
1 GRL (\x => PML ty x) where
2   mkGoal (MkPaper t) = Elem GOALty t Nothing
3   mkGoal (MkSect t)  = Elem GOALty t Nothing
4   mkGoal (MkBib)    = Elem GOALty "Bib" Nothing
5   mkGoal (MkAbs)    = Elem GOALty "Abs" Nothing
6
7   mkGoal (MkAuth t) =
8     Elem TASKty ("Authoring " ++ t) Nothing
9   mkGoal (MkRev t) =
10    Elem TASKty ("Reviewing " ++ t) Nothing
11
12   mkIntent (AddAction x y) =
13     ILink IMPACTSty MAKES (mkGoal x) (mkGoal y)
14
15   mkStruct (AddElem x y) =
16     SLink ANDty (mkGoal x) [(mkGoal y)]
```

Listing 10.10: Interpretation semantics from §10.11 implemented in Idris

Use of PML in its current form will result in type signatures and link declarations that are verbose. This is the same as was seen with the DSML implementation of the GRL given in §10.6. Similarly, use of type alias and syntax declarations can be used to reduce the verbosity. For example:

```
syntax [a] "==>" [b] = AddAction a b
syntax [a] "&=" [b]   = AddElem a b
```

10.7.5 Model Construction

The running example that was introduced in §10.6.4, can now be modelled using the domain specific PML. A full listing of the resulting model is given in Appendix A. Below are examples of how a paper and review task are declared.

```
paper : PAPER
paper = MkPaper "My First Paper"

rIntro : REVIEW
rIntro = MkRev "Intro"
```

Comparing how these constructs were declared using the GRL, domain specific terms have now been given to the elements. The full model declaration is similar in appearance to the one seen for the GRL. For instance, compare populating the model with elements in Listing 10.4 for the GRL with the one for the PML in Listing 10.11.

```
1 modelElements : GModel
2 modelElements = emptyModel
3   \= paper
4   \= abst  \= wabs  \= rabs
5   \= bib   \= wbib  \= rbib
6   \= intro \= wIntro \= rIntro
7   \= meth  \= wMeth \= rMeth
8   \= res   \= wRes  \= rRes
9   \= disc  \= wDis  \= rDis
```

Listing 10.11: Insertion of elements into a PML model for academic paper writing.

Insertion of structural information for the GRL was presented in Listing 10.5. Listing 10.12 presents the same procedure but for PML.

```
1 modelStructure : GModel
2 modelStructure = modelElements
3   \= (paper &= abst)  \= (paper &= bib)
4   \= (paper &= intro) \= (paper &= meth)
5   \= (paper &= res)   \= (paper &= dics)
```

Listing 10.12: Insertion of structural information into a PML model for academic paper writing.

To complete the PML model, the intentional links must now be inserted. For the GRL this was detailed in Listing 10.6. Listing 10.13 presents the same procedure but for PML.

```

1 paperPlan : GModel
2 paperPlan = modelStructure
3   \= (wabs ==> abst)  \= (wbib ==> bib)
4   \= (wIntro ==> intro) \= (wMeth ==> meth)
5   \= (wRes ==> res)   \= (wDis ==> dis)
6   \= (rabs ==> abs)   \= (rbib ==> bib)
7   \= (rIntro ==> intro) \= (rMeth ==> meth)
8   \= (rRes ==> res)   \= (rDis ==> dis)

```

Listing 10.13: Insertion of contribution links into a PML model for academic paper writing.

Modulo the use of syntax extensions (see Listing 10.7) notice, that there is now no unnecessary leakage from the GRL of contribution values. However, with the construction used above a more verbose syntax is required to assign components to a paper, each element must be assigned using a separate declaration. This, can be resolved through creation and use of a ‘hidden’ element used to denote valid components of a paper. This will allow for the more compact notation to be restored.

10.8 Experimental Evaluation

NovoGRL is a re-engineering of the GRL for use as an EDSL within Idris such that new semantic layers can be constructed on-top of the GRL. §10.6 and §10.7 have presented the results of two case studies illustrating how such languages are constructed. §10.6 detailed modelling of the GRL itself in NovoGRL, and §10.7 the construction of a language for modelling academic writing.

For both DSMLs and the NovoGRL, each model constructed for the thesis was cross-validated using the standard implementation of the GRL in the jUCMNav tool².

Further, the evaluation mechanism of the NovoGRL was tested during constructed using examples taken from Amyot et al. [Amy+10]. The results were also cross-validated using the tooling from the jUCMNav tool. The models constructed and

²<http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome>

tests are presented as part of the implementation of the NovoGRL and available online [dMH15e].

10.9 Discussion

This section presents a discussion over NovoGRL and its limitations.

10.9.1 Re-Domaining

NovoGRL allows for provisioning of new semantic overlays for the GRL. Although the GRL has been replicated itself, this new functionality, as demonstrated using PML, allows for domain specific constructs to be presented and used by the modeller. The GRL is now extensible. The problems that were once seen when constructing DSMLs over interpretation correctness can now be avoided. The interpretation process forces DSML designers to ensure that their language can be translated into GRL constructs with the GRL interface. Dependent types and the techniques presented provides a concrete link between goal-oriented DSMLs and their host language.

10.9.2 The GRL is now Programmable

The GRL is a visual modelling language and requires modellers to use a bespoke environment to construct and evaluate models. Modellers interact with their models using visual tooling. With NovoGRL this environment is now programmable and scriptable, as the language is an EDSL within Idris itself. NovoGRL models *are* Idris programs. Although a DSL can be constructed, the ‘as-a-library’ provision of NovoGRL allows for it to be used elsewhere. Either to embed GRL modelling into applications or as the base for a new DSML. The latter is how SIF was constructed.

10.9.3 Guarantees over Domain Unique Properties

The NovoGRL framework allows for the construction of DSMLs that are domain specific variants of the GRL. When constructing goal-graphs (GModel) the framework ensures the structural correctness of the model as a GRL instance. Although, the proof search mechanism of Idris can be used to enforce correctness of relations (See §10.4.5), other domain unique properties such as uniqueness of certain node types are not catered for. For example, the implementation of the PML given in §10.7 does not

restrict the creation of combined paper planning models, something that should be restricted. This lack of further property enforcement limits the ability of this framework to provide guarantees towards non-GRL domain specific properties. Future work can be to investigate how the framework can be extended to include arbitrary domain specific checks of the constructed model.

10.9.4 Contribution and Satisfaction Modelling

This framework does not offer a means to provide interpretations of the satisfaction and contribution values from the modelled domain to that of the GRL. For the PML the full range, nor the semantic meaning, for satisfaction and contribution values are appropriate for modelling paper creation/todo management. Satisfaction values such as DENIED, WEAKSATIS, and SATISFIED could be replaced with names more endemic to the domain such as TODO, STARTED, and DONE. Similar considerations should be made towards the translation of contribution values between the two domains. Future work will be to consider how the mapping of these values from the modelled domain to the GRL and the reverse transformation should be facilitated.

10.9.5 Performance

As detailed in Chapter 5 §5.6.2 this research was primarily focused on construction of functionally correct modelling languages. Producing a language with efficient evaluation was not a prime concern. However, as also mentioned in Chapter 5 §5.6.2 future work will be to investigate the performance of the NovoGRL evaluation mechanism when operating on large specifications.

10.9.6 Future Work

Aside from the existing areas of future work mentioned in this discussion, several other areas can be considered.

The version of the GRL modelled in NovoGRL is not complete, and only allows for modelling of single systems. Further work will be to provide a complete representation of the GRL that includes notions of *beliefs*, *actors*, and *indicators*. *Actors* represent discrete units within the problem scenario. Modelling of actors will allow for more discrete components in a DSML to also be modelled.

Further, current property checks when constructing NovoGRL models are performed at run time, during model construction. The PML case study demonstrated how checks, when promoted to the type level, can become compile time and runtime checks using Idris' proof search mechanism. Future work will be to promote the property checks to the type level so that they become compile, and not run time checks.

A further area of improvement will be to address error reporting. As NovoGRL is an EDSL, Idris' own error reporting mechanisms are used to report errors. Work has been performed to allow for better EDSL error reporting [Chr14]. Future work will be to improve upon the error reporting for $G\star$ and DSMLs built using it.

10.10 Summary

This chapter has presented NovoGRL a framework for constructing GRL-oriented DSMLs. This framework was used to reconstruct the GRL, and present the PML, a requirements based modelling language for modelling the act of academic paper writing.

NovoGRL has been designed and constructed using language-oriented techniques. Dependent types were used for implementing the language and for property checking to provide runtime and compile time guarantees over the correctness of various modelling constructs. Ensuring that all expressions are well-formed through typing, and use of Idris' proof search mechanism for additional expression construction checks. Providing for a verified and efficient implementation of the formal language descriptions.

Further, the approach taken has allowed for the structural logic to be separated from the semantic logic. The resulting languages are semantic overlays that are structurally equivalent to the GRL but use domain specific terminology and constructs. Thus, the resulting domain language will have a semantically different abstract syntax, and different restrictions on the configuration of nodes and edges in comparison to the GRL. It is this separation that allows for the GRL to be re-targeted for different domains.

TYPES AS (META) MODELLERS

The *Types as (Abstract) Interpreters* approach detailed in Chapter 4 §4.7 introduced how dependent types can provide guarantees over the correctness of a language w.r.t. to a named ‘abstraction’. This was achieved by modelling the abstraction directly within the type of the language. This technique allowed for the construction of: *Well-Typed (Abstract) Interpreters*. This chapter details how this approach can also be applied to the construction of a DSML and its meta-model, resulting in a new approach called: *Types as (Meta) Modellers*.

When building a DSML from a host language, an important design decision is determining how to interpret expressions from one language to another—i.e. from SIF to NovoGRL. If the languages are not structurally isomorphic, that is they are structurally dimorphic, then this structural difference makes modelling their interpretations at the type level more cumbersome. This chapter also details several implementation techniques used to aid practitioners when working with structurally dimorphic languages using the *Types as (meta) Modellers* approach.

The techniques detailed in this chapter details how the interpretation semantics for SIF were implemented in Idris. If one examines the structures of a SIF model w.r.t. to the resulting GRL model, the two structures are not structurally isomorphic to one another: SIF is list-recursive, and the GRL model is a directed graph.

Furthermore, the implementation of the interpretation semantics detailing the transformation of a SIF model to a NovoGRL model is too extensive for inclusion

directly in this thesis. It is available in Appendix A. Thus, to illustrate the techniques employed when using this approach a smaller DSML for modelling the planning of academic paper writing, PTODO, is introduced.

11.1 Modelling with Differently Shaped Languages

Chapter 10 introduced NovoGRL and within §10.7 described PML, a DSML used for organising paper writing. The PML is a re-domaining of NovoGRL, even though the NovoGRL and PML are semantically different they are structurally equivalent. For example, compare the definitions for the ADTs that represent the PML, and NovoGRLs intermediate language, GExpr. They are repeated from Chapter 10 for convenience. First, the PML:

```
data PML : PTy -> GTy -> Type where
  MkPaper  : String -> PML (ElemTy PaperTy) ELEM
  MkSect   : String -> PML (ElemTy SecTy)   ELEM
  AddElem  : PML (ElemTy PaperTy) ELEM
            -> PML (ElemTy x) ELEM
            -> {auto prf : ValidPElem x}
            -> PML SLinkTy STRUCT
  MkBib    : PML (ElemTy BibTy) ELEM
  MkAbs    : PML (ElemTy AbsTy) ELEM
  MkAuth   : String -> PML (ElemTy AuthTy) ELEM
  MkRev    : String -> PML (ElemTy RevTy)  ELEM
  AddAction : PML (ElemTy x) ELEM
            -> PML (ElemTy y) ELEM
            -> {auto prf : ValidAction x y}
            -> PML ALinkTy INTENT
```

and second, GExpr:

