

**Transformations of
Specifications and Proofs
to Support an
Evolutionary Formal Software
Development**

Axel Schairer

Dissertation
zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen
Fakultäten der
Universität des Saarlandes

Saarbrücken, 2005

Veröffentlicht im Shaker Verlag, Aachen, 2006
www.shaker.de

Tag des Kolloquiums	21. 07. 2006
Dekan	Prof. Dr.-Ing. Thorsten Herfet
Prüfungsausschuss:	
Vorsitzender	Prof. Dr. Gert Smolka
Akademischer Mitarbeiter	Dr. Dieter Hutter
Gutachter	Prof. Dr. Jörg H. Siekmann Universität des Saarlandes
	Prof. Dr. Bernd Krieg-Brückner Universität Bremen
	Dr. Alan Smaill University of Edinburgh, Scotland, UK

Abstract

Like other software engineering activities, formal modelling needs to deal with change: bugs and omissions need to be corrected, and changes from the outside need to be dealt with. In the context of axiomatic specifications and (partly) interactive proofs, the main obstacle is that changes invalidate proofs, which then need to be rebuilt using an inhibitive amount of resources.

This thesis proposes to solve the problem by considering the state of a formal development consisting of (potentially buggy) specification and (potentially partial) proofs as one entity and transforming it using pre-conceived transformations. These transformations are operationally motivated: how would one patch the proofs on paper given a consistent transformation for the specification? They are formulated in terms of the specification and logic language, so as to be usable for several application domains.

In order to make the approach compatible with the architecture of existing support systems, development graphs are added as an intermediate concept between specification and proof obligations, and the transformations are extended to work in the presence of the indirection. This leads to a separation of a framework for propagating transformations through development graphs and a reference instantiation that commits to concrete languages and proof representation. The reference instantiation works for many practically relevant scenarios. Other instantiations can be based on the framework.

Zusammenfassung

Wie bei allen anderen Software Engineering-Aktivitäten muss es auch bei der formalen Modellierung möglich sein, Änderungen vorzunehmen. So können Modellierungsfehler korrigiert, Unvollständigkeiten ergänzt und externe Änderungen in das Modell eingearbeitet werden. Für axiomatische Spezifikationen und (zumindest teilweise) interaktive Beweise ist das größte Hindernis dabei, dass Änderungen die Beweise ungültig machen. Diese müssen dann erneut konstruiert werden, was einen unverantwortbar hohen Einsatz von Ressourcen erfordert.

Diese Arbeit schlägt vor, dieses Problem dadurch zu lösen, dass eine formale Entwicklung bestehend aus einer (möglicherweise fehlerhaften) Spezifikation und (möglicherweise unvollständigen) Beweisen als eine Einheit betrachtet und nach vorgegebenen Regeln transformiert wird. Diese Regeln werden Transformationen genannt und sind operational motiviert: Wie würde ein Mensch intuitiv die Beweise anpassen, wenn eine in sich konsistente Transformation auf die Spezifikation angewendet wird? Die Transformationen sind unabhängig von der Anwendungsdomäne anwendbar. Um den Ansatz mit dem Aufbau existierender Werkzeuge vereinbar zu machen, führen wir ein zusätzliches Konzept als Mittler zwischen Spezifikation und Beweisverpflichtungen ein, so genannte Development Graphs. Das führt zu einer Trennung zwischen einem Rahmenwerk, das Transformationen über generische Development Graphs hinweg propagiert, und einer Referenzinstantiierung, die sich auf konkrete Sprachen und eine konkrete Repräsentation der Beweise festlegt. Die Referenzinstanz erlaubt die Behandlung von vielen in der Praxis vorkommenden Szenarien. Andere Instanzen können aus dem Rahmenwerk abgeleitet werden.

Ausführliche Zusammenfassung

Es wird heute allgemein akzeptiert, dass ein rein wasserfallartiger Software-Entwicklungsprozess nicht angemessen ist. Wie bei allen anderen Software Engineering-Aktivitäten auch muss es daher bei der formalen Modellierung möglich sein, Änderungen vorzunehmen. So können Modellierungsfehler korrigiert, Unvollständigkeiten verbessert und externe Änderungen in das Modell eingearbeitet werden. Gerade das Entdecken und Eliminieren von Fehlern in einem frühen Stadium ist einer der Hauptvorteile der formalen Modellierung und Entwicklung.

Für axiomatische Spezifikationen und (zumindest teilweise) interaktive Beweise ist das größte Hindernis dabei, dass Änderungen die vorher konstruierten Beweise potentiell ungültig machen. Diese müssen dann erneut konstruiert werden, was einen unverantwortbar hohen Einsatz von Ressourcen erfordert. Gerade gegen Ende einer Entwicklung wird so das Projektrisiko immer größer.

Diese Beobachtung beruht auf der Annahme, dass Änderungen an den Spezifikationen zuerst unabhängig von den Beweisen durchgeführt werden, und dass dann in einem zweiten Schritt die Beweise nachgeführt oder neu konstruiert werden. Wir schlagen deswegen vor, dieses Problem dadurch zu lösen, dass eine formale Entwicklung bestehend aus einer (möglicherweise fehlerhaften) Spezifikation und (möglicherweise unvollständigen) Beweisen als eine Einheit betrachtet wird und nach vorgegebenen Regeln transformiert wird. Diese Regeln werden Development Transformations genannt und sind operational motiviert: Wie würde ein Mensch intuitiv die Beweise anpassen, wenn eine in sich konsistente Transformation auf die Spezifikation angewendet wird? Die Transformationen sind auf der Ebene der Spezifikations- und Logiksprache formuliert. Damit werden sie unabhängig von der Anwendungsdomäne anwendbar.

Um den Ansatz mit dem Aufbau existierender Werkzeuge, wie zum Beispiel die in unserer Gruppe entwickelten Softwaretools VSE oder INKA/MAYA, vereinbar zu machen, führen wir ein dort zusätzlich vorhandenes Konzept als Mittler zwischen Spezifikation und Beweisverpflichtungen ein, so genannte Development Graphs. Development Graphs dienen einer logischen Strukturierung der Spezifikationen und der davon abgeleiteten Beweisverpflichtungen. Die Transformationen werden so erweitert, dass sie die zusätzlich eingeführten Development Graphs berücksichtigen. Das führt zu einer Trennung zwischen einem Rahmenwerk, das die Transformationen über generische Development Graphs hinweg propagiert, und einer

Referenzinstantiierung, die sich auf konkrete Sprachen und eine konkrete Repräsentation der Beweise festlegt. Die Referenzinstanz erlaubt die Behandlung von vielen in der Praxis vorkommenden Szenarien. Andere Instanzen können aus dem Rahmenwerk abgeleitet werden.

Wir entwickeln eine konkrete Instanz des vorgestellten Rahmenwerkes. In einer prototypischen Implementierung mechanisieren wir Transformationen der vorgestellten Instanz. So weisen wir nach, dass die vorgestellten Konzepte praktisch anwendbar sind, und Korrekturen und Ergänzungen der Spezifikation und der Beweise mit vertretbarem Aufwand möglich werden.

Acknowledgements

This thesis would not have been written without the help and support that I received. In particular, I am indebted to Professor Jörg H. Siekmann who let me be a member of his group in Saarbrücken and provided an inspiring environment in which to work, and to Professor Alan Bundy who hosted me as a visiting researcher in Edinburgh for a year. My work was substantially influenced by my stay in Edinburgh, and I am very grateful for the funding of the German Academic Exchange Service (DAAD). I would also like to thank Professor Bernd Krieg-Brückner and Dr. Alan Smaill for agreeing to serve as my examiners and for the feedback and encouragement that they provided.

I am particularly grateful for the support that my supervisor Dieter Hutter has provided over all those years. He has always been available to discuss ideas, provide feedback and offer advice whenever I needed it. I would also like to thank Dieter for the time and energy he spent on commenting numerous notes and drafts of this thesis. I guess he is at least as happy as I am that this is the final iteration.

I would like to thank Till Mossakowski for invaluable discussions and comments. In particular I am very grateful for the helpful feedback he provided on numerous notes that would eventually become the framework part of my thesis, and for his comments on the near-final version of the relevant chapter.

My thanks are due to the members of our group at the DFKI in Saarbrücken and the DReaM group in Edinburgh, both for input and for the positive and encouraging atmosphere. In particular, Serge Autexier, Heiko Mantel, Werner Stephan, Toby Walsh and Andreas Wolpers have provided helpful discussions and valuable insights.

Perhaps the most important source of help over all those years, however, was the support that Julia and my parents provided. If it was not for them, I would not be writing these words now.

This thesis was set in Palatino and Helvetica using \LaTeX , Makoto Tatsuta's proof package and Paul Taylor's commutative diagrams package.

Contents

Abstract	iii
Zusammenfassung	v
Ausführliche Zusammenfassung	vii
Acknowledgements	ix
List of Figures	xv
I Introduction and Motivation	1
1 Introduction	3
1.1 Software Engineering	3
1.1.1 Development Artifacts and Workflow	3
1.1.2 Software Development Process	4
1.2 Formal Methods	6
1.2.1 Formal Artifacts	7
1.2.2 Formal Methods and the Development Process	8
1.3 Management of Change	10
1.3.1 Development Graphs	10
1.3.2 Proof Replay and Reuse	10
1.4 Our Approach	12
1.5 Structure of the Thesis	15
2 Example Scenarios and Supporting Transformations	17
2.1 Overview	17
2.2 Example Scenarios	17
2.2.1 Changes Resulting from Corrections	18
2.2.2 Changes as Part of the Development Process	20
2.3 Support by Transformations	23
2.3.1 Extending and Restricting the Signature	25
2.3.2 Changing Existing Signature Entries	27

CONTENTS

2.3.3	Adding and Removing Axioms	28
2.3.4	Changing Formulae	29
2.3.5	Changing Induction Schemes	31
2.3.6	Completeness and Adequacy	33
2.4	Summary	33
 II Transformation Framework		35
3	Context and Overview over the Framework	37
3.1	Overview	37
3.2	Abstract Logic: Institutions	39
3.3	Development Graphs	41
3.4	Specification Language	49
3.5	Proof Representation	51
3.6	Formal Developments	55
3.7	Integration with Existing Tools	57
3.8	Transformations	58
3.9	Summary	61
4	Development Graph Transformations	63
4.1	Overview	63
4.2	Changing the Graph Structure	65
4.2.1	Adding and Deleting Nodes	65
4.2.2	Adding and Deleting Links	65
4.3	Changing the Content of Nodes or Links	67
4.3.1	Adding, Deleting, and Moving Axioms	72
4.3.2	Changing Axioms	74
4.3.3	Extending and Restricting Signatures	79
4.3.4	Translating Development Graphs	87
4.4	Generic Construction of Translations	94
4.5	Relation to Basic DG-Operations	96
4.6	Summary	97
 III A Reference Instantiation		99
5	Formal Developments	101
5.1	Overview	101
5.2	Concrete Logic	101
5.3	Concrete Specification Language	106

5.3.1	Specification in the Small	106
5.3.2	Specification in the Large	108
5.3.3	Mapping to Development Graphs	110
5.4	Concrete Proof Representation	111
5.5	Summary	117
6	Specification Transformations	119
6.1	Overview	119
6.2	Adding and Deleting Elements	120
6.2.1	Theories	120
6.2.2	Axioms	121
6.2.3	Signature Items	121
6.2.4	Uses and Satisfies Clauses	124
6.3	Changing Elements	126
6.3.1	Signature Item Names	126
6.3.2	Function and Predicate Arities	128
6.3.3	Generatedness Constraints	132
6.3.4	Formula and Term Occurrences	134
6.4	Summary	137
7	Proof Transformations	141
7.1	Overview	141
7.2	General Pattern of Proof Transformations	142
7.3	Adding and Deleting Assumptions	143
7.4	Mapping Proofs	146
7.5	Restricting the Signature	146
7.6	Translating Proofs	146
7.7	Changing Occurrences	148
7.7.1	Replacing Occurrences	149
7.7.2	Special Cases	162
7.7.3	Induction Schemata	164
7.8	Auxiliary Transformations	166
7.9	Summary	168
8	Mechanising Transformations	169
8.1	Overview	169
8.2	Original Specification	169
8.3	Missing Axioms	174
8.4	Missing Theory	177
8.5	Missing Slot	180
8.6	Missing Action	182

CONTENTS

8.7	Stronger Precondition	184
8.8	Summary	194
IV	Related Work and Conclusions	197
9	Related Work	199
9.1	Management of Change	199
9.2	Proof Reuse and Replay	200
9.3	Correctness-Preserving Transformations	201
9.4	Advanced Programming IDEs	203
9.5	Requirements Traceability	205
10	Conclusions and Outlook	207
10.1	Conclusions	207
10.2	Further Work	209
	References	213
	Index	225
V	Appendix	229
A	Category, Functor, Natural Transformation	231
B	Functors Lifted to Sets and Tuples	233
C	Details of the Definition of FolEqGen	237
D	Details of the Case Study	241
D.1	Development Trace	241
D.2	Developments	241
D.3	Transformations	247
D.4	Original Specification Text	247

List of Figures

1.1	Development artifacts and workflow	4
1.2	Waterfall-like process	5
1.3	Iterative process	5
1.4	Relationship between specification and proofs	8
1.5	Feedback from proofs to specification	9
1.6	Changes to specification triggered elsewhere	9
1.7	Traditional approach: edit and reuse	13
1.8	Proposed new approach: transformation	13
2.1	Example set of basic transformations	26
3.1	Visualisation of the state of a formal development	38
3.2	Example development graph	43
3.3	Development graph calculus example	47
3.4	Commuting diagrams for η and <i>concl</i>	54
3.5	Development well-formedness	56
3.6	Development transformation	57
3.7	Layers of development transformation	59
4.1	Induced transformations	64
4.2	Respecting morphisms	69
4.3	Moving axioms	73
4.4	Signature adjustment respects the morphisms	81
4.5	Moving signature symbols from one node to another one	82
4.6	Signature adjustment: sentence functor	83
4.7	Signature adjustment: sketch of inherited axioms	84
4.8	Translation of global axioms	91
4.9	Extended proof representation conditions	92
4.10	Definition of translations	95
4.11	Uniqueness of definition of translations	96
4.12	Development graph transformations	98

LIST OF FIGURES

5.1	Logic language abstract syntax	105
5.2	Specification language abstract syntax	107
5.3	Proof calculus rules	115
6.1	Adding links	125
6.2	Realization of adding links	125
6.3	Renaming signature items	128
6.4	Development graph translation for adding arguments	132
6.5	Specification transformations	139
6.6	Dependency between concrete transformations	140
7.1	Sentence replacement for context and focus	157
8.1	Initial development graph	172
8.2	Initial proof	173
8.3	Proof with completed subproof for b-1	176
8.4	Development graph with boolean	178
8.5	Development graph with added link	179
8.6	Proof with crash action	183
8.7	Proof with weakened property	186
8.8	Part of the original proof for (a)	187
8.9	Part of the new proof for (a)	189
D.1	History of example	242
D.2	Development graph	243
D.3	Contents of nodes	244
D.4	Proof tree	245
D.5	Proof obligation	246
D.6	Goal with missing axioms	248
D.7	Goal with added axioms	249
D.8	Goal with strengthened preconditions	250

Part I

Introduction and Motivation

Chapter 1

Introduction

1.1 Software Engineering

Software engineering is concerned with developing software systems that satisfy customers' needs and expectations. The crucial technical questions are *what* needs to be built and *how* it can be constructed in a goal-directed way. The things that are constructed are usually called development *artifacts* (or deliverables, if they are handed to or discussed with the customer). Which artifacts are derived from others and how is captured in the *workflow*, and the distribution of the activities over time is called the development *process*.

1.1.1 Development Artifacts and Workflow

Complex and mission-critical systems should almost always be developed in a top-down fashion: requirements are gathered and the system is built such that it meets these requirements. This involves core development activities, e.g. requirements analysis, design, implementation, or integration [Som95], [Bal00]. These activities produce documents that are related: the design should meet the requirements, for instance, and the implementation should implement the design. In a top-down approach, development activities produce more concrete artifacts from more abstract ones along these relationships: the design activity, e.g., derives a design document from the requirements document. This is called workflow and is illustrated for a very coarse-grained set of artifacts and workflow activities in Figure 1.1.

In addition to the workflow activities, there are also validation and verification activities. These are concerned with whether the artifacts are

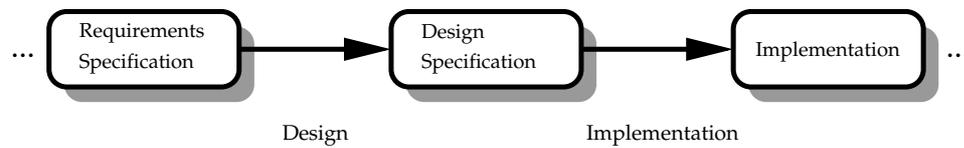


Figure 1.1: Development artifacts and workflow

internally consistent and the relationships between them are as required. For example, design reviews ensure that the design document is consistent and satisfies the requirements; unit tests ensure that the implementation of a component does what the design document says it should do.

1.1.2 Software Development Process

Given a fixed workflow, a development process describes or legislates how the different activities are spread over time. The simplest possibility is a waterfall-like process (cf. [Roy70]), where each artifact is completed and frozen before a more concrete document is derived from it. Thus, workflow activities correspond to distinct phases in the process. Another possibility is to proceed in iterations (or spirals, cf. [Boe88]) and produce new versions of the artifacts in each iteration. These two possibilities are extremes, and intermediate possibilities exist. Figures 1.2 and 1.3 illustrate a waterfall-like and an iterative process. Criteria to choose between these possibilities include process visibility, the level of goal-directedness that can be achieved, and the necessary development resources. These are, of course, dependent on assumptions about the system to be built and the stakeholders in the project.

As far as resources are concerned, a *waterfall-like* process assumes that it is possible to cast, e.g., the design in stone before the implementation work has started. This means that each artifact is built only once, and no duplicate work is spent on updates further down the workflow line: the downstream artifacts do not exist yet. The perceived problem with the waterfall process is that it does not work very well in practice: experience shows that downstream activities are going to reveal errors in earlier artifacts no matter what, at a time when no resources are left for fixing them. This results in an inacceptably high risk for the project.

An *iterative process*, on the other hand, assumes that the design is unlikely to be correct without insights resulting from implementation activities. Thus, starting implementation work before the design is complete is considered preferable to completing a misguided design first. The perceived problem with the iterative process is that constantly revising arti-

1.1. Software Engineering

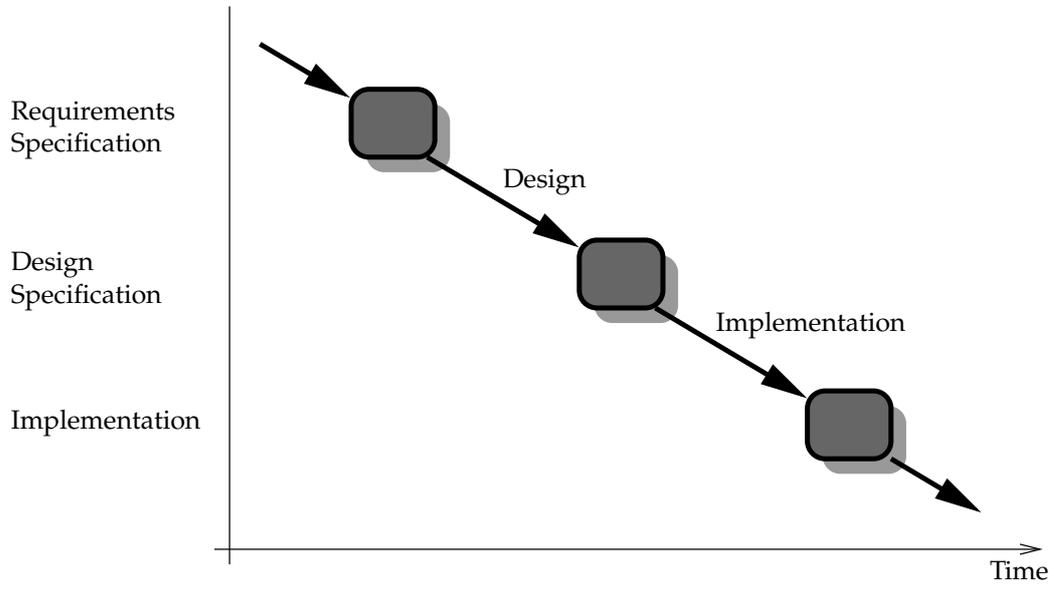


Figure 1.2: Waterfall-like process

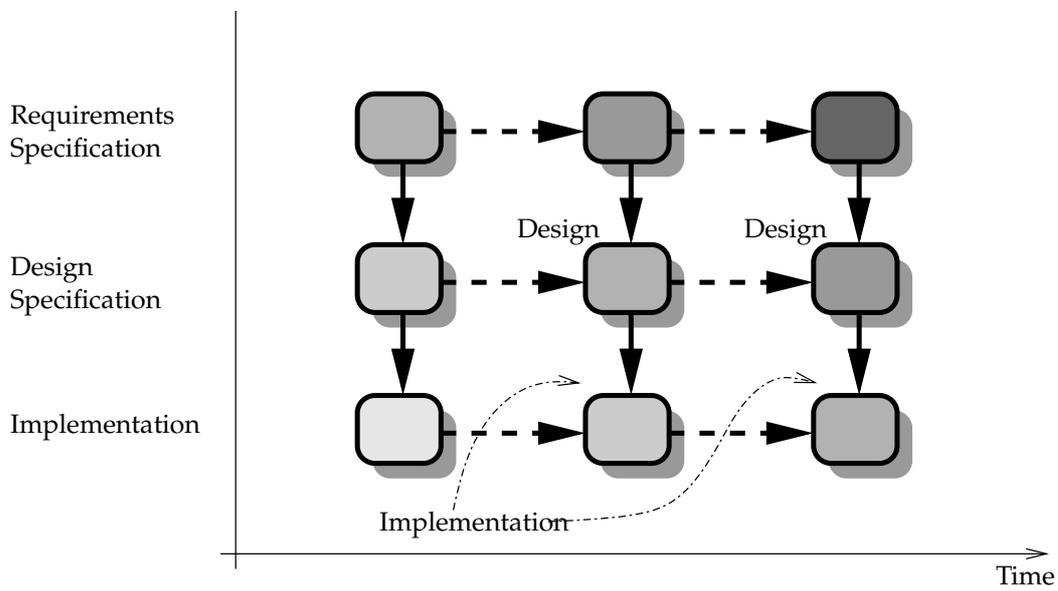


Figure 1.3: Iterative process

facts and updating all downstream artifacts accordingly simply costs too much.

Almost all current processes strive for an iterative, or evolutionary, process, cf. [Boe88], [JBR99], [Bec00], [LKB02]. This means that the workflow and validation and verification activities need to address the incremental nature of the process: they need to make use of the respective artifact from the previous iteration, because it would be too expensive to produce all artifacts from scratch in each iteration. There is a whole wave of proposed development strategies that promise to make changes cheap [McC96], [Bec00], [Hig02], [Lar04]. These are called *agile* processes. As usual, cautiousness is required with the extreme positions and the benefits that the proponents claim for their favourite ideology; as an example compare the Extreme Programming literature, e.g. [Bec00], with a critical view, e.g. [SR03].

Even though, it is generally accepted today that a pure waterfall model is inadequate, and that it is an important goal to make making changes cheaper so as to enable an iterative, evolutionary process.

1.2 Formal Methods

Formal methods can be used throughout the whole development to improve the quality of the artifacts and the final product. For example, the requirements document and the design document can both contain a formal model of the system to be built on different levels of detail and abstraction. Because of the formal syntax and semantics, the meaning of the models is unambiguous. It is thus possible to analyse with mathematical rigour, whether the design satisfies the requirements. Similarly, it is possible to analyse with mathematical rigour whether each model is complete and internally consistent. This provides a way to introduce objectivity and rigour into an otherwise subjective and informal review process.

Experience shows that whenever informal documents are formalised and scrutinised rigorously, unexpected errors and omissions show up. The main benefit of formality, therefore, is not primarily that a specification is provably correct – as is often claimed. Rather the main benefit is that errors, omissions, and misconceptions, which would otherwise go undetected, are found as early as possible. It is argued, therefore, that the biggest benefit of using formal methods is on the abstract development levels, where no executable code exists that could be tested [Rus01].

1.2.1 Formal Artifacts

In a formal setting, the specific formal artifacts are *formal specifications* (sometimes also called models) and formal, mechanised *proofs*.

A formal specification captures, in a formal language, relevant aspects of the system and its relevant properties. A formal specification can contain, e.g., a formalisation of the requirements, a formalisation of the design, and a formal statement of the postulate that the design satisfies the requirements. This example of a formal specification is relevant for the design activity: the postulated property ensures that the relationship between requirements and design is as intended. Similarly, on another level of abstraction, a formal specification models design and programs and postulates that the implementation satisfies the design. These postulates link two different informal artifacts. One specification thus typically consist of parts that are associated with different informal artifacts. Additionally, specifications also contain formal postulates that correspond to properties of a single informal artifact, e.g. its internal consistency, or the claim that some aspects are consequences of others. Examples for the latter are security models, which describe the requirements that should be met by the design in terms of security functions that are expected to enforce a certain property, which cannot be implemented directly.

Given a specification, postulated properties give rise to *proof obligations*: the resulting proof obligations are sufficient conditions for the postulated properties to hold. They are derived mechanically from the specification and are formulated in a logical language. They propose that a certain conclusion follows logically from assumptions, where the notion of logical entailment is formally captured by the logic.

Formal proofs are objects that represent evidence for such a proposition. Partial proofs are proofs that have “holes” representing open goals still to be proven. These holes need to be closed before the proof is complete. While working on a development, many proofs are partial proofs. The failure to find a proof for a proof obligation often means that the obligation is mistaken, pointing to a problem in the model or property statements of the specification. The relationship between specifications and proofs is illustrated in Figure 1.4.

Tool support for this approach is available and is used by now in an industrial setting for developments that are mission-critical, e.g. [SRS⁺00], [SRS⁺02]. Users of a support tool like, e.g., the Verification Support Environment (VSE, [AHL⁺00]) enter structured specifications with models and properties. They let the system generate proof obligations and then use the reasoning subsystem of the tool to discharge the proof obligations by con-

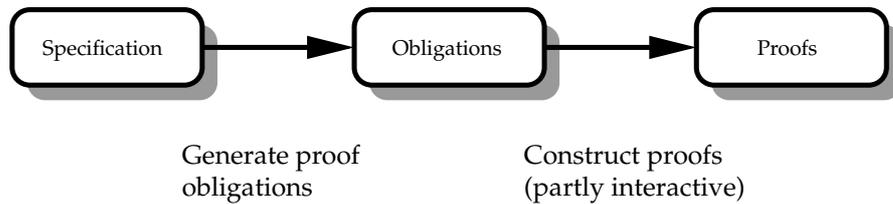


Figure 1.4: Relationship between specification and proofs

structuring formal proofs. Experience shows that, despite of the heuristics used, finding proofs cannot be fully automated and requires a significant amount of user interaction.

1.2.2 Formal Methods and the Development Process

The choice of a development process has, of course, consequences on how formal methods fit into the development as a whole. For a *waterfall-like* process, it is important to get early, abstract artifacts right because the earlier an error is introduced the more expensive it is to correct later. In this situation, formal specifications and proofs are used to “debug” the artifacts [BN04]. The consequence is that formal specification and proof work are necessarily evolutionary in nature: they are more like a formal reflection about the current state of the development in progress, rather than a way to prove facts that were already evident beforehand. Whenever a problem is detected, i.e. a proof cannot be constructed as expected, the specification is changed accordingly. This leads to changed proof obligations, which in turn invalidate the respective proof. This is illustrated in Figure 1.5.

For an *iterative* process, the formal specification changes with each iteration anyway. These changes may either be the outcome of a formal activity, e.g. a bug is detected when a proof fails (cf. Figure 1.5), or may be due to other forces (cf. Figure 1.6), e.g. changed requirements due to a changing business context or additional functionality that is introduced within the current iteration.

In any case, the consequence is that specifications and their associated proof obligations necessarily change frequently in a formal development process. It is, therefore, not acceptable to construct proofs from scratch each time the proof obligation changes. The cost and the project risk associated with these changes would be too high. This is independent of the chosen process and the current phase or workflow activity.

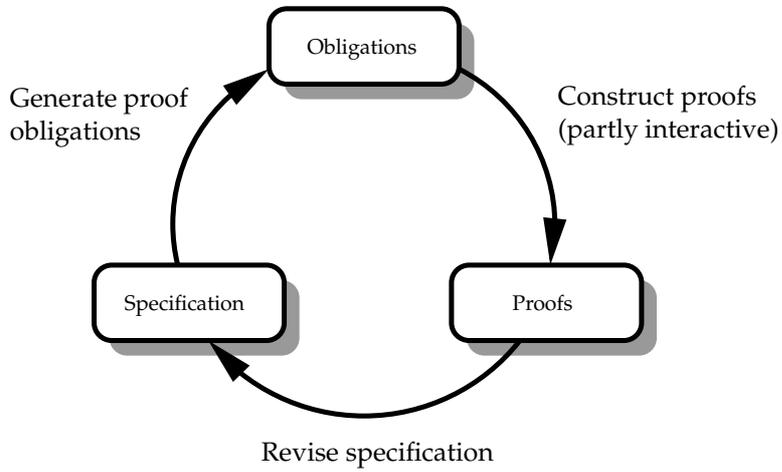


Figure 1.5: Feedback from proofs to specification

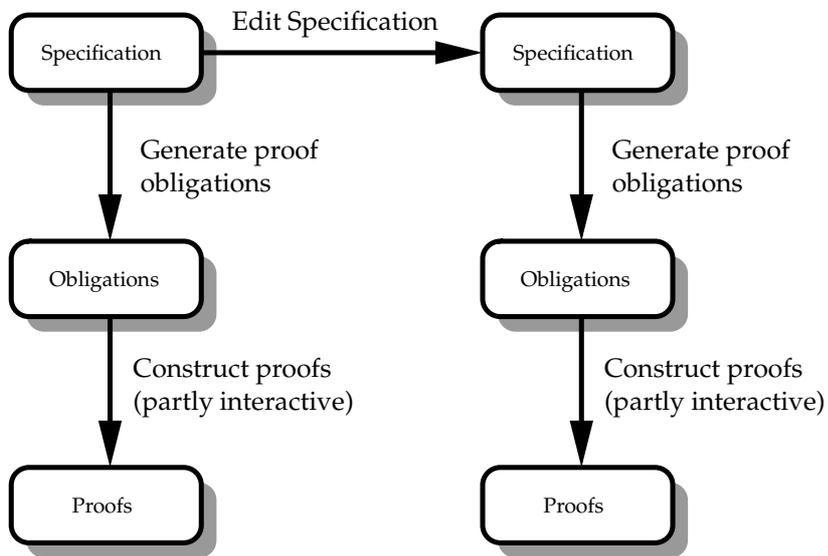


Figure 1.6: Changes to specification triggered elsewhere

1.3 Management of Change

Management of change is the name used for concepts and techniques that aim at reducing the cost of changes to formal specifications. The idea invariably is to make sure that constructing the proofs as illustrated in Figures 1.5 and 1.6 does not start from scratch but rather reuses the old versions of the proofs. There are two main ideas that have been followed in the existing literature. If an obligation is not changed (or is changed in a way that is transparent to the associated proof) the proof can be kept. Another idea is to derive the new proofs from information that is available in – or about – the old proofs.

1.3.1 Development Graphs

Development graphs [Hut00] have been introduced to exploit the structure of a formal specification to minimise the effect of changes. Like any other description of a software system, formal specifications present the system in a structured way. They divide the system description into modules that are connected by explicit interfaces. Since the presentation is formal, the dependency of modules on other modules can be extracted and encoded in a development graph. In the graph, nodes correspond to modules and links represent different kinds of dependency relationships.

Proof obligations are now computed indirectly from the development graph rather than directly from the specification. Each proof obligation is rooted at a link. This provides a notion of locality: changes to the graph only affect those proof obligations that are reachable from the changed part of the graph. Proofs associated to links that are unreachable can be determined as still valid by reasoning about the *structure* of the graph without considering the *contents* of the modules at all, cf. [AHMS00], [MAH01], [AH02], [MHAH04]. This is implemented in the MAYA system [AHMS99], [AHMS02] and the idea is also used in the correctness management of the VSE system [AHL⁺00]. The approach makes a huge difference and is indispensable in practice for realistically sized developments. However, it still invalidates those proofs that are reachable, whether they are essentially affected or not.

1.3.2 Proof Replay and Reuse

Another idea is to use the old version of proofs to construct a new proof when the proof obligation has changed.

1.3. Management of Change

After a proof has been constructed in a theorem prover, using interaction by the user and proof automation, there is knowledge about the proof itself and about how it was found. The latter may consist of, say, a proof script that reproduces the proof when it is run by the theorem prover on the original proof obligation: it simulates the choices and inputs of the user. The former might be a proof object that represents the detailed proof rules that were applied, in which case the proof can be inspected and checked easily. Not all systems represent or store both kinds of information, though. Also, the distinction between the two kinds of information is not so clear as may seem at first: a proof object can be seen as a very detailed proof script that eliminates most or all search; and a proof script can be seen as a way to externalise the proof object (e.g. to store it on disk).

When a proof obligation associated with a proof has changed, the proof script can be rerun on the changed obligation. The hope is that the script represents the essence of the proof that still works for the new proof obligation, and therefore a proof for the new obligation will be produced. Usually, the script needs to be adapted before it can be replayed. Depending on how detailed the proof script is, this is called *proof reuse* or *proof replay*. In special cases, heuristics are available to reuse and partly adapt an old proof to the new proof obligation. In essence, these heuristics redo the search that was done by the proof automation in the original proof and try to adapt the user's previous inputs where possible [RS93], [FH94], [KW94], [Kol97], [MW97], [Sch98], [MS98], [BK04].

While these methods for heuristic replay are an important help where applicable, they face the problem that changes can have a dramatic effect on the search space. It turns out that they are not appropriate as a general tool for the organised adaption of proofs in the context of the formal software development. For example it is a well-known fact that adding axioms – i.e. changing the theory monotonically – changes the search space so that proof scripts often fail to reprove the *unchanged* conjecture, although clearly the proof which the script found in the first place is not invalidated by an additional assumption. An example was discussed on the Isabelle mailing list.

“The automatic proof of Cantor's theorem is very fragile, since it has to construct the diagonal set by unification. When you replace the type 'a by nat, you greatly increase the search space, since there are many rules that specifically apply to the type of natural numbers.” [Pau00]

This is a problem of redoing too much search when replaying the script.

An idea for fixing this is to tighten the control of the replay by representing more details. But this also has its problems:

“Isabelle does not (unlike, say, PVS) let you refer to assumptions by number. That leads to brittle proofs that cannot be re-played if you make minor changes.” [Pau00]

The result is that in practice after changes, proof scripts are often patched, or other proof representations are adapted, manually and ad hoc. What is desperately needed is support for the organised adaption of proofs that can be used as a matter of routine and that is aware of the type of change being made, in order to avoid time-consuming manual, ad-hoc patching. Our suggestion is to use transformations on whole developments, i.e. transform specification and proofs as one entity.

1.4 Our Approach

In current support systems, changes to a formal development are carried out by editing the formal artifacts, e.g., in a text editor. The support system then computes new proof obligations. In a second, decoupled step, the proofs have to be adapted. This is in contrast to how changes to a development are discussed informally on a blackboard, where changes to a specification are considered together with the effects of these changes to the property of the system and their proofs: adding a new state transition is going to add another case to case distinctions over the possible transitions. Only that new case is considered.

Therefore, we propose to view the state of a development as one entity consisting of a specification *and* the proofs. This means that proofs are an integral part of the formal development artifacts, rather than being conceptually separated as in traditional approaches. Instead of editing the specification and adapting the proofs in a second step (cf. Figure 1.7), we propose to change the entire development using preconceived formal transformations that map a development consisting of specification and proofs to a new development (cf. Fig. 1.8).

Because such a transformation knows what change is carried out on the level of the formal specifications, it can determine what needs to be changed on the level of proofs. Thus, proofs will be patched as necessary. Of course, in general this will introduce holes in the proofs that need to be looked at afterwards. However, large parts of the old proof can be reused.

Over time, the evolution of the formal artifacts is a sequence of developments where each of the developments results from applying a trans-

1.4. Our Approach

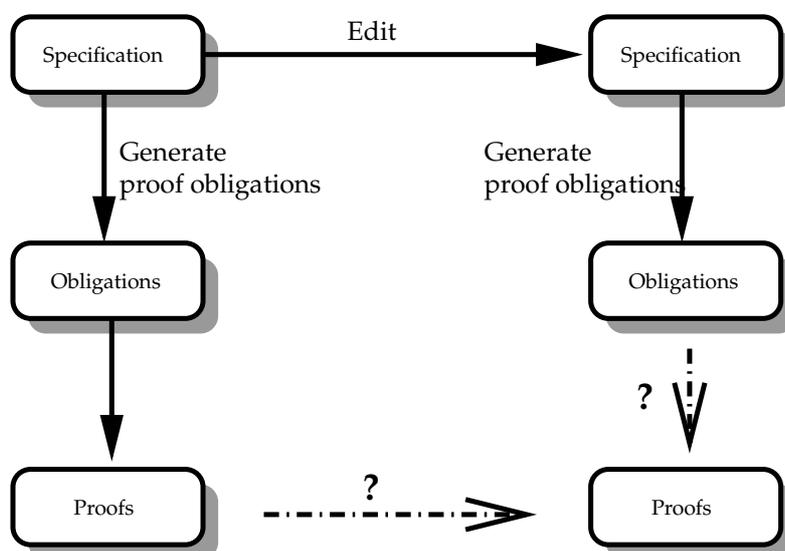


Figure 1.7: Traditional approach: edit and reuse

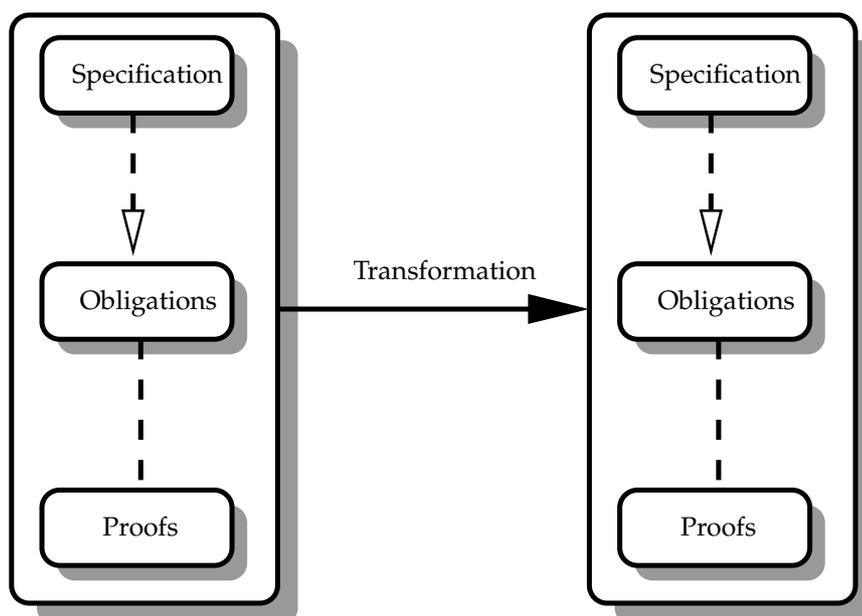


Figure 1.8: Proposed new approach: transformation

Chapter 1. Introduction

formation to the preceding development. Each of these developments is a self-contained entity without reference to the one it resulted from. Therefore, the history is not needed to determine the meaning of a development. Similarly, the development alone includes all the information needed to determine whether all proof obligations have an associated proof that is complete. The proposed approach is compatible with the architecture and methodology of existing support tools. In particular, it integrates well with development graphs, and it is completely orthogonal to any existing proof replay or reuse. This means that transformations can be used *in addition* to any other existing methods for manipulating formal artefacts that are provided by a tool.

Each development transformation rule, or simply development transformation, is a binary relation over well-formed developments. If two developments are in relation, we also say that the first development can be mapped to the second one by the transformation. In practice, we are interested in the special case of partial functions. In particular this means that each development transformation has to map a specification from its domain to another specification, and it has to map (partial) proofs for the old specification to (partial) proofs for the new specification. This has several consequences. On the one hand, for each change to the specification, proofs are adapted automatically as far as possible by the respective development transformation. In order to be able to adapt proofs *usefully*, development transformations need to use information about the relationship between the old and new specification. Also, it is unreasonable to expect useful adaption of proofs of arbitrary changes to the specification. Thus, useful transformations have to restrict the ways in which specifications can be changed compared to what is possible when freely editing specification text. On the other hand, each change a user wants to make to the specification needs to be supported by a development transformation, unless adequate support is provided already by existing techniques of course. Therefore, it is essential to find an adequate set of transformations that is restricted enough to enable useful patching of proofs and that is powerful enough to enable users to carry out the changes they want to make. These are conflicting interests which need to be balanced against each other.

Our proposed solution to this conflict is to provide a set of transformations that correspond to elementary changes of the specification. In the context of the axiomatic specification of abstract datatypes (ADTs), such elementary changes are, e.g., adding or removing theories from a specification, adding and removing axioms, replacing subformula occurrences in an axiom of a theory by another formula, or changing the signature of

a theory by making a binary predicate ternary, i.e. adding an additional type to the arity of the predicate.

The proposed set is complete in the sense that for each well-formed development $dev = (S, P)$ and for each well-formed specification S' , there is a development $dev' = (S', P')$ that results from dev by applying exclusively development transformations from the set. In order for the set of transformations to be adequate, we need to provide evidence that for practically relevant transformations from dev to dev' , closing the resulting proofs P' is less time-consuming than following the traditional approach. We will do this by revisiting example scenarios that are the motivation for our work and discussing the treatment of these examples with the set of transformations.

1.5 Structure of the Thesis

The rest of this thesis is structured as follows. Chapter 2 describes example scenarios that are representative for the kinds of changes appearing in practice and motivates the use of transformations to support the changes. Our approach is then developed in detail in Part II and III.

Part II describes an abstract formulation of the ideas as a framework that is to a large extent independent of the concrete specification language and proof formalism. Chapter 3 describes abstractions of specification language, development graphs, logic, and proof calculus. This forms the basis for a detailed description of development graph transformations together with their interfaces to abstract specification and proof transformations in Chapter 4.

In Part III, we describe the details of an instance of the framework for a concrete choice of specification language and proof representation, which is described in Chapter 5. Using this instantiation, Chapter 6 works out specification transformations and associates them with the resulting transformations to proof obligations. This is afforded through the development graph transformations that have been described in Part II. Chapter 7 then provides the proof transformations that are needed to propagate the specification transformations to the proofs. Finally, Chapter 8 revisits the scenarios for which our approach has been developed. It presents example transformations in the context of a small case study and shows how these transformations are successfully applied in the context of a concrete specification and proofs.

Part IV concludes the thesis, discussing related work, our contributions, and questions that provide the starting point for further work.

Chapter 2

Example Scenarios and Supporting Transformations

2.1 Overview

Many formal developments for industrial projects and academic case studies have been carried out by our group over the years: published examples include [RSW⁺99], [LUV00], [MG00], [RSB00], [SRS⁺00], [MSK⁺01], [SRS⁺02], and [Sch03]. These cover different application domains like, e.g., robot controllers, chip cards for digital signatures, operating systems, or multi-agent systems. Different specification formalisms and languages, and different proof calculi and representations were used. In all of them we had to deal with changing specifications and the need to adjust the proofs.

In this chapter, we first present examples that are representative for the kinds of changes that occurred regularly and frequently. The examples are presented in the form of brief scenarios. Then, we show how these types of changes are supported by transformations in the approach that we propose in this thesis.

2.2 Example Scenarios

We distinguish two types of changes based on whether they represent corrective actions or whether they are anticipated as part of the development activity. Section 2.2.1 shows examples of changes that are due to errors that were noticed when proofs failed. Section 2.2.2 contains examples where the development methodology itself presupposes changes to the specification. The classification is independent of whether the overall development

process is waterfall-like or iterative.

2.2.1 Changes Resulting from Corrections

Invariants for State Transition Systems. The first example is an explicit specification of a state transition system. We specify states and possible state transitions as predicates on prior and resulting states. This induces a definition of possible system traces, i.e. of a set of all those sequences of states that describe a possible execution of the system with the given state transitions. Postulated properties are formulated as invariant properties: each state in each system trace is required to satisfy a predicate. Detailed examples can be found, e.g., in [MG00], [SRS⁺00].

In the specification, state transitions are divided into classes that can be specified uniformly. For a specification of a file system, we specify state transitions that correspond to creating, reading, writing, and removing files. The pre- and postconditions of, e.g., all write-transitions are specified together: the concrete data to be written and the file to write to are parameters of the definition. The other transitions are specified similarly. A straightforward proof of an invariant now proceeds by an induction over the length of the sequence and a case distinction over the different classes of possible transitions for the step case. There is one subproof corresponding to the case for the create-transitions, one for the read-transitions and so on. Each of these subproofs establishes that the invariant property is in fact invariant over transitions from the respective class, where the parameters are universally closed over.

After finishing the subproofs for the create- and read-transformations, we discover that we cannot finish the subproof for the write-transition because of an error in the specification of the transition relation. We therefore change the post-condition to correct the error and reconsider the subproof for the write-transition. Informally, we argue that changing the pre- or postcondition of one class of transitions, e.g. the write-transitions, only has an effect on the subproof for this very transition class, but not on the other subproofs. In other words, we know in advance which parts of a proof are possibly invalidated by the change. Even more importantly, we know that the rest of the proof is still valid.

Similar effects can be seen by looking at typical forms of the invariants: quite frequently, an invariant property is a conjunction of several predicates on states. Each of these conjuncts is established separately and results in a subproof for the respective conjunct. A subproof can refer to the complete induction hypothesis, i.e. to an assumption corresponding to

2.2. Example Scenarios

each of the conjuncts. In general, this is needed. In practice, only some of the assumptions are used in each subproof, however. A change to one conjunct in the invariant property invalidates the subproof for the respective conjunct, and it also potentially invalidates all the other subproofs, because the induction hypothesis changes as well. Subproofs that do not refer to the hypothesis for the changed conjunct are still valid, however. And even those subproofs that do refer to the changed hypothesis are not necessarily invalidated as a whole; depending on how and where in the proof the hypothesis is used, we expect some part of the subproof to be still valid. In this case, however, we do not know in advance, which part of the proof can be kept, but we expect to be able in principle to find out by inspecting the proof in detail.

In the course of the development, sometimes an invariant turns out to be too weak, and an additional conjunct is added. This adds a case, and thus a new subproof, to the proof. It also enlarges the set of available assumptions in subproofs, because the conjunct is also added to the induction hypothesis. Similarly, removing a conjunct (because we discovered that the invariant is actually too strong) removes one case and the corresponding subproof, but also retracts assumptions from other branches of the proof that are no longer available. Some branches do not use the respective part of the induction hypothesis, and therefore we expect substantial parts of the proof still to be valid.

High-level Specification and Consistency Meta-Proofs. In [LUV00], a generic chip card for digital signatures was specified as a reference model for a standardisation effort. Many proofs were constructed to the effect that the properties claimed by the standardisation document were actually met by an abstract state machine described also in the document.

As one issue with the adequateness of the specification, the consistency of the model was analysed: if it was possible to construct a refinement that is executable, the specification would be known to be consistent. In the course of providing an executable model however, it was found that a part of the specification could not be implemented: some cryptographic functions were over-constrained, rendering the specification in fact inconsistent. However, only a consistent subset of the constraints was needed for the proof that dealt with the primitives. All other proofs simply ignored the specification of these primitives.

In the given situation, the axiomatisation of the cryptographic primitives was weakened, and a lemma was proven so that it matched the subset that the existing proof used. Finally, the weaker cryptographic primi-

tives were refined to complete the model.

None of this was actually problematic or interesting in any sense; these were tedious technical changes that affected details in large portions of the proofs, however. The only interesting point was that the new axioms implied the lemma that was used in the original proofs. Nevertheless, the effort required for fixing this was considerable, since many of the proofs had to be patched ad hoc.

2.2.2 Changes as Part of the Development Process

Information Flow Proofs by Unwinding. Some of the developments, e.g. [SRS⁺00], [MSK⁺01], [SRS⁺02], and [Sch03], dealt with the verification of information flow properties [Man03], [MS05]. Simplified, an information flow property is a closure property of sets of execution traces of state-transition systems. Such properties are usually not proven directly; rather a technique called *unwinding* is used. Using an *unwinding theorem*, the closure property is reduced to conceptionally simpler properties over single state transitions from one state to another one. These properties are called *unwinding conditions*, and examples are given in [Man00], [Man03]. Their exact form does not matter much for our purposes.

However, all of those conditions involve binary relations over states, called *unwinding relations*. A helpful, but only partially correct, intuition for these relations is that they relate a state to other states from which observers can extract no more information than they can in the original state. For each possible abstract observer, a different relation is needed. An unwinding theorem essentially says: if there exists binary relations such that the unwinding conditions are satisfied, then the information flow property holds.

The main problem here is that of finding unwinding relations for each observer, so that the unwinding conditions can be verified. It turns out that the intuition does not suffice to guess appropriate relations, so a trial-and-error approach is chosen. As is reported in [SRS⁺00], [Sch03], most effort is actually spent in trying to find proofs for the unwinding conditions and change the unwinding relations whenever a goal cannot be closed with the current guess. Since the relation appears in assumptions and in conjectures, it is not clear in advance whether a stronger or a weaker definition is better. A typical way to start is to define, for each observer, the relation to be the equality relation and then adjust it when necessary.

After a couple of iterations, the definition of the relation spans a page of specification text and more, and is by case distinctions over character-

2.2. Example Scenarios

istics of the state [HMSS05]. Each of the cases is again a complex formula. In the process of working out a viable relation, cases are added or existing ones changed. Again, when we change one of the cases, we expect the sub-proofs that we have completed so far using another case of the definition to be unaffected.

Usually, not only do we need to modify the definition of the unwinding relation, but we also find omissions and errors in the specification of the transition relation similar to the discussion of the invariant proofs in the preceding section.

Modelling Fault-Tolerant Systems. As the last example scenario we consider the case study [MG00] on modelling reliable broadcast using the technique of *fault models* described in [Gär99]. We will describe this example in more detail than the other example scenarios because we shall use it as the basis for giving the overview of our approach in Section 2.3. The presentation follows our simplified reconstruction [SH02] of the original case study.

The scenario consists of two steps: first a system is modelled without reference to the faults it is supposed to be tolerant of. Then the system specification is adjusted to account for the possible fault, taking advantage of the insights gained from the first model.

In the first step, a network of processes running concurrently is formalised and verified. Processes, connected by directed channels, run a local non-deterministic program to deal with arriving messages and to resend them. It is verified that only messages that have been broadcast to the network are delivered to processes, and that they are delivered to each process at most once. The system model is that of a state transition system described by an initial state and possible state transitions that the processes can take, called actions. The state transition system is explicitly specified using axiomatic specifications of abstract datatypes.

The required property of the network is specified as a property of states: a state s is safe iff only messages that were actually broadcast are delivered to any process p , and if no message delivered to any process p is delivered more than once. This is expressed by

$$\forall s : State. \text{safety}(s) \Leftrightarrow \begin{cases} (\forall p : Proc. p \in \text{procs}(s) \Rightarrow \text{delivered}(p, s) \subseteq \text{broadcast}(s)) \\ \wedge (\forall p : Proc. p \in \text{procs}(s) \Rightarrow \text{nodups}(\text{delivered}(p, s))) \end{cases} \quad (2.1)$$

All states that can be reached from the initial state are postulated to have

Chapter 2. Example Scenarios and Supporting Transformations

this property, so the proof obligation roughly reads

$$\forall tr : Trace. admissible(tr) \Rightarrow safety(res(tr)) , \quad (2.2)$$

where $admissible(tr)$ is true iff tr is a possible system behaviour, and $res(tr)$ is the resulting state after the actions in tr have been taken, i.e. each admissible sequence of states ends in a state satisfying the property. The proof is by induction over the length of traces. In the step case, the proof is split by a case distinction over possible actions, and for each action it is shown that it conserves the safety property.

In the second step, fault assumptions are added to the specification: processes are no longer assumed to work reliably, rather each process can either be up (i.e. still be running) or down (i.e. have crashed). This is done by adding parts to the specification related to the added “functionality,” but it also involves changing parts of the existing specification, cf. [MG00, Sect. 4.1]. In particular, the representation of processes in the state is changed to hold additional information about whether the respective process is up or down, and a predicate $isup$ is defined so that $isup(p)$ holds iff the process p is up. Also, an additional crash action is added. It can be executed by running processes, and its effect is to crash the process. All other actions are restricted to be executable by running processes only, thereby strengthening the preconditions of the actions. Obviously, after these changes the safety property given by (2.2) and (2.1) does no longer hold and the verification proofs are no longer valid.