```
data GExpr : GTy -> Type where
  Elem  : GElemTy -> String -> Maybe SValue
        -> GExpr ELEM
  ILink : GIntentTy -> CValue
        -> GExpr ELEM -> GExpr ELEM
        -> GExpr INTENT
  SLink : GStructTy -> GExpr ELEM
        -> List (GExpr ELEM) -> GExpr STRUCT
```

Not only are both languages indexed by GTy, but the constructors for each value in GTy share a common shape modulo minor differences. This commonality aides in the construction of an interpreter between the two languages. Each expression in PML is interpreted directly into a corresponding expression from GExpr.

However, the structure of the resulting model GModel bears little actual resemblance to the structure of a data structure specified to work with a TODO list. PML prohibits domain specific properties of the domain model to be enforced in the language itself. For example, restricting the resulting goal models to modelling one, and only one, paper at a time, and ensuring that a paper can be linked to one abstract and one bibliography. Further, the natural structure of a TODO list, is that of a list of TODO items and tasks. A graph does not represent the natural structure of a TODO list for paper writing.

These domain specific properties can be enforced through construction of a DSML for academic paper writing that is not based on the GRL. Interpretation semantics can then be given to detail how constructs in the language are to be mapped to the GRL. The concept of *Well-Typed (Abstract) Interpreters* is then used to provide a verified interpreter from the language to the NovoGRL that is constructed directly within the types of the DSML itself. To enable the construction of a *Well-Typed (Abstract) Interpreter* for differently shaped languages several implementation techniques that use dependent types have to be developed.

11.2 The Paper Planning Modelling Language

This section introduces the PTODO language for modelling the construction of academic papers. The host language for PTODO is that of the NovoGRL, as presented in Chapter 10.

11.2.1 Language Design

The modelling requirements of PTODO mirror that of PML presented in Chapter 10 §10.7. Papers are to be modelled with a single abstract and bibliography, and will have multiple sections. Tasks performed during the creation process are related to the writing and reviewing of the components. For each component of the paper, a single review and writing task must be modelled, and initialised with a default satisfaction value. This satisfaction value is sourced from the implementation of the GRL. The TODO list

$$\begin{aligned}
 e &= \text{Abstract} \mid \text{Biblio} \mid \text{Section } t \mid \text{Paper } t e e \{e_1, \dots, e_n\} \\
 &\quad \mid \text{Write } v e \mid \text{Review } v e \mid \text{PTodo } e \{e_1, \dots, e_m\} \\
 t &= \text{a String Value} \\
 v \in \mathcal{Q} &= \text{taken from NovoGRL.}
 \end{aligned}$$

Figure 11.1: Abstract syntax for the PTODO modelling language.

will comprise of the paper paired with a list of the TODO items. Use of the GRL’s evaluation semantics will allow for the modelled TODO list to be checked for satisfiability when presented with an initial strategy that represents the state of each of the presented tasks. Figure 11.1 presents the abstract syntax for PTODO.

11.2.2 Type System

To ensure that only well-formed expressions can be constructed, a type-system is defined. The types for PTODO are used to distinguish between the different types of component within a paper, the paper, and the different TODO items, and the TODO list itself.

$$\mathcal{T} = \mathcal{P} \mid \mathcal{A} \mid \mathcal{S} \mid \mathcal{B} \mid \mathcal{J} \mid \mathcal{L}$$

With semantic meaning of: \mathcal{P} the type given to papers; \mathcal{A} the type given to abstracts; \mathcal{S} the type given to sections; \mathcal{B} the type given to bibliographies; \mathcal{J} the type given to TODO items; \mathcal{L} the type given to the TODO list itself.

Using these types, the following typing judgements ensure that only well-formed expressions can be constructed.

11.2.3 Implementation

Listing 11.1 details a simple implementation of PTODO in Idris. Rather than model the language expressions as an inductive ADT, PTODO can be implemented across several bespoke data types. This provides a more natural style of implementation. The types for PTODO can be modelled using a mixture of:

- 1) ADTs to represent the core constructs in the language of: a paper (`Paper`), TODO item (`Item`), a paper’s components (`Comp`), and the TODO list itself—`TODOList`;

$$\begin{array}{c}
 \frac{}{\Gamma \vdash \text{Abstract} : \mathcal{A}} \quad \frac{}{\Gamma \vdash \text{Biblio} : \mathcal{B}} \quad \frac{\Gamma \vdash t : \text{String}}{\Gamma \vdash \text{Section } t : \mathcal{S}} \\
 \frac{\Gamma \vdash t : \text{String} \quad \Gamma \vdash a : \mathcal{A} \quad \Gamma \vdash b : \mathcal{B} \quad \Gamma \vdash ss : \text{List } (\mathcal{S})}{\Gamma \vdash (\text{Paper } t \ a \ b \ ss) : \mathcal{P}} \\
 \\
 \frac{\Gamma \vdash c : \mathcal{S} \quad \Gamma \vdash v : \mathcal{Q}}{\Gamma \vdash (\text{Write } c \ v) : \mathcal{J}} \quad \frac{\Gamma \vdash c : \mathcal{S} \quad \Gamma \vdash v : \mathcal{Q}}{\Gamma \vdash (\text{Review } c \ v) : \mathcal{J}} \\
 \\
 \frac{\Gamma \vdash c : \mathcal{A} \quad \Gamma \vdash v : \mathcal{Q}}{\Gamma \vdash (\text{Write } c \ v) : \mathcal{J}} \quad \frac{\Gamma \vdash c : \mathcal{A} \quad \Gamma \vdash v : \mathcal{Q}}{\Gamma \vdash (\text{Review } c \ v) : \mathcal{J}} \\
 \\
 \frac{\Gamma \vdash c : \mathcal{B} \quad \Gamma \vdash v : \mathcal{Q}}{\Gamma \vdash (\text{Write } c \ v) : \mathcal{J}} \quad \frac{\Gamma \vdash c : \mathcal{B} \quad \Gamma \vdash v : \mathcal{Q}}{\Gamma \vdash (\text{Review } c \ v) : \mathcal{J}} \\
 \\
 \frac{\Gamma \vdash p : \mathcal{P} \quad \Gamma \vdash is : \text{List } (\mathcal{J})}{\Gamma \vdash (\text{PTODO } p \ is) : \mathcal{L}}
 \end{array}$$

Figure 11.2: Typing rules for PTODO.

- 2) an enumerated type (CTy) to distinguish between components in the paper, representing sections, abstracts, and bibliography.

Notice how in Listing 11.1, Comp is indexed by CTy to distinguish at the type level different components. Using this implementation a TODO planner for academic paper planning can be constructed.

11.2.4 Interpretation Semantics

With the syntax, types, and implementation for PTODO defined, interpretation semantics can now be given that details how constructs from PTODO map to those in the NovoGRL. Figure 11.3 details these semantics.

The rules given in Figure 11.3 detail how each of the paper components, and paper itself, are transformed into a goal model in which the paper is represented by a root goal node, and each component is linked to the root node using an ‘and’ decomposition rule. Each of the TODO items are translated into a NovoGRL declaration pairing, the task node from the NovoGRL and an intentional link with a contribution value of Make linking the task node to the component being affected. When presented with a paper TODO list (PTODO), the resulting goal-graph that represents the paper itself

```

1 data CTy = STy | ATy | BTy
2
3 data Comp : CTy -> Type where
4   Sect : String -> Comp STy
5   Abst : Comp ATy
6   Bibl : Comp BTy
7
8 data Paper : Type where
9   MkPaper : (title : String) -> (abst : Comp ATy)
10            -> (bibl : Comp BTy) -> (sects : List (Comp STy))
11            -> Paper
12
13 data Item : Type where
14   Review : Comp ty -> (s : SValue) -> Item
15   Write : Comp ty -> (s : SValue) -> Item
16
17 data TODOList : Type where
18   MyList : String -> Paper -> List Item -> TODOList

```

Listing II.1: Initial implementation of the PTODO modelling language.

has the elements and declarations from the TODO items inserted. The result from this interpretation is a complete goal-graph and is evaluated using the NovoGRL evaluation semantics.

However, the types of the expressions on the RHS in Figure II.3 are not equivalent. A NovoGRL model has type \mathcal{M} (GModel), and expressions are lists of single instances with type \mathcal{L} (GLang INTENT or GLang STRUCT) or \mathcal{N} (GLang ELEM). This will affect how the interpreter will be constructed. How to deal with interpretation results will be important.

Further, the implementation of PTODO given in Listing II.1 is distributed over several data structures. This will further affect how the meta-model for PTODO can be modelled within the types. A question arises of: *How to distribute and combine the results in type?*

11.3 Lists of Dependent Types

When representing language expressions as an inductive ADT, a natural pattern is to collect a sequence (or set) of expressions in a container. One commonly used container is that of the List data type. However, in dependently typed languages the values detailed complicates the matter of the collection of values described using a dependent

$$\begin{aligned}
 \llbracket \text{PTODO} \rrbracket &: \text{PTODO} \rightarrow \text{GRL} \\
 \llbracket \text{Abstract} \rrbracket &= \text{Goal "Abstract" Unknown} & (11.1) \\
 \llbracket \text{Biblio} \rrbracket &= \text{Goal "Biblio" Unknown} & (11.2) \\
 \llbracket \text{Section } t \rrbracket &= \text{Goal } t \text{ Unknown} & (11.3) \\
 \llbracket \text{Paper } t \text{ a } b \{s_1, \dots, s_n\} \rrbracket &= \emptyset \uplus p \uplus^* \{\llbracket a \rrbracket, \llbracket b \rrbracket\} \uplus^* cs & (11.4) \\
 &\quad \uplus (p \wedge cs) \uplus (p \wedge \{\llbracket a \rrbracket, \llbracket b \rrbracket\}) \\
 &\text{where} \\
 &\quad p = \text{Goal } t \text{ Unknown} \\
 &\quad cs = \{\llbracket s_1 \rrbracket, \dots, \llbracket s_n \rrbracket\} \\
 \llbracket \text{Review } v \text{ e} \rrbracket &= (n, n \xrightarrow{\text{Make}} \llbracket e \rrbracket) & (11.5) \\
 &\text{where} \\
 &\quad n = \text{Task ("Reviewing" ++ e) } v \\
 \llbracket \text{Write } v \text{ e} \rrbracket &= (n, n \xrightarrow{\text{Make}} \llbracket e \rrbracket) & (11.6) \\
 &\text{where} \\
 &\quad n = \text{Task ("Writing" ++ e) } v \\
 \llbracket \text{PTODO } p \{i_1, \dots, i_n\} \rrbracket &= p \uplus^* es \uplus^* is & (11.7) \\
 &\text{where} \\
 &\quad (es, is) = \{(a, b) \mid (a, b) \leftarrow \llbracket i \rrbracket, i \leftarrow \{i_1, \dots, i_n\}\}
 \end{aligned}$$

Figure 11.3: Interpretation semantics for converting PTODO expressions into GRL constructs.

type. Collection of items implies that all items must have the same type, and as such must also have the same value. Such a limitation hampers the ability to construct abstract interpreters using list inductive ADTs, the interpretations in the type must also be collected. However, they are not all the same value.

11.3.1 A List of TODO Items

For example, Listing 11.2 lists a simple dependently typed ADT to represent items in a TODO list. Here, NovoGRL is used as the meta-model such that each TODO item is represented by a Goal element.

With the `Item` data structure if one wishes to collect a list of TODO items into


```

1 data Item : GLang ELEM -> Type where
2   Done : (title : String)
3     -> (desc : Maybe String)
4     -> Item (MkGoal title (Just SATISFIED))
5   TODO : (title : String)
6     -> (desc : Maybe String)
7     -> Item (MkGoal title Nothing)

```

Listing 11.2: A simple dependently typed ADT to represent items in a TODO list.

a collection such as `List`, the interpretation (the value contained within the type) of the collected expressions must be the same. For example, given a list of `TODO` items (detailed in Listing 11.3), the list `todos` will fail to type check as each of the elements within the list have different types due to their interpretation.

The first element interprets to: `MkGoal "Abstract" (Just SATISFIED)`. The second element to: `MkGoal "Types as (Meta) Modellers" Nothing`. Finally, the third to: `MkGoal "Introduction" Nothing`. The `List` data structure has a type that is parameterised by the *single* type given to all elements contained within the list. To collect a list of elements with type `Item` the value specified within `Item` must be *exactly* the same for each element. Further, when an enumerated type is used to provide scopes over a data structure, for example to model a type-system, the value of this enumeration will only add more points that will be fixed.

The question is: *How to collect lists of dependent types such that the value within the type can differ?*

```

1 todos : ?myTypeIs
2 todos = [ Done "Abstract" Nothing
3         , TODO "Types as (Meta) Modellers" Nothing
4         , TODO "Introduction" Nothing]

```

Listing 11.3: A list of `TODO` items with elements of type `Item`, with an unknown type.

11.3.2 `DList`: A Data Structure for Collecting Dependent Types.

To address the collection of values in types dependently typed languages such as `Agda` provide a means to existentially quantify values in a parameterised type, collecting the

value in the type as the structure is built. This is the `All` data type. During the research project, no such comparable structure existed within the standard library of Idris. This section introduces `DList` a comparable structure to `All` that allows for the collection of values at the type and value level. `DList` differs further from `All` in that `DList` is a ‘proof-carrying’ living and breathing list. `All` represents a predicate that is used as an external proof for a separate list.

```

1 data DList : (aTy : Type)
2           -> (elemTy : aTy -> Type)
3           -> (as : List aTy)
4           -> Type where
5   Nil    : DList aTy elemTy Nil
6   (::)   : (elem : elemTy x)
7         -> (rest : DList aTy elemTy xs)
8         -> DList aTy elemTy (x::xs)

```

Listing 11.4: Definition of `DList`, a data structure to allow collection of Dependent Types.

`DList` is a generalised cons-style ADT that allows for a *value* contained within the type of a dependent type to be collected at the type-level. All elements within the list come from the same family of indexed types and that the index within the type of the element can differ. With `DList`, the family of indexed types is constrained to a singular instance.

Listing 11.4 presents the definition of `DList`. In this definition: `aTy` is the type of the value contained within the list element type; `elemTy` is the type of the elements within the list; and `as` is the `List` containing the collected values within the type. Using `DList`, lists can now be constructed for dependent types, collecting a single value from within the type. Listing 11.5 illustrates how the list of `foos` from Listing 11.3 can be specified using `DList`.

The benefit of this approach is that a single library of operations involving `DList` data structures can be specified and developers can create lists of dependent types. Other approaches are more cumbersome and do not allow for generic list style operations to be provided for a data structure of type `DList`. Appendix C details these other approaches further. `DList` and other containers available for use in Idris have been made available online [dMH15b].

```

1 todos : DList (GLang ELEM) Item
2       [ MkGoal "Abstract" (Just SATISFIED)
3         , MkGoal "Types as (Meta) Modellers" Nothing
4         , MkGoal "Introduction" Nothing]
5 todos = [ Done "Abstract" Nothing
6         , TODO "Types as (Meta) Modellers" Nothing
7         , TODO "Introduction" Nothing]

```

Listing II.5: Using `DList` to collect a list of elements of type `Foo`.

Unfortunately, the `DList` data structure only collects a single value from the type. Dependent types that are parameterised using multiple elements must have all but a single value fixed. A possible work around would be to weaken the relations between the parameters in the type and allow arbitrary values to be collected in the type.

11.3.3 Relation to Abstract Interpreters

With `DList` comes the ability to collect the terms within a list of dependent types. Listing II.6 details a naïve representation of the actual list of `TODO` items modelled using `DList`.

`TODOList` is indexed by an instance of a goal-graph (`GModel`) that is populated using the list of elements obtained from the `DList` presented as an argument in the `MyList` constructor. However, this does not result in a correct `NovoGRL` model instance. Missing are the definitions of the tasks required to complete each goal item. Also missing is a means to specify contribution links between goals and the missing tasks. Further, this implementation will result in a disconnected graph and will not evaluate. Use of `DList` alone is not enough to add these features to this simple `TODO` list implementation. To address these features a second technique needs to be introduced.

```

1 data TODOList : GModel -> Type where
2   MyList : String
3         -> DList (GLang ELEM) Item es
4         -> TODOList (insertMany es emptyModel)

```

Listing II.6: A naïve modelling of a `TODO` List with `NovoGRL` as the meta-model.

11.4 Working with Interpretation Results

When working with values in the type for modelling, a crucial design decision is over how the interpretation is represented. The data type `TODOList` from Listing 11.6 used `NovoGRL` as the meta-model and required that the value in the type be the same type as the meta-model itself. This restricts what form the meta-model can take. This is not important when working with structurally isomorphic modelling languages, however, for structurally dimorphic languages this presents a problem.

When working with inductive ADTs the language constructs are being built up in both the domain language and the host language. If the resulting constructs in the abstract interpretation cannot be represented in the type of the domain language this will make combination of the abstract interpretation constructs harder to achieve. For instance, §11.1 detailed that when interpreting a `PToDo` instance to its `NovoGRL` representation, intermediate results needed to be calculated that were not of type `GModel`. These results represented lists of declarations *to be inserted* into a `GModel` instance. For example, Listing 11.7 shows how the `TODO List` example from the previous section (§11.3) can be extended with a data type to represent tasks. `Task` is indexed by a pair representing the interpretation of an `Action` to elements from `NovoGRL`. This interpretation is that of a `Task` element and an intentional links between the task element and the goal stored in the type representing items.

```

1 data Task : (GLang ELEM, GLang INTENT) -> Type where
2   Action : (title : String) -> (desc : Maybe String)
3         -> (value : SValue) -> (todo : Item e)
4         -> Task (Pair (MkTask title (Just value))
5                   (MkTask title (Just value) ==> e | MAKES))

```

Listing 11.7: A simple dependently typed ADT to represent tasks in a `TODO` list.

With `Task`, the data type for modelling `TODOList`'s can be improved to include these tasks lists. However, a problem now arises of how to collect the `NovoGRL` representations at the type level, and use them to construct a valid `GModel` instance. Naïvely, one might think to use `DList` to collect tasks into a list. Producing a parameter with type:

```

tasks : DList (GLang ELEM, GLang INTENT)
        (\(e,i) => Task (e,i))
        reprs

```

However, use of an anonymous function results in a highly verbose type signature for an instance of `DList`. A means to reduce this verbosity and to aid in the interpretation, collection, and processing of the results is the use of a custom data type. This data type is dedicated to the collection of the interpretation results for the expressions in the meta-modelling language. Each constructor in the data type can be used to store the interpretation (final or intermediate) for each expression in the host language. For example, the index of `Item` can be wrapped using a record `TaskRepr` with projections `getElem` and `getIntent` used to access the individual elements in the pair. `TaskRepr` subsumes the functionality required from a pair with a simple type. Listing II.8 presents a rewriting of `TODOList` using `TaskRepr`, and the definition of `TaskRepr`.

```

1 record TaskRepr where
2   constructor MkTRep
3   getElem    : GLang ELEM
4   getIntent  : GLang INTENT
5
6 data TODOList : GModel -> Type where
7   MyList : (name : String)
8           -> DList (GLang ELEM) Item es
9           -> DList TaskRepr Task rs
10          -> TODOList (insertMany (map getIntent rs)
11                      (insertMany (map getElem rs)
12                      (insertMany es emptyModel)))

```

Listing II.8: A revised modelling of `TODOList` with more complex types.

Unfortunately, use of `TaskRepr` results in a more complicated type. How the `GModel` instance is constructed is exposed. The list of `NovoGRL` model constructions, `rs`, needs to be processed and the results inserted within the `GModel` instance. Fortunately, a function can be used to abstract away the computation required. Listing II.9 demonstrates one such function: `buildModel`.

The use of custom data types and functions to work with interpretation results allows for greater control over the construction of the meta model. A natural extension to the use of custom data types is the convergence of all interpretation results into a single ADT. This way the actual meta-model values and types are not exposed in the type of the DSML. However, when modelling the DSML as an inductive ADT this will lead to ambiguity at the type level over which constructor should be used with

```

1 buildModel : List (GLang ELEM) -> List TaskRepr -> GModel
2 buildModel es ts = insertMany tis (insertMany tes modelAndGoals)
3   where
4     tis : List (GLang INTENT)
5     tis = map getIntent ts
6
7     tes : List (GLang ELEM)
8     tes = map getElem ts
9
10    modelAndGoals : GModel
11    modelAndGoals = insertMany es emptyModel
12
13 data TODOList : GModel -> Type where
14   MyList : (name : String)
15           -> DList (GLang ELEM) Item es
16           -> DList TaskRepr Task rs
17           -> TODOList (buildModel es rs)

```

Listing 11.9: A second revised modelling of TODOList with added functions.

which function, and which result should be inserted into what. The next section details a technique used to avoid such problems.

11.5 Type Threading

When modelling interpretation results in the type of an inductive ADT one must be able to distinguish between different kinds of constructors. How a constructor is interpreted may differ, not to mention the results themselves.

For example, imagine that both the `Item` (Listing 11.2) and `Task` (Listing 11.7 or Listing 11.11) ADTs were to be represented as a single ADT: `TODOList`. An enumerated type (`Ty`) distinguishes between constructors for items and tasks. Both original representations of these data structures have different interpretation results: `Task` is a pair of an element-intentional link pairing; and `Item` a single element. A single ADT (`IRes`) can be used to coalesce the different interpretation results together. However, it is not clear when using `IRes` at the type level which constructor is being referred to.

Type Threading is a technique in which a set of data types are indexed by the same enumerated type. Allowing for the view from above (type-level) to correspond to the view from below—value-level. This technique was seen in the definition of `SiFExpr` in Chapter 5 §5.4.3, and also in the construction of `NovoGRL`. Chapter 10

```

1  data Ty = TyITEM | TyTASK
2
3  data IRes : Ty -> Type where
4      IResItem : GLang ELEM -> IRes TyITEM
5      IResTask : GLang ELEM -> GLang INTENT -> IRes TyTASK
6
7  interpItem : String -> Maybe SValue -> IRes TyITEM
8  interpItem title sval = IResItem $ MkGoal title sval
9
10 interpTask : String -> SValue -> IRes TyITEM -> IRes TyTASK
11 interpTask t s (IResItem to) = IResTask elem link
12 where
13     elem : GLang ELEM
14     elem = MkTask t (Just s)
15
16     link : GLang INTENT
17     link = (elem ==> to | MAKES)
18
19 data TODOItem : (ty : Ty) -> IRes ty -> Type where
20     Done : (title : String) -> (desc : Maybe String)
21         -> TODOItem TyITEM (interpItem title (Just SATISFIED))
22
23     TODO : (title : String) -> (desc : Maybe String)
24         -> TODOItem TyITEM (interpItem title Nothing)
25
26     Action : (title : String) -> (desc : Maybe String)
27             -> (value : SValue) -> (todo : TODOItem TyITEM e)
28             -> TODOItem TyTASK (interpTask title value e)
    
```

Listing 11.10: A revised implementation of `Item` and `Task` as a single inductive ADT with interpretation results indexed by the same enumerate type to distinguished different kinds of constructors.

used the enumerated type `GTy` to ensure that the values from the presented domain language are correctly mapped to the constructs in `GCore`. This is a quintessential use of dependent types, detailing how the values used in the type can be leveraged to provide greater precision in describing programs. In this case the values in the types are used to constraint and enforce the relation between several different data structures when they are used together.

With the ability to thread types, `IRes` and the `TODOItem` can now both be indexed using the same enumerated type, `Ty`. This ensures that the right functions and interpretation results are used together. Listing 11.10 details the implementation of `TODOList`, `IRes` contains the interpretation results for both items and tasks, and these results can be distinguished through the value of type `Ty` specified in the type. The functions

`interpItem` and `interpTask` are used to perform the actual interpretation of values. Notice that as with Listing 11.11, use of functions to compute the interpretation result provides for more succinct code to be written in the type of `TODOItem`.

With `TODOItem`, the implementation of `TODOList` can now be updated. Listing 11.12 presents the revised implementation. As before, the interpretation of `TODOList` results in the construction of a `GModel` instance, with the function `buildModel` used to perform the actual building. Inspecting the constructor arguments for `MyList`, two `DList` types are used to model the list of items and tasks, and to collect the interpretation results. An anonymous function allows the `TODOItem` type (indexed by two parameters) to have the correct shape for `DList`, and to ensure that only the interpretation result is collected. The resulting `NovoGRL` model for this `TODO` list has, as its root, an element representing the root goal: The completion of all items in the `TODO` list.

11.6 Interpreter for PToDo

The formal definition and interpretation semantics for `PToDo` were given in §11.2. This chapter has discussed several techniques, and illustrated their uses of examples closely related to the `PToDo` language. This section details how the actual interpreter for `PToDo` is constructed, showcasing the use of all the techniques described in this chapter. The full implementation is not described here, and is provided in Appendix A, and made available online [dMH15e].

11.6.1 Representing the Types

The enumerated types `Ty` and `CTy` are used to index data types to model the type system of `PToDo`. `Ty` represents core things in the language—papers, `TODO` items, and paper components). `CTy` distinguishes between different aspects of a paper. Their implementation is:

```
data Ty = PAPERty | ITEMty | COMPTy
```

```
data CTy = STy | ATy | BTy
```

An alternative implementation would be to combine the types into a single ADT. However, as it will be shown the view of interpretation results need not be concerned with


```

1  buildModel : String
2      -> List (IRes TyITEM)
3      -> List (IRes TyTASK) -> GModel
4  buildModel n is ts = insertMany tis $ insertMany tes m
5      where
6      root : GLang ELEM
7      root = MkGoal n Nothing
8
9      es : List (GLang ELEM)
10     es = map (\(IResItem x) => x) is
11
12     toRoot : GLang STRUCT
13     toRoot = (root &= es)
14
15     m : GModel
16     m = insert toRoot $ insertMany (root::es) emptyModel
17
18     ties : List (GLang ELEM, GLang INTENT)
19     ties = map (\(IResTask e i) => (e,i)) ts
20
21     tis : List (GLang ELEM)
22     tis = map fst ties
23
24     tes : List (GLang INTENT)
25     tes = map snd ties
26
27  data TODOList : GModel -> Type where
28     MyList : (name : String)
29         -> DList (IRes TyITEM) (\res => TODOItem TyITEM res) is
30         -> DList (IRes TyTASK) (\res => TODOItem TyTASK res) ts
31         -> TODOList (buildModel name is ts)
    
```

Listing II.11: A revised implementation of `TODOList` using `TODOItem` from Listing II.10.

the precise notion of a component as it gets interpreted to the same result. Further, one could also parameterise the constructor `COMPty` to store a value with type `CTy`. However, this will increase the complexity of the implementation unnecessarily.

11.6.2 Interpretation Results & Functions

Figure II.3 details the interpretation semantics for `PTODO`. Each of the core language constructs are either going to be interpreted into an element from `NovoGRL`, a complete `GModel` instance, or a pairing between an element and a intentional link. `InterpRes` (Listing II.12) is used to represent these results, and uses `Ty` to distinguish at the type level the different constructors.

```

1 data InterpRes : Ty -> Type where
2   ResComp  : GLang ELEM          -> InterpRes COMPTy
3   ResPaper : GModel             -> InterpRes PAPERty
4   ResITEM  : GLang ELEM -> GLang INTENT -> InterpRes ITEMty

```

Listing 11.12: An ADT to collect intermediate interpretation results for PTODO.

With `InterpRes`, functions can now be defined to perform the interpretation. The full details of which are provided in Appendix A. Here only the type signatures are given.

The first function, `interpComp`, interprets a paper’s components into NovoGRL elements. Corresponding to Rules 11.1, 11.2 & 11.3 from Figure 11.3. The next function converts the paper (an abstract, a bibliography, and a list of sections) into a NovoGRL `GModel` instance—Rule 11.4.

```
interpComp : String -> InterpRes COMPTy
```

Examining the type signature for `interpPaper`, a question arises over which argument refers to which interpretation result w.r.t. the paper’s components. At the type level it is not clear if the argument labelled `a` is the abstract or the bibliography. This is a limitation of the design of this interpreter, if the developer has passed in the wrong arguments to `interpPaper` an incorrectly constructed `GModel` will arise.

```
interpPaper : String
             -> InterpRes COMPTy
             -> InterpRes COMPTy
             -> List (InterpRes COMPTy)
             -> InterpRes PAPERty
```

However, this limitation can be addressed. Extending `Ty` to represent individual paper components would be beneficial, and allow for more precision to be specified in the type signature. This would require the addition of extra constructors to `InterpRes` and more functions to perform the required interpretation. For each value in `Ty` a function needs to be constructed. An alternative approach would be to parameterise `InterpRes` further with `CTy`. Implying that both `ITEMty` and `COMPTy` can also be paired with a `CTy` value. For this example, this is not an appropriate solution, `TODO` items do not have a bibliography, abstract, nor sections.

The function `interpTODO` interprets individual `TODO` items, corresponding to Rule 11.5 and Rule 11.6 from Figure 11.3. Unlike the definition of `interpPaper`, no

disambiguation is needed to distinguish between different components. The final function is `interpTODOs` that constructs the `GModel` instance representing a TODO list.

```
interpTODO : String
            -> SValue
            -> InterpRes COMPTy
            -> InterpRes ITEMty
```

The `interpTODOs` function, uses the interpretation results for a paper, and combines it with the interpretation results for each item in the TODO list. This function provides an implementation of Rule II.7.

```
interpTODOs : InterpRes PAPERTy
            -> List (InterpRes ITEMty)
            -> GModel
```

11.6.3 Data Structures

The functions from the previous section can now be used to enhance the data structures, originally presented in §II.2.3. Listing II.13 presents these new functions. Now, the interpreter for converting the TODO list to a `GModel` has been distributed and embedded through the types of the program. Notice, that in the revised types for `PTODO`, the use of `DList` in `Paper` and `TODOList` to collect the interpretations of paper components. Further, the values within each data type are used to populate the `GModel`. As with the *Well-Typed (Abstract) Interpreter*, a valid `TODOList` instance can only be constructed if a valid `GModel` instance is also constructed. Thus, providing stronger correctness-by-construction guarantees for `PTODO` models.

11.6.4 Model Evaluation

With the ability to define models, a question arises over how they are to be evaluated. Using the `NovoGRL` as the meta-model, allows for the *existing* evaluation semantics to be used on the resulting meta-model. With `PTODO`, the meta-model can be made accessible by bringing the meta-model down into the value level, as follows:

```

1 data Comp : InterpRes COMPTy -> CTy -> Type where
2   Sect : (t:String) -> Comp (interpComp t          ) STy
3   Abst :                Comp (interpComp "Abstract") ATy
4   Bibl :                Comp (interpComp "Biblio"  ) BTy
5
6 data Paper : InterpRes PAPERty -> Type where
7   MkPaper : (t : String)
8             -> Comp a ATy
9             -> Comp b BTy
10            -> DList (InterpRes COMPTy) (\x => Comp x STy) ss
11            -> Paper (interpPaper t a b ss)
12
13 data TODO : InterpRes ITEMty -> Type where
14   Review : (c : Comp a ty)
15           -> (s : SValue)
16           -> TODO (interpTODO "Review " s a)
17
18   Write  : (c : Comp a ty)
19           -> (s : SValue)
20           -> TODO (interpTODO "Writing "s a)
21
22 data TODOList : GModel -> Type where
23   MyList : String
24           -> Paper m
25           -> DList (InterpRes ITEMty) TODO ts
26           -> PaperToDos (interpTODOs m ts)

```

Listing 11.13: A revised implementation of PToDo with the interpreter to NovoGRL distributed and embedded through out the types.