According to the methodology adopted by [Gär99], the property is replaced by a weaker property: in (2.1),

$$delivered(p, s) \subseteq broadcast(s)$$

is weakened to

$$isup(p) \Rightarrow delivered(p, s) \subseteq broadcast(s)$$

and similarly for the other conjunct. As a consequence of these changes, the proof for (2.2) now has an additional case for the crash action. Also, it remains to be shown that the weaker safety property is sufficient for the induction to go through in the cases for all other actions. Since the induction hypothesis is weakened, too, this is not completely obvious. The additional case to prove is, however, that a process was up before an action when it is up after the action, and this turns out to be provable rather easily. It still has to be addressed in each subproof. The proof idea and the overall structure of the proofs are essentially unchanged, however.

There are a number of technical consequences that are trivial to hand-wave over in an informal description of the changes. The changes turned out to be hard to carry out efficiently and reliably in a mechanised theorem prover. Examples for problems include changing existing symbols of the signature and induction schemata.

2.3 Support by Transformations

The example scenarios we have presented suggest that certain types of changes appear over and over again, and in different contexts. For one thing, changes to certain parts of the specification correspond to certain parts of the proofs. In some cases we know which portions of the proofs will likely be affected and which ones will be guaranteed not to be affected, in other cases we do not know, but we know how to find out by looking at the proofs in detail, at least in principle.

Another recurring theme is that the whole specification is changed systematically. This changes technical details all over the specification and proofs, but the *relevant* changes are localised. An example appeared explicitly with the failure transformations, where processes were changed to either be up or down, but the only real change was how this interacted with the predicate *isup* and the crash action. Similar effects, however, also appear in the other scenarios: adding, e.g., a transition to a state transition system changes the induction scheme – and this only really matters where a case distinction is being made.

Support for some of these changes can be provided by observing *non-dependencies* between changes and proofs. Development graphs afford this on the level of formulae and proof obligations cf. [Hut00], [AHMS00], [AH02]: if an assumption is changed that is not visible in a proof obligation, the corresponding proof can be kept unchanged. As we have shown, the same principle can also be invoked at a more fine grained level for parts of axioms and parts of proofs.

However, we claim that for the scenarios presented in the preceding section, another view on the problem is more adequate. The reason for this is that, intuitively, proofs can sometimes be kept because of known *dependencies* between changes and proof obligations. We would expect there to be some relationship between the two views: non-dependency is a special case of a known dependency – the trivial one. Therefore, the approach we suggest allows, as special cases, those changes that make use of known non-dependencies.

We have presented our examples on a fairly abstract level, without ref-

Chapter 2. Example Scenarios and Supporting Transformations

erence to a concrete specification language and proof calculus. We hope it has thus become plausible that the abstract effects are independent of the concrete setting. However, to show that the abstract findings transfer to the concrete, and that our approach can handle them, we need to commit ourselves to some concrete choice of a formalism. We have decided to look at algebraic specifications of abstract datatypes, cf. [EM85], [LEW96].

Specifications will deal with *signatures* and *sentences*. A signature determines types, and functions and predicates with their arities. Sentences include formulae and generatedness constraints. They play the role of assumptions or conjectures. In this context, changes to a specification can have the following consequences for the resulting proof obligations.

- Additional proof obligations can be generated or proof obligations can disappear by adding or removing conjectures.
- Assumptions can be added or removed from proof obligations by adding or removing axioms.
- Formulae in proof obligations can change.
- Induction schemata can change.
- The language for the assumptions and the conjecture can be changed, e.g. by adding or removing signature symbols, or by changing existing ones.

In general, changing the specification will have more than one of the effects.

The first effect is rather uninteresting from our perspective: when a new obligation is created there is no old proof to make use of. Similarly, when an obligation disappears, there is nothing to reuse the existing proof for. We will, therefore, concentrate on the other effects and will motivate in the rest of this section how *transformations* on specification and proofs can be used to provide support. Since our emphasis is on changes that are independent of the structure of the specification in terms of modules, it is not necessary to complicate the exposition with details about structuring mechanisms; the general problems that need to be solved can be shown with unstructured specifications. However, in the rest of the thesis we will present our problem and the suggested solutions in the context of structured specifications and development graphs.

It is not feasible to provide separate specialised transformations for each particular application domain. Instead, we will provide a set of *basic transformations* that are specific to the specification language, the way

2.3. Support by Transformations

proof obligations are generated, and the notion of proof, but not specific to an application domain. Basic transformations can be used in sequence to achieve the effect of a domain specific transformation. Each basic transformation maps specification and proofs to a new consistent state, though potentially new open goals will occur in proofs and some parts of old proofs will be missing. In this new state, another transformation can be applied, or the proofs can be inspected and changed as usual with the theorem prover. Where it matters, we will assume proofs are sequent calculus proof trees for first order logic, cf. [Gen35], [Fit96].

As the result of working through the examples, we have come up with the set of basic transformations given in Fig. 2.1. In the rest of this section we will describe some of the transformations in more detail and show how they are employed in the case study introduced earlier. The list of basic transformations is open ended, there is no indication that the transformations listed there would be sufficient for every conceivable application.

2.3.1 Extending and Restricting the Signature

A group of transformations, i.e. *add type*, *add function*, and *add predicate* and the corresponding *remove* transformations change the specification in a way that is not reflected in the formulae occurring in the proof obligations: they add or remove signature symbols. *Add* transformations are applicable only if the name of the signature item, e.g. a type, to be added is new in the relevant name space of the specification signature. Name conflicts with bound variables in axioms or lemmata of the specification can be avoided by α -renaming. The result of the transformation on the signature, terms and formulae, therefore, is an inclusion homomorphism. Semantically, for each model M of the new specification, its reduct with respect to the inclusion homomorphism is a model of the old specification. Existing proofs can be retained in the new specification: none of the proof steps in any of the old proofs can be invalidated by applying the homomorphism to the goals in the proof, unless name conflicts with Eigenvariables introduced in the proof arise. These can be avoided by naming the conflicting Eigenvariables away, either by implicit or explicit α -renaming of the Eigenvariables, depending on whether the Eigenvariables are implicitly or explicitly bound in the proof tree.

Example 2.1 In the example, new datatypes for dealing with crashed processes are added and functions and predicates are defined to operate on them, e.g. an enumeration type $UpDown = up \mid down$ and a predicate $isup : Proc$ are added. ◦

Chapter 2. Example Scenarios and Supporting Transformations

- *Add type, add function, add predicate* – add a definition to the signature
- *Remove type, remove function, remove predicate* – remove a definition from the signature
- *Rename signature item* – consistently rename a signature item throughout the whole specification
- *Add axiom* – add an axiom to the collection of axioms
- *Remove axiom* – remove an axiom from the collection of axioms
- *Add constructor* – add a new constructor function to the list of constructors of an existing generated type
- *Remove constructor* – remove a constructor from the list of constructors of a generated type
- *Add argument, remove argument* – change the arity of a function or predicate by adding a new argument or by removing an argument from the definition
- *Swap arguments* – reorder the arity of a function or predicate by swapping two of its arguments
- *Replace occurrence* – replace the occurrence of a formula or term in an axiom or lemma by another formula or term

Special cases of *Replace occurrence* are:

- *Wrap* – replace a sub-formula A of an axiom or lemma by another formula B that has A wrapped somewhere inside
- *Unwrap* – replace a sub-formula B of an axiom or lemma that has A wrapped inside by A

Figure 2.1: Example set of basic transformations

2.3. Support by Transformations

The *remove* transformations are applicable only if the signature item to be removed is not used in the specification, i.e. a type τ can only be removed if none of the functions or predicates mentions τ in its arity, and if none of the axioms or lemmata quantifies over τ . The same restriction applies to formulae and terms introduced in the proof, e.g. no formula introduced by a cut rule may quantify over τ if the transformation is to be applicable. In this case, the inclusion homomorphism from the new signature to the old signature maps every formula or term in the axioms, lemmata, and proofs to itself. The old proofs are, therefore, proofs for the new proof obligations.

Example 2.2 In the example, several constants that were introduced by other transformations (cf. Sect. 2.3.2) were removed from the specification when they were no longer needed. \circ

The transformation *rename signature item* is applicable when the new name of, e.g., a type does not already occur in the relevant name space. In this case, the renaming corresponds to an isomorphism between the old and the new signature, and similarly between the old and new specification. Semantically, this does not have any effects, since the concrete names of signature items do not matter for the meaning of terms and formulae. Modulo α -renaming of bound variables and Eigenvariables, the image of a proof under the isomorphism, i.e. the result of applying the isomorphism to terms and formula in the proof, is again a proof.

Example 2.3 In the example, signature items were renamed after they had been changed to address fault tolerance, e.g. type *Action* was renamed to *CrashAction*. \circ

2.3.2 Changing Existing Signature Entries

The transformation *add argument* changes the signature of the specification by changing the arity of a function or predicate. This means that if, e.g.,

$$f : \tau_1 \times \cdots \times \tau_m \rightarrow \tau$$

is changed to

$$f : \tau_1 \times \cdots \times \tau_m \times \tau_{m+1} \rightarrow \tau,$$

terms constructed with the function f will no longer be well-formed. There are also consequences concerning induction schemes when f is a constructor function, cf. Sect. 2.3.5 below. For now, assume that f is a non-constructor function. Then replacing each term of the form $f(t_1, \dots, t_m)$

with a term $f(t_1, \dots, t_m, t)$, where t is an arbitrary but fixed term of type τ_{m+1} , in the specification, proof obligations, and proofs ensures all formulae are again syntactically correct. The transformed proof obligations correspond to the new specification and the transformed proofs are valid. This can be shown by induction over the tree structure of a proof, using the fact that whenever two formulae were equal before the transformation, they will again be equal afterwards.

Example 2.4 In the example, the arity of the function *mkproc* was changed by adding an argument of type *UpDown*. The new argument represents information on whether the process is up or down. See Sect. 2.3.5 for more details. ◦

2.3.3 Adding and Removing Axioms

The transformations *add axiom* and *remove axiom* have direct and obvious consequences on the proof obligations. For each axiom that acts as a conjecture, there is exactly one proof obligation, so adding or removing axioms in such positions adds or removes proof obligations. When a new proof obligation $\Gamma \vdash B$ is generated, no old proof is available for the new proof obligation, so a trivial open proof with root goal $\Gamma \vdash B$ is associated with the new goal. When an old proof obligation is removed, the proof for it is removed together with the proof obligation.

On the other hand, adding or removing axioms adds or removes formulae from the assumptions of proof obligations for other lemmata.

Let A be a new axiom. Each proof obligation in which A becomes visible is changed by adding the new axiom A to the assumptions of the proof obligation. Throughout the whole proof, it can be introduced to any goal and be used as an additional assumption. In particular, proofs to be constructed for open goals can use the newly introduced axiom.

Example 2.5 In the fault tolerance example, defining axioms for *isup* are added, e.g.

$$\forall i : PID, b : BMsg, m : MMSet. isup(mkproc(i, b, m, up))$$

and the respective negation. ◦

Conversely, if A were to be removed from the set of axioms, and thereby from the set of available assumptions for a proof, the proof is still a valid proof. However, if A was used in the proof, it will now become a new open goal. The proof itself is still valid. In particular, for proofs that do not use A at all, no additional goals are created.

2.3.4 Changing Formulae

The transformation *replace occurrence* replaces an occurrence of a formula in the specification.

Example 2.6 As an example consider (2.1) where the subformula

$$\text{delivered}(p, s) \subseteq \text{broadcast}(s) ,$$

call it $A(p, s)$, is replaced by the formula $\text{isup}(p) \Rightarrow A(p, s)$. ◦

The old proof for (2.2) uses the definition in (2.1) several times to expand the definition of *safety* for each subproof resulting from a case split over all possible actions. All these proof steps are still valid if only instances of $A(X, Y)$ are replaced by the corresponding instances of $\text{isup}(X) \Rightarrow A(X, Y)$ throughout the proof, since the steps do not depend on the concrete form of the formula. However, at some point, the concrete structure of A is used in the old proof, e.g. the top-level formula $A(p_1, s_1)$ is unified with another formula of the form $t_1 \subseteq t_2$ for some terms t_1, t_2 . This step is not valid if $A(p_1, s_1)$ is replaced by the new formula. At this point in the proof, the cut rule can be used to introduce the old formula as a top-level formula, resulting in an additional open goal of the form $\Gamma, \text{isup}(p_1) \Rightarrow A(p_1, s_1) \vdash \Delta, A(p_1, s_1)$. In this particular case, however, since it is known that $A(p, s)$ has been replaced by $\text{isup}(p) \Rightarrow A(p, s)$, a more specific transformation, i.e. *wrap connective* is applicable.

2.3.4.1 Wrap Connective.

If a formula occurrence A is replaced by $A \circ B$ or $B \circ A$, where ‘ \circ ’ is a connective, and A has the same polarity in the new formula (the polarity of A is preserved in $A \wedge B$, $A \vee B$, $B \wedge A$, $B \vee A$, and in $B \Rightarrow A$, but not in $\neg A$ nor in $A \Rightarrow B$), the transformation *wrap connective* is applicable. In Example 2.6, parts of a simplified proof for (2.2) looks like the following

$$\frac{\dots \quad \Gamma' \vdash A(p, \text{res}(tr.a)) \quad \dots}{\vdots \xi_1} \quad \vdots \xi_2$$

$$\Gamma, \forall s. \text{safety}(s) \Leftrightarrow \forall p. A(p, s) \vdash \text{ad}(tr.a) \Rightarrow \text{safety}(\text{res}(tr.a))$$

where parts of the proof for the base case of the induction and for all but one specific action a have been omitted. When $A(p, s)$ is replaced by $\text{isup}(p) \Rightarrow A(p, s)$, ξ_1 is transformed to ξ'_1 as described for the general

Chapter 2. Example Scenarios and Supporting Transformations

replace occurrence. Then, the top-level formula $isup(p) \Rightarrow A(p, s)$ is decomposed by inserting a $\Rightarrow: r$ rule such that ξ_2 can be used if it is transformed to ξ'_2 as described for the case of adding assumptions above. This yields the following transformed proof.

$$\frac{\dots \quad \frac{\Gamma'', isup(p) \vdash A(p, res(tr.a))}{\Gamma'' \vdash isup(p) \Rightarrow A(p, res(tr.a))} \Rightarrow: r \quad \dots}{\Gamma, \forall s. safety(s) \Leftrightarrow \forall p. (isup(p) \Rightarrow A(p, s)) \vdash ad(tr.a) \Rightarrow safety(res(tr.a))} \begin{array}{c} \vdots \xi'_2 \\ \vdots \xi'_1 \end{array}$$

Because the proof is by induction, a similar formula also occurs in the antecedent of the sequent as the induction hypothesis. There, instead of adding $isup(p)$ to the assumptions, the proof is split by a $\Rightarrow: l$ rule and a new open goal is produced. Note that the new open goal for the case of each action is mentioned in [MG00, p. 483] as being there ‘because of the additional condition in the guard which was added’ to the actions. Thus, the new open goal produced by the applications of the *wrap connective* transformation corresponds perfectly to the new proof that the authors of the case study had to construct after they had transformed the system in one big step and reproduced the proofs manually, cf. in particular their Fig. 6 with an overview of their new proof tree.

2.3.4.2 Wrap Quantifier.

A specialised basic transformation for wrapping formulae into quantifiers, *wrap quantifier*, is similar to *wrap connective*. When formula $A(a)$ (where a is a constant) is replaced by, e.g., $\forall x. A(x)$, then again, the part ξ_1 of the proof in which the form of A does not matter is transformed by replacing the formula occurrence. Where $A(a)$ was used in the old tree, a proof step is introduced that eliminates the quantifier. If this is on the left hand side, a witness for the variable x can be chosen, and with the choice of a the rest of the proof using $A(a)$ can be reused essentially unchanged. If the \forall -quantifier is eliminated on the right hand side, the resulting formula is $A(a')$, where a' is a new Eigenvariable. In this case the old proof can be reused if it is transformed as if by the general case of *replace occurrence*, where $A(a)$ is replaced by $A(a')$, and a new open goal will be generated that reads $\Gamma, A(a') \vdash \Delta, A(a)$. In the example, intermediate transformations replace terms introduced by earlier transformations.

2.3.5 Changing Induction Schemes

Add constructor adds a constructor to a type τ by making a non-constructor function $f : \tau_1 \times \dots \times \tau_m \rightarrow \tau$ a constructor function. Consequently, it changes the induction scheme for τ , but does not change the proof obligations that are generated. If inside a proof the induction rule for τ is used, the proof has to be changed to respect the new induction schema. Applying induction over τ splits the proof in m branches, where m is the number of constructors and the i th branch is the proof for the case of the i th constructor c_i .

$$\frac{\Gamma \vdash \Delta, \dots \Rightarrow A(c_1(\dots)) \quad \dots \quad \Gamma \vdash \Delta, \dots \Rightarrow A(c_m(\dots))}{\Gamma \vdash \Delta, \forall x : \tau. A(x)}$$

This means that the transformation adds a new child to the goal at which the induction was applied, and the child goal is the obligation to be shown for the new constructor f .

$$\frac{\dots \quad \Gamma \vdash \Delta, \dots \Rightarrow A(c_i(\dots)) \quad \dots \quad \Gamma \vdash \Delta, \dots \Rightarrow A(f(\dots))}{\Gamma \vdash \Delta, \forall x : \tau. A(x)}$$

Thus, the application of the induction rule is again a sound proof step. The new goal is left open. Since the old constructors are not changed, the subproofs above them are still valid (unless, of course, another induction over t takes place there, in which case the transformation is applied recursively). The transformation, therefore, manages to keep all of the old proofs but produces one new open goal for each induction step over t . Similarly, *remove constructor* removes a constructor and the corresponding cases from induction steps.

Example 2.7 In the example, a new action is added by changing the definition of type *Action* from

$$Action = B_1 : PID \times Msg \rightarrow Action \mid \dots \mid B_5 : PID \rightarrow Action$$

to

$$Action = B_1 : PID \times Msg \rightarrow Action \mid \dots \mid B_5 : PID \rightarrow Action \\ \mid Crash : PID \rightarrow Action .$$

The subproofs in the proof for (2.2) of the actions B_1 to B_5 are kept and a new case for the action *Crash* is added as an open goal. This corresponds to the fact that the subproof for *Crash* had to be constructed from scratch in the case study. \circ

Chapter 2. Example Scenarios and Supporting Transformations

The transformation *add argument*, if applied to a constructor function c , also changes the induction schema for the target type of c . To ease the presentation of the transformation, consider the special case where $c : \tau_1 \rightarrow \tau$ is the only constructor, and assume that $\tau \neq \tau_1$. An application of the induction rule for τ before and after adding an argument of type $\tau_2 \neq \tau$ to c produces a proof step of the form

$$\frac{\Gamma \vdash \Delta, \forall y : \tau_1. A(c(y)) \quad \vdots \xi}{\Gamma \vdash \Delta, \forall x : \tau. A(x)} \quad \text{and} \quad \frac{\Gamma \vdash \Delta, \forall y : \tau_1, z : \tau_2. A(c(y, z)) \quad \vdots \xi'}{\Gamma \vdash \Delta, \forall x : \tau. A(x)}$$

respectively. The subproof ξ can be transformed by a combination of the transformations described for *wrap quantifier* (this takes care that the quantifier binding z is removed before A is used in the proof) and *add argument* for non-constructor functions (this takes care that formulae with terms constructed by c are again well-formed).

Example 2.8 In the example, the function

$$mkproc : PID \times BMsg \times MMSet \rightarrow Proc$$

constructs process terms, where the arguments represent the process ID and the local state of the process. For each of the arguments corresponding to a local variable there is an update function, e.g.

$$\forall p : Proc, b : BMsg. updatebbuf(p, b) = mkproc(procpid(p), b, procD(p)) .$$

In the main proof for (2.2), lemmata are used which state that *updatebbuf* does not change the other slots of a process, e.g.

$$\forall p : Proc, b : BMsg. procD(p) = procD(updatebbuf(p, b)) .$$

These lemmata are proved using induction (i.e. a case split) over p . The arity of *mkproc* is changed to $PID \times BMsg \times MMSet \times UpDown \rightarrow Proc$ and the new lemma reads

$$\forall p : Proc, b : BMsg. updatebbuf(p, b) = mkproc(procpid(p), b, procD(p), a) .$$

The induction produces open goals of the form

$$\begin{array}{l} \Gamma, updatebbuf(p, b) = mkproc(\dots, a) \\ \vdash \Delta, updatebbuf(p, b) = mkproc(\dots, a') \end{array}$$

where a' is an Eigenvariable introduced for the universally bound variable resulting from the changed induction scheme. In the example these goals can be closed by subsequent *wrap quantifier* and *change occurrence* transformations. ◦

2.3.6 Completeness and Adequacy

To assess the usefulness of a set of basic transformations, natural questions to be asked are whether the set is *complete* in the sense that every well-formed specification can be transformed into an arbitrary well-formed specification using only transformations from the set, or whether the transformations are *adequate* in that a reasonable portion of old proofs is conserved.

The question of completeness can be answered formally for this specific set of transformations, but in a totally uninteresting way: a subset of the basic transformations given above, i.e. *add* and *remove* transformations for signature items and axioms suffice to reduce any specification to the empty specification, and also suffice to build any specification from scratch. In the course, all proof effort would be lost, however.

Adequateness can hardly be shown formally, as a characterisation of the parts of proofs that are expected to be kept for an arbitrary overall transformation does not seem to be possible in general, mirroring the general problem of devising a complete set of transformations for expressive languages and properties [Voe01]. We have derived the set of basic transformations described above using the experience from development case studies and projects, and have found that they provided adequate support where the existing support mechanisms did not help. We take this as an indication that the approach is worthwhile even though no more formal assessment of its adequateness seems to be possible. Time and more practical experience will tell.

We expect that using the approach in practice over time, we will need to provide additional basic transformations for cases in which the existing set is found not to be adequate, e.g. particularly useful compositions of basic transformations that can, when considered as a whole, be given better proof support.

2.4 Summary

In this chapter, we have briefly presented four example scenarios in which changes to formal specifications and proofs were found to be necessary or beneficial in prior case studies and industrial projects, and that could not be handled adequately by the existing methods and techniques. One of them was presented in some more detail, and we have provided examples of the changes that were problematic. We have also sketched how our proposed approach of transforming specification and proofs is able to

Chapter 2. Example Scenarios and Supporting Transformations

affect these changes. The transformations we have presented address issues that are orthogonal to those addressed by both, heuristic replay or the management of change using development graphs.

Part II

Transformation Framework

Chapter 3

Context and Overview over the Framework

3.1 Overview

The transformations that we provide in this thesis do not exist in isolation. Rather they aim at solving problems that occur in the setting we have described in Section 1.2, and which is provided by our support tools VSE [AHL⁺00] or MAYA/INKA [AHMS02, AHMS99]. Here, a formal development consists of structured specifications and proofs. The specifications are translated into development graphs, proof obligations are extracted from development graphs, and proofs are required to cover the proof obligations, cf. Figure 3.1 for a visualisation. Obviously, our contributions will be most effective if they can be used *in addition* to any other support that is already available in this setting.

In Chapter 2, we have presented example scenarios for our techniques. For reasons of simplicity and brevity, the scenarios were presented without regard to the structure of the specifications, thereby completely ignoring development graphs and proof obligations. This was suitable for the motivation of our work, since the information that our techniques use is orthogonal to the structure of the specification. However, the transformations are required to work *in the presence* of structured specifications and development graphs if they are to be usable in addition to the existing support. In fact, handling the transformations in the context of structured specifications and development graphs is one of the main contributions of this thesis.

It is much easier to present the core ideas of transformations for structured specifications if details of the concrete specification language and

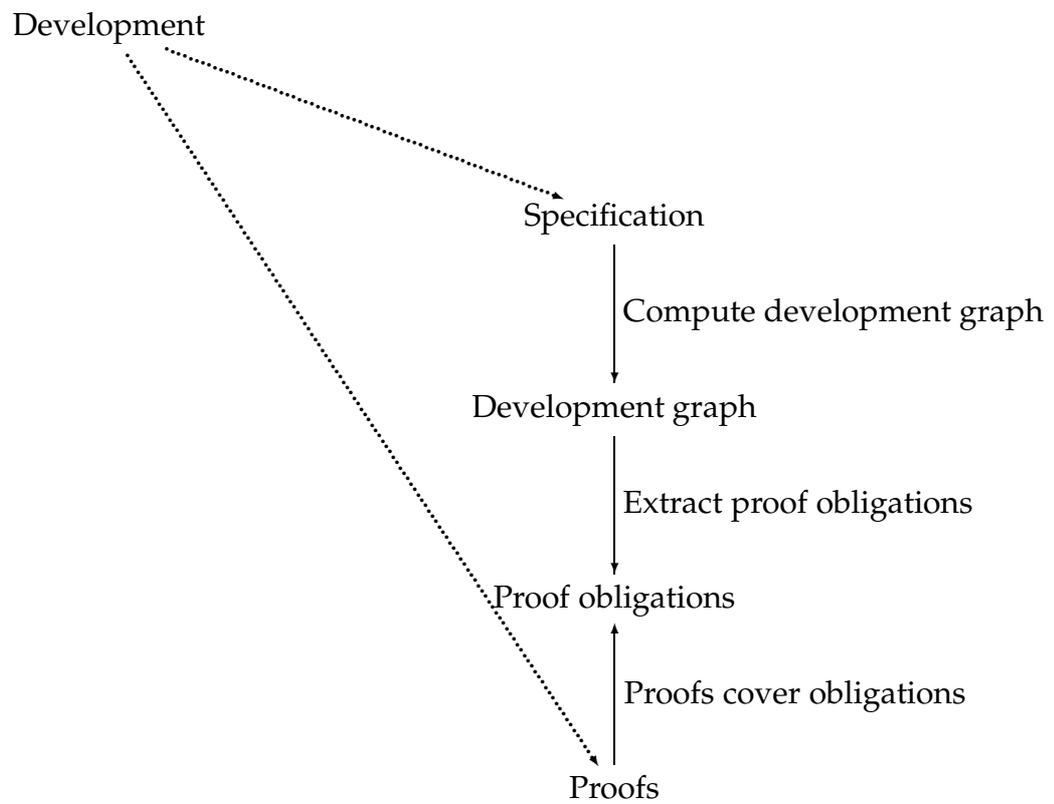


Figure 3.1: Visualisation of the state of a formal development

the concrete logic and proof calculus are omitted. For this reason, in a first step we abstract from these details and present classes of abstract development graph transformations. This forms a framework that we instantiate later in this thesis by defining a concrete logic, a concrete representation of partial proofs, and a concrete specification language. The main benefit for the thesis is that we can reduce the complexity of our presentation by explaining the concrete transformations in two steps. An additional benefit is that the framework provides a basis for other instantiations.