```

evalTODO : TODOList m -> List GoalNode
evalTODO {m} l = evalModel m Nothing

```

The type-level value is made accessible using Idris' built in mechanisms for working with implicit values¹. The function `evalTODO` returns the list of goal nodes in the `GModel` instance `m` with their satisfaction value. For this example implementation, nothing further is computed with the results. However, with further engineering the results from the evaluation can be transformed back into the values of `Item` and reported back to the user accordingly.

¹*Programming in Idris* details this further [Idris].

11.7 Discussion

This section presents a discussion over the techniques presented in this chapter and areas of future work.

11.7.1 Direct or Abstract Interpreters

This thesis has demonstrated two different styles of construction for DSMLs within a dependently typed language. Both of these styles are provided in the `SIF` evaluator—see Chapter 5 §5.6.3.

The first, is a direct style of construction and requires for the separate implementation of the DSML from the interpretation to the meta-model itself. Naturally, this separation allows for different meta-models and interpretation semantics to be considered for a DSML. However, this separation only allows for the correctness of construction guarantees between a DSML and meta-modelling language to be checked once the model has been constructed and checked explicitly. Mistakes in model construction are now not guaranteed to be discovered unless the model is actively checked.

The second technique requires the meta-model to be constructed directly within the types of the DSML itself. This technique provides stronger guarantees between the host and domain language. The meta-model instance is constructed at the same time as the DSML. Any mistakes made when constructing the two languages will be detected at the same time.

The abstract interpretation technique also allows for the meta-modelling language to be presented at the type-level. Allowing for a modelling language to be used as a host language when both languages are structurally dimorphic. Further allowing for the meta-modelling language to be made available for use in programs where use of the meta-modelling language might not be apparent, or the implementation of an interpreter non-obvious. The resulting DSML can differ in both syntax and structural properties, but can nonetheless be modelled using the host language.

For example, compare the syntax and structural properties of `SIF` compared to `NovoGRL`. They differ considerably, a `GModel` is a directed graph, and `SIF` is structurally the union of two trees. However, both `SIF` and `NovoGRL` are two goal-oriented languages. A more interesting use of the GRL may be the representation, at the type level, of access control to data based on environmental conditions. The goal (file access) might be restricted based on several system environmental resources, such as current

user, time, user's permissions *et cetera*. The values of these resources can be used to populate a predefined goal model as a strategy, and if the goal is satisfied then access could be granted.

Although the DSML and meta-modelling language are technically inseparable the DSML can still be defined separately from the meta-model. The reference implementation for SIF (Chapter 5 §5.4.4) details how the pattern `ABSTRACT FACTORY` was combined with *type-threading* and interfaces to separate the abstract syntax of the language from the meta-model.

11.7.2 Efficacy of InterpRes

The use of a bespoke indexed ADT (`InterpRes`) to capture intermediate interpretation results resulted in `PToDo` becoming over engineered. The use of distinct data structures was sufficient, and not only made the index `Ty` redundant but also the pervasive use of `InterpRes`. The result of interpreting the types `Paper` and `Comp` w.r.t. to the constructors in those types did not differ. When using the *types as (meta) modellers* approach, pervasive use of an `InterpRes` like ADT is not required.

However, in the case that an inductive ADT is used, provision of `InterpRes` is crucial. As noted in §11.4, interpretation results when used at the type level must have the same type. Otherwise, the result cannot be stored. For example, the 'Abstract-Builder' backend for SIF made use of abstract interpretation over an inductive ADT. Here `InterpRes` was used to collect the intermediate results (GRL statements) for processing. These intermediate results varied from lists of declarations, to a single declaration, and to the meta-model instance itself. Further, look at the use of the GRL to model `PToDo`. The GRL EDSL is indexed by `GTy`, an enumerated type, but the resulting model will be of type `GModel`: Two different types.

Another aspect to consider is that in Idris, types are first class constructs. Use of this option has not been explored in the construction of DSMLs using the *types as (meta) modellers* approach. However, it is reasonable to suggest that language expressions for a DSML can be additionally indexed over the type of the intermediate interpretation results, and that this type be used to represent the interpretation result within the type of the data structure. This resembles how `DList` allows for collection of values at the type level. How this approach would work is left for future study.

11.7.3 Collecting Dependently Typed Values

This chapter has introduced `DLiSt` and the reasons behind its construction. When working with dependent types `DLiSt` has proven useful in simplifying the work associated with collecting lists of dependently typed values.

In the area of containers for dependent types, future work will be to explore how other more container types can be constructed, for example: Balanced Binary Trees, Dictionaries, and Graphs. Key to the construction of these data types will be how the values in the types can be collected such that the container's structure can be preserved and the values made accessible at the type level.

11.7.4 Resource Usage

An aspect to consider with these two techniques is the lifetime and use of the meta-model construct itself. Both `PTODO` and `SIF` are based on the GRL, and make use of the meta-model for evaluation purposes.

A side effect of modelling information in types is that the extra information in the type will impact memory usage and execution speed. Dependently typed programs will be slower and larger to run if this information is not taken care of. Erasure, is a technique that identifies information used during compile time that is irrelevant during runtime [Miso8]. This irrelevant data is then erased when producing the program executable.

With the *types as (meta) modellers* approach the meta-model can be used by the program to present evaluation results. The DSML is not evaluated directly. For example, the `evalTODO` function from §II.6.4 drops the meta-model down to the value level for use. It is this 'drop' that makes the meta-model relevant. Programs that make use of the meta-model at the value level will not have this information erased.

11.7.5 Correctness Guarantees

The *types as (meta) modellers* approach uses a variety of techniques to provide correctness guarantees over a DSML and its interpretation into the meta-modelling language. These approaches are realised using dependent types.

Well-typed interpreters are used to ensure that only well-formed and well-typed expressions are constructed. For `NovoGRL`, this approach was used to also ensure correct transformation between the semantic domains. Modelling type-systems with

enumerated types also allowed for *type-threading* and assurances to be made in connecting expressions to intermediate interpretation results.

Construction of the meta-model directly in the type ensures that a valid DSML instance can only be constructed if the corresponding meta-model can also be constructed. An important aspect to consider with such modelling is: *When is the correctness of a model checked?* For both P`TODO` and S`IF` the intermediate results of interpreting several expressions are declarations that *are* inserted into a model. This insertion happens at a later point past the definition site. The correctness check for the meta-model only occurs when these declarations are being inserted into the model itself. However, this is a property resulting from the declarative nature of how the NovoGRL was constructed. These guarantees cannot be made *a priori* as the complete modelling information is not known in advance.

An aspect not considered is the correctness of the resulting meta-model. The approach detailed in this chapter only provides guarantees towards the shape and transformation of models. The arising semantics post transformation are not reasoned upon. This is a result of implementing the languages as EDSLs in Idris and not restricting the typing environment of the languages to variables only defined within the context of the language expressions. The current implementation choice allows for *free variables* to be inserted into the model. The solution to this problem is to also model the typing environment directly within the modelling of the languages. The *Well-Typed Interpreter* from The Idris Community [Idr15] details how such a restriction can be implemented.

11.8 Summary

This chapter has demonstrated and discussed how the approach of *Well-Typed (Abstract) Interpreters* can be applied to modelling and provide stronger construction guarantees between a DSML and its meta-modelling language. However, this approach does not provide guarantees over the correctness of the resulting model semantics, semantically incorrect models can still be constructed. How this is to be addressed is left for future work.

Meta-modelling languages are not always structurally equivalent to that of the domain model. The techniques presented in this chapter can be used to bridge the gap between the two structures. Especially when pairing declarative modelling languages

(e.g. NOVOGRL) with inductive languages (e.g. SIF), or those whose expressions are distributed across several data types—PTODO.

Further, this work has investigated the construction of list oriented containers for dependently typed values. The `DList` data structure was developed to allow the values in a collection of types to be collected at the type level. The shape of the containers was mirrored in the value in the type. Future work opens up to the area of constructing dependently typed containers.

CONCLUSION

Design Patterns are supposed to be the well-described, well-tested, well-evaluated pairing between a problem and a solution for a given context. Unfortunately, it is the case that the patterns being developed, presented, and published are not well-described, not well-tested, and are sometimes not ‘true’ patterns. This thesis has seen the design and introduction of tools and techniques for supporting and enabling machine checkable design pattern documents. This thesis concludes with a retrospective of the research hypothesis presented in Chapter 1 and the approach taken to address the hypothesis by the contributions. This thesis ends with a look at future research directions, ostensibly asking: *What’s next?*

12.1 Language-Oriented Design of DSMLs

Existing work saw the use of the GRL to reason about design patterns as requirements models. However, use of the GRL is complicated by its pictorial language constructs and inability to be re-domained to support domain specific concepts such as those for design patterns. Through adoption of existing techniques from programming language theory a formal specification for a subset of the GRL was developed—Chapter 10. This specification treated the GRL as a declarative language for the construction of goal-graphs. The language-oriented specification presented an abstract syntax and type-system to replace the use of UML.

The resulting formal specification, following programming language theory, was used as the host language when detailing interpretation semantics from a domain specific language to the GRL itself. The link between a domain language and its meta-modelling language can now be described through language transformations. This was demonstrated with various minor modelling languages throughout this thesis: GRL in Chapter 10 §10.7; PML in Chapter 10 §10.6; and PTODO in Chapter 11 §11.2. More importantly this approach was used for the design of SIF itself—Chapter 5.

With this language-oriented approach, clear and precise language descriptions are presented, together with clear transformation descriptions. DSML design has been enhanced using these concepts from programming language theory.

12.2 Better Implemented DSMLs

Given a language oriented design for DSMLs specification, dependent types can be used to provide greater guarantees between DSMLs and their meta-modelling language. The rich and expressive type-system in a dependently typed language allowed for stronger guarantees to be made in the implementation of both the DSMLs and the meta-modelling language that followed their formal descriptions. By embedding a domain specific type system within the data structures representing these languages, greater and succinct reasoning about language expressions could be made: *correctness-by-construction*.

Further, dependent types allowed for the transformation of a DSML to its meta-modelling language to be implemented as a verified staged interpreter based on the *Well-Typed Interpreter* approach. This was seen in the design of NovoGRL (Chapter 10) where direct transformation occurred, and in the design of SIF where this transformation occurred between expressions at the value and type level.

However, the link between a DSML and its meta-modelling language can be made stronger still through the *types as (meta) modellers* approach presented in Chapter 11. When building expressions in the domain language, the corresponding expressions in the meta-modelling language are also constructed in the type of the domain language. Structurally correct domain models can only be constructed if the resulting meta-model is also structurally correct. This was the construction method of SIF, and was demonstrated in the construction of PTODO in Chapter 11.

Further, the implementation of SIF was parameterised by the context of operation—

Chapter 5 §5.4.3. This enforced, at the type level, that a problem and solution can only be paired if they are indexed by the same domain. Additionally, dependent types were used in the implementation of SIF to facilitate the ‘hot-swapping’ of meta-modelling through implementation of the ABSTRACT FACTORY pattern—Chapter 5 §5.4.4. This allowed for direct and abstract interpretation constructions to be explored without affecting the infrastructure common to all backends.

12.3 Machine Checkable Design Patterns

Chapter 5 introduced SIF, a GOML for modelling design patterns as goal-oriented requirement models and has been designed as a DSML using as its meta-modelling language NOVOGRL. Allowing for formal modelling of design patterns using concepts specific to patterns but ultimately relying on a more general purpose modelling language to evaluate the models. Chapter 5 §5.5 presented the results of using SIF to reason about existing patterns determining the suitability of the presented solution to solve the presented problem.

Patterns are not *just* requirements models, patterns are well-written and presented design documents. FREYJA is an active pattern document schema presented in Chapter 6 that allows not only for a pattern template to be presented but also for a SIF model to be embedded directly within the metadata of the pattern document itself, and made accessible programmatically. These design pattern documents are not only machine readable but also machine checkable.

The machine checkable aspect of these documents was demonstrated through introduction of the FRIGG tool in Chapter 7. This utility was designed to facilitate machine aided interaction with pattern documents allowing users to: compute various metrics for a pattern document; produce pattern documents in various output formats; query the document; and evaluate the embedded SIF model to check for satisfaction. Although, the FRIGG tool is not as feature rich as existing work it nonetheless demonstrates how machine checkable design pattern documents can be used.

12.4 Better Pattern Evaluation and Publication

Machine checkable design pattern documents can enhance Design Pattern Engineering for the creation of new patterns. Chapter 8 introduced the PREMES evaluation framework that provides demonstrable and reproducible reporting on the quality of a design pattern document. Several indicators for pattern quality were identified and presented in Chapter 8 §8.3. When combined with tailorable testing techniques, a PATTERN REPORT CARD (Chapter 8 §8.4) can be produced. These report cards provide summative and formative feedback on the quality of a design pattern. The PATTERN REPORT CARDS element of PREMES was used to evaluate several known patterns attesting to their quality both good and bad—Section 8.6.

Anecdotal evidence indicated that when performed by hand the PREMES framework is cumbersome. However, the FRIGG tool and FREYJA schema demonstrate (see Chapter 9) how machine checkable design patterns can help with framework execution, and also towards pattern publication. PREMES and SIF are linked through FREYJA, as the schema allows for evaluation data to be stored as metadata inside, or directly calculated from, the design pattern document itself. Further, the FREYJA encoding also presents well-defined transformations of the pattern document to other publication formats. How machine checkable design patterns can help in pattern application was not explored.

12.5 Linked Concerns in Pattern Engineering

Design patterns are more than just software artefacts, they are *design documents*. This thesis has presented a pantheon of tooling (SIF, FRIGG, FREYJA) to support and enhance design pattern engineering through machine checkable design pattern documents. Dependent types were used to aid in the correctness of language and tool construction.

With the presented tooling, pattern engineers (and auditors) have a means to interact with pattern documents. Pattern auditors have a tool to aid in execution of the PREMES framework; Pattern writers have a tool to convert pattern documents into alternative formats for publication; and Pattern users have a tool to view and interact with patterns, allowing for inspection of document subsections and arbitrary querying

of the document's contents.

The tutorial presented in Chapter 9 demonstrates how several of the stages in design pattern creation can be brought together and each stage linked. Specifically, Figure 9.1 and Figure 1.2 illustrates how the thesis contributions bring together pattern creation, evaluation, and publication. Unfortunately, this thesis did not investigate pattern identification, this is still a concern to be linked in. Nor did this thesis investigate the use of machine checkable design pattern documents as part of the *Application* stage of pattern engineering. There is still future work still to consider.

Regardless, of these deficiencies the work presented in this thesis does show how the concerns can be linked, and made more robust through the introduction of tooling to aid in the design, writing, and evaluation of design patterns.

12.6 Future Work

The chapters detailing thesis contributions already detailed potential areas of future work. This thesis concludes by highlighting several of those areas further.

12.6.1 Improve Accuracy of Sif Models

The current language specification for SIF does not support fine-grained modelling of problems nor specifications. How requirements, traits, and properties interact with each other cannot be modelled. Further, the quantitative values used for detailing satisfaction and contribution are too obtuse. This lack of detail hampers the ability of accurate pattern models to be constructed. With greater specification will come greater accuracy.

Although SIF has this restriction, the NovoGRL language does not. Future work will be to improve the accuracy of modelling using SIF by revisiting the mapping of concepts from SIF to NovoGRL.

However, a second direction will be to look the efficacy of using NovoGRL. The GRL was used due to its previous use in modelling design patterns. Alternate GOMLs such as i^* and Tropos present extra modelling concepts for richer goal models to be constructed. Future work will be to look at how SIF could be re-targeted, and the domain specific concepts presented mapped onto, these languages.

12.6.2 Modelling Pattern Languages

The engineering of pattern languages was not considered in this thesis. A second avenue of research is the design and construction of a modelling language for reasoning about patterns *and* pattern languages. Primarily, investigating how patterns can be composed and combined. Heyman [Hey13] already investigated the composition of design patterns as software artefacts as part of a formal analysis of software architectures. In essence this details the construction of pattern languages. Future work will be to investigate how the SIF language can be extended to reason about pattern languages.

12.6.3 Machine Checkable Pattern Application

The contributions towards machine checkable design patterns does not support checking of pattern application. Behavioural and structural models used to represent a solution are made accessible within FREYJA.

Existing work has already shown how access to these models can enhance the use of design patterns—Chapter 2 §2.6. Future work will be to investigate how machine checkable pattern application can be ascertained, and greater access to the underlying models as well. This will also include investigation of how the properties and traits within a SIF model can be linked to the model constructs presented.

12.6.4 Efficacy and Autonomic use of Premes

A lacking aspect in this thesis is that the efficacy and usability of PATTERN REPORT CARDS and the PREMES framework was not explored in greater detail. This efficacy and usability should be explored. Part of the framework requires selection of grade mappings, and the assignment and calculation of weighting values. Such a selection will affect the calculation of the final score. Chapter 9 demonstrated how the satisfaction values from the GRL can be used to generate these grades when provided with a mapping from the qualitative data to a quantitative value. Further, the usability of the presented framework was not investigated.

12.6.5 Domain Modelling

A novel feature of NovoGRL is the ability to re-target the semantics to other domains. In effect allowing for the GRL to be re-skinned. Future work will be to investigate

and target other domains were goal-modelling would be useful. For example, goal-modelling has been used for modelling the security of socio-technical systems. Novo-GRL could be re-targeted to this domain to allow for better and more domain friendly modelling.

A secondary re-targeting would be the further exploration of the *types as (meta) modellers* approach to further enhance the link between domain model and meta-model. The techniques employed do not provide guarantees towards model evaluation, only structurally correct models can be constructed. Future work will be to explore how guarantees towards evaluation semantics can be given.



ELECTRONIC APPENDICES

Accompanied with this thesis are several electronic appendices providing copies of the: software developed; example Domain Specific Modelling Languages created; example model instances; and the raw data from the PREMES evaluation. The contents are:

Directory	Description
tutorial	The SIF models, configuration files, and pattern documents created as part of the tutorial—Chapter 9.
dsm1-examples	The DSML examples from Chapter 10 in which NovoGRL is used to construct modelling languages for planning academic paper writing.
premes-results	The raw results and utilities used as part of the PREMES analysis from Chapter 8.
sif-prelude	A copy of the current SIF prelude.
software	A directory containing the software projects representing the main thesis contributions: SIF, NovoGRL, FREYJA, and FRIGG.
dependencies	A directory containing the Idris projects that support the main thesis contributions.
idris	A directory containing the version of Idris that the thesis contributions compile with.

GRL FORWARD EVALUATION ALGORITHM

There are no standardised evaluation algorithms detailed in the official specification for the GRL [UTN12]. Rather, the standard details several exemplary algorithms for how models can be evaluated for satisfaction. These algorithms were also detailed in Amyot et al. [Amy+10]. Common to all algorithms is the definition of a *Strategy* that represents a list of nodes in the model and a predetermined satisfaction value. One of the exemplary algorithms presented was a forward propagation algorithm for determining goal satisfaction in which a predetermined set of nodes were given an initial satisfaction value. As part of the NOVOGRL framework a variant of this algorithm was implemented that took into account the subset of the GRL that was implemented. This appendix details the tailored algorithm in operation only. Amyot et al. [Amy+10] contains a more detailed description of the algorithm and the reasoning behind its specification.

B.1 Overview

The algorithm operates by visiting each node in the model, and calculates a satisfaction value for the node if and only if the node's children are also satisfied. A queue is used to keep track of the nodes that need to be visited, with nodes being removed from the queue once they have been satisfied. The traversal is detailed in Algorithm B.1. To ensure that model evaluation is total and guaranteed to terminate, a model is valid for evaluation if all leaf nodes in the model have an initial satisfaction value. This is formally presented in Definition 38. Prior to satisfaction evaluation, a node's initial satisfaction value is set either through a strategy or through a default node value of

None.

Definition 38 (Evaluation Ready). *Given a goal graph $M = \langle gs, ls \rangle$ where $gs = \{g_1, \dots, g_n\}$, $g_i \in \mathcal{N}$ and $ls = \{l_1, \dots, l_n\}$, $l_i \in \mathcal{L}$. Let $leafs \subset gs$, be the leaf nodes in M , and $rest \subset gs$ where $leafs \cup rest = gs$. M is said to be evaluation ready if:*

$$\forall n \in leafs, isSatisfied(n) \equiv True \vee getSValue n \equiv None$$

Algorithm B.1: GRL Model Satisfaction

Data: The model $m = \langle gs, rs \rangle \in \mathcal{M}$
Result: $\{\langle n_1, q_1 \rangle, \dots, \langle n_n, q_n \rangle\}$ the nodes from m together with satisfaction values.