The rest of this chapter provides an abstract formulation of specification language, development graphs, and proofs as the basis for the framework. We will also give an overview of the framework itself.

3.2 Abstract Logic: Institutions

Development graphs, introduced in [Hut00], provide a kernel language for expressing the semantics of structured specifications: a concrete specification is translated into a concrete development graph, and the development graph is given a meaning by associating a flat, i.e. unstructured, theory to each of its nodes. Such a theory is, of course, relative to a logic. On the other hand, which particular syntax, semantics, and proof theory are used is immaterial for the formulation of development graphs. Thus, an abstract formulation of logic is appropriate. We use a definition of development graphs similar to the ones in [MAH01], [AH02], or [MHAH04] where the abstract notion of logic is given by an *institution* [GB84].

An institution is an abstract description of a family of languages (indexed by abstract signatures) and the semantics (i.e. satisfaction and logical consequence) for each of these languages. Additionally, an institution specifies mappings between languages together with respective mappings between the semantics such that satisfaction is preserved.

Since the notion of abstract signatures and abstract sentences of an institution will play a role for the framework, we give a definition first. Institutions are presented in category-theoretic terms in the literature. We stick to that convenient terminology and notation. However, no category theory except for the elementary definition of categories, functors, and natural transformations is presupposed, see e.g. [Pie91] (Definitions 1.1.1, 2.1.1, and 2.3.1) or [BW96] (Paragraphs 2.1.3, 3.1.1, and 4.2.10). For ease of reference, Appendix A provides definitions using the notation that we use in this thesis.

In the following the standard definition of an institution (as given in [GB84]) is provided.

Definition 3.1 (Institution) An institution $\mathcal{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ consists of

- a category \mathbf{Sig} (of abstract signatures and signature morphisms),
- a functor $\mathbf{Sen} : \mathbf{Sig} \rightarrow \mathbf{Set}$ (mapping each signature $\Sigma \in |\mathbf{Sig}|$ to the set of Σ -sentences and each \mathbf{Sig} -morphism $\sigma : \Sigma \rightarrow \Sigma'$ to a function translating Σ -sentences to Σ' -sentences),
- a functor $\mathbf{Mod} : \mathbf{Sig} \rightarrow \mathbf{Set}^{\text{op}}$ (mapping each signature to the set of Σ -models and each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ to the σ -reduct operator), and
- a satisfaction relation $\models_{\Sigma} \subseteq \mathbf{Mod}(\Sigma) \times \mathbf{Sen}(\Sigma)$ for each signature $\Sigma \in |\mathbf{Sig}|$

such that translating sentences along $\mathbf{Sen}(\sigma)$ and taking the reduct of models along $\mathbf{Mod}(\sigma)$ preserves the satisfaction relation, i.e. for any \mathbf{Sig} -morphism $\sigma : \Sigma \rightarrow \Sigma'$, any $\varphi \in \mathbf{Sen}(\Sigma)$, and any $M' \in \mathbf{Mod}(\Sigma')$ the satisfaction condition

$$M' \models_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi) \quad \text{iff} \quad \mathbf{Mod}(\sigma)(M') \models_{\Sigma} \varphi \quad (3.1)$$

is satisfied. As usual we overload the symbol \models_{Σ} and write $\Gamma \models_{\Sigma} \varphi$ if φ is a logical consequence of Γ , i.e. if all Σ -models that satisfy each member of Γ also satisfy φ .

Example 3.2 First order logic can be given as an institution

$$\mathcal{I}_{\text{FOL}} = (\mathbf{Sig}_{\text{FOL}}, \mathbf{Sen}_{\text{FOL}}, \mathbf{Mod}_{\text{FOL}}, \models_{\text{FOL}})$$

by defining $\mathbf{Sig}_{\text{FOL}}$ to be the category with algebraic signatures as objects and algebraic signature morphisms as arrows. For each signature $\Sigma \in |\mathbf{Sig}_{\text{FOL}}|$, $\mathbf{Sen}_{\text{FOL}}(\Sigma)$ is defined to be the set of all first order formulae over the signature Σ , and for each $\mathbf{Sig}_{\text{FOL}}$ -arrow σ , $\mathbf{Sen}_{\text{FOL}}(\sigma)$ is the function resulting from extending the signature morphism σ homomorphically to closed formulae. Finally, $\mathbf{Mod}_{\text{FOL}}(\Sigma)$ is the class of all Σ -algebras and for each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, $\mathbf{Mod}_{\text{FOL}}(\sigma)$ is the reduct functor that maps each Σ' -model M' to the σ -reduct of M' , i.e. $\mathbf{Mod}_{\text{FOL}}(\sigma)(M') = M'|_{\sigma}$. \circ

We refer to [GB84] for detailed examples of various logics defined as institutions.

It is important to note that arrows in the category \mathbf{Sig} of signatures (called \mathbf{Sig} -morphisms or signature morphisms) play a role similar to algebraic signature morphism (cf. [EM85, Def. 8.1] or [LEW96, Def. 4.1]).

Sig-objects are not restricted to algebraic signatures, however, and **Sig**-arrows are not restricted to algebraic signature morphisms. In particular, the notion of abstract signatures does not expose any internal structure of signatures like, e.g., the notion of signature symbols or function arities. Therefore, the preservation of function arities is not required. Similarly, the sentence functor does not expose any internal structure of how formulae are defined w.r.t. the signature. Thus, the mapping of sentences along σ is described by $\mathbf{Sen}(\sigma)$ transparently rather than by a homomorphic extension of a symbol mapping. However, algebraic signatures together with signature morphisms, i.e. symbol mappings extended to terms and formulae, is an important example of a category of signatures. We will use it as an example whenever this is convenient.

3.3 Development Graphs

Development graphs [Hut00] serve as a way to represent the semantics of the structuring concepts that a specification language provides and provide a uniform proof theory on the level of structuring constructs. This proof theory is parametric in the semantics of the basic, unstructured building blocks of the specification language. Technically, this is achieved by defining development graphs relative to an institution.

In practice, structured specifications are represented as graphs such that nodes correspond to unstructured basic specifications and that the structure of the specification is represented in the structure of the graph, cf. [AHMS00], [MHAH04].

A development graph relative to an institution determines a collection of theories (its nodes) that are connected by signature morphisms (its links). Sentences making up the axiomatisation of the theories are inherited along links, and thus large theories can be presented piece-wise and in a structured way. Links can either be definitorial (i.e. they define the axiomatisation of theories) or be postulated (i.e. they postulate that some theory is included in another one). Postulated links are also called theorem links.

The following definition is similar to the original one in [Hut00] (except that, like [MAH01], [AH02], or [MHAH04], we use institutions instead of consequence relations and morphisms, which were used to abstract from the concrete logic in the original statement).

Definition 3.3 (Development graphs) *Let $\mathcal{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ be an institution. A development graph $D = (N, A, C)$ relative to \mathcal{I} consists of*

Chapter 3. Context and Overview over the Framework

- a finite set N of nodes, together with, for each node $n \in N$
 - a signature $\Sigma^n \in |\mathbf{Sig}|$ and
 - a finite set of sentences $\Phi^n \subseteq \mathbf{Sen}(\Sigma^n)$ (the local axioms of the node),
- a finite set A of definitorial and a finite set C of postulated, directed links, together with, for each link $l \in A \cup C$ from n_1 to n_2 (where $n_1, n_2 \in N$)
 - a signature morphism $\sigma^l : \Sigma^{n_1} \rightarrow \Sigma^{n_2}$ and
 - an indication whether l is a global or local link,

such that N , A , and C are pairwise disjoint, (N, A) is an acyclic, directed graph and $(N, A \cup C)$ is a possibly cyclic directed graph. We write $l : n_1 \xrightarrow{\sigma} n_2$ for a local link l from n_1 to n_2 with signature morphism σ , and similarly $g : n_1 \xrightarrow{\sigma} n_2$ for a global link g .

The sets of links written as A and C are meant to be mnemonic: the links in A are assumed and the links in C are conjectured.

Example 3.4 Consider the structured specification of the behaviour and some properties of a state transition system, which results in the simple development graph sketched in Figure 3.2. The behaviour of the system, i.e. the state transition table, is specified in node “Behaviour” in terms of the possible states and the supported commands. Thus, the node “Behaviour” imports the definitions of commands and states from the nodes “Commands” and “States”. A property, node “Property”, is formulated in terms of the possible states and, therefore, also imports from “States”. The system is supposed to satisfy the property and this is expressed by the postulated link from “Property” to “Behaviour”.

Postulated arrows are drawn with an arrow head and tail to distinguish them from definitorial links. For ease of presentation, we assume that all morphisms are identity or inclusion morphisms. Note that all links are global in this example. ◦

We define the semantics of a development graph by associating a theory to each node. The theory of a node is defined to be the smallest set closed under logical consequence that includes the local axioms of the node, that inherits the local axioms of other nodes along local definitorial links, and that inherits the theories of other nodes along global definitorial links. Since definitorial links do not include cycles, this fixpoint exists and is unique.

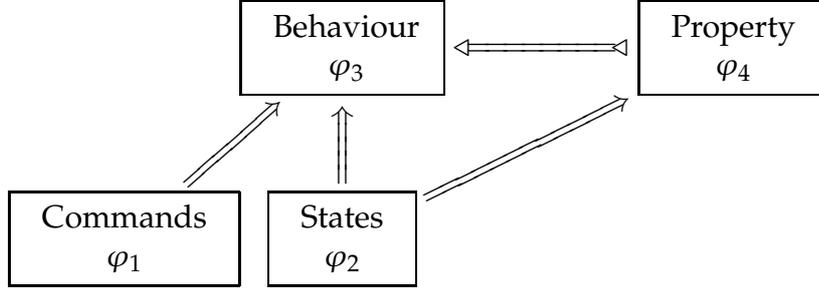


Figure 3.2: Example development graph

Postulated links, which we have ignored in the semantics so far, postulate relationships between theories or their models, respectively. For our work, the exact details of the model-theoretic semantic definitions are immaterial. We are only interested in the proof theory, i.e. in questions of whether the axioms of a node are entailed by the theory of another node. In particular we are interested in whether we can construct a proof. We express this using the notion of *global axioms* of a node. (For the details of the relationship between the model-theoretic and the proof-theoretic semantics cf., e.g., [AH02], [BCH⁺04].)

Definition 3.5 (Global axioms) Let $D = (N, A, C)$ be a development graph. The set of global axioms Φ_D^n of node n wrt. the development graph D is defined inductively for all $n \in N$ as the smallest sets satisfying the conditions

- $\Phi^n \subseteq \Phi_D^n$,
- $\text{Sen}(\sigma)(\Phi^{n'}) \subseteq \Phi_D^n$ for all $n' \xrightarrow{\sigma} n \in A$, and
- $\text{Sen}(\sigma)(\Phi_D^{n'}) \subseteq \Phi_D^n$ for all $n' \xRightarrow{\sigma} n \in A$.

Note that the sets of global axioms are independent of the postulated links C of D .

Example 3.6 For the development graph given in Figure 3.2, the global set of axioms of the node “Behaviour”, e.g., includes all local axioms from the nodes “Commands”, “States”, and itself. Similarly, “Property” includes the axioms from “State” and itself. Thus if the formula φ_1 is an axiom in the node “Commands”, φ_2 is an axiom in node “States”, φ_3 is an axiom in node “Behaviour”, and finally φ_4 is an axiom in node “Property”, then the global set of axioms of “Behaviour” is $\{\varphi_1, \varphi_2, \varphi_3\}$ and the global set of axioms of “Property” is $\{\varphi_2, \varphi_4\}$. \circ

A postulated local link is implied by a development graph, if the local axioms of the source of the link are all logical consequences of the global set of axioms of the target node of the link. Similarly, a postulated global link conjectures that the global axioms of the source are consequences of the global axioms of the target.

Definition 3.7 (Implied links) *A postulated local link $l : n_1 \xrightarrow{\sigma} n_2 \in C$ is implied by the development graph $D = (N, A, C)$ iff*

$$\Phi_D^{n_2} \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi) \quad \text{for all } \varphi \in \Phi^{n_1} \quad (3.2)$$

and similarly, a postulated global link $g : n_1 \xrightarrow{\sigma} n_2 \in C$ is implied by D iff

$$\Phi_D^{n_2} \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi) \quad \text{for all } \varphi \in \Phi_D^{n_1}. \quad (3.3)$$

The development graph D is said to be satisfied, if each postulated link in C is implied by the development graph.

Whether a link is implied or not only depends on the set of nodes N and the set A of definitorial links, but not on the set of postulated links C . We will, therefore, say that a postulated link is implied by N and A to mean that it is implied by some development graph $D = (N, A, C)$.

The definition suggests a direct way of verifying a development graph: show for each postulated link that it is implied by proving (3.2) or (3.3) for each of the axioms.

Example 3.8 For the example development graph, we have to show, for each member of the global set of axioms of “Property” that it is implied by the global set of axioms of “Behaviour”. I.e. we have to show

$$\varphi_1, \varphi_2, \varphi_3 \models \varphi_2$$

and

$$\varphi_1, \varphi_2, \varphi_3 \models \varphi_4$$

Note in particular, that we need to prove this for an axiom of “States”, in the example this is φ_2 , because these axioms are inherited by “Property”. On the other hand, such an axiom is also a member of the global set of axioms of “Behaviour”, thus it should not be necessary to look at the axioms in detail. \circ

As the example suggests, some postulated links can be shown to be implied simply by considering the graph structure: e.g. a postulated link that is subsumed by a parallel definitorial link with the same morphisms

3.3. Development Graphs

is implied without consideration for the concrete axioms. Also, a global postulated link can be decomposed into a parallel local link and additional local and global links. These two aspects can be expressed and refined by a development graph calculus [Hut00]. A development graph calculus rule

$$\frac{N, A, C'}{N, A, C}$$

states that the development graph $D = (N, A, C)$ is satisfied if only the development graph $D' = (N, A, C')$ is. As usual, we will call the development graph below the line the *conclusion* of the rule and the development graph above the line its *premiss*.

This suggests another way of verifying a development graph: apply the calculus to simplify the graph first, and only then prove (3.2) for the remaining links – it will turn out that we only need to consider proofs for local postulated links as in (3.2), but never for global links as in (3.3).

For one of the calculus rules we need the notion of a path in a development graph. Basically, a path is a sequence of links along which the local axioms of the source node are inherited by the target node. This is the case if all links, or all but the first link, are global (since the inheritance relationship along local links is not transitive).

Definition 3.9 (Global definitorial paths) *Given a development graph $D = (N, A, C)$ we say that there is a (definitorial) global path from $n \in N$ to $n' \in N$ with path morphism σ , written as $n \xrightarrow{\sigma} n'$, iff*

- $n = n'$ and σ is the identity morphism, or
- there is a node $n'' \in N$ such that
 - there is a global definitorial link $n \xrightarrow{\sigma''} n''$, and
 - there is a global path $n'' \xrightarrow{\sigma'} n'$

such that $\sigma = \sigma' \circ \sigma''$, i.e. $n \xrightarrow{\sigma''} n'' \xrightarrow{\sigma'} n'$.

We say that there is a (definitorial) local path (or simply path) from n to n' with morphism σ , written as $n \xrightarrow{\sigma} n'$, iff

- there is a global path $n \xrightarrow{\sigma} n'$, or
- there is a node $n'' \in N$ such that
 - there is a local definitorial link $n \xrightarrow{\sigma''} n''$, and

Chapter 3. Context and Overview over the Framework

– there is a global path $n'' \xrightarrow{\sigma'} n'$

such that $\sigma = \sigma' \circ \sigma''$.

Note that there is a trivial path from each node to itself with the identity morphism.

The following definition is a slight variation of the calculus given in [AH02]. Since the rules do not change N or A , we omit them from the rules and only write C and C' in the conclusions and premisses of the rules.

Definition 3.10 (Development graph calculus) *The development graph calculus consists of the axiom stating that the empty set of postulated links is trivially satisfied, and the following rules:*

Decomposition:

$$\frac{C \cup \{n_1 \xrightarrow{\sigma} n_2\} \cup \bigcup_{n_0 \xrightarrow{\sigma'} n_1 \in A} \{n_0 \xrightarrow{\sigma \circ \sigma'} n_2\} \cup \bigcup_{n_0 \xrightarrow{\sigma'} n_1 \in A} \{n_0 \xrightarrow{\sigma \circ \sigma'} n_2\}}{C \cup \{n_1 \xrightarrow{\sigma} n_2\}}$$

Subsumption:

$$\frac{C}{C \cup \{n_1 \xrightarrow{\sigma} n_2\}} \text{ there is a path } n_1 \xrightarrow{\sigma} n_2 \text{ in } A$$

Without proof we state that the rules are sound, cf. [AH02], [BCH⁺04] for a thorough analysis.

Example 3.11 The development graph of Example 3.4 is reprinted as the topmost development graph in Figure 3.3 with some morphisms added for clarity. Application of the *decomposition rule* to the global postulated link with morphism σ_1 leads to the second graph in which the link has been decomposed in the local link with morphism σ_1 and the new global link with morphism $\sigma_1 \circ \sigma_2$. Another application of the rule to the new global postulated link results in the third graph. Finally, since we have assumed that all morphisms are identity or inclusion morphisms, we have $\sigma_3 = \sigma_1 \circ \sigma_2$ and the *subsumption rule* is applicable. The result is the bottommost development graph. \circ

Exhaustive analytic application of the rules starting with a given development graph terminates with a uniquely determined set of postulated

3.3. Development Graphs

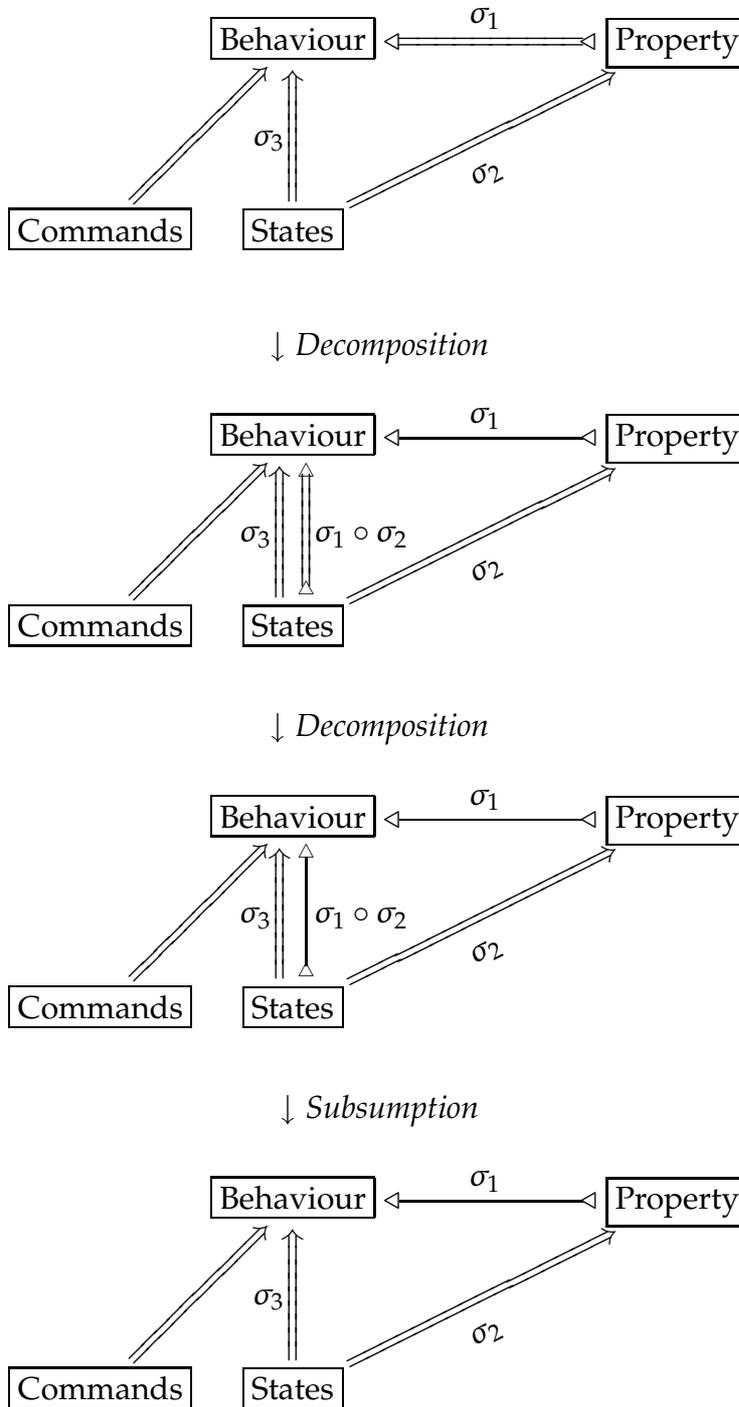


Figure 3.3: Development graph calculus example

links that only contains local links. Each of the links carries a signature morphism that is the composition of link morphisms that appeared already in the original development graph. Since we will be using these results later, we state them as a theorem.

Theorem 3.12 Let $D = (N, A, C)$ be a development graph. Repeated application of arbitrary rules of the development graph calculus until no further rule is applicable will reduce the verification of D to the verification of a unique $D' = (N, A, C')$ such that

- each link in C' is a local link, and
- for each link $n_1 \xrightarrow{\sigma} n_2$ in C' , there is a path $n_1 \xrightarrow{\sigma'} n'$ in A and a link $n' \xrightarrow{\sigma''} n_2$ (or $n' \xRightarrow{\sigma''} n_2$) in C , and $\sigma = \sigma'' \circ \sigma'$.

Example 3.13 For the example development graph of Figure 3.3, the set C' is the singleton consisting of the local postulated link from “Property” to “Behaviour”. It is trivially the case that the morphism σ_1 for the postulated link is the morphism along a path in the original development graph, namely the path consisting of the original global postulated link with morphism σ_1 . If the subsumption rule was not applicable and the local postulated link with morphism $\sigma_1 \circ \sigma_2$ was also in C' , then the condition would also be satisfied. ◦

For the proof of Theorem 3.12, we need the concept of the depth of a node, i.e. of the length of the longest definitorial path that leads to the node:

Definition 3.14 (Depth of node) Let (N, A) be an acyclic graph. The depth of a node $n \in N$ with respect to A is defined by

$$\#_A(n) = \begin{cases} 0 & \text{if there is no link } l \in A \text{ with target } n \\ 1 + \max \left\{ \#_A(n') \mid n' \xrightarrow{\sigma} n \in A \text{ or } n' \xRightarrow{\sigma} n \in A \right\} & \text{otherwise.} \end{cases}$$

Since the graph (N, A) is acyclic, the depth $\#_A$ is well-defined, and the set of nodes N is well-founded wrt. \leq on the depth of nodes.

Proof of 3.12 We note that the calculus is terminating. Let an auxiliary strict order \sqsubset over links be defined by the usual smaller than relation on the depth of links, where the depth of a global link $n \xrightarrow{\sigma} n'$ is the depth $\#_A(n)$ of n , i.e. the longest definitorial path in A to the source node n of the link, and the depth of a local link is -1. In both rules, the premiss is strictly

3.4. Specification Language

smaller according to the multiset order corresponding to \sqsubseteq (cf. [BN98, Definition 2.5.3]): *decomposition* replaces a link of depth k ($0 < k$) by finitely many links of depths -1 or k' ($k' < k$, the longest path to each n_0 in the premiss is at least one link shorter than the longest path to n_1) and *subsumption* removes an element. Since \sqsubseteq is terminating (i.e. a well-founded ordering for links), so is the multiset order (cf. [BN98, Theorem 2.5.5]).

The calculus is also locally confluent because the rules are permutable. Since the calculus is terminating, this implies overall confluence.

Thus, first, exhaustive application of rules yields a unique result. Second, the decomposition rule is applicable whenever there is a global link in the conclusion; thus when no rule is applicable, all links are local. Finally, the condition for the link morphisms is invariant over both rules, and is satisfied initially for the trivial path from the source node to itself. \square

Thus we can view the development graph calculus as a function computing a unique set of local postulated links (the normal form of the original postulated links wrt. the definitorial links) such that if these links are implied, the original development graph is satisfied.

Definition 3.15 (Normal form of C wrt. A and N) For a given development graph $D = (N, A, C)$, let $C \downarrow_{N,A}$ be the normal form of C wrt. N and A , i.e. the set of postulated local links resulting from D by exhaustive application of the development graph calculus rules to D .

3.4 Specification Language

Specification languages usually address both the formulation of small, basic specifications and the construction of structured specifications from simpler ones. Thus, we distinguish two different aspects of a specification language, called *specification in the small* and *specification in the large*. Whereas the latter is concerned with how complex specifications can be presented in a modular fashion by conjoining smaller specifications in a structured way, the former deals with the semantics of the smallest specification entities, sometimes called *basic specifications* or *theories*.

Syntactical well-formedness of specification is often defined in the form of a context-free grammar together with additional static well-formedness conditions like, e.g., type correctness conditions of expressions or the condition that identifiers be defined textually before they are used in the specification text. Using techniques from compiler construction like, e.g., lexing and parsing, syntax-directed translation, and static analysis and type-

checking, it is usually decidable mechanically and with moderate cost of resources whether a syntactical entity is a statically well-formed specification, and if so to describe the specification in a form that is independent of the concrete syntax. This allows us to ignore syntactic issues and assume that static well-formedness conditions can be checked fully automatically.

A specification language defines so called *dynamic* well-formedness conditions. These are additional conditions that need to hold in order for the specification to be well-defined. The conditions are called dynamic because they are not decidable mechanically in general. The conditions are typically expressed as a proposition in some logic. A specification is called satisfied if these dynamic conditions hold. In practice, proofs are constructed to show that the conditions hold. The conditions are therefore called *proof obligations*, and a specification is called verified if there is a proof for each proof obligation.

An example for such a dynamic well-formedness condition results from a specification that presents the natural numbers, addition, and specifies that the associativity of the addition is a logical consequence. This specification is well-formed only if associativity is indeed a logical consequence of the other definitions. Thus, a proof obligation arises: it requires a proof of the proposition that associativity is a consequence of the definition of the natural numbers and addition. Constructing proofs for such proof obligations is called *verifying a specification*.

For practical reasons, we do not derive proof obligations from specifications directly. Rather, a specification is first translated into a development graph, and proof obligations are then derived from the development graph. Basic specification units are translated into nodes of the development graph, thus expressing these basic units in the logic underlying the development graph. The structure of a specification, on the other hand, is translated into the structure of the development graph, i.e. into definitorial and postulated links.

For our purposes, therefore, a specification language for a given logic consists of a set of (statically) well-formed specifications, and a mapping from the set into the set of development graphs for the logic. Of course, we require the mapping to be correct in the sense that proof obligations represented by a resulting development graph imply the proof obligations for the given specification.

Definition 3.16 (Specification language) *A specification language for an institution \mathcal{I} is a set $Spec$ of statically well-formed specifications and a function dg , mapping each specification $S \in Spec$ to a development graph such that the specification S is satisfied if only the corresponding development graph $dg(S)$ is satisfied.*

As an example, [AHMS00] describes the mapping dg for a subset of the statically well-formed CASL-specification; it uses a definition of development graphs that is similar to the one we have given above.

3.5 Proof Representation

A well-formed specification S can be verified indirectly by showing that its associated development graph $D = dg(S)$ is satisfied. This is called *verifying the development graph*. A development graph $D = (N, A, C)$ can be verified by showing that A implies the set of obligation links $C \downarrow_{N,A}$. Each of these links $n_1 \xrightarrow{\sigma} n_2 \in C \downarrow_{N,A}$ in turn can be shown to be implied by constructing a proof for

$$\Phi_D^{n_2} \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi) \quad (3.4)$$

for each $\varphi \in \Phi^{n_1}$, cf. (3.2) on page 44. The general form of obligations for a given signature Σ is thus $\Gamma \models_{\Sigma} \varphi$, where Γ is a set of Σ -sentences and φ is a Σ -sentence. Proofs for these obligations are mechanised using a proof calculus for the institution, and proofs are represented using, e.g., proof objects or proof scripts.

Definition 3.17 (Proof obligations) *The set of proof obligations $obl(l)$ of a postulated link $l : n_1 \xrightarrow{\sigma} n_2$ with $\sigma : \Sigma' \rightarrow \Sigma$ is defined by*

$$obl(l) = \{(\Phi_D^{n_2}, \mathbf{Sen}(\sigma)(\varphi)) \mid \varphi \in \Phi^{n_1}\} .$$

obl is overloaded for development graphs, and the $|\mathbf{Sig}|$ -indexed family of sets of proof obligations $obl(D)$ for a development graph $D = (N, A, C)$ is defined by

$$(obl(l))_{\Sigma} = \bigcup_{\substack{l : n_1 \longrightarrow n_2 \in C \downarrow_{N,A} \\ \text{with } \Sigma^{n_2} = \Sigma}} obl(l) .$$

A pair (Γ, φ) in $(obl(D))_{\Sigma}$ is typically written as $\Gamma \models_{\Sigma} \varphi$ for clarity, and in this case we say that (Γ, φ) is a Σ -proof obligation. Where it is convenient we write $\Gamma \models_{\Sigma} \varphi \in obl(D)$ to mean $(\Gamma, \varphi) \in (obl(D))_{\Sigma}$.

For the framework we only need a few assumptions about the representation of proofs. First, proof obligations correspond to conclusions of proofs, i.e. we can express that a particular proof is a proof for a given proof obligation. Second, we assume that proofs can be proof sketches or partial proofs, i.e. contain holes representing open conjectures. Third,

proofs for different signatures and their conclusions are related in a way that is consistent with the relationship between signatures given by signature morphisms.

We represent proof obligation in $obl(l)$ as proof states, called *goals*. Technically, for each signature Σ we assume a set of Σ -goals and a mapping from proof obligations $\Gamma \models_{\Sigma} \varphi$ into the set of Σ -goals. We also require a set of partial Σ -proofs. Each Σ -proof has a conclusion, which in turn is a Σ -goal. For convenience we assume that for each proof goal there is a (distinguished) partial proof.¹ For goals and proofs we assume mappings along signature morphisms, and we further constrain these definitions so that proofs can be translated along signature morphisms meaningfully: mapping the formulae of a proof obligation and mapping its proof along the same signature morphism preserves the relationship between proof obligation and proof.

Example 3.18 The sequent calculus represents proof goals as a sequent. A proof obligation of the form $\Gamma \models_{\Sigma} \varphi$ is represented by the sequent $\Gamma \vdash \varphi$. In a natural deduction style calculus, the same proof state would be represented as the tuple consisting of the formula φ that we want to prove and the active hypotheses Γ , which we can use in the proof. \circ

Note that in both examples there are goals that do not correspond to proof obligations: sequents can have several formulae to the right of the turnstile and both representations will have to deal with free variables in formulae (e.g. Eigenvariables) in the course of a proof. This will also complicate the definition of how proofs are mapped along signature morphisms since, e.g., name capture with free variables should be avoided. How this is achieved in detail is immaterial for the framework, however.

We require that proofs can be mapped along signature morphisms. We do not require, however, that the image of a proof is structurally similar to its pre-image, or that the holes of the two proofs are related by the signature morphisms. Because we have not modelled the internal structure of proofs such a requirement cannot even be expressed formally.²

Our assumptions are captured by the following definition.

¹This is a reasonable assumption for partial proofs: usually there is a trivial proof for each goal that assumes the goal itself.

²Modelling the structure of proofs on this level of abstraction, e.g., along the lines of [LS86], introduces many details that are then not used in the framework to any good effect, but that restrict its applicability. Therefore, we decided to remove these definitions.

3.5. Proof Representation

Definition 3.19 (Proof representation) Let $\mathcal{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ be an institution. A proof representation $\mathcal{P} = (\mathbf{Goal}, \eta, \mathbf{Prf}, \mathit{concl})$ for the institution \mathcal{I} consists of

- a functor $\mathbf{Goal} : \mathbf{Sig} \rightarrow \mathbf{Set}$ (mapping each signature $\Sigma \in |\mathbf{Sig}|$ to the set of Σ -goals, and each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ to a function translating Σ -goals to Σ' -goals),
- a natural transformation $\eta : (2^{\mathbf{Sen}} \times \mathbf{Sen}) \rightarrow \mathbf{Goal}$ (mapping Σ -proof obligations to the corresponding proof goal)³,
- a functor $\mathbf{Prf} : \mathbf{Sig} \rightarrow \mathbf{Set}$ (mapping each signature Σ to the set of partial Σ -proofs, and each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ to a function translating Σ proofs into Σ' -proofs), and
- a natural transformation $\mathit{concl} : \mathbf{Prf} \rightarrow \mathbf{Goal}$ (mapping Σ -proofs to the goal that they prove, i.e. their conclusion).

Additionally, we assume that for each Σ -proof obligation $\Gamma \models_{\Sigma} \varphi$ there is some proof⁴ for the goal $g = \eta_{\Sigma}(\Gamma, \varphi)$.

In the following, we write \mathbf{Obl} for the functor $2^{\mathbf{Sen}} \times \mathbf{Sen}$, i.e. $\mathbf{Obl} = 2^{\mathbf{Sen}} \times \mathbf{Sen}$. Also we write concl instead of concl_{Σ} when the signature is clear from the context.

Remark 3.20 Note that the definition of proofs we have just given does not exclude incorrect proof representations, i.e. representations in which a proof can be found for a non-theorem. In fact, the framework is independent of whether the proof representation is correct⁵; it only depends on the way in which proof obligations, goals, and proofs for different signatures are related according to the definition. Furthermore, defining what a correct proof representation is would necessitate introducing details about the relationship between proofs and their holes or assumptions. These details would not be used in the framework but would potentially restrict the applicability of the framework.⁶

³For the definition of the notation $2^{\mathbf{A}}$ and $\mathbf{A} \times \mathbf{B}$, i.e. of functors lifted to sets and cross-products, see Appendix B.

⁴Recall that our notion of proofs allows partial proofs with holes. These holes, i.e. open goals, stand for arbitrary assumptions, independently of whether complete proofs for the open goals exist or not.

⁵Of course, these “proofs” are checked by a proof checker independently to ensure correctness of formal developments.

⁶Cf. Remark 7.1 on page 156 for an example of why the additional freedom is beneficial.

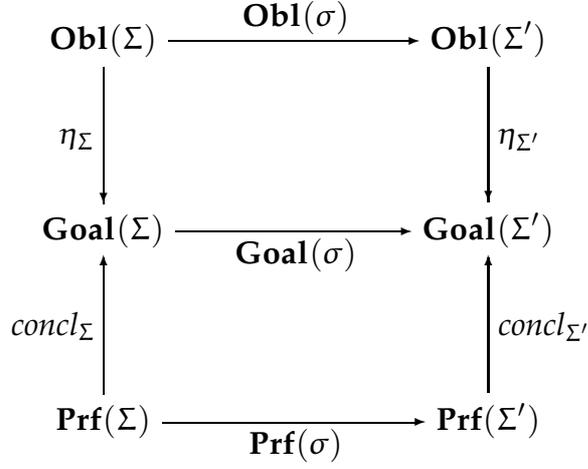


Figure 3.4: Commuting diagrams for η and concl

The requirement that η be a natural transformation (cf. Appendix A) amounts to the conditions that η assigns to each signature Σ a **Set**-arrow $\eta_\Sigma : \mathbf{Obl}(\Sigma) \rightarrow \mathbf{Goal}(\Sigma)$ such that for every signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, the top square in Figure 3.4 commutes, i.e. we have

$$\eta_{\Sigma'} \circ \mathbf{Obl}(\sigma) = \mathbf{Goal}(\sigma) \circ \eta_\Sigma .$$

Similarly, for each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, the bottom square in Figure 3.4 commutes, i.e. we have

$$\mathit{concl}_{\Sigma'} \circ \mathbf{Prf}(\sigma) = \mathbf{Goal}(\sigma) \circ \mathit{concl}_\Sigma .$$

The effect of these conditions is that the mapping $\mathbf{Prf}(\sigma)$ of proofs along signature morphisms is uniform with respect to the mapping of sentences. If we translate a proof along $\mathbf{Prf}(\sigma)$, its conclusion is translated as if we had translated the sentences of the proof obligation along $\mathbf{Sen}(\sigma)$ directly. The formulation abstracts from the details of exactly how a goal or a proof should be translated, for instance how Eigenvariables or active hypotheses should be translated for natural deduction goals, or the rule justifications for sequent calculus proofs.

The relationship between proofs and proof obligations that has been described informally up to now can be defined formally:

Definition 3.21 (Covered proof obligation) Let $o = (\Gamma, \varphi)$ be a Σ -proof obligation. The proof obligation is said to be covered by a proof p , written $\text{covers}(p, o)$, iff p is a Σ -proof such that $\text{concl}(p) = \eta_{\Sigma}(\Gamma, \varphi)$.⁷

The proof obligation o is said to be *verified* by p if p covers o and additionally p is complete (i.e. non-partial). Since we have not defined formally what distinguishes complete from non-complete proofs, we do not formally define verification of proof obligations either.

3.6 Formal Developments

We define the abstract notion of the state of a formal development (called development from hereon) and of well-formedness, or consistency, that we require of such developments. Intuitively, a development consists of a structured specification and proofs such that the proofs cover the proof obligations resulting from the development graph of the specification. However, we do not need to require the proof obligations to be verified, i.e. the proofs to be complete – as we have said before, we have not even defined completeness of proofs formally.

Definition 3.22 (Formal Development) Let $\mathcal{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ be an institution, let (Spec, dg) be a specification language for \mathcal{I} , and finally let $\mathcal{P} = (\mathbf{Goal}, \eta, \mathbf{Prf}, \text{concl})$ be a proof representation for \mathcal{I} . A formal development $dev = (S, P)$ consists of a specification $S \in \text{Spec}$ and a set P of proofs, such that each proof obligation $(\Gamma \models_{\Sigma} \varphi) \in \text{obl}(dg(S))$ is covered by a Σ -proof $p \in P$.

Again, informally, a development is called *verified* or *completed* if each proof obligation is not only covered, but also verified by a proof in P .

We can now explain more detailed what was meant with the informal visualisation that we have provided in Figure 3.1 on page 38: the well-formedness condition for a development dev is visualised by Figure 3.5, using the concepts that we have introduced above. The dashed arrows from a development dev to specification S and proofs P represent the fact that dev consists of S and P . π_1 and π_2 project a development $dev = (S, P)$ to its components S and P , respectively. The solid lines between S and D and between D and O describe the mapping from specification to the associated development graph and from development graph to the associated proof obligations. Finally, the solid line between P and O visualises that each Σ -proof obligation in O is covered by a Σ -proof in P . In other

⁷Note that this defines whether a Σ -proof covers a Σ' -obligation, for arbitrary signatures Σ and Σ' . If the two signatures are distinct the proof does not cover the obligation.

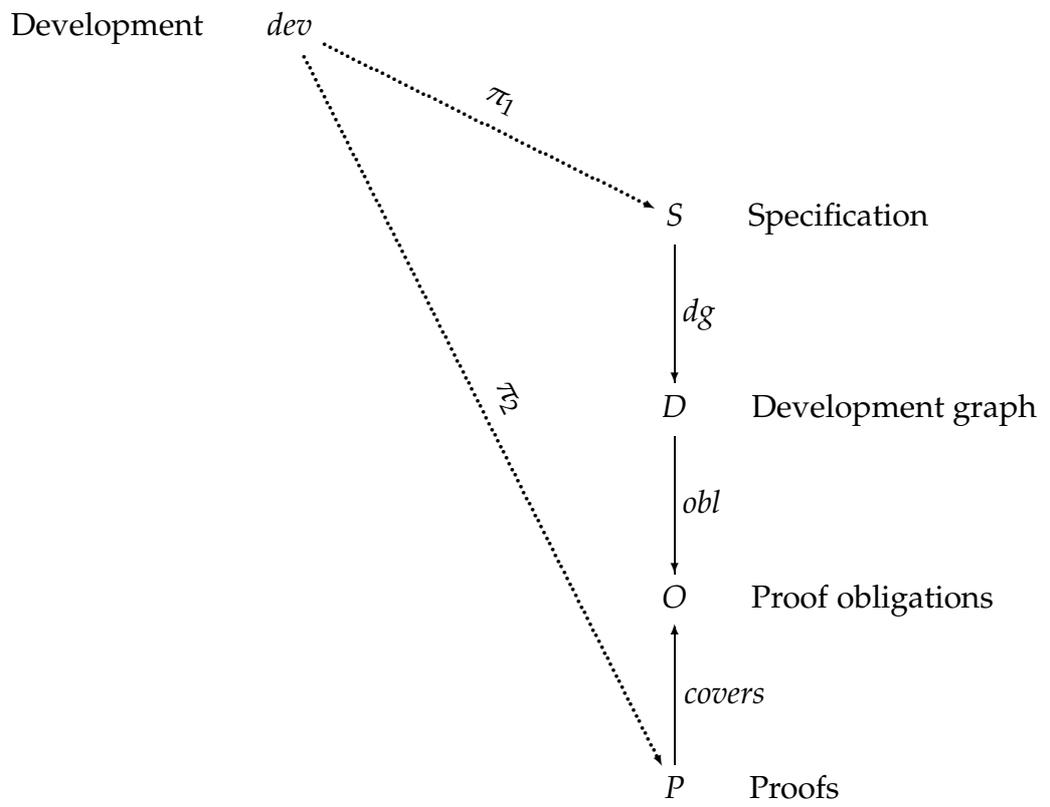


Figure 3.5: Development well-formedness

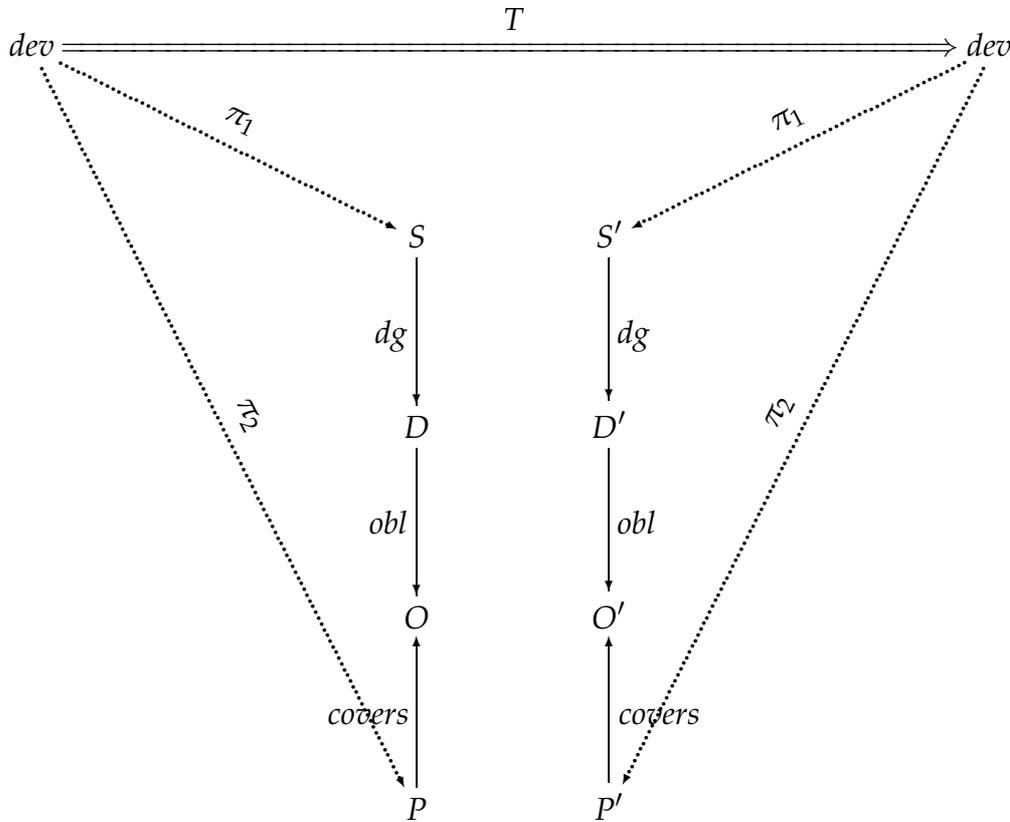


Figure 3.6: Development transformation

words, a formal development is well-formed iff each proof obligation arising from the development graph associated to the specification is covered by a proof.

3.7 Integration with Existing Tools

Let us come back to the issue we raised at the beginning of this chapter: we want to integrate the types of transformations that we motivated in Chapter 2 into existing support tools. This means that our overall goal is to provide transformations on developments, i.e. specifications and proofs, such that well-formedness of developments is preserved. In Figure 3.6, the double arrow between dev and dev' visualises the transformation taking dev to dev' .

The condition on transformations is the following: if dev' results from

dev by the transformation T , then dev' has to be well-formed, i.e. each obligation in

$$obl(dg(\pi_1(dev')))$$

is covered by a proof in $\pi_2(dev')$, provided dev is well-formed, i.e. if each obligation in

$$obl(dg(\pi_1(T(dev))))$$

is covered by a proof in $\pi_2(dev)$. In terms of the diagram this means that the triangle underneath dev' commutes if only the left triangle underneath dev commutes. Of course, we will break down development transformations into transformations on specifications, development graphs, proof obligations, and proofs. Figure 3.7 illustrates this approach, where T_S , T_D , T_O , and T_P represent transformations on specification, development graph, obligation, and proof transformations, respectively. The idea is that if the boxes in Figure 3.7 commute, and if T on developments is defined component-wise in terms of T_S and T_P (in which case the trapezium dev, S, S', dev' and the outermost trapezium dev, P, P', dev' both commute), then the well-formedness condition for dev implies the well-formedness of dev' .

Chapter 4 presents an abstract view of the box marked by (b) in Figure 3.7 in terms of the abstract concepts that we have introduced. Several classes of transformations on development graphs and corresponding transformations on proof obligations are identified and related to each other. This provides us with a language and classification on which we base the concrete transformations corresponding to the boxes (a) and (c) in the figure, which we will present in Part III. The immediate benefit is that we can treat (a) and (c) separately in Part III using the framework that we provide here to bridge the gap visualised by the box (b).

3.8 Transformations

So far we have not explained our notion of transformation in detail: a transformation rule (or simply transformation) is a binary relation T that leaves a property Q invariant. For example, a development transformation is a relation T between developments such that if $T(dev, dev')$ and dev is well-formed, then dev' is also well-formed.

Definition 3.23 (Transformation) *Let a set X (the domain) be given, and let Q be an unary predicate over X . Further, let T be a binary predicate over X . T is a Q -transformation rule (or Q -transformation for short) over X iff $Q(x)$ and $T(x, x')$ implies $Q(x')$ for all $x, x' \in X$.*

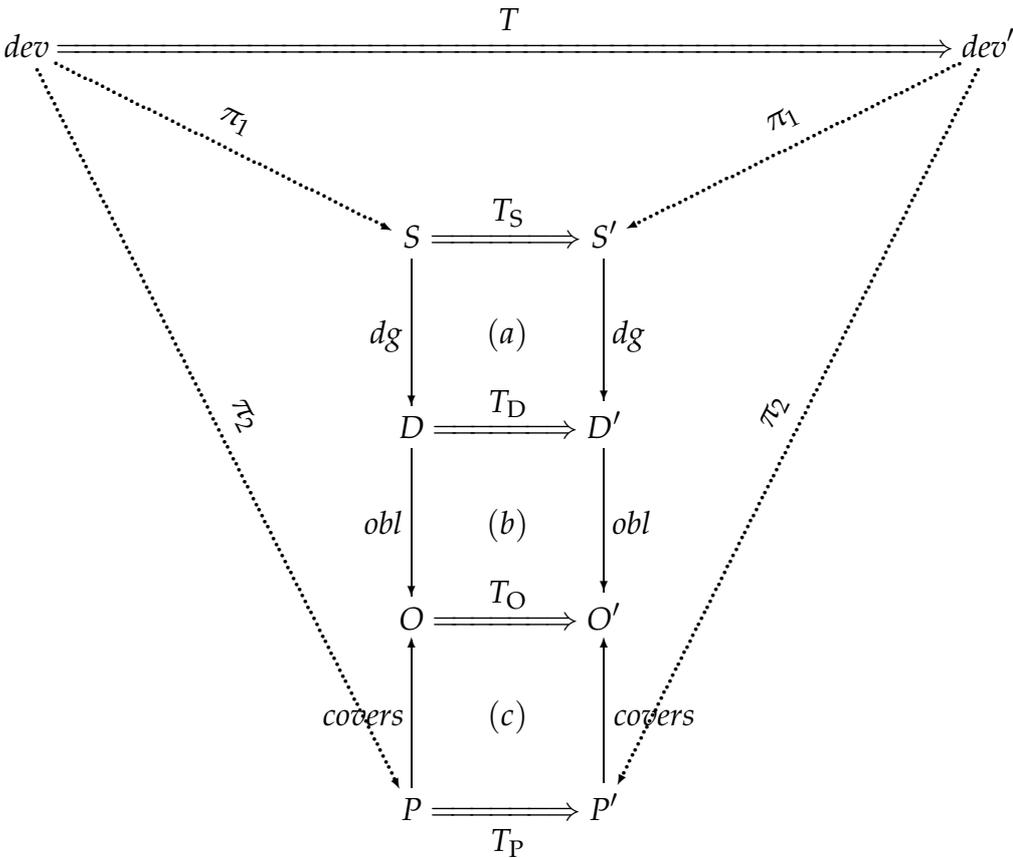


Figure 3.7: Layers of development transformation

A pair (x, x') is said to be a *T-transformation instance* if $T(x, x')$. In this case we also say that x' results from x by a *T-transformation*. In case the graph of T is the graph of a partial function over X we also call T a transformation function, and we write $T(x)$ for x' if $T(x, x')$. Where it is clear from the context we will often omit X and Q .

Thus development transformations T are transformations over developments (S, P) consisting of specification and proofs, where Q is the condition that a pair is a development, i.e. it satisfies the well-formedness condition for developments given in Definition 3.22. Development graph transformations T_D are arbitrary transformations over development graphs, proof obligation transformations T_O are transformations over proof obligations, and proof transformations T_P are arbitrary transformations over proofs – in all three cases, Q is well-formedness and is already guaranteed by the fact that Q holds for all elements of the domain.

Note that in Figure 3.7, (a) corresponds to transformations over specification/development graph pairs, (b) corresponds to development graph/proof obligation pairs, and (c) corresponds to transformations over proof obligation/proof pairs. The invariants are that the development graph results from the specification, that the proof obligations are the ones corresponding to the development graph, and that the proof covers the proof obligation, respectively. Transformations involving two artefacts such that transformations at one end necessitate transformations at the other have been coined *coupled transformations* in [Läm04].

The main property of transformations is that if we start out with a well-formed development and successively apply development transformations, then the resulting development is also well-formed. In practice, we are interested in transformation rules that are partial functions: these allow us to actually transform one development into another one by applying the partial function, if only the original development is in the domain of the transformation. Later we only describe transformations that can be seen easily to be computable (with moderate cost), and thus we need not be concerned with whether the relation is decidable (or the partial function computable).

A partial transformation function can be built from any transformation T by restricting the graph of T such that the result is the graph of a partial function. Since any partial function that is contained in the relation T will do, we can often leave unspecified the concrete choices involved in making a relation right-unique. Restricting a transformation relation to a transformation function is a special case of the fact that each subrelation of a transformation T is again a transformation.

3.9. Summary

These facts are trivial, but since we rely on them, we make them explicit in the following theorem.

Theorem 3.24 Let T_1 ($1 \leq i \leq n$) be Q transformations over X and let T be defined by

$$T(x_0, x_n) \text{ iff there is a sequence } \langle x_1, \dots, x_n \rangle \in X^* \\ \text{such that } T_i(x_{i-1}, x_i) \text{ holds for all } 1 \leq i \leq n .$$

Then T is also a Q -transformation. In particular $Q(x_0)$ implies $Q(x_n)$.

If T is a Q -transformation over X and $T' \subseteq T$ then T' is also a Q -transformation over X .

Proof of 3.24 The proofs are by straightforward induction over n and trivial application of modus ponens. \square

In particular, if the graph of a partial function $f : X \hookrightarrow X$ is a subset of T , i.e. $T(x, f(x))$, then f is a Q -transformation function. Theorem 3.24 implies that if f_i are Q -transformation functions, x is in the domain of $f_n \circ \dots \circ f_1$ and $Q(x)$, then $Q((f_n \circ \dots \circ f_1)(x))$.

3.9 Summary

We have introduced an abstract view of the context for describing our transformations. Transformations are orthogonal to the structuring concepts that development graphs provide and we want the transformations to be applicable in addition to development graphs. Thus, we need to integrate our transformations into existing tools that use development graphs. In this chapter we have made explicit the assumptions that this entails for the framework in which the transformations will be formulated. We have explained the role of a framework for development graph and proof obligation transformations to ease the presentation by distinguishing generic aspects from the ones that are dependent on the concrete logic and proof representation.