```

1 next  $\leftarrow$  enqueue*(gs,  $\emptyset$ )
2 while next  $\neq$   $\emptyset$  do
3   c  $\leftarrow$  dequeue(next)
4   if  $\neg isSatisfied(c)$  then
5     cs'  $\leftarrow$  {isSatisfied(x) | x  $\leftarrow$  children(c)}
6     if  $\forall t \in cs, t \equiv True$  then
7       s  $\leftarrow$  calcEval(c)
8       update c with s in m
9     else
10    enqueue(c, next)
11 return { $\langle v, getSValue(v) \rangle$  | v  $\leftarrow$  vertices(m)}

```

B.2 Calculating Node Satisfaction

Recall from the definition of a goal-graph given in Chapter 10 §10.2.1 that during goal-graph construction the intentional edges were reversed during insertion into the graph. Thus, by design, all edges originating from a node n go to the children of n in the GRL model. The satisfaction value for a node is calculated through combination of the satisfaction results for both the intentional and decomposition edges. This combination is detailed in Algorithm B.2

B.2.1 Decomposition Edge Satisfaction

Given a node $n = \langle e, t, q, sTy \rangle$ in a goal-graph $M = \langle gs, ls \rangle$, the decomposition satisfaction value for n is determined by weighted comparison of the child nodes. The comparison is different for the three kinds of decomposition value. The calculation function is presented in Function B.3.

Function B.2: calcEval**Data:** let $n \in \mathcal{N}$ be the node to be evaluated.**Data:** let $m \in \mathcal{M}$ be the associated goal-graph.**Result:** A satisfaction value: $s \in \mathcal{Q}$.

```

1  $d \leftarrow \text{calcDecomp}(n, m)$ 
2  $c \leftarrow \text{calcContrib}(d, n, m)$ 
3 return  $c$ 

```

Function B.3: calcDecomp**Data:** $n \in \mathcal{N}$ **Result:** $q \in \mathcal{Q}$

```

1  $cs \leftarrow \{c \mid \text{isDecompEdge}(c), c \leftarrow \text{children}(n)\}$ 
2 if  $cs \neq \emptyset$  then
3    $cs' \leftarrow \{\text{getSValue}(c) \mid c \leftarrow cs\}$ 
4   switch  $\text{getDecompTy}(c)$  do
5     case  $\wedge$  do return  $\text{getDecompAnd}(cs')$ 
6     case  $\vee$  do return  $\text{getDecompIOR}(cs')$ 
7     case  $\oplus$  do return  $\text{getDecompXOR}(cs')$ 
8 else
9   return  $\text{None}$ 

```

And Decomposition

The ‘AND’ comparison is the *maximum* satisfaction value from the list of satisfaction values from the child decomposition nodes, calculated using the following precedence rules.

$$\text{Denied} < (\text{Conflict} \equiv \text{Undecided}) < \text{wDenied} < \text{None} < \text{wSatisfied} < \text{Satisfied}$$
Function B.4: getDecompAND**Data:** $qs = \{q_1, \dots, q_n\}$ where $q_i \in \mathcal{Q}$ **Result:** $\text{res} \in \mathcal{Q}$ such that res is the minimum value in $\{q_1, \dots, q_n\}$ according to the AND precedence rules.

```

1  $\text{res} \leftarrow \text{foldr}((\lambda x, y \rightarrow \text{max}^{\text{and}}(x, y)), \text{Satisfied}, qs)$ 
2 return  $\text{res}$ 

```

OR Decomposition

The ‘IOR’ comparison is the *maximum* satisfaction value from the list of satisfaction values from the child decomposition nodes, calculated using the following precedence rules:

Denied < wDenied < None < wSatisfied < (Conflict \equiv Undecided) < Satisfied

‘XOR’ comparison is calculated using the same precedence rules as for ‘IOR’ but takes the minimum value from the list presented.

B.2.2 Intentional Edge Satisfaction

Unlike decomposition edges, intentional edges are weighted with a contribution value that denotes the magnitude of the effect that one node has on another. The calculation of a nodes satisfaction value is determined through weighted analyses of these edges. It is also here that the satisfaction value for decomposition edges are taken into account.

Function B.5: calcContrib

Data: $d \in \mathcal{Q}$
Data: $n \in \mathcal{N}$
Result: $q \in \mathcal{Q}$

```

1  $cs \leftarrow \{c \mid \neg \text{isDecompEdge}(c), c \leftarrow \text{children}(n)\}$ 
2  $wcs \leftarrow \{\text{weightedContribution}(c) \mid c \leftarrow cs\}$ 
3  $counts \leftarrow \text{adjustCounts}(\text{append}(d, wcs))$ 
4 if noUnknown from counts > 0 then
5   | return Unknown
6 else
7   |  $x \leftarrow \text{cmpWSandWD}(counts)$ 
8   |  $y \leftarrow \text{cmpSatAndDen}(counts)$ 
9   | return combineContribs( $x, y$ )

```

Function B.5 outlines the complete steps. The first step is to determine the set of weighted contribution values for the edges. The `weightedContribution()` is implemented as a lookup table presented in Table B.2. Using these values and the decomposition satisfaction value, a count is then taken for each possible satisfaction value.

If there is more than one ‘unknown’ value then the satisfaction for the node is reported as Unknown. Otherwise, the final satisfaction value is determined through comparison of the number of weakly satisfied and weakly denied values, and satisfied and denied values. These comparisons are outlined in Functions B.6 & B.7, and their comparison in the lookup table presented in Table B.1.

Function B.6: cmpWSAndWD**Data:** The tuple counts = $\langle \text{noSat}, \text{noWeakS}, \text{noWeakD}, \text{noDen}, \text{noKnown} \rangle$.**Result:** A satisfaction value $q \in \mathcal{Q}$

```

1 switch counts do
2   case noWeakS > noWeakD do return wSatisfied
3   case noWeakD > noWeakS do return wDenied
4   case noWeakS  $\equiv$  noWeakD do return None
5   otherwise do return Unknown

```

Function B.7: cmpSatAndDen**Data:** The tuple counts = $\langle \text{noSat}, \text{noWeakS}, \text{noWeakD}, \text{noDen}, \text{noKnown} \rangle$..**Result:** A satisfaction value $q \in \mathcal{Q}$

```

1 switch counts do
2   case noSatis > 0  $\wedge$  noDen > 0 do return Conflict
3   case noSatis > 0  $\wedge$  noDen  $\equiv$  0 do return Satisfied
4   case noDen > 0  $\wedge$  noSatis  $\equiv$  0 do return Denied
5   case noSatis  $\equiv$  0  $\wedge$  noDen  $\equiv$  0 do return None
6   otherwise do return None

```

	Denied	Satisfied	Conflict	None
Weakly Denied	Denied	wSatisfied	Conflict	wDenied
Weakly Satisfaction	wDenied	Satisfied	Conflict	wSatisfied
None	Denied	Satisfied	Conflict	None

Table B.1: Look-Up table for combineContribs. Column index is for the result of cmpSatAndDen. Row index is for the result of cmpWSandWD.

	Make	Help	Some Positive	Unknown	Some Negative	Hurt	Breaks
Denied	Denied	wSatisfied	wSatisfied	None	wSatisfied	wSatisfied	Satisfied
Weakly Denied	wDenied	wSatisfied	wSatisfied	None	wSatisfied	wSatisfied	wSatisfied
Weakly Satisfied	wSatisfied	wDenied	wDenied	None	wDenied	wDenied	wDenied
Satisfied	Satisfied	wDenied	wDenied	None	wDenied	wDenied	Denied
Conflict	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
None	None	None	None	None	None	None	None

Table B.2: Look-Up table for weightedContribution. Column index is for the contribution value. Row index is for the Satisfaction value.



COLLECTING DEPENDENT TYPES: ALTERNATIVE APPROACHES

Chapter 11 §11.3 introduced a cons-style data structure for collecting dependent types. This chapter details alternative approaches that are not as flexible as that provided by `DList`. As a reminder the aim of these approaches is to allow for a set of elements of the dependent type to be collected into a list. The dependent type is:

```
data FTy = A | B | C

data Foo : FTy -> Type where
  ||| Strings
  FStr   : String -> Foo A
  ||| Naturals
  FNat   : Nat -> Foo B
  ||| Pairs
  FPair  : Foo A -> Foo B -> Foo C
```

and the list to be constructed is:

```
foos : ?myTypeIs
foos = [ FStr "Hello World"
        , FNat 42
        , FPair (FStr "Hello" (FNat 34)) ]
```

C.1 Using Wrapper Types

An naïve approach to the problem is to introduce a secondary wrapper type that is used to explicitly collect lists of dependent types specifically for `Foo`. For example:

```
data FNode = mkNode (Foo x)
```

This can facilitate the construction of lists of the form:

```
fs : List FNode
fs = [mkNode (FStr "Hello World")
      , mkNode (FNat 42)
      , mkNode (FPair (FStr "Hello") (FNat 34))]
```

However, this is cumbersome and requires the programmer having to extract the value contained within an instance of `FNode` either through pattern matching or accessor functions. Further, the values within the type are no longer exposed, if you are working at the type level, you have just lost information.

C.2 Heterogeneous Vectors

As an alternate approach Heterogeneous vectors (`HVect`) can be used. `HVect` are a data structure for collecting lists of arbitrary typed elements. For example:

```
fs : HVect [Foo A, Foo B, Foo C]
fs = [FStr "Hello World"
      , FNat 42
      , FPair (FStr "Hello") (FNat 34)]
```

However, this may result in a program that has several unsafe constructs in which the permissible set of elements contained within the list structure is malleable. `HVect` are too loose.

C.3 List of Dependent Pairs

Dependent Pairs allow for the value in the type to be more dynamically presented. This construct is used to introduce proofs that a value exists within a specific type. For example, a list of `Foo` typed elements type can be specified using Dependent Pairs as:

```
fs : List (x ** Foo x)
fs = [( _ ** FStr "Hello World")
      , ( _ ** FNat 42)
      , ( _ ** FPair (FStr "Hello") (FNat 34))]
```

Notice how the list elements are now comprised of the element to be collected and proof that the value within the type exists. More information about dependent pairs is available in the Idris manual [Idris]. Unfortunately, like the solution presented in §C.1 one has to work with dependent pairs and extract the element from the pair construct. This will result in cumbersome code and lots of calls to the dependent pair's accessor functions or pattern matching. Further, access to the values in type is still not granted.

C.4 Custom Lists

The final alternate solution is the creation of a custom list that allows for the value in the type to be collected within the type much the same way the values of the elements are also collected. Take, for example the following data structure:

```
data FList : List FTy -> Type where
  Nil    : FList Nil
  (::)   : FList x -> FList xs -> FList (x::xs)
```

FList facilitates direct access to the collected elements at the value level, and at the type level the values within the type. The collection of values in the type mirrors the collection of the values. The list of foos can now be collected within a data structure.