Chapter 4

Development Graph Transformations

4.1 Overview

When we formulate the transformations as described in Chapter 2 for structured specification languages like VSE-SL or CASL, and then determine their effects on the associated development graphs, we obtain several classes of development graph transformations. These classes are described systematically in this chapter. Each transformation class falls in one of two categories. The first changes the structure of the development graph but leaves the contents of nodes and links common to both graphs unchanged. The second keeps the structure of the development graph fixed and changes the contents of nodes and links, i.e. signatures, axioms, and signature morphisms. Some of the classes roughly correspond to basic development graph operations that have been defined for the management of development graphs before, e.g. in [Hut00], [AHMS00]. The relationship is briefly discussed in Section 4.5. In particular, the existing basic development graph operations can be represented as development graph transformations. Therefore, the question of whether the set of transformations that are provided by a concrete instantiation of the framework is complete, cf. the discussion in Section 2.3.6 on page 33, is not an issue: it cannot be less complete than the original setup, which is complete. Similarly, adequacy then reduces to the question of whether there are changes that are expensive to carry out and that can be carried out more efficiently using one of our transformations.

Transforming a development graph from D to D' means that the corresponding proof obligations change from $O = obl(D)$ to $O' = obl(D')$.

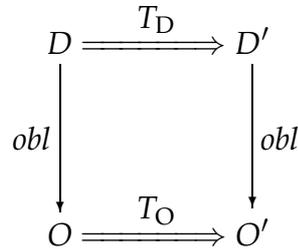


Figure 4.1: Induced transformations

Proofs are associated with O , and we want to use these to construct proofs for O' . We do not, therefore, want to compute O' from D' . Rather we want to derive O' from O directly, giving us a chance to derive proofs for O' from those for O .

The question is thus whether we can present a proof obligation transformation T_O relating O and O' , cf. Figure 4.1. O and O' are both sets of obligations, and the question is whether for each proof obligation $o' \in O'$ there is a similar proof obligation $o \in O$. Later, we will use this association to transform the proof for o into a proof for o' .

For each of the classes of development graph transformations we will thus describe the effects it has on proof obligations by describing associations between old and new proof obligations and by corresponding transformations between each pair of old and new proof obligation, such that the diagram in Figure 4.1 commutes. In this case we say informally that development graph transformations induce the proof obligation transformations, or that the proof obligation transformations can be used to propagate the development graph transformations to proof obligations.

We concentrate on describing which changes to obligations may possibly be caused by changes to a development graph, and how new and corresponding old proof obligation are identified, together with a formal notion of their difference. We are interested in the way in which changes are propagated in order to mechanize the propagation. The propagation itself is, however, not part of the trusted kernel: the resulting formal developments are self-contained and are checked for their well-formedness independently from their history. Of course, the mechanical propagation should still be correct because otherwise the resulting developments are of no use to the software engineer. Thus we provide proofs where it is useful to convince ourselves that the propagation is sound.

We list classes of development graph transformations. Instances of

4.2. Changing the Graph Structure

these classes can be used later on as a language to express what effects specification transformations have on the level of development graphs. Similarly, since we know how these classes of development graphs transformations induce proof obligation transformations, we can determine which instances of the transformations we need to support for proof obligations and proofs.

For the rest of the chapter we assume that we are given an arbitrary but fixed institution $\mathcal{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ and a fixed proof representation $(\mathbf{Goal}, \eta, \mathbf{Prf}, \mathit{concl})$ for \mathcal{I} .

4.2 Changing the Graph Structure

This section deals with transformations that add or delete whole nodes or links of a development graph, *without changing other nodes or links*. These transformations change the structure of the graph.

4.2.1 Adding and Deleting Nodes

Adding new nodes to a development graph necessarily produces isolated nodes. Otherwise the old development graph would have contained links from or to non-existent nodes, in contradiction of the definition of a development graph. On the other hand, deleting a node which is not isolated would produce a development graph with links from or to a non-existing node, again contradicting the conditions for development graphs.

Definition 4.1 (DG transformation: Tr_{nodes}) *The development graphs transformation Tr_{nodes} relates a development graph $D = (N, A, C)$ and another development graph $D' = (N', A, C)$.*

Note that all nodes not in $N \cap N'$ are necessarily isolated. Obviously, isolated nodes do not influence the set of local or global axioms of any other node. Similarly, isolated nodes do not influence proof obligation links. Therefore, isolated links can be ignored as far as proof obligations are concerned. This means that Tr_{nodes} can be propagated to proof obligations with the identity transformation on proof obligations.

4.2.2 Adding and Deleting Links

Adding a definitorial link to a development graph potentially changes the set of proof obligations determined by the development graph calculus

in the following way. The new link adds additional paths along which sentences can be inherited. It also possibly influences the set of proof obligation links that a development graph $D = (N, A, C)$ is mapped to by $C \downarrow_{N,A}$, because the new link may be applicable to decompose a global postulated link, or it may lead to the subsumption of a local postulated link. It does *not* change the signatures and local axioms that are associated with the nodes of the development graph, however: signatures and local axioms are associated to nodes independently of other nodes and links; the relationship between the signatures of nodes connected by a link is given by the condition that the link carries a morphism from the source signature to the target signature, and these are not changed. Adding a definitorial link, therefore, entails

1. that there possibly exist new proof obligation links and some old ones might be obsolete, and
2. that for proof obligation links l that have not changed, the set of proof obligations $obl(l)$ may be different, depending on whether the global set of axioms in the target node of l has changed.

Removing definitorial links has the same effects: additional proof obligation links may appear because the link was needed to subsume a local postulated link, others might disappear because decomposing a global postulated link might now produce less new local postulated links. Adding or removing postulated links possibly leads to new proof obligation links or to old ones becoming obsolete.

We integrate all transformations that deal with adding or removing links into the following definition.

Definition 4.2 (DG transformation: Tr_{links}) *The development graph transformation Tr_{links} relates a development graph $D = (N, A, C)$ and another development graph $D' = (N, A', C')$.*

If we compare the two sets of proof obligation links $L = C \downarrow_{N,A}$ and $L' = C' \downarrow_{N,A'}$ of the development graphs D and D' that are in relation Tr_{links} , there are (i) links that are in L but not in L' , (ii) links that are in L' but not in L , and (iii) links that are in both. Note that by definition of $C \downarrow_{N,A}$, all links in L (and L') are local links.

- (i) For any proof obligation link $l \in L \setminus L'$ that contributes to the proof obligations for D but not for D' , the old proofs for the proof obligations of D and l are no longer needed. They can be kept in an archive in case they are needed in the development later on. We will not

4.3. Changing the Content of Nodes or Links

deal with this kind of archive in this thesis, so we assume that they are thrown away.

- (ii) For any proof obligation link $l \in L' \setminus L$ that contributes to the proof obligations for D' but not for D , there is no old proof. So a new proof needs to be created for each proof obligation for D' and l .
- (iii) Let the link $l : n_1 \xrightarrow{\sigma} n_2$ be a proof obligation link in $L \cap L'$ (and thus local). The proof obligations for D and l and for D' and l , respectively, are given by Definition 3.17 as

$$\begin{aligned} & \{ \Phi_D^{n_2} \vdash_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi) \mid \varphi \in \Phi^{n_1} \} \text{ and} \\ & \{ \Phi_{D'}^{n_2} \vdash_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi) \mid \varphi \in \Phi^{n_1} \} , \end{aligned}$$

where it is obvious that each proof obligation

$$o' = (\Phi_{D'}^{n_2} \vdash_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi))$$

in the second line corresponds to a proof obligation

$$o = (\Phi_D^{n_2} \vdash_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi))$$

in the first line that only differs from o' in the set of assumptions.

The development graph transformation Tr_{links} can, therefore, be propagated to proof obligations

- by throwing obsolete proof obligations away,
- by adding new proof obligations, and
- by changing existing proof obligations by adding or removing assumptions.

4.3 Changing the Content of Nodes or Links

We will introduce transformations that change the contents of nodes (i.e. signatures and local axioms) or links (i.e. morphisms) without changing the graph structure. In such a case there is an association between old and new proof obligation links: these links are determined by the development graph calculus from the graph structure, and the graph structure is invariant.

The informal notion of changing the contents of a development graph is formally expressed by two graphs that have the same structure but different contents. The relationship between two development graphs with identical structure can be adequately expressed by a bijection between nodes and links that respects the graph structure, i.e. by a graph isomorphism.

Definition 4.3 (Development graph isomorphism) Let $D = (N, A, C)$ and $D' = (N', A', C')$ be two development graphs. D and D' have the same structure if there are bijections $h_N : N \rightarrow N'$, $h_A : A \rightarrow A'$, and $h_C : C \rightarrow C'$, respectively, such that

- for each link $l \in A$ from node $n_1 \in N$ to $n_2 \in N$, the link $h_A(l) \in A'$ is a link from node $h_N(n_1) \in N'$ to $h_N(n_2) \in N'$,
- for each link $l \in C$ from node $n_1 \in N$ to $n_2 \in N$, the link $h_C(l) \in C'$ is a link from node $h_N(n_1) \in N'$ to $h_N(n_2) \in N'$, and
- $h_A(l)$ or $h_C(l)$ is a local (global) link iff l is a local (global) link.

We say that $h = h_N \cup h_A \cup h_C$ is a development graph isomorphism, and since N , A , and C are disjoint, we write $h(n)$ and $h(l)$ for $h_N(n)$, $h_A(l)$, and $h_C(l)$ without the risk of ambiguity.

All the remaining transformations follow a common idea. Proof obligations are computed using the normal form $C \downarrow_{N,A}$, which results from the development graph by exhaustive application of the *decomposition* and the *subsumption* rule. The *decomposition* rule only depends on the *structure* of the development graph. A development graph isomorphism h between D and D' ensures that the structure of both graphs is the same. Therefore, D' can be decomposed exactly as D . This does not hold for the *subsumption* rule: the applicability condition of the *subsumption* rule depends on the link morphisms, and link morphisms are not invariant over development graphs isomorphisms. However, it turns out that all transformations that we will introduce in this section relate D and D' in such a way that the application of *both* rules can in fact be simulated: whenever a link is subsumed in D , then the h -image of the link is also subsumed in D' . In this case we say that a development graph bijection *respects* morphisms. The formal definition is as follows.¹

¹Recall that the notion of paths that we have introduced in Definition 3.9 only considers *definitorial* links. In contrast, in the following definition we do not distinguish between definitorial and postulated links.

4.3. Changing the Content of Nodes or Links

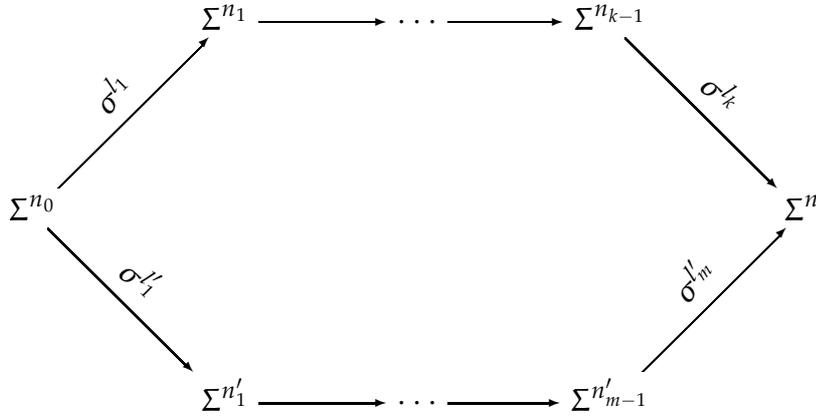


Figure 4.2: Respecting morphisms

Definition 4.4 (Respecting morphisms) Let h be a development graph bijection between $D = (N, A, C)$ and D' . We say that h respects morphisms of D and D' iff for each two sequences of (local or global, definitorial or postulated) links

$$\left\langle n_0 \xrightarrow{\sigma^{l_1}} n_1, \dots, n_{k-1} \xrightarrow{\sigma^{l_k}} n \right\rangle \in (A \cup C)^*$$

and

$$\left\langle n_0 \xrightarrow{\sigma^{l'_1}} n'_1, \dots, n'_{m-1} \xrightarrow{\sigma^{l'_m}} n \right\rangle \in (A \cup C)^*$$

we have

$$\begin{aligned} \sigma^{l_k} \circ \dots \circ \sigma^{l_1} &= \sigma^{l'_m} \circ \dots \circ \sigma^{l'_1} \\ \text{iff } \sigma^{h(l_k)} \circ \dots \circ \sigma^{h(l_1)} &= \sigma^{h(l'_m)} \circ \dots \circ \sigma^{h(l'_1)}, \end{aligned}$$

cf. Figure 4.2.

If a development graph bijection respects morphisms of D and D' then any link in D is subsumed by a path in D iff the h -image of the link is subsumed by the h -image of the path in D' . However, the decomposition rule introduces new postulated links that are not in D or D' . If such a new link is subsumed by a path in D , it is *not* clear *a priori* whether the result of simulating the decomposition rule in D' also produces a link that is subsumed by the h -image of the path in D' . The following theorem states that this is the case. It asserts that h can be complemented canonically by a bijection \bar{h} between the newly introduced links such that morphisms are

Chapter 4. Development Graph Transformations

still respected: each newly added link for D is in parallel to a sequence of links that already exists in D with the same morphism. Similarly, each newly added link in D' is in parallel to a sequence of links with the same morphism in D' . Links in D and D' , however, are related by h , which respects morphisms, and so the morphisms of the new links are related by \bar{h} .

Theorem 4.5 Let h be a development graph isomorphism between $D = (N, A, C)$ and $D' = (N', A', C')$ that respects the morphisms. Then there is a bijection \bar{h} between $C \downarrow_{N,A}$ and $C' \downarrow_{N',A'}$ such that for each link $l : n_1 \xrightarrow{\sigma^l} n_2 \in C \downarrow_{N,A}$

- $\bar{h}(l)$ is a link from $h(n_1)$ to $h(n_2)$, and
- there is a sequence of links l_1, \dots, l_m and a sequence of nodes k_0, \dots, k_m such that
 - $k_0 = n_1$ and $k_m = n_2$,
 - l_i is a link from k_{i-1} to k_i (for $1 \leq i \leq m$),
 - $\sigma^l = \sigma^{l_m} \circ \dots \circ \sigma^{l_1}$, and
 - $\sigma^{\bar{h}(l)} = \sigma^{h(l_m)} \circ \dots \circ \sigma^{h(l_1)}$

We say that h induces \bar{h} .

Proof of 4.5 Let $D = (N, A, C)$ and $D' = (N', A', C')$ be development graphs and h a development graph bijection that respects the morphisms. We proceed by induction over the length of the derivation of $C \downarrow_{N,A}$.

Base case: If $C = C \downarrow_{N,A}$ then both $C' = C' \downarrow_{N',A'}$ and $\bar{h} = h \cap (C \times C')$ satisfies the conditions of the theorem. It is easy to see that the decomposition rule is applicable for C if it is for C' . For the subsumption rule, we need to appeal to Definition 4.4 to see that a local link l in C is subsumed by a path in A iff $h(l)$ is subsumed by a path in A' .

Step case: The derivation has at least one rule application.

- Assume that we apply the subsumption rule, i.e. l is subsumed by a path in A . Then Definition 4.4 says that $h(l)$ is subsumed by a path in A' , and thus the subsumption rule is also applicable to C' . The two premisses now read $(N, A, C \setminus \{l\})$ and $(N', A', C' \setminus \{h(l)\})$. h restricted to $(C \setminus \{l\}) \times (C' \setminus \{h(l)\})$ trivially respects the morphisms. Thus, the theorem holds by appeal to the induction hypothesis.

4.3. Changing the Content of Nodes or Links

- Assume that we apply the decomposition rule to link $l \in C$. Thus, $h(l)$ is also a global link, and we can apply the decomposition rule to $h(l) \in C'$. For each local or global link l_1 in A with target n_1 there is a corresponding link $h(l_1)$ in A' with corresponding source and target, and there are no more links in A' with target $h(n_1)$. Thus, for each new local link $l_2 : n_0 \xrightarrow{\sigma^l \circ \sigma^{l_1}} n_2$ in the premiss for C , there is a corresponding local link $l'_2 : h(n_0) \xrightarrow{\sigma^{h(l)} \circ \sigma^{h(l_1)}} h(n_2)$, and similarly for each new global link. We define a new bijection h_1 for the premisses by extending the graph of h by the pairs (l_2, l'_2) . This extended h also respects the morphisms for the premisses. Thus, again we can appeal to the induction hypothesis. □

This means that if h respects morphisms of D and D' , we can simulate the application of development graph calculus rules in D by applying the corresponding rules in D' , yielding a set of proof obligation links for D and D' that are related by the canonical complement \bar{h} of h . Thus, given a proof obligation link

$$l : n \xrightarrow{\sigma} n'$$

for D , we compare it to the corresponding proof obligation link

$$\bar{h}(l) = l' : h(n) \xrightarrow{\sigma'} h(n')$$

for D' . The differences between the local signatures and axioms of n , n' and σ on the one side and $h(n)$, $h(n')$, σ' on the other, respectively, directly influence the difference between proof obligations resulting from l and from l' :

$$\left\{ \Phi_D^{n'} \models_{\Sigma^{n'}} \mathbf{Sen}(\sigma)(\varphi) \mid \varphi \in \Phi^n \right\} \quad vs. \quad \left\{ \Phi_{D'}^{h(n')} \models_{\Sigma^{h(n')}} \mathbf{Sen}(\sigma')(\varphi) \mid \varphi \in \Phi^{h(n)} \right\}$$

However, the proof obligations also refer to axioms in $\Phi_D^{n'}$ and $\Phi_{D'}^{h(n')}$ that are inherited by nodes n' and $h(n')$ according to the definitorial links in D and D' . Since the local axioms of other nodes may have changed as well, this implies there are indirect influences on the changed proof obligations.

The different classes of transformations that we present in the rest of this chapter are instances of this pattern – an exception to this occurs when moving axioms, cf. Section 4.3.1.

4.3.1 Adding, Deleting, and Moving Axioms

A relatively simple class of development graph transformations is given by keeping the signatures and link morphisms constant and only adding, removing, or moving axioms.

Definition 4.6 (DG transformation: Tr_{axioms}) *Two development graphs $D = (N, A, C)$ and $D' = (N', A', C')$ are in relation Tr_{axioms} , iff there exists a development graph isomorphism h between D and D' and the signatures of the nodes and the morphisms of the links are unchanged, i.e.*

- $\Sigma^n = \Sigma^{h(n)}$ for every $n \in N$ and
- $\sigma^l = \sigma^{h(l)}$ for every $l \in A \cup C$.

The local axioms Φ^n of a node n may be different from those of $h(n)$. Since the link morphisms are unchanged by h the development graph bijection trivially respects the morphisms. Using Theorem 4.5 we know for each proof obligation link $l : n \xrightarrow{\sigma} n'$ for D that the corresponding proof obligation link for D' is from $h(n)$ to $h(n')$, carries the same morphism σ , and that $h(n)$ and $h(n')$ have the same signatures as n and n' . The node $h(n)$ may, however, have different axioms than n . Since there is a proof obligation for D for each axiom in Φ^n , and similarly for D' and $\Phi^{h(n)}$, we have to distinguish the cases that (i) φ is in Φ^n but not in $\Phi^{h(n)}$, (ii) that φ is not in Φ^n but is in $\Phi^{h(n)}$, and (iii) that φ is in both.

- (i) The proof obligation for l and $\varphi \in \Phi^n$ does not have a corresponding proof obligation in $obl(D')$, so the proof obligation can be thrown away.
- (ii) The proof obligation o for $h(l)$ and $\varphi \in \Phi^{h(n)}$ does not have a corresponding proof obligation in $obl(D)$, and thus we need to add o as a new proof obligation.
- (iii) Let $\Phi_D^{n'} \models_{\Sigma^{n'}} \mathbf{Sen}(\sigma)(\varphi)$ be the proof obligation for D , l , and $\varphi \in \Phi^n$. We know that φ , and σ are unchanged, and that $\Sigma^{n'} = \Sigma^n$. Thus, the only possible difference is between $\Phi_D^{n'}$ and $\Phi_{D'}^{h(n')}$, and the corresponding proof obligation for D' , $h(l)$, and φ reads $\Phi_{D'}^{h(n')} \models_{\Sigma^{n'}} \mathbf{Sen}(\sigma)(\varphi)$.

Axioms can be moved from one node to another one by removing them from one node and adding them to another one. Note that moving an axiom may include translating the sentence according to the relationship in

4.3. Changing the Content of Nodes or Links

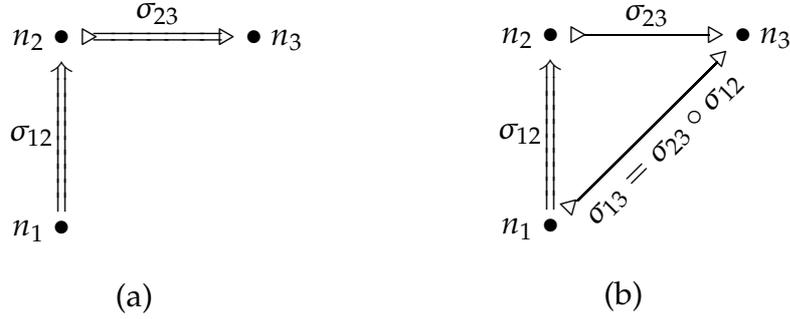


Figure 4.3: Moving axioms

terms of signature morphisms between the old and new node of the axiom. One would think that moving an axiom along a link and mapping it correspondingly along the morphism of the link would allow us to leave many obligations unchanged because we can determine that the axiom is still visible, although it is inherited from another node. As it turns out, where the axiom is used as an assumption, this is handled completely satisfactorily by case (iii) above: it so happens that $\Phi_{D'}^{h(n')}$ is the same as $\Phi_D^{n'}$ in this case. However, if the axiom played the rôle of a conjecture things are different. The reason is that we have associated proof obligations with proof obligation links. For instance, consider the simple development graph given in Figure 4.3(a). Again, postulated links, like the one linking n_2 and n_3 , have been drawn with an arrow head and tail to distinguish them from definitorial links. Applying the development graph calculus yields the graph in Figure 4.3(b) containing two local proof obligation links. If φ is a local axiom of node n_1 , then one of the proof obligations for the link between n_1 and n_3 reads

$$\Phi_D^{n_3} \models_{\Sigma^{n_3}} \mathbf{Sen}(\sigma_{13})(\varphi). \quad (4.1)$$

Moving φ from n_1 to n_2 , where it now reads $\varphi' = \mathbf{Sen}(\sigma_{12})(\varphi)$, makes the proof obligation (4.1) obsolete for the link between n_1 and n_3 , but introduces the obligation

$$\Phi_D^{n_3} \models_{\Sigma^{n_3}} \mathbf{Sen}(\sigma_{23})(\varphi') \quad (4.2)$$

for the link between n_2 and n_3 . Since $\sigma_{13} = \sigma_{23} \circ \sigma_{12}$ and $\varphi' = \mathbf{Sen}(\sigma_{12})(\varphi)$, (4.1) and (4.2) turn out to be the same proof obligation. Of course, the argument is independent of whether the axiom is mapped along a link that was used to decompose a global postulated link. However, the argument only works if the link is a definitorial link.

Moving an axiom φ' backwards over a link is somewhat more complicated: since $\mathbf{Sen}(\sigma)$ is not surjective in general it is not clear whether there is a sentence φ such that $\varphi' = \mathbf{Sen}(\sigma)(\varphi)$ for the given φ' . However, if such a φ exists then the argument above can be repeated, so the proof obligation remains unchanged but is attached to another link.

It is possible to define a separate class of transformations for moving axioms, or to relax the idea that proof obligation links can be inspected one by one. It would then be necessary to search, for each newly introduced proof obligation associated with $h(n) \xrightarrow{\sigma} h(n')$, whether the same proof obligation was already associated with another proof obligation link $n'' \xrightarrow{\sigma''} n'$ in the original development graph (note the same target node). This also captures a further generalisation, which allows an arbitrary, undirected path along which the axiom is moved, sometimes along, sometimes backwards over a definitorial link. No matter which of these strategies is chosen, the effects that occur for proof obligations are the same.

The development graph transformation Tr_{axioms} can be propagated to proof obligations

- by throwing obsolete proof obligations away,
- by adding new proof obligations, and
- by changing existing proof obligations by adding or removing assumptions.

These are exactly the same effects on proof obligations that we determined for the development graph transformation Tr_{links} .

4.3.2 Changing Axioms

The examples in Chapter 2 show an important class of transformations that are concerned with axioms that have to be changed, i.e. the transformations deal with replacing subformula or subterm occurrences of a given formula in a controlled fashion. Of course, this can be simulated by removing the old axiom and adding the new one using Tr_{axioms} , which we have introduced in the preceding section. However, in practice it is vitally important to know the relationship between the old and the new axiom and to use it to transform the proofs. This relationship describes two things. Given two sets of axioms, first it describes which axiom in the second set

4.3. Changing the Content of Nodes or Links

results from which axiom in the first. Second, it describes for each such pair how the two axioms relate in detail. For instance, if we have

$$\begin{aligned}\Gamma_1 &= \{\varphi_1, \varphi_3, \varphi_4\} \quad \text{and} \\ \Gamma_2 &= \{\varphi_1, \varphi_2 \Rightarrow \varphi_3, \varphi_4\} ,\end{aligned}$$

then a possible association is

$$\begin{aligned}\varphi_1 &\text{ is associated with } \varphi_1 , \\ \varphi_3 &\text{ is associated with } \varphi_2 \Rightarrow \varphi_3 , \text{ and} \\ \varphi_4 &\text{ is associated with } \varphi_4 .\end{aligned}$$

For the second pair, the detailed relationship between the two formulae is that the original formula becomes the conclusion of an implication, with a new precondition.

In the context of development graph transformations, we can abstract from the second aspect. All we use is the association between axioms.

The following transformations assume that the structure of the development graph, the signatures and the link morphisms are unchanged. However, in each node several axioms may be changed.