```
fs : FList [A, B, C]
fs : [FStr "Hello World"
     , FNat 42
     , FPair (FStr "Hello") (FNat 34)]
```

However, this *is a* custom list construct for collections of Foo elements. Access has now been lost to all functions that use list operations Custom operations now have to be explicitly constructed for FList Secondly for each custom list that one uses, distinct operations must also be specified. The DList data structure is in fact a generic version of FList that can be used to collect a dependent data structure and present at the type-level the collection of a single value from each type.

BIBLIOGRAPHY

- [AC99] L. Augustsson and M. Carlsson. ‘An Exercise in Dependent Types: A Well-Typed Interpreter’. In: *In Workshop on Dependent Types in Programming, Gothenburg*. 1999.
- [AF10] O. Ajaj and E. B. Fernández. ‘A Pattern for the WS-Trust Standard for Web-Services’. In: *AsianPlop2010*. Ed. by H. Wahizaki and N. Yoshioka. Vol. 1. GRACE-TR-2010-01. Center for global research in advanced software science and engineering (GRACE). Tokyo, Japan: GRACE, Mar. 2010, pp. 9–20.
- [AG09] C. Ashford and P. Gauthier. *OSS Design Pattern. A Pattern Approach to the Design of Telecommunications Management Systems*. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-01396-6. DOI: 10.1007/978-3-642-01396-6.
- [Ale+77] C. Alexander et al. *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.
- [Ale79] C. Alexander. *The Timeless Way of Building*. Later printing. New York: Oxford University Press, 1979. ISBN: 0195024028.
- [AM11] D. Amyot and G. Mussbacher. ‘User Requirements Notation: The First Ten Years, The Next Ten Years (Invited Paper)’. In: *Journal of Software* 6.5 (2011).
- [Amy+10] D. Amyot et al. ‘Evaluating Goal Models within the Goal-Oriented Requirement Language’. In: *International Journal of Intelligent Systems* 25.8 (2010), pp. 841–877. ISSN: 1098-111X. DOI: 10.1002/int.20433.
- [Aug98] L. Augustsson. ‘Cayenne—A Language with Dependent Types’. In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP ’98. Baltimore, Maryland, USA: ACM, 1998, pp. 239–250. ISBN: 1-58113-024-4. DOI: 10.1145/289423.289451.
- [Bar92] H. Barendregt. ‘Lambda Calculi with Types’. In: *Handbook of Logic in Computer Science*. Oxford University Press, 1992, pp. 117–309.
- [BC87] K. Beck and W. Cunningham. *Using Pattern Languages for Object-Oriented Programs*. Technical Report CR-87-43. Apple Computer, Inc. and Tektronix, Inc., 1987.

- [Ben86] J. Bentley. ‘Programming Pearls: Little Languages’. In: *Commun. ACM* 29.8 (Aug. 1986), pp. 711–721. ISSN: 0001-0782. DOI: 10.1145/6424.315691.
- [BF99] F. L. Brown and E. B. Fernandez. ‘The Authenticator Pattern’. In: *Proceedings of the 1999 conference on Pattern languages of programs*. PLoP ’99. 1999.
- [BH06] E. Brady and K. Hammond. ‘A Verified Staged Interpreter is a Verified Compiler’. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. GPCE ’06. Portland, Oregon, USA: ACM, 2006, pp. 111–120. ISBN: 1-59593-237-2. DOI: 10.1145/1173706.1173724.
- [BH12] E. Brady and K. Hammond. ‘Resource-Safe Systems Programming with Embedded Domain Specific Languages’. In: *Proceedings of the 14th International Conference on Practical Aspects of Declarative Languages*. PADL’12. Philadelphia, PA: Springer-Verlag, 2012, pp. 242–257. ISBN: 978-3-642-27693-4. DOI: 10.1007/978-3-642-27694-1_18.
- [BHS07] F. Buschmann et al. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. Wiley series in software design patterns. John Wiley & Sons, 2007. ISBN: 9780471486480.
- [BKS11] M. Bunke et al. ‘Application-Domain Classification for Security Patterns’. In: *PATTERNS 2011, The Third International Conferences on Pervasive Patterns and Applications*. Rome, Italy: ThinkMind, 2011, pp. 138–143. ISBN: 978-1-61208-158-8.
- [Bra05] E. Brady. ‘Practical Implementation of a Dependently Typed Functional Programming Language’. PhD thesis. Durham University, 2005.
- [Bra13] E. Brady. ‘Idris, a general-purpose dependently typed programming language: Design and implementation’. In: *Journal of Functional Programming* 23 (05 Sept. 2013), pp. 552–593. ISSN: 1469-7653. DOI: 10.1017/S095679681300018X.
- [Bra15a] E. Brady. ‘Cross-Platform Compilers for Functional Languages’. English. Online. Submitted to TFP 2015. 2015.
- [Bra15b] E. Brady. ‘Resource-Dependent Algebraic Effects’. English. In: *Trends in Functional Programming*. Ed. by J. Hage and J. McCarthy. Vol. 8843. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 18–33. ISBN: 978-3-319-14674-4. DOI: 10.1007/978-3-319-14675-1_2.
- [BRD98] A. M. Braga et al. ‘Tropyc: A Pattern Language for Cryptographic Software’. In: *Pattern Languages of Programs PLoP 1998*. 1998.

- [Bre+04] P. Bresciani et al. ‘Tropos: An Agent-Oriented Software Development Methodology’. English. In: *Autonomous Agents and Multi-Agent Systems* 8.3 (2004), pp. 203–236. ISSN: 1387-2532. DOI: 10.1023/B:AGNT.0000018806.20944.ef.
- [BZ10] I. Bayley and H. Zhu. ‘Formal Specification of the Variants and Behavioural Features of Design Patterns’. In: *Journal of Systems and Software* 83.2 (2010), pp. 209–221. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.09.039.
- [Cas+15] D. Castro et al. ‘Structure, Semantics and Speedup: Reasoning about Structured Parallel Programs using Dependent Types’. Manuscript under consideration with the Journal of Functional Programming, 2015.
- [CB13] T. Clark and B. Barn. ‘Domain Engineering for Software Tools’. English. In: *Domain Engineering*. Ed. by I. Reinhartz-Berger et al. Springer Berlin Heidelberg, 2013, pp. 187–209. ISBN: 978-3-642-36653-6. DOI: 10.1007/978-3-642-36654-3_8.
- [CCM03] J. Clark et al. *RELAX NG Compact Syntax Tutorial*. The Organization for the Advancement of Structured Information Standards (OASIS). 26th Mar. 2003.
- [Che+03] B. H. Cheng et al. ‘Using Security Patterns to Model and Analyze Security Requirements’. In: *Security* (2003).
- [Chr14] D. R. Christiansen. ‘Reflect on Your Mistakes! Lightweight Domain-Specific Error Messages’. Online. Submitted to Post-Proceedings of TFP 2014. 2014.
- [CO08] D. Crocker and P. Overell. *Augmented BNF for Syntax Specifications: ABNF*. Request for Comments 5234. Internet Engineering Task Force. IETF, Jan. 2008.
- [Cor13] M. Corporation. *Common Attack Pattern Enumeration and Classification Repository*. Online. 2013.
- [Cue+09] A. Cuevas et al. ‘A Security Pattern for Untraceable Secret Handshakes’. In: *Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09. Third International Conference on*. June 2009, pp. 8–14. DOI: 10.1109/SECURWARE.2009.9.
- [Cue+10] A. Cuevas et al. ‘Security Patterns for Untraceable Secret Handshakes with Optional Revocation’. In: *International Journal On Advances in Security* 3.1&2 (Sept. 2010), pp. 68–79. ISSN: 1942-2636.
- [CW97] J. O. Coplien and B. Woolf. ‘A Pattern Language for Writers’ Workshops’. In: *C PLUS PLUS REPORT* 9 (1997), pp. 51–60.
- [DE05] J. Dietrich and C. Elgar. ‘A formal description of design patterns using OWL’. In: *Software Engineering Conference, 2005. Proceedings. 2005 Australian*. Mar. 2005, pp. 243–250. DOI: 10.1109/ASWEC.2005.6.

- [Del+12] M. Delescluse et al. ‘Making neurophysiological data analysis reproducible: Why and How?’ In: *Neuronal Ensemble Recordings in Integrative Neuroscience* 106.3–4 (2012), pp. 159–170. ISSN: 0928-4257. DOI: 10.1016/j.jphysparis.2011.09.011.
- [DFo6] A. Dearden and J. Finlay. ‘Pattern Languages in HCI: A Critical Review’. In: *Human-Computer Interaction* 21.1 (2006), pp. 49–102. DOI: 10.1207/s15327051hci2101\3.
- [DFLo7] N. Delessy et al. ‘A Pattern Language for Identity Management’. In: *Computing in the Global Information Technology, 2007. ICCGI 2007. International Multi-Conference on*. Mar. 2007, p. 31. DOI: 10.1109/ICCGI.2007.5.
- [dGM13] L. S. da Silva Júnior et al. *An Approach to Formalise Security Patterns*. Tech. rep. Montréal Québec: École Polytechnique de Montréal, 2013.
- [Don+07] J. Dong et al. ‘Composing Pattern-Based Components and Verifying Correctness’. In: *J. Syst. Softw.* 80.11 (Nov. 2007), pp. 1755–1769. ISSN: 0164-1212. DOI: 10.1016/j.jss.2007.03.005.
- [ESP07] R. Erber et al. ‘Patterns for Authentication and Authorisation Infrastructures’. In: *Proceedings of the 18th International Conference on Database and Expert Systems Applications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 755–759. ISBN: 0-7695-2932-1. DOI: 10.1109/DEXA.2007.114.
- [Feh+14] C. Fehling et al. *Cloud Computing Patterns. Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Vienna, 2014. ISBN: 978-3-7091-1568-8. DOI: 10.1007/978-3-7091-1568-8.
- [Fer+11] E. B. Fernández et al. ‘Using Security Patterns to Develop Secure Systems’. In: *Software Engineering for Secure Systems: Industrial and Research Perspectives*. Ed. by H. Mouratidis. IGI Global, 2011. Chap. 2, pp. 16–31. DOI: 10.4018/978-1-61520-837-1.ch002.
- [Fero7] E. B. Fernández. ‘Security Patterns and Secure Systems Design’. In: *Dependable Computing*. Ed. by A. Bondavalli et al. Vol. 4746. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, pp. 233–234. ISBN: 978-3-540-75293-6. DOI: 10.1007/978-3-540-75294-3_18.
- [Fer13] E. B. Fernández. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley Software Patterns Series. Wiley, 2013. ISBN: 9781119970484.
- [Fino4] S. Fincher. *Extended Pattern Language Markup Language*. Online. 2004.
- [Fow10] M. Fowler. *Domain-Specific Languages*. 1st ed. Addison-Wesley Signature Series. Addison-Wesley Professional, Oct. 2010. ISBN: 0321712943.

- [Fra13] U. Frank. ‘Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines’. English. In: *Domain Engineering*. Ed. by I. Reinhartz-Berger et al. Springer Berlin Heidelberg, 2013, pp. 133–157. ISBN: 978-3-642-36653-6. DOI: 10.1007/978-3-642-36654-3_6.
- [Fre+04] E. Freeman et al. *Head First Design Patterns*. O’ Reilly & Associates, Inc., 2004. ISBN: 0596007124.
- [FS03] E. B. Fernández and J. Sinibaldi. ‘More Patterns for Operating System Access Control’. In: *Proceedings of the 2003 European Conference on Pattern Languages of Programs (EuroPLoP)*. 2003, pp. 381–398.
- [FW03] E. B. Fernández and R. Warriier. ‘Remote Authenticator/Authorizer’. In: *Proceedings of the 10th Conference on Pattern languages of programs*. PLoP 2003. 2003.
- [FWY08] E. B. Fernández et al. ‘Abstract Security Patterns’. In: *Proceedings of the 15th Conference on Pattern Languages of Programs*. PLoP ’08. Nashville, Tennessee: ACM, 2008, 4:1–4:2. ISBN: 978-1-60558-151-4. DOI: 10.1145/1753196.1753201.
- [Gabo2] R. P. Gabriel. *Writers’ workshops & the work of making things*. Addison-Wesley, 2002.
- [Gam+94] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [GB08] S. Greenberg and B. Buxton. ‘Usability Evaluation Considered Harmful (Some of the Time)’. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’08. Florence, Italy: ACM, 2008, pp. 111–120. ISBN: 978-1-60558-011-1. DOI: 10.1145/1357054.1357074.
- [Gio+03] P. Giorgini et al. ‘Formal Reasoning Techniques for Goal Models’. English. In: *Journal on Data Semantics I*. Ed. by S. Spaccapietra et al. Vol. 2800. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 1–20. ISBN: 978-3-540-20407-7. DOI: 10.1007/978-3-540-39733-5_1.
- [Gio+05] P. Giorgini et al. ‘Modeling security requirements through ownership, permission and delegation’. In: *13th IEEE International Conference on Requirements Engineering (RE’05)*. Aug. 2005, pp. 167–176. DOI: 10.1109/RE.2005.43.
- [Gra92] R. B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN: 0-13-720384-5.
- [Gue+14] E. Guerra et al. ‘Patterns for Preparing for a Test Driven Development Session’. In: *Pattern Languages of Programs (PLoP) 2014*. Hillside, 2014.

- [GY01] D. Gross and E. Yu. ‘From Non-Functional Requirements to Design through Patterns’. English. In: *Requirements Engineering 6.1* (2001), pp. 18–36. ISSN: 0947-3602. DOI: 10.1007/s007660170013.
- [Haro4] N. B. Harrison. *Advanced Pattern Writing: Patterns for Experienced Pattern Authors*. Online. 2004.
- [Har99] N. B. Harrison. *The Language of Shepherding: A Pattern Language for Shepherds and Sheep*. Online. 1999.
- [Hau06] M. Hause. ‘The SysML Modelling Language’. In: *Fifteenth European Systems Engineering Conference*. 2006.
- [HCo7] S. Henninger and V. Corrêa. ‘Software Pattern Communities: Current Practices and Challenges’. In: *Proceedings of the 14th Conference on Pattern Languages of Programs*. PLOP ’07. Monticello, Illinois: ACM, 2007, 14:1–14:19. ISBN: 978-1-60558-411-9. DOI: 10.1145/1772070.1772087.
- [HCS04] S. Halkidis et al. ‘A Qualitative Evaluation of Security Patterns’. In: *Information and Communications Security*. Ed. by J. Lopez et al. Vol. 3269. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, pp. 251–259. ISBN: 978-3-540-23563-7. DOI: 10.1007/978-3-540-30191-2_11.
- [Hey+07] T. Heyman et al. ‘An Analysis of the Security Patterns Landscape’. In: *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*. SESS ’07. Washington, DC, USA: IEEE Computer Society, 2007, p. 3. ISBN: 0-7695-2952-6. DOI: 10.1109/SESS.2007.4.
- [Hey13] T. Heyman. ‘A Formal Analysis Technique for Secure Software Architectures (Een formele analysetechniek voor veilige softwarearchitecturen)’. English. PhD thesis. Katholiek Universiteit Leuven, 6th Mar. 2013.
- [HF10] K. Hashizume and E. B. Fernández. ‘Symmetric Encryption and XML Encryption Patterns’. In: *Proceedings of the 16th Conference on Pattern Languages of Programs*. PLoP ’09. Chicago, Illinois: ACM, 2010, 13:1–13:8. ISBN: 978-1-60558-873-5. DOI: 10.1145/1943226.1943243.
- [HFH09] K. Hashizume et al. ‘Digital Signature with Hashing and XML Signature Patterns’. In: *EuroPLoP*. 2009.
- [Idr15] The Idris Community. *Programming in Idris. A Tutorial*. 2015. URL: <http://docs.idris-lang.org/en/latest/tutorial/>.
- [ISO96] ISO/IEC, ed. *Extended BNF*. Information Technology—Syntactic Meta-Language ISO/IEC 14977 : 1996(E). International Standards Organisation (ISO), 1996.
- [Jac12] D. Jackson. *Software Abstractions. Logic, Language, and Analysis*. MIT Press, 2012. ISBN: 0-262-01715-6.

- [JN95] N. D. Jones and F. Nielson. ‘Handbook of Logic in Computer Science (Vol. 4)’. In: ed. by S. Abramsky et al. Oxford, UK: Oxford University Press, 1995. Chap. Abstract Interpretation: A Semantics-based Tool for Program Analysis, pp. 527–636. ISBN: 0-19-853780-8.
- [Jür02] J. Jürjens. ‘UMLSec: Extending UML for Secure Systems Development’. English. In: *UML 2002 – The Unified Modeling Language*. Ed. by J.-M. Jézéquel et al. Vol. 2460. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 412–425. ISBN: 978-3-540-44254-7. DOI: 10.1007/3-540-45800-X_32.
- [KH10] C. Kruschitz and M. Hitz. ‘Bringing Formalism and Unification to Human-Computer Interaction Design Patterns’. In: *Proceedings of the 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems*. PEICS ’10. Berlin, Germany: ACM, 2010, pp. 20–23. ISBN: 978-1-4503-0246-3. DOI: 10.1145/1824749.1824754.
- [Kie+03] D. M. Kienzle et al. ‘Security Patterns Repository Version 1.0’. Online [Accessed 2012-02-27]. 2003.
- [Kin+75] J. P. Kincaid et al. *Derivation of New Readability Formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy Enlisted Personnel*. Tech. rep. Research branch rept. ADA006655. Feb. 1975, p. 41.
- [KLL09] R. K. Ko et al. ‘Business process management (BPM) standards: A Survey’. In: *Business Process Management Journal* 15,5 (2009), pp. 744–791. DOI: 10.1108/14637150910987937.
- [Kru09] C. Kruschitz. ‘XPLML: a HCI Pattern Formalizing and Unifying Approach’. In: *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*. CHI EA ’09. Boston, MA, USA: ACM, 2009, pp. 4117–4122. ISBN: 978-1-60558-247-4. DOI: 10.1145/1520340.1520627.
- [Lav+06] M.-A. Laverdière et al. ‘Security Design Patterns: Survey and Evaluation’. In: *Electrical and Computer Engineering, 2006. CCECE ’06. Canadian Conference on*. 2006, pp. 1605–1608. DOI: 10.1109/CCECE.2006.277727.
- [LFM11] L. López et al. ‘Making Explicit Some Implicit i★ Language Decisions’. English. In: *Conceptual Modeling—ER 2011*. Ed. by M. Jeusfeld et al. Vol. 6998. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 62–77. ISBN: 978-3-642-24605-0. DOI: 10.1007/978-3-642-24606-7_6.

- [LHM14] T. Li et al. ‘Integrating Security Patterns with Security Requirements Analysis Using Contextual Goal Models’. English. In: *The Practice of Enterprise Modeling*. Ed. by U. Frank et al. Vol. 197. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2014, pp. 208–223. ISBN: 978-3-662-45500-5. DOI: 10.1007/978-3-662-45501-2_15.
- [Luc+03] D. Lucrédio et al. ‘MVCASE Tool–Working with Design Patterns’. In: *Proceedings of the 3rd Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2003)*. 2003, pp. 261–275.
- [LW12] C. Le Goues and W. Weimer. ‘Measuring Code Quality to Improve Specification Mining’. In: *Software Engineering, IEEE Transactions on* 38.1 (Jan. 2012), pp. 175–190. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.5.
- [Man+13] A. Mana et al. ‘Towards Computer-oriented Security Patterns’. In: *PLOP ’13: Proceedings of the 20th Conference on Pattern Languages of Programs*. Hillside. 2013.
- [Mar98] M. Marchesi. ‘OOA metrics for the Unified Modeling Language’. In: *Software Maintenance and Reengineering, 1998. Proceedings of the Second Euromicro Conference on*. Mar. 1998, pp. 67–73. DOI: 10.1109/CSMR.1998.665739.