We represent changes to the set of local axioms for a given node n by a binary relation q_n , called a *sentence replacement*. Each pair of axioms that is in relation q_n represents the change of a *local* axiom φ to a local axiom φ' . Note that we do not require q_n to be a partial function, and thus an axiom φ can be changed to more than one other axiom, say φ'_1 and φ'_2 . This is necessary for the following reason. Below, we extend q_n to q_n^* such that q_n^* represents the changes to the *global* set of axioms. To this end, we inherit the changes represented by q_n along definitorial links. In this process, two pairs of sentences (φ_1, φ'_1) and (φ_2, φ'_2) that are in relation q_n can be mapped to the pairs $(\mathbf{Sen}(\sigma)(\varphi_1), \mathbf{Sen}(\sigma)(\varphi'_1))$ and $(\mathbf{Sen}(\sigma)(\varphi_2), \mathbf{Sen}(\sigma)(\varphi'_2))$, which are then required to be in relation $q_{n'}^*$ for the target node n' . It is entirely possible that $\mathbf{Sen}(\sigma)(\varphi_1) = \mathbf{Sen}(\sigma)(\varphi_2)$ and $\mathbf{Sen}(\sigma)(\varphi'_1) \neq \mathbf{Sen}(\sigma)(\varphi'_2)$. In this case, $\mathbf{Sen}(\sigma)(\varphi_1)$ is replaced by two sentences, i.e. $\mathbf{Sen}(\sigma)(\varphi'_1)$ and $\mathbf{Sen}(\sigma)(\varphi'_2)$, and thus sentence replacements are represented by binary relations rather than partial functions on sentences.

Definition 4.7 (Sentence replacement) *Let $\Gamma \subseteq \mathbf{Sen}(\Sigma)$ be a set of sentences. A sentence replacement for Γ is a binary relation $q \subseteq \Gamma \times \mathbf{Sen}(\Sigma)$. We also write q for the function on sets of sentences defined by*

$$q(\Gamma) = (\Gamma \setminus \{\varphi \mid (\varphi, \varphi') \in q\}) \cup \{\varphi' \mid (\varphi, \varphi') \in q\} . \quad (4.3)$$

Chapter 4. Development Graph Transformations

Thus, we write $q(\Gamma)$ for the set that results from Γ by the replacement. Using sentence replacements, the definition of the development graph transformation is as follows.

Definition 4.8 (DG transformation: Tr_{occ}) *The two development graphs $D = (N, A, C)$ and D' are in relation Tr_{occ} iff there exist a development graph isomorphism h between D and D' and an N -indexed family of sentence replacements $(q_n)_{n \in N}$ such that*

- *the signatures of h -related nodes and the morphisms of h -related links are equal:*

$$Tr_{axioms}(D, D') \quad \text{and}$$

- *the local axioms of h -related nodes are related by q_n :*

$$\Phi^{h(n)} = q_n(\Phi^n) \quad \text{for every node } n \in N.$$

We say that D and D' are related by Tr_{occ} wrt. h and $(q_n)_{n \in N}$.

Obviously, Tr_{occ} is a subrelation of Tr_{axioms} as given in Definition 4.6 for adding, removing, and moving axioms: we can simulate changing a formula by removing the old one and adding a new one. However, the intention for Tr_{occ} is different since Tr_{occ} makes explicit use of the knowledge that φ and φ' are similar, whereas in Tr_{axioms} we assumed that different axioms are not usefully related. In the case of Tr_{axioms} , our best bet was to remove the old and add the new sentence. Our intention with Tr_{occ} is to propagate the information about similarities of the old and new sentence to proof obligations so that the information can be used to transform the covering proofs. Since proof obligations are formulated in terms of the *global* set of axioms of the source of their associated proof obligation link, we need to determine the changes to *global* set of axioms of those nodes.

Definition 4.9 (Global changes induced by Tr_{occ} , h and $(q_n)_{n \in N}$) *Let $D = (N, A, C)$ and D' be two development graphs related by Tr_{occ} wrt. the development graph bijection h and $(q_n)_{n \in N}$. Then the global changes induced by Tr_{occ} , h , and $(q_n)_{n \in N}$ is the N -indexed family of sentence replacements $(q_n^*)_{n \in N}$ consisting of the smallest sets $q_n^* \subseteq \mathbf{Sen}(\Sigma^n) \times \mathbf{Sen}(\Sigma^n)$ such that*

- $q_n \subseteq q_n^*$,
- for each $\varphi \in \Phi^n$, if $(\varphi, \varphi') \notin q_n$ for any $\varphi' \neq \varphi$ then $(\varphi, \varphi) \in q_n^*$,

4.3. Changing the Content of Nodes or Links

- $(\varphi, \varphi') \in q_{n_1}$ and $n_1 \xrightarrow{\sigma} n_2 \in A$ imply

$$(\mathbf{Sen}(\sigma)(\varphi), \mathbf{Sen}(\sigma)(\varphi')) \in q_{n_2}^*,$$

and

- $(\varphi, \varphi') \in q_{n_1}^*$ and $n_1 \xrightarrow{\sigma} n_2 \in A$ imply

$$(\mathbf{Sen}(\sigma)(\varphi), \mathbf{Sen}(\sigma)(\varphi')) \in q_{n_2}^* .$$

$(q_n^*)_{n \in N}$ is well-defined since the definitorial links are acyclic, and $(q_n^*)_{n \in N}$ relates the global axioms of node n and node $h(n)$. This is captured by the following theorem.

Theorem 4.10 Let $D = (N, A, C)$ and D' be in relation Tr_{occ} wrt. h and $(q_n)_{n \in N}$, and let $(q_n^*)_{n \in N}$ be the global changes induced by Tr_{occ} , h , and $(q_n)_{n \in N}$. Then, for each node $n \in N$, $\Phi_{D'}^{h(n)} = q_n^*(\Phi_D^n)$ holds.

Proof of 4.10 The idea is that the global set of axioms of n is given by

$$\Phi_D^n = \{ \varphi \mid (\varphi, \varphi') \in q_n^* \} ,$$

i.e. the projection of the graph of q_n^* on its domain, and the global axioms of $h(n)$ by

$$\Phi_{D'}^{h(n)} = \{ \varphi' \mid (\varphi, \varphi') \in q_n^* \} ,$$

i.e. the projection on the codomain. The proof is then by straightforward induction on the depth of nodes, using the fact that each member of Φ_D^n is either a local axiom of n or is a σ -instance of a local axiom of some node m such that $m \xrightarrow{\sigma} n$ or $m \xrightarrow{\sigma} n$ is in A , and similarly for $\Phi_{D'}^{h(n)}$ and $h(n)$. \square

This allows us to propagate the changes to the proof obligations as follows.

Theorem 4.11 Let D and D' be development graphs in relation Tr_{occ} wrt. h and $(q_n)_{n \in N}$, and let $(q_n^*)_{n \in N}$ be the global changes induced by Tr_{occ} , h , and $(q_n)_{n \in N}$. Then for each proof obligation for D' of the form

$$\Phi_{D'}^{h(n_2)} \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi') \quad \varphi' \in \Phi^{h(n_1)} \quad (4.4)$$

there is a proof obligation for D of the form

$$\Phi_D^{n_2} \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi) \quad \varphi \in \Phi^{n_1} \quad (4.5)$$

Chapter 4. Development Graph Transformations

with

$$\Phi_{D'}^{h(n_2)} = q_{n_2}^*(\Phi_D^{n_2}) \quad \text{and} \quad (4.6)$$

$$\Phi^{h(n_1)} = q_{n_1}(\Phi^{n_1}). \quad (4.7)$$

Proof of 4.11 Due to the definition of Tr_{occ} the link morphisms remain unchanged, and thus the development graph bijection h trivially respects the morphisms. Thus, according to Theorem 4.5, the development graph bijection h induces a bijection \bar{h} between the proof obligation links for D and D' such that for every proof obligation link $l : n_1 \xrightarrow{\sigma} n_2 \in C \downarrow_{N,A}$ we have $\bar{h}(l) : h(n_1) \xrightarrow{\sigma} h(n_2)$, i.e. the two links have the same link morphism. Additionally, the local and global sets of axioms of two nodes n and $h(n)$ are related by the sentence replacements q_n and q_n^* , respectively. \square

We first look at the consequences this has for the conjectures of the proof obligations by making a case distinction over whether $\varphi' \in q_{n_1}(\Phi^{n_1})$ is in Φ^{n_1} or not.

1. If φ' is a member of Φ^{n_1} then there is an old proof obligation for D
 - (a) with the same conjecture, and
 - (b) the changes in the assumptions are represented by $q_{n_2}^*$.
2. Otherwise, there is a pair $(\varphi, \varphi') \in q_{n_1}$, and thus there is an old proof obligation for D that differs from the new one as follows:
 - (a) the conjecture φ has been changed to φ' , and
 - (b) the changes in the assumptions are represented by $q_{n_2}^*$.

We can represent both cases uniformly along the following lines: Let q^* be the sentence replacement over Σ^{n_1} -sentences that is the relation $\{(\varphi, \varphi)\}$ for case 1 and that consists of the single pair (φ, φ') for case 2. The new proof obligation uniformly results from the old one by applying q^* to the singleton set $\{\varphi\}$ consisting of the conjecture and $q_{n_2}^*$ to the assumptions.

The development graph transformation Tr_{occ} can thus be propagated to proof obligations

- by mapping the assumptions and the conjecture according to two sentence replacements.

4.3.3 Extending and Restricting Signatures

Dually to adding axioms and keeping the signatures, signatures can also be extended while all axioms are kept unchanged. In terms of the examples in Chapter 2, we add signature symbols to a node.

We formalize this by assuming signature extensions to be signature morphisms. Then, if the signature of a node n is extended, which we now write as $\Sigma^{h(n)} = \sigma\Sigma^n$ for an appropriate σ , an existing proof obligation

$$\Gamma \models_{\Sigma^n} \varphi \quad \text{becomes} \quad \mathbf{Sen}(\sigma)(\Gamma) \models_{\sigma\Sigma^n} \mathbf{Sen}(\sigma)(\varphi),$$

i.e. the whole proof obligation is simply mapped along σ . The covering proof can then be mapped along $\mathbf{Prf}(\sigma)$.

The situation in which the signature of a node is restricted without changing the axioms is only marginally more complicated. Obviously, axioms can only be kept unchanged if the signature restriction has no influence on the axioms, i.e. if the removed signature items do not occur in the local axioms. Formally, we express this simply by saying that, whenever a restriction is fit, it is the converse of an extension, i.e. there is a morphism $\sigma : \Sigma^{h(n)} \rightarrow \Sigma^n$ such that all axioms of node n can be represented as $\mathbf{Sen}(\sigma)(\varphi)$ for some φ , and φ is an axiom in $h(n)$. For signature restrictions, we expect φ to be determined uniquely, and thus require $\mathbf{Sen}(\sigma)$ to be injective for signature extensions σ . For practical reasons we do not want to restrict signature extensions to be inclusions on the level of sentences: this would force any sentence representation to make the set $\mathbf{Sen}(\Sigma)$ of sentences over Σ a subset of the set $\mathbf{Sen}(\sigma\Sigma)$ of sentences over $\sigma\Sigma$ for every extension σ – which is technically not the case in most concrete support tool implementations. What is usually the case, though, is that $\mathbf{Sen}(\sigma)(\mathbf{Sen}(\Sigma)) \subseteq \mathbf{Sen}(\sigma\Sigma)$. Thus our definition of signature extensions is the following:²

Definition 4.12 (Signature extension) *If a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ is a signature extension, then $\mathbf{Sen}(\sigma)$ is an injective mapping.*

Since every function is trivially surjective on its range, $\mathbf{Sen}(\sigma)$ is a bijective function between $\mathbf{Sen}(\Sigma)$ and $\mathbf{Sen}(\sigma)(\mathbf{Sen}(\Sigma))$. Thus, the inverse exists for the range, and we assume that in practice for concrete institutions and the corresponding notions of signature extension, it can easily enough be determined.

We can now extend signature extensions and restrictions to whole development graphs.

²Note that the reverse is not required: not every injective mapping is required to be a signature extension.

Definition 4.13 (DG transformation: $Tr_{signatures}$) Development graphs $D = (N, A, C)$ and D' are in relation $Tr_{signatures}$ wrt. a development graph isomorphism h between D and D' and an N -indexed family of signature morphisms $(\sigma_n)_{n \in N}$ iff

- for each node $n \in N$ there is a signature extension σ_n such that

$$\Sigma^{h(n)} = \sigma_n \Sigma^n \quad \text{and} \quad \Phi^{h(n)} = \mathbf{Sen}(\sigma_n)(\Phi^n) \quad (4.8)$$

(for extensions) or

$$\sigma_n \Sigma^{h(n)} = \Sigma^n \quad \text{and} \quad \mathbf{Sen}(\sigma_n)(\Phi^{h(n)}) = \Phi^n, \quad (4.9)$$

(for restrictions) and

- for each link $l \in A \cup C$ from n_1 to n_2 with link morphism σ^l the diagram

$$\begin{array}{ccc}
 \Sigma^{n_1} & \xrightarrow{\sigma^l} & \Sigma^{n_2} \\
 \sigma_{n_1} \updownarrow & & \downarrow \sigma_{n_2} \\
 \Sigma^{h(n_1)} & \xrightarrow{\sigma^{h(l)}} & \Sigma^{h(n_2)}
 \end{array} \quad (4.10)$$

commutes in **Sig**.

In (4.10), we have drawn the links between Σ^{n_i} and $\Sigma^{h(n_i)}$ as two-sided arrows to visualise that the arrow may either be up or down, depending on whether the signature of node n_i is extended or restricted. If, e.g., the left arrow points upwards and the right one downwards, the condition reads $\sigma_{n_2} \circ \sigma^{h(l)} \circ \sigma_{n_1} = \sigma^{h(l)}$.

This means that the signature of each node is either extended or restricted. The side condition (4.10) on the new signature morphisms ensures that the mapped axioms are mapped along links in essentially the same way as before. Otherwise, the definition would allow arbitrary mappings along links, and this would preclude us from propagating $Tr_{signatures}$ to proof obligations. Formally, the side condition ensures that h respects the morphisms, so that there is a bijection \bar{h} between proof obligation links and Theorem 4.5 is applicable.

Theorem 4.14 Let $D = (N, A, C)$ and D' be in relation $Tr_{signatures}$ wrt. h and $(\sigma_n)_{n \in N}$. Then h respects the morphisms.³

Proof of 4.14 In Figure 4.4, the four trapeziums (e.g. σ_{n_0} , σ^{l_1} , σ_{n_1} , and

³Cf. Definition 4.4 on page 69.

4.3. Changing the Content of Nodes or Links

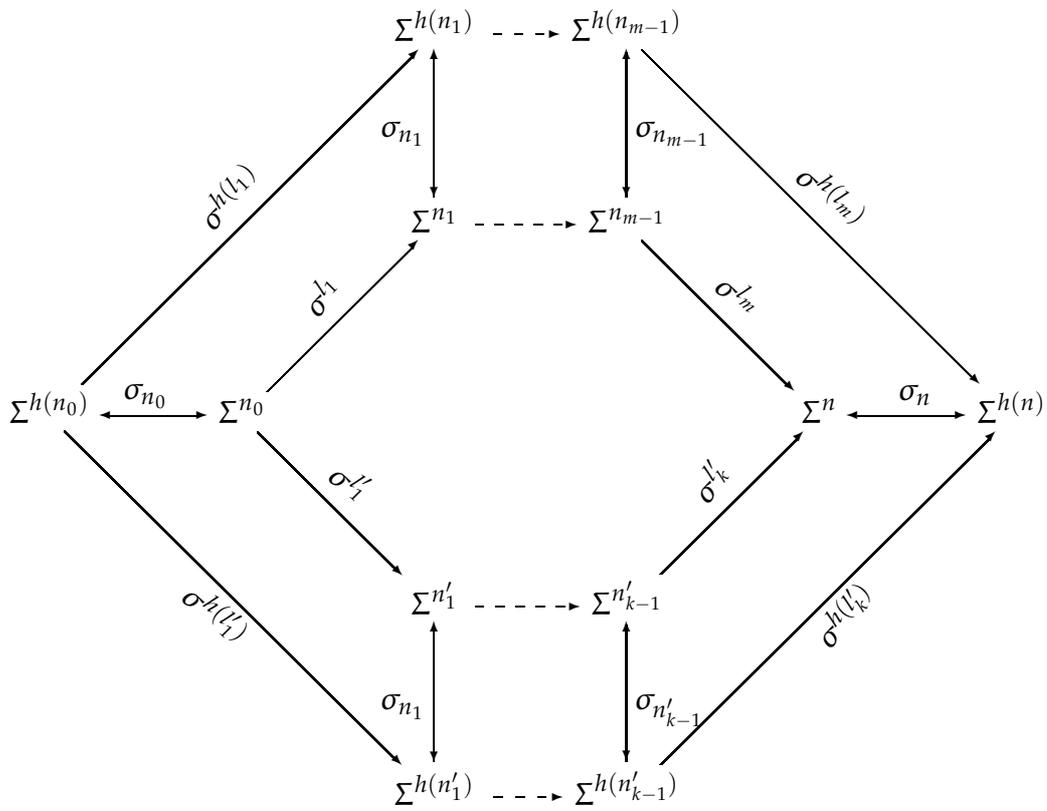


Figure 4.4: Signature adjustment respects the morphisms

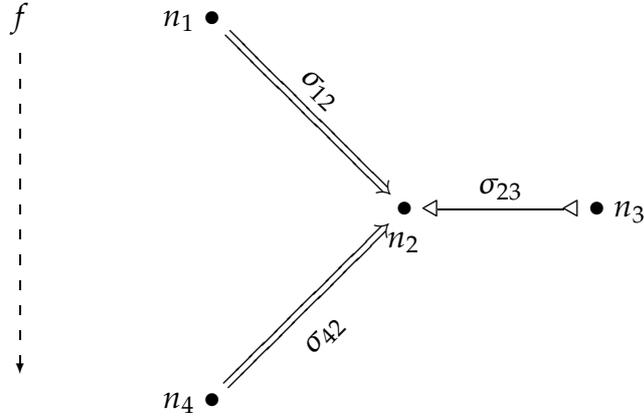


Figure 4.5: Moving signature symbols from one node to another one

$\sigma^{h(l_1)}$) commute because of (4.10). Similarly, the two boxes commute because each of them consists of finitely many boxes that in turn commute due to (4.10). Thus the outermost polygon commutes in **Sig** iff the innermost polygon commutes, as required. \square

The definition of $Tr_{signatures}$ allows us to add signature symbols to a node and propagate them to all other nodes in which they become visible. In this case, the signatures of all nodes are extended, all arrows in (4.10) point downwards. Similarly, it is possible to remove symbols and also remove them from any other nodes in which they are no longer visible, provided that the symbol is not used in any of those nodes either. Here, the signature of each node is restricted, i.e. all arrows point upwards.

Finally it allows us to move signature symbols from one node to another one: assume that we move a symbol f from n_1 to n_4 in the development graph of Figure 4.5. For simplicity, let us further assume that both σ_{12} and σ_{42} are injections. Thus, the signature of n_1 is restricted by removing f , the signature of n_4 is extended by adding f , and the signature of n_2 and n_3 stay the same. The morphism between n_1 and n_2 is changed to be the injection morphism from the restricted signature of n_1 into the signature of n_2 , and similarly for n_4 . It can easily be checked that the original development graph and the resulting one are in relation $Tr_{signatures}$ according to Definition 4.13.

Let us investigate how $Tr_{signatures}$ is propagated to proof obligations: assume that $D = (N, A, C)$ and $D' = (N', A', C')$ are related by $Tr_{signatures}$ wrt. h and $(\sigma_n)_{n \in N}$. First, we show that the signature extensions σ_n relate

4.3. Changing the Content of Nodes or Links

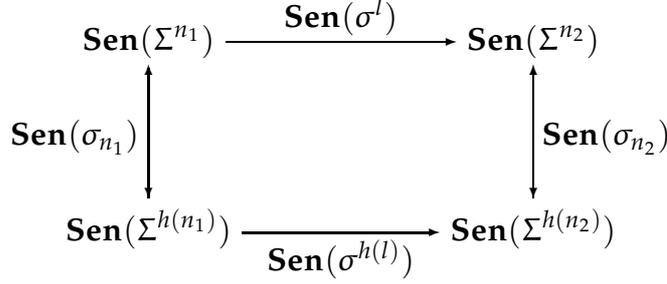


Figure 4.6: Signature adjustment: sentence functor

the sets of global axioms of each pair of h -related nodes n and $h(n)$ in D and D' . Then we will have a look at the proof obligation links.

Lemma 4.15 Let $D = (N, A, C)$ and $D' = (N', A', C')$ be two development graphs in relation $Tr_{signatures}$ wrt. h and $(\sigma_n)_{n \in N}$. Then, for each node $n \in N$ either

$$\Phi_{D'}^{h(n)} = \mathbf{Sen}(\sigma_n)(\Phi_D^n) \quad \text{or} \quad \Phi_D^n = \mathbf{Sen}(\sigma_{h(n)})(\Phi_{D'}^{h(n)}) \quad (4.11)$$

holds.

Proof of 4.15 First, note that since \mathbf{Sen} is a functor and Diagram (4.10) on page 80 commutes for each link $l \in A \cup C$, the diagram in Figure 4.6 also commutes in \mathbf{Set} , where again the double-headed arrows stand for downwards or upwards arrows, depending on whether the respective node is extended or restricted, respectively. We proceed by induction over the depth of nodes in A .⁴

Base case: We have $\#_A(n) = 0 = \#_{A'}(h(n))$, so there are no links in A with target n , and therefore no links in A' with target $h(n)$. According to Definition 3.5 we have thus $\Phi_D^n = \Phi^n$ and $\Phi_{D'}^{h(n)} = \Phi^{h(n)}$. Depending on whether the signature of node n is extended or restricted, assumption (4.8) or (4.9) of Definition 4.13 directly yields (4.11).

Step case: First, we show that for every member $\varphi \in \Phi_D^n$ there is a member $\varphi' \in \Phi_{D'}^{h(n)}$ such that $\varphi' = \mathbf{Sen}(\sigma_n)(\varphi)$ or $\varphi = \mathbf{Sen}(\sigma_{h(n)})(\varphi')$ – then we show the other direction.

⁴For the definition of the depth of a node n wrt. the definitorial links A , written $\#_A(n)$, cf. Definition 3.14 on page 48.

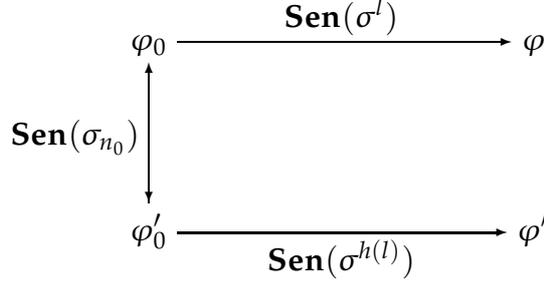


Figure 4.7: Signature adjustment: sketch of inherited axioms

\Rightarrow : According to Definition 3.5, each member $\varphi \in \Phi_D^n$ is either a member of Φ^n or it is inherited via an incoming definitorial link, global or local, and we distinguish these cases:

1. $\varphi \in \Phi^n$: Depending on whether the signature of n is extended or restricted, by (4.8) or (4.9) there is a member $\varphi' \in \Phi^{h(n)}$ such that $\varphi' = \mathbf{Sen}(\sigma_n)(\varphi)$ or $\varphi = \mathbf{Sen}(\sigma_n)(\varphi')$ as required.
2. φ inherited by local link $l : n_0 \xrightarrow{\sigma} n$: There is an axiom $\varphi_0 \in \Phi^{n_0}$ such that $\varphi = \mathbf{Sen}(\sigma)(\varphi_0)$. Because of (4.8) or (4.9), there is an axiom $\varphi'_0 \in \Phi^{h(n_0)}$ such that $\varphi'_0 = \mathbf{Sen}(\sigma_{h(n_0)})(\varphi_0)$ or $\varphi_0 = \mathbf{Sen}(\sigma_{n_0})(\varphi'_0)$. Let $\varphi' = \mathbf{Sen}(\sigma^{h(l)})(\varphi'_0)$. This is visualised in Figure 4.7. Since h is a development graph isomorphism, $h(l)$ is a link from $h(n_0)$ to $h(n)$ and we have $\varphi' \in \Phi^{h(n)}$, because φ'_0 is inherited along $h(l)$. Finally, the diagram in Figure 4.6 commutes, and thus we have $\varphi' = \mathbf{Sen}(\sigma_n)(\varphi)$ or $\varphi = \mathbf{Sen}(\sigma_n)(\varphi')$ as required.
3. φ inherited by global link $l : n_0 \xrightarrow{\sigma} n$: The argument is similar to case 2. The difference is that we have $\varphi_0 \in \Phi_D^{n_0}$, i.e. in the set of global axioms of node n_0 rather than the local axioms. Accordingly, instead of (4.8) or (4.9), we use the induction hypothesis to conclude that $\varphi'_0 \in \Phi_{D'}^{h(n_0)}$ such that $\varphi'_0 = \mathbf{Sen}(\sigma_{h(n_0)})(\varphi_0)$ or $\varphi_0 = \mathbf{Sen}(\sigma_{n_0})(\varphi'_0)$. Then again, the diagram in Figure 4.6 commutes, and we infer $\varphi' = \mathbf{Sen}(\sigma_n)(\varphi)$ or $\varphi = \mathbf{Sen}(\sigma_n)(\varphi')$ as required.

\Leftarrow : Since h is a bijection and the relevant conditions (4.8), (4.9), (4.10) and (4.11) are symmetric, we get the backward direction by the same argument, if only we use the bijection h^{-1} .

□

4.3. Changing the Content of Nodes or Links

We can now say that if two development graphs $D = (N, A, C)$ and $D' = (N', A', C')$ are in relation $Tr_{signatures}$ wrt. h and $(\sigma_n)_{n \in N}$, for each proof obligation link

$$l' : h(n_1) \xrightarrow{\sigma^{l'}} h(n_2) \in C' \downarrow_{N', A'}$$

there is a proof obligation link

$$l : n_1 \xrightarrow{\sigma^l} n_2 \in C \downarrow_{N, A}$$

such that

- $\Phi^{n_1} = \mathbf{Sen}(\sigma_{n_1})(\Phi^{h(n_1)})$ or $\mathbf{Sen}(\sigma_{n_1})(\Phi^{n_1}) = \Phi^{h(n_1)}$, and
- $\Phi_D^{n_2} = \mathbf{Sen}(\sigma_{n_2})(\Phi_D^{h(n_2)})$ or $\mathbf{Sen}(\sigma_{n_2})(\Phi_D^{n_2}) = \Phi_D^{h(n_2)}$.