- [MCo1] W. E. McUmber and B. H. C. Cheng. ‘A General Framework for Formalizing UML with Formal Languages’. In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE ’01*. Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 433–442. ISBN: 0-7695-1050-7.
- [MCI4] H. Mehnert and D. R. Christiansen. ‘Tool Demonstration: An IDE for Programming and Proving in Idris’. In: *Dependently Typed Programming 2014*. Vienna, 2014.
- [McBo5] C. McBride. ‘Epigram: Practical Programming with Dependent Types’. English. In: *Advanced Functional Programming*. Ed. by V. Vene and T. Uustalu. Vol. 3622. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 130–170. ISBN: 978-3-540-28540-3. DOI: 10.1007/11546382_3.
- [MD97] G. Meszaros and J. Doble. ‘A Pattern Language for Pattern Writing’. In: *Pattern languages of program design 3*. Ed. by R. C. Martin et al. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, pp. 529–574. ISBN: 0-201-31011-2.
- [MFo6] P. Morrison and E. B. Fernández. ‘The Credentials Pattern’. In: *Proceedings of the 2006 conference on Pattern languages of programs. PLoP ’06*. Portland, Oregon: ACM, 2006, 9:1–9:4. ISBN: 978-1-60558-372-3. DOI: 10.1145/1415472.1415483.

- [MG07] H. Mouratidis and P. Giorgini. ‘Secure TROPOS: A Security-Oriented Extension of the TROPOS Methodology’. In: *International Journal of Software Engineering and Knowledge Engineering* 17.02 (2007), pp. 285–309. DOI: 10.1142/S0218194007003240.
- [MGPO3] M. Manso et al. ‘No-redundant Metrics for UML Class Diagram Structural Complexity’. English. In: *Advanced Information Systems Engineering*. Ed. by J. Eder and M. Missikoff. Vol. 2681. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 127–142. ISBN: 978-3-540-40442-2. DOI: 10.1007/3-540-45017-3_11.
- [Miso8] R. N. Mishra-Linger. ‘Irrelevance, Polymorphism, and Erasure in Type Theory’. PhD. Portland State University, 2008.
- [ML05] S. Mahmood and R. Lai. ‘Measuring the Complexity of a UML Component Specification’. In: *Quality Software, International Conference on* (2005), pp. 150–160. ISSN: 1550-6002. DOI: 10.1109/QSIC.2005.39.
- [MM04] C. McBride and J. McKinna. ‘The View from the Left’. In: *Journal of Functional Programming* 14 (01 2004), pp. 69–111. ISSN: 1469-7653. DOI: 10.1017/S0956796803004829.
- [MMZ07] F. Massacci et al. ‘Computer-aided Support for Secure Tropos’. In: *Automated Software Engineering* 14.3 (2007), pp. 341–364. ISSN: 1573-7535. DOI: 10.1007/s10515-007-0013-5.
- [MMZ10] F. Massacci et al. ‘Security Requirements Engineering: The SI* Modeling Language and the Secure Tropos Methodology’. In: *Advances in Intelligent Information Systems*. Ed. by Z. W. Ras and L.-S. Tsay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 147–174. ISBN: 978-3-642-05183-8. DOI: 10.1007/978-3-642-05183-8_6.
- [MWA06] G. Mussbacher et al. ‘Formalizing Architectural Patterns with the Goal-Oriented Requirement Language’. In: *Nordic Pattern Languages of Programs Conference (VikingPlop2006)*. 2006.
- [Mylo6] J. Mylopoulos. ‘Goal-Oriented Requirements Engineering, Part II’. In: *Requirements Engineering, 14th IEEE International Conference*. Sept. 2006, pp. 5–5. DOI: 10.1109/RE.2006.27.
- [Nor09] U. Norell. ‘Dependently Typed Programming in Agda’. In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI ’09. Savannah, GA, USA: ACM, 2009, pp. 1–2. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481862.
- [OFK15] S. Overbeek et al. ‘A Language for Multi-Perspective Goal Modelling: Challenges, Requirements and Solutions’. In: *Computer Standards & Interfaces* 38 (2015), pp. 1–16. ISSN: 0920-5489. DOI: 10.1016/j.csi.2014.08.001.

- [Pea89] G. Peano. *Arithmetices Principia: Nova Methodo*. Fratres Bocca, 1889.
- [Pet62] C. A. Petri. ‘Kommunikation mit Automaten’. ger. PhD thesis. Universität Hamburg, 1962.
- [Pri+04] T. Priebe et al. ‘A Pattern System for Access Control’. In: *Research Directions in Data and Applications Security XVIII*. Ed. by C. Farkas and P. Samarati. Vol. 144. IFIP International Federation for Information Processing. Springer Boston, 2004, pp. 235–249. ISBN: 978-1-4020-8127-9. DOI: 10.1007/1-4020-8128-6_16.
- [Sari6] V. Sarcar. *Java Design Patterns. A tour of 23 gang of four design patterns in Java*. Apress, 2016. ISBN: 978-1-4842-1802-0. DOI: 10.1007/978-1-4842-1802-0.
- [Sch+06] M. Schumacher et al. *Security Patterns: Integrating Security and Systems Engineering*. Wiley series in software design patterns. John Wiley & Sons, 2006. ISBN: 9780470858844.
- [Scho3] M. Schumacher. *Security Engineering with Patterns: Origins, Theoretical Models, and New Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003. ISBN: 3540407316.
- [Scho6] D. Schmidt. *How to Hold a Writer’s Workshop*. 2006.
- [SD11] E. Schulte and D. Davison. ‘Active Documents with Org-Mode’. In: *Computing in Science Engineering* 13.3 (May 2011), pp. 66–73. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.41.
- [Sef15] A. Seffah. *Patterns of HCI Design and HCI Design of Patterns. Bridging HCI Design and Model-Driven Software Engineering*. Human–Computer Interaction Series. Springer International Publishing, 2015. ISBN: 978-3-319-15687-3. DOI: 10.1007/978-3-319-15687-3.
- [Shi+10] Y. Shiroma et al. ‘Model-Driven Application and Validation of Security Patterns’. In: *Proceedings of the 10th Conference on Pattern Languages of Programs*. PLoP ’10. Oct. 2010.
- [SI15] P. Scupelli and P. S. Inventado. *Developing an Open, Collaborative Design Pattern Repository*. Focus Group at PLoP ’15. 2015.
- [Sier12] J. Siek. *Crash Course on Notation in Programming Language Theory*. 2012.
- [Smi87] R. Smith. ‘Panel on Design Methodology’. In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*. OOPSLA ’87. Orlando, Florida, USA: ACM, 1987, pp. 91–95. ISBN: 0-89791-266-7. DOI: 10.1145/62138.62151.
- [TCK13] T. Thimthong et al. ‘Evaluating Design Patterns of Commercial Web Applications using Net Easy Score’. In: *I.J. Information Technology and Computer Science* 5.8 (July 2013). DOI: 10.5815/ijitcs.2013.08.09.

- [The13] The Hillside Group. *Patterns Catalog*. Online. 2013.
- [UTN12] UTN. *User Requirements Notation. Language Definition*. Series Z: Languages and General Software Aspects for Telecommunication Systems ITU-T Z.151. International Telecommunications Union. 2012.
- [vLam01] A. van Lamsweerde. ‘Goal-Oriented Requirements Engineering: A Guided Tour’. In: *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. 2001, pp. 249–262. DOI: 10.1109/ISRE.2001.948567.
- [VM11] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley Professional Computing Series. Addison-Wesley Professional, 2011.
- [WCo2] T. Winn and P. Calder. ‘Is this a pattern?’ In: *Software, IEEE* 19.1 (2002), pp. 59–66. ISSN: 0740-7459. DOI: 10.1109/52.976942.
- [Wei06] M. Weiss. ‘Credential Delegation: Towards Grid Security Patterns’. In: *The Nordic Conference on Pattern Languages of Programs*. 2006, pp. 65–70.
- [Wel+06] L. Welicki et al. ‘Patterns Meta-Specification and Cataloging: Towards Knowledge Management in Software Engineering’. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 679–680. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176670.
- [WF11] T. Wellhausen and A. Fießler. ‘How to Write a Pattern? A Rough Guide for First-Time Pattern Authors’. In: *Proceedings of the 16th European Conference on Pattern Languages of Programs*. EuroPLoP ’11. Irsee, Germany: ACM, 2011, 5:1–5:9. ISBN: 978-1-4503-1302-5. DOI: 10.1145/2396716.2396721.
- [WLA05] L. Welicki et al. ‘A Model for Meta-Specification and Cataloging of Software Patterns’. In: *Proceedings of the European Conference on Pattern Languages of Programming 2005 (EuroPLoP 2005)*. 2005, pp. 261–275.
- [WLA06] L. Welicki et al. ‘Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models’. In: *Proceedings of the European Conference on Pattern Languages of Programming 2006 (EuroPLoP 2006)*. 2006, pp. 261–275.
- [WMo8] M. Weiss and H. Mouratidis. ‘Selecting Security Patterns that Fulfill Security Requirements’. In: *International Requirements Engineering, 2008. RE ’08. 16th IEEE*. Sept. 2008, pp. 169–172. DOI: 10.1109/RE.2008.32.
- [WN06] P. Wadler and M. Naftalin. *Java Generics and Collections*. O’Reilly Media, 2006. ISBN: 9780596551506.

- [Xia+15] T. Xia et al. ‘Two-level Checklists and Perspectives: Software Reading Techniques for Pattern Writer’s Workshop’. Online. Submitted to the International Conference on Pattern Languages of Programs 2015 (PLoP’15). 2015.
- [YB97] J. Yoder and J. Barcalow. ‘Architectural Patterns for Enabling Application Security’. In: *Proceedings of the Conference on Pattern Languages of Programs (PLoP 1997)*. Monticello/IL, 1997.
- [Yu97] E. Yu. ‘Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering’. In: *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*. Jan. 1997, pp. 226–235. DOI: 10.1109/ISRE.1997.566873.
- [YWG04] T. Yi et al. ‘A Comparison of Metrics for UML Class Diagrams’. In: *SIGSOFT Softw. Eng. Notes* 29.5 (Sept. 2004), pp. 1–6. ISSN: 0163-5948. DOI: 10.1145/1022494.1022523.
- [YWM08] N. Yoshioka et al. ‘A Survey on Security Patterns’. In: *Progress in Informatics* 5 (2008), pp. 33–47. DOI: 10.2201/NiiPi.2008.5.5.
- [ZZP02] Y. Zhou et al. ‘Policy Enforcement Pattern’. In: *PLoP 2002*. 2002.

LIST OF FIGURES

1.1	Pattern Engineering according to Yoshioka et al. [YWMo8].	3
1.2	The <i>Pattern Engineering Process</i> with thesis contributions placed <i>in situ</i>	8
2.1	Example pattern diagram from de Muijnck-Hughes and Duncan [dMD12] describing a proposed pattern language for Predicate-Based Encryption.	15
2.2	Pattern Engineering according to Yoshioka et al. [YWMo8].	16
3.1	Legend for the GRL [Amy+10].	29
3.2	An example goal model for the problem of ‘Information Secrecy’.	31
3.3	An example goal model for the problem of ‘Information Secrecy’ with a solution using ‘Symmetric Cryptography’.	32
3.4	Goal model from Figure 3.3 after evaluation.	33
4.1	The syntax for ARITH.	41
4.2	Interpretation and evaluation semantics for the types in ARITH.	44
4.3	Interpretation and evaluation semantics for ARITH.	45
4.4	Interpretation semantics for abstracting ARITH expressions into a ‘Cast Nine’ abstraction.	57
5.1	Abstract Syntax for the SIF modelling language.	64
5.2	Types in the SIF modelling language.	64
5.3	Interpretation semantics for converting SIF expressions into GRL constructs.	67
5.4	Architecture of the SIF Evaluator.	69
5.5	eBNF grammar for the SIF Domain Specific Language	71
7.1	Feature-Set and dependency overview for FRIGG.	109
8.1	Schematic overview of the stages and indicators of the PREMES evaluation framework.	115
9.1	Tools and technologies presented in this thesis and their placement in the pattern engineering process.	136

10.1	Schematic illustrating how DSML relate to, and are interpreted into, No-voGRL goal models.	161
10.2	Language definition for G^*	171
10.3	Interpretation semantics for G^*	174
10.4	Language definition for GEXPR.	175
10.5	Interpretation semantics for GEXPR.	177
10.6	Language definition for the GRL.	182
10.7	Interpretation semantics for converting GRL expressions into GEXPR expressions.	186
10.8	Generic instantiation of a GRL model for academic paper writing.	187
10.9	Model instance of a GRL model for an academic paper.	188
10.10	Abstract syntax for language declarations in the PML.	192
10.II	Interpretation semantics for converting PML expressions into GEXPR expressions.	193
11.1	Abstract syntax for the PTODO modelling language.	204
11.2	Typing rules for PTODO.	205
11.3	Interpretation semantics for converting PTODO expressions into GRL constructs.	207

LIST OF TABLES

4.2	Example pairings of values and their types.	42
4.4	Java <i>Generics</i> used to describe collections with the type of the elements being described within the type of the collection.	46
5.3	Summary of the patterns modelled using SIF during evaluation.	79
5.5	Requirements for the problem of <i>Abstract Data Types</i>	80
5.7	Evaluation results for the factory patterns modelled in SIF.	81
5.9	Requirements for the problem of Information Secrecy.	82
5.11	Evaluation results for the Information Secrecy patterns modelled in SIF.	84
8.2	Grade descriptor for the <i>Pattern Coherency</i> indicator.	119
8.4	Grade descriptor for the <i>Pattern Atomicity</i> indicator.	119
8.6	Grade descriptor for the <i>Problem Independence</i> indicator.	120
8.8	Sample grading scheme for <i>Solution Complexity</i> indicator.	121
8.10	Sample grading scheme for the <i>Solution Effectiveness</i> indicator.	122
8.12	Grading scheme used for the <i>Presentation Accessibility</i> indicator.	124
8.13	Summative pattern report cards for several existing patterns.	128
9.2	SIF evaluation results for AUTHENTICATION THROUGH SHIBBOLETHS & AUTHENTICATION THROUGH ID CARDS.	147
9.4	Mappings from GRL satisfaction values to quantitative values used for pattern evaluation.	153
9.6	Report Cards Grades for the AUTHENTICATION THROUGH SHIBBOLETHS and AUTHENTICATION THROUGH ID CARDS patterns.	153
10.2	Various representations of the GRL: pictorial, abstract syntax, and Idris.	183
B.1	Look-Up table for combineContribs. Column index is for the result of cmpSatAndDen. Row index is for the result of cmpWSandWD.	239
B.2	Look-Up table for weightedContribution. Column index is for the contribution value. Row index is for the Satisfaction value.	240

LIST OF DEFINITIONS

1	Definition (Pattern Coherency)	116
2	Definition (Pattern Atomicity)	116
3	Definition (Problem Independence)	116
4	Definition (Solution Appropriateness)	116
5	Definition (Solution Complexity)	117
6	Definition (Solution Effectiveness)	117
7	Definition (Pattern Structure)	117
8	Definition (Pattern Legibility)	117
9	Definition (Presentation Accessibility)	117
10	Definition (Weighted Solution Satisfaction)	120
11	Definition (Weighted Adherence to Pattern Template)	122
12	Definition (Goal Graph)	160
13	Definition (GRL Goal-Graph)	163
14	Definition (Goal Nodes)	164
15	Definition (Intentional Link)	164
16	Definition (Structural Links)	164
17	Definition (Goal Edges)	164
18	Definition (Node Types)	165
19	Definition (Contribution Values)	165
20	Definition (Satisfaction Values)	165
21	Definition (Decomposition Link Types)	165
22	Definition (Intentional Link Types)	165
23	Definition (Well-Formed Intentional Link)	166
24	Definition (Well-Formed Decomposition Link)	166
25	Definition (Valid Intentional Link)	167
26	Definition (Valid Decomposition Link)	167
27	Definition (Goal Uniqueness)	167
28	Definition (Valid Goal Intentional Link)	168
29	Definition (Strongly Valid Goal Intentional Link)	168
30	Definition (Valid Goal Decomposition)	169
31	Definition (Strongly Valid Node Decomposition)	169
32	Definition (Structural Span)	169

LIST OF DEFINITIONS

33	Definition (Valid Structural Span)	170
34	Definition (Goal-Graph Correctness)	170
35	Definition (Valid Element Insertion)	178
36	Definition (Valid Intentional Link Insertion)	178
37	Definition (Valid Decomposition Link Insertion)	178
38	Definition (Evaluation Ready)	236

LIST OF SOFTWARE

List of active software repositories containing up-to-date versions of the code developed as part of the research project.

- [dMH15a] J. de Muijnck-Hughes. *Config: Utilities for processing configuration file formats*. 2015. URL: <https://github.com/jfdm/idris-config>.
- [dMH15b] J. de Muijnck-Hughes. *Containers: Containers for Idris*. 2015. URL: <https://github.com/jfdm/idris-containers>.
- [dMH15c] J. de Muijnck-Hughes. *Edda: A Document Processing Engine inspired by Pandoc*. 2015. URL: <https://github.com/jfdm/edda>.
- [dMH15d] J. de Muijnck-Hughes. *Freyja: A Design Pattern Document Description Schema & Tooling*. 2015. URL: <https://github.com/jfdm/freyja-schema>.
- [dMH15e] J. de Muijnck-Hughes. *GRL: The Goal Requirements Language in Idris*. 2015. URL: <https://github.com/jfdm/idris-grl>.
- [dMH15f] J. de Muijnck-Hughes. *Readability: Collecting Readability Metrics for Documents*. 2015. URL: <https://github.com/jfdm/idris-read>.
- [dMH15g] J. de Muijnck-Hughes. *Sif-Lang: A Requirements-Based Model Checker for Design Patterns*. 2015. URL: <https://github.com/jfdm/sif-lang>.
- [dMH15h] J. de Muijnck-Hughes. *UML: A UML Modelling Library*. 2015. URL: <https://github.com/jfdm/idris-uml>.
- [dMH15i] J. de Muijnck-Hughes. *XML: Modelling and processing XML using DOM and XPath*. 2015. URL: <https://github.com/jfdm/idris-xml>.

LIST OF PUBLICATIONS

List of publications that were made during the research project.

- [DdM14] I. Duncan and J. de Muijnck-Hughes. ‘Security Pattern Evaluation’. In: *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*. Apr. 2014, pp. 428–429. DOI: 10.1109/SOSE.2014.61.
- [dMD12] J. de Muijnck-Hughes and I. Duncan. ‘Thinking Towards a Pattern Language for Predicate Based Encryption Crypto-Systems’. In: *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*. 2012, pp. 27–32. DOI: 10.1109/SERE-C.2012.34.
- [dMD13] J. de Muijnck-Hughes and I. Duncan. ‘Issues Affecting Security Design Pattern Engineering’. In: *Proceedings of the Second International Workshop on Cyberpatterns*. Oxford Brookes University. July 2013, pp. 54–61.
- [dMD15] J. de Muijnck-Hughes and I. Duncan. ‘What’s the PREMES behind your Pattern?’ In: *Proceedings of the 22nd Conference on Pattern Languages of Programs (PLoP ’15)*. To appear in the post-conference proceedings. Pittsburgh, PA, USA: ACM, Oct. 2015.