Here we have used the fact that h respects morphisms as shown in Theorem 4.14. For simplicity, let us assume that both n_1 and n_2 are extended rather than restricted. Thus, we can write each proof obligation

$$\Phi_D^{h(n_2)} \models_{\Sigma^{h(n_2)}} \mathbf{Sen}(\sigma^{l'})(\varphi') \quad \varphi' \in \Phi^{h(n_1)}$$

for D' in the form

$$\mathbf{Sen}(\sigma_{n_2})(\Phi_D^{n_2}) \models_{\sigma_{n_2} \Sigma^{n_2}} \mathbf{Sen}(\sigma^{l'}) (\mathbf{Sen}(\sigma_{n_1})(\varphi)) \quad \varphi \in \Phi^{n_1} \quad (4.12)$$

for some proof obligation

$$\Phi_D^{n_2} \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma^l)(\varphi) \quad \varphi \in \Phi^{n_1} \quad (4.13)$$

for D . The following theorem allows us to replace $\mathbf{Sen}(\sigma^{l'}) \circ \mathbf{Sen}(\sigma_{n_1})$ in (4.12) by $\mathbf{Sen}(\sigma_{n_2}) \circ \mathbf{Sen}(\sigma^l)$.

Theorem 4.16 Let $D = (N, A, C)$ and D' be development graphs in relation $Tr_{signatures}$ wrt. h and $(\sigma_n)_{n \in N}$. Then each link $l \in C \downarrow_{N, A}$ is mapped by \bar{h} to a link $l' = \bar{h}(l)$ such that the diagram

$$\begin{array}{ccc} \mathbf{Sen}(\Sigma^{n_1}) & \xrightarrow{\mathbf{Sen}(\sigma^{l'})} & \mathbf{Sen}(\Sigma^{n_2}) \\ \mathbf{Sen}(\sigma_{n_1}) \updownarrow & & \updownarrow \mathbf{Sen}(\sigma_{n_2}) \\ \mathbf{Sen}(\Sigma^{h(n_1)}) & \xrightarrow{\mathbf{Sen}(\sigma^{\bar{h}(l)})} & \mathbf{Sen}(\Sigma^{h(n_2)}) \end{array}$$

commutes in **Set**.

Proof of 4.16 By Theorem 4.5 and the diagram of Figure 4.6, for each link $l \in C \downarrow_{N,A}$ there is a sequence of links l_1, \dots, l_m and nodes k_1, \dots, k_{m-1} such that

$$\begin{array}{ccccccc}
 \mathbf{Sen}(\Sigma^{n_1}) & \xrightarrow{\mathbf{Sen}(\sigma^{l_1})} & \mathbf{Sen}(\Sigma^{k_1}) & \longrightarrow & \dots & & \\
 \uparrow \mathbf{Sen}(\sigma_{n_1}) & & \uparrow \mathbf{Sen}(\sigma_{k_1}) & & & & \\
 \mathbf{Sen}(\Sigma^{h(n_1)}) & \xrightarrow{\mathbf{Sen}(\sigma^{h(l_1)})} & \mathbf{Sen}(\Sigma^{h(k_1)}) & \longrightarrow & \dots & & \\
 & & \uparrow \mathbf{Sen}(\sigma_{k_{m-1}}) & & & & \\
 \dots & \longrightarrow & \mathbf{Sen}(\Sigma^{k_{m-1}}) & \xrightarrow{\mathbf{Sen}(\sigma^{l_m})} & \mathbf{Sen}(\Sigma^{n_2}) & & \\
 & & \downarrow \mathbf{Sen}(\sigma_{k_{m-1}}) & & \downarrow \mathbf{Sen}(\sigma_{n_2}) & & \\
 \dots & \longrightarrow & \mathbf{Sen}(\Sigma^{h(k_{m-1})}) & \xrightarrow{\mathbf{Sen}(\sigma^{h(l_m)})} & \mathbf{Sen}(\Sigma^{h(n_2)}) & &
 \end{array}$$

commutes in **Set**. The outer rectangle thus commutes also as required. \square

Because of Theorem 4.16, we can write (4.12) as

$$\mathbf{Sen}(\sigma_{n_2})(\Phi_D^{n_2}) \models_{\sigma_{n_2} \Sigma^{n_2}} \mathbf{Sen}(\sigma_{n_2})(\mathbf{Sen}(\sigma^l)(\varphi)) \quad \varphi \in \Phi^{n_1}$$

or

$$\mathbf{Obl}(\sigma_{n_2})(\Phi_D^{n_2}) \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma^l)(\varphi) \quad \varphi \in \Phi^{n_1}. \quad (4.14)$$

A similar argument can be applied to the cases where the signature of one the nodes is extended and the other restricted, and where both signatures are restricted. In case the signature of n_2 is restricted, the old proof obligation (4.13) and the corresponding new one (4.14) read

$$\mathbf{Obl}(\sigma_{h(n_2)})(\Phi_{D'}^{h(n_2)}) \models_{\Sigma^{h(n_2)}} \mathbf{Sen}(\sigma^{\bar{h}(l)})(\varphi') \quad \varphi' \in \Phi^{h(n_1)}$$

and

$$\Phi_{D'}^{h(n_2)} \models_{\Sigma^{h(n_2)}} \mathbf{Sen}(\sigma^{\bar{h}(l)})(\varphi') \quad \varphi' \in \Phi^{h(n_1)}.$$

instead. Note that σ_{n_2} is applied in the other direction.

As the overall result, the development graph transformation $Tr_{signatures}$ can be propagated to proof obligations

- by mapping existing proof obligations along signature extensions and
- by mapping existing proof obligations backwards along signature extension.

4.3.4 Translating Development Graphs

Signature extensions and restrictions relate nodes in different development graphs via signature morphisms. We will weaken this strong requirement in the following and study transformations that relate associated nodes by translations. A *sentence translation* is an arbitrary mapping from sentences over one signature to sentences in another signature; a sentence translation is associated with a *signature translation* that relates the source and target signature.

The following example captures this idea: Let f be a function symbol in a signature Σ and let Σ' be the signature that is like Σ except that f has an additional parameter. Rewriting each Σ -term using the rewriting rule

$$f(T_1, \dots, T_n) \mapsto f(T_1, \dots, T_n, t)$$

for a fixed Σ' -term t exhaustively is a sentence translation.

Remark 4.17 Note that signature and sentence *translations* are mappings similar to signature and sentence *morphisms*. Like morphisms, they can be used to formulate development graph transformations that do not change the structure of the development graph. These are called *development graph translations* in the following.

Many concrete development graph transformations can be formulated by *development graph translations*. A development graph translation applies a sentence translation to each node of the development graph such that the independent translations of the nodes are consistent over the whole graph. This requires that the signature morphisms on links can be adjusted appropriately: for instance, if a development graph contains the link $n_1 \xrightarrow{\sigma} n_2$ and f is a symbol in Σ^{n_1} to which we add an additional required argument of type $\tau \in \Sigma^{n_1}$, then we need to add an additional argument of type $\sigma\tau$ to σf in Σ^{n_2} .

Before we can define development graph translations, we need to define signature and sentence translations and clarify their relationship with the underlying institution and proof representation. The idea is simple: We extend **Sig** to **XSig** by adding all signature translations as additional arrows. We also extend the functor **Sen** to **XSen** such that **XSen** has **XSig** as its domain. This necessitates explicit additional definitions for the cases that are not covered by the original functor **Sen**. Since the definition of categories and functors requires that arrows are closed under concatenation, we need to ensure that the category **XSig** and all the functors are well-defined, i.e. there are enough morphisms in the respective categories. This

allows us to present concrete signature translations together with the respective definitions for the functors in the reference instantiation one after another, independently of each other.

Definition 4.18 (Signature and sentence translations) *An extended category of signatures is a category \mathbf{XSig} such that $|\mathbf{XSig}| = |\mathbf{Sig}|$ and each \mathbf{Sig} -arrow $\sigma : \Sigma \rightarrow \Sigma'$ is also an \mathbf{XSig} -arrow. An extended sentence functor for \mathbf{XSig} is a functor $\mathbf{XSen} : \mathbf{XSig} \rightarrow \mathbf{Set}$ such that \mathbf{Sen} and \mathbf{XSen} agree on objects and on \mathbf{Sig} -arrows, i.e.*

- $\mathbf{XSen}(\Sigma) = \mathbf{Sen}(\Sigma)$ for any $\Sigma \in |\mathbf{Sig}| = |\mathbf{XSig}|$ and
- $\mathbf{XSen}(\sigma) = \mathbf{Sen}(\sigma)$ for any \mathbf{Sig} -arrow σ .

An \mathbf{XSig} -arrow $\vartheta : \Sigma \rightarrow \Sigma'$ is called a signature translation from Σ to Σ' . Similarly, if $\vartheta : \Sigma \rightarrow \Sigma'$ is a signature translation then $\mathbf{XSen}(\vartheta) : \mathbf{Sen}(\Sigma) \rightarrow \mathbf{Sen}(\Sigma')$ is called a sentence translation.

Recall that signature morphisms preserve satisfaction, cf. Definition 3.1 on page 40. Signature translations, on the other hand, do not have to satisfy the satisfaction condition (3.1). The exact statement of the satisfaction condition relies on the standard notion of σ -reducts. No standard notion of mapping models backwards over signature translations exists, however. Thus, we do not need to define the equivalent of a σ -reduct for translations, and consequently, no extension of the functor \mathbf{Mod} to \mathbf{XSig} is required.

We will continue to use σ for \mathbf{Sig} -morphisms (and \mathbf{XSig} -arrows that are \mathbf{Sig} -morphisms, i.e. satisfy the satisfaction condition) and we will use ϑ to range over signature translations that may or may not obey the satisfaction condition.

We can now define the notion of a *development graph translation* from one development graph to another one: it consists of a bijection between the two development graphs, a translation from each node to its image under the bijection, and of link morphisms for the new links. The formal definition is as follows.

Definition 4.19 (DG transformations: $Tr_{\text{translate}}$) *Let $D = (N, A, C)$ and $D' = (N', A', C')$ be development graphs. A development graph translation $(h, (\vartheta_n)_{n \in N})$ from D to D' consists of*

- a development graph bijection h from D to D'
- a signature translation $\vartheta_n : \Sigma^n \rightarrow \Sigma'^{h(n)}$ for each node $n \in N$

4.3. Changing the Content of Nodes or Links

such that for each node $n \in N$ we have

$$\Phi^{h(n)} = \mathbf{XSen}(\vartheta_n)(\Phi^n) \quad (4.15)$$

and for each link $link\ l : n_1 \xrightarrow{\sigma^l} n_2$ from $A \cup C$, the diagram

$$\begin{array}{ccc} \Sigma^{n_1} & \xrightarrow{\sigma^l} & \Sigma^{n_2} \\ \vartheta_{n_1} \downarrow & & \downarrow \vartheta_{n_2} \\ \Sigma^{h(n_1)} & \xrightarrow{\sigma^{h(l)}} & \Sigma^{h(n_2)} \end{array} \quad (4.16)$$

commutes in \mathbf{XSig} . Two development graphs D and D' are in relation $Tr_{translate}$ iff there is a development graph translation from D to D' .

Theorem 4.20 Let $(h, (\vartheta_n)_{n \in N})$ be a development graph translation between $D = (N, A, C)$ and D' . Then h respects the morphisms.⁵

Proof of 4.20 The proof is similar to the one for Theorem 4.14 on page 80, except that instead of signature adjustments σ_{n_i} we now have sentence translations ϑ_{n_i} between the nodes n_i and $h(n_i)$. The diagrams commute in \mathbf{XSig} instead of in \mathbf{Sig} . \square

The signature translations ϑ_n consistently map the local axioms of the nodes, cf. (4.15), and also the global sets of axioms, as the following lemma shows:

Lemma 4.21 Let $(h, (\vartheta_n)_{n \in N})$ be a development graph translation between $D = (N, A, C)$ and D' . Then for each node $n \in N$,

$$\Phi_{A'}^{h(n)} = \mathbf{XSen}(\vartheta_n)(\Phi_A^n),$$

i.e. the global set of axioms is mapped along ϑ_n .

The proof uses the following lemma.

⁵Cf. Definition 4.4 on page 69.

Lemma 4.22 Let $(h, (\vartheta_n)_{n \in N})$ be a development graph translation between $D = (N, A, C)$ and $D' = (N', A', C')$. Then for any link $l : n_1 \xrightarrow{\sigma^l} n_2 \in A \cup C$, the diagram

$$\begin{array}{ccc}
 \mathbf{Sen}(\Sigma^{n_1}) & \xrightarrow{\mathbf{Sen}(\sigma^l)} & \mathbf{Sen}(\Sigma^{n_2}) \\
 \mathbf{XSen}(\vartheta_{n_1}) \downarrow & & \downarrow \mathbf{XSen}(\vartheta_{n_2}) \\
 \mathbf{Sen}(\Sigma^{h(n_1)}) & \xrightarrow{\mathbf{Sen}(\sigma^{h(l)})} & \mathbf{Sen}(\Sigma^{h(n_2)})
 \end{array} \quad (4.17)$$

commutes in **Set**.

Proof of 4.22 Since **XSen** is a functor and diagram (4.16) commutes, diagram (4.17) with **Sen** replaced by **XSen** also commutes. And since **XSen** and **Sen** agree on signatures and signature morphisms, diagram (4.17) commutes also. \square

Proof of 4.21 We proceed by induction on the depths of nodes in N .

Base case: Let $\#_A(n) = 0$, i.e. there are no links in A with target n . Thus there are no links in $A' = h(A)$ with target $h(n)$, either. In this case, $\Phi_{A'}^{h(n)}$ degenerates to $\Phi^{h(n)}$ and similarly $\Phi_A^n = \Phi^n$, and thus

$$\Phi_{A'}^{h(n)} = \Phi^{h(n)} \stackrel{(4.15)}{=} \mathbf{XSen}(\vartheta_n)(\Phi^n) = \mathbf{XSen}(\vartheta_n)(\Phi_A^n).$$

Step case: We first show that for each $\varphi \in \Phi_D^n$, $\mathbf{XSen}(\vartheta_n)(\varphi) \in \Phi_{D'}^{h(n)}$, and then the other direction: for each member $\varphi' \in \Phi_{D'}^{h(n)}$, there is a member $\varphi \in \Phi_D^n$ such that $\varphi' = \mathbf{XSen}(\vartheta_n)(\varphi)$.

\Rightarrow : Each member φ of Φ_D^n is either a member of Φ^n , or is inherited via a local or global link with target n .

1. $\varphi \in \Phi^n$: By (4.15), $\mathbf{XSen}(\vartheta_n)(\varphi) \in \Phi^{h(n)}$, and $\Phi^{h(n)} \subseteq \Phi_{D'}^{h(n)}$ as required.
2. φ is inherited by a local link $l : n_0 \xrightarrow{\sigma} n$: There is a local axiom $\varphi_0 \in \Phi^{n_0}$ such that $\varphi = \mathbf{Sen}(\sigma^l)(\varphi_0)$. Because of (4.15), there is an axiom $\varphi'_0 \in \Phi^{h(n_0)}$ such that $\varphi'_0 = \mathbf{XSen}(\vartheta_{n_0})(\varphi_0)$. Since h is a development graph bijection, there is a link $h(n_0) \xrightarrow{\sigma^{h(l)}} h(n)$ in

4.3. Changing the Content of Nodes or Links

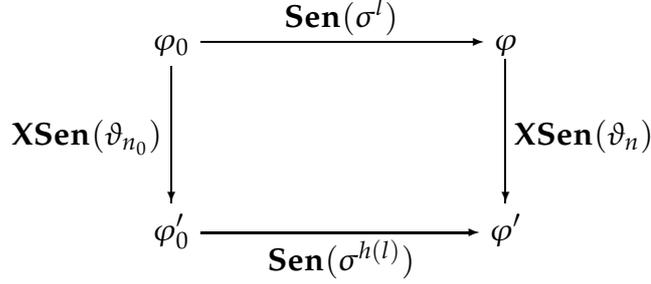


Figure 4.8: Translation of global axioms

D' , and thus $\varphi' = \mathbf{Sen}(\sigma^{h(l)})(\varphi'_0)$ is in $\Phi_{D'}^{h(n)}$. Additionally, h is a development graph translation, so (4.17) is applicable, and we have $\varphi' = \mathbf{Sen}(\sigma^{h(l)})(\varphi'_0) = \mathbf{XSen}(\vartheta_n)(\varphi)$, as required. This is visualised in the diagram in Figure 4.8.

3. φ is inherited by a global link $l : n_0 \xrightarrow{\sigma} n$: There is an axiom $\varphi_0 \in \Phi_{D'}^{n_0}$ such that $\varphi = \mathbf{Sen}(\sigma^l)(\varphi_0)$. By appeal to the induction hypothesis, there is an axiom $\varphi'_0 \in \Phi_{D'}^{h(n_0)}$ such that $\varphi'_0 = \mathbf{XSen}(\vartheta_{n_0})(\varphi_0)$. Again, since h is a development graph bijection, there is a link $h(n_0) \xrightarrow{\sigma^{h(l)}} h(n)$ in D' , and thus $\varphi' = \mathbf{Sen}(\sigma^{h(l)})(\varphi'_0)$ is in $\Phi_{D'}^{h(n)}$, and by (4.17) again we have $\varphi' = \mathbf{Sen}(\sigma^{h(l)})(\varphi'_0) = \mathbf{XSen}(\vartheta_n)(\varphi)$, as required. This is visualised in Figure 4.8.

\Leftarrow : Each member of $\Phi_{D'}^{h(n)}$ is either a local axiom of $h(n)$ or is inherited along a local or global link.

1. $\varphi' \in \Phi_{D'}^{h(n)}$: Again, (4.15) directly yields the required result.
2. φ' inherited via a local link: Similarly to the argument for the other direction, in the node $h(n_0)$ there is a local axiom φ'_0 such that $\varphi' = \mathbf{Sen}(\sigma^{h(l)})(\varphi'_0)$. According to (4.15) there is a local axiom φ_0 in n_0 such that $\varphi'_0 = \mathbf{XSen}(\vartheta_{n_0})(\varphi_0)$, and (4.17) ensures $\mathbf{XSen}(\vartheta_n)(\mathbf{Sen}(\sigma^l)(\varphi_0)) = \varphi'$.
3. φ' inherited via a global link: We refer to the induction hypothesis instead of to (4.15) to show that there is a suitable φ_0 .

□

Similarly to sentences, we define extended goal and proof functors \mathbf{XGoal} and \mathbf{XPrf} . The idea is that we can meaningfully talk about map-

$$\begin{array}{ccc}
 \mathbf{XObl}(\Sigma) & \xrightarrow{\mathbf{XObl}(\vartheta)} & \mathbf{XObl}(\Sigma') \\
 \downarrow \eta_\Sigma & & \downarrow \eta_{\Sigma'} \\
 \mathbf{XGoal}(\Sigma) & \xrightarrow{\mathbf{XGoal}(\vartheta)} & \mathbf{XGoal}(\Sigma') \\
 \uparrow \text{concl}_\Sigma & & \uparrow \text{concl}_{\Sigma'} \\
 \mathbf{XPrf}(\Sigma) & \xrightarrow{\mathbf{XPrf}(\vartheta)} & \mathbf{XPrf}(\Sigma')
 \end{array}$$

Figure 4.9: Extended proof representation conditions

ping a goal along a translation: the mapping should respect the mapping of sentences, and similarly for proofs.

Definition 4.23 (Extended proof representation) *Let a proof representation $(\mathbf{Goal}, \eta, \mathbf{Prf}, \text{concl})$ for the institution $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ be given, and let \mathbf{XSig} and \mathbf{XSen} be an extended category of signatures for \mathbf{Sig} and an extended sentence functor for \mathbf{XSig} and \mathbf{Sen} , respectively. $(\mathbf{XGoal}, \eta, \mathbf{XPrf}, \text{concl})$ is an extended proof representation for the institution, \mathbf{XSig} , and \mathbf{XSen} iff the following conditions are all met.*

- $\mathbf{XGoal} : \mathbf{XSig} \rightarrow \mathbf{Set}$ and $\mathbf{XPrf} : \mathbf{XSig} \rightarrow \mathbf{Set}$ are functors that agree with the functors \mathbf{Goal} and \mathbf{Prf} on objects and \mathbf{Goal} - and \mathbf{Prf} -arrows, respectively:
 - $\mathbf{XGoal}(\Sigma) = \mathbf{Goal}(\Sigma)$ for any $\Sigma \in |\mathbf{Sig}| = |\mathbf{XSig}|$ and
 - $\mathbf{XGoal}(\sigma) = \mathbf{Goal}(\sigma)$ for any \mathbf{Sig} -arrow σ .
 - $\mathbf{XPrf}(\Sigma) = \mathbf{Prf}(\Sigma)$ for any $\Sigma \in |\mathbf{Sig}| = |\mathbf{XSig}|$ and
 - $\mathbf{XPrf}(\sigma) = \mathbf{Prf}(\sigma)$ for any \mathbf{Sig} -arrow σ .
- η and concl are also natural transformations for the the extended obligation and proof functors, i.e. the diagram in Figure 4.9 commutes in \mathbf{Set} , where $\mathbf{XObl} = 2^{\mathbf{XSen}} \times \mathbf{XSen}$.

For a signature morphism σ we expect $\mathbf{Prf}(\sigma)$ to map proofs rather directly, preserving their structure and gaps. For an arbitrary translation ϑ

4.3. Changing the Content of Nodes or Links

this expectation is not warranted in general. Of course, the idea is to introduce concrete translations that map proofs sensibly. The framework, however, abstracts from questions about how well translations map to proofs. Similarly to our treatment of signature adjustments, we now study how a development graph translation propagates to proof obligations. Let h and $(\vartheta_n)_{n \in N}$ be a development graph translation from D to D' . Then, by appeal to Theorem 4.5 we can say that for each proof obligation link $l' : h(n_1) \xrightarrow{\sigma^{l'}} h(n_2)$ for D' there is a proof obligation link $l : n_1 \xrightarrow{\sigma^l} n_2$ for D such that

$$\begin{aligned} \Phi^{h(n_1)} &= \mathbf{XSen}(\vartheta_{n_1})(\Phi^{n_1}) && \text{because of (4.15), and} \\ \Phi_{D'}^{h(n_2)} &= \mathbf{XSen}(\vartheta_{n_2})(\Phi_D^{n_2}) && \text{because of Lemma 4.21.} \end{aligned}$$

We can thus write each proof obligation

$$\Phi_{D'}^{h(n_2)} \models_{\Sigma^{h(n_2)}} \mathbf{Sen}(\sigma^{l'}) (\varphi') \quad (\varphi' \in \Phi^{h(n_1)})$$

for D' in the form

$$\mathbf{XSen}(\vartheta_{n_2})(\Phi_D^{n_2}) \models_{\vartheta_{n_2} \Sigma^{n_2}} \mathbf{Sen}(\sigma^{l'}) (\mathbf{XSen}(\vartheta_{n_1})(\varphi)) \quad (\varphi \in \Phi^{n_1}) \quad (4.18)$$

for some proof obligation

$$\Phi_D^{n_2} \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma^l) (\varphi) \quad (\varphi \in \Phi^{n_1})$$

for D . The following theorem allows us to replace $\mathbf{Sen}(\sigma^{l'}) \circ \mathbf{XSen}(\vartheta_{n_1})$ by $\mathbf{XSen}(\vartheta_{n_2}) \circ \mathbf{Sen}(\sigma^l)$ in (4.18).

Theorem 4.24 Let $D = (N, A, C)$ and D' be development graphs such that $Tr_{translate}(D, D')$, i.e. there is a development graph translation $h, (\vartheta_n)_{n \in N}$. Each link $l \in C \downarrow_{N,A}$ is mapped by the induced bijection \bar{h} on proof obligation links to a link $l' = \bar{h}(l)$ such that

$$\begin{array}{ccc} \mathbf{Sen}(\Sigma^{n_1}) & \xrightarrow{\mathbf{Sen}(\sigma^{l'})} & \mathbf{Sen}(\Sigma^{n_2}) \\ \mathbf{XSen}(\vartheta_{n_1}) \downarrow & & \downarrow \mathbf{XSen}(\vartheta_{n_2}) \\ \mathbf{Sen}(\Sigma^{h(n_1)}) & \xrightarrow{\mathbf{Sen}(\sigma^{\bar{h}(l)})} & \mathbf{Sen}(\Sigma^{h(n_2)}) \end{array}$$

commutes in **Set**.

Proof of 4.24 The proof is analogous to the proof of Theorem 4.16 on page 85, except that we appeal to diagram (4.17) instead of the diagram of Figure 4.6. \square

We can now write (4.18) as

$$\mathbf{XSen}(\vartheta_{n_2})(\Phi_D^{n_2}) \models_{\vartheta_{n_2}\Sigma^{n_2}} \mathbf{XSen}(\vartheta_{n_2})(\mathbf{Sen}(\sigma^l)(\varphi)) \quad (\varphi \in \Phi^{n_1})$$

or

$$\mathbf{XObI}(\vartheta_{n_2}) \left(\Phi_D^{n_2} \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma^l)(\varphi) \right) \quad (\varphi \in \Phi^{n_1}).$$

Thus, each development graph transformation from the class $Tr_{translate}$ is propagated to proof obligations

- by mapping proof obligations along signature translations.

4.4 Generic Construction of Translations

Most classes of development graph transformations are fixed on the level of development graphs: given an institution, the definitions are complete. Development graph translations are different in that they refer to sentence translations. Interesting translations need to be defined *in addition to the institution*.

We expect the process of defining interesting translations to be open-ended. Therefore, it is not appropriate to require \mathbf{XSig} and the extended sentence functor and extended proof representation to be defined once and for all in advance. Rather we would like to add specific translations when they are needed without being concerned about other existing translations. We will briefly explain why this is indeed easily possible.

Given an extended category of signatures \mathbf{XSig}_1 for \mathbf{Sig} , and extended functors \mathbf{XSen}_1 , \mathbf{XGoal}_1 and \mathbf{XPrf}_1 , such that $(\mathbf{XGoal}_1, \eta, \mathbf{XPrf}_1, concl)$ is an extended proof representation for the institution, we define another extended category of signatures \mathbf{XSig}_2 with appropriate extended functors \mathbf{XSen}_2 , \mathbf{XGoal}_2 , \mathbf{XPrf}_2 by the following construction: Let \mathbf{XSig}_2 have all arrows of \mathbf{XSig}_1 plus a set Θ of explicitly given new ones, disjoint from the collection of \mathbf{XSig}_1 -arrows. (For formal reasons, this entails further arrows; we address this aspect below.) First, for each $\vartheta \in \Theta$ we explicitly define the values of the new extended functors at those points ϑ , i.e. we provide definitions for $\mathbf{XSen}_2(\vartheta)$, $\mathbf{XGoal}_2(\vartheta)$, and $\mathbf{XPrf}_2(\vartheta)$, such that the translation ϑ makes the diagram in Figure 4.9 on page 92 commute. As we

4.4. Generic Construction of Translations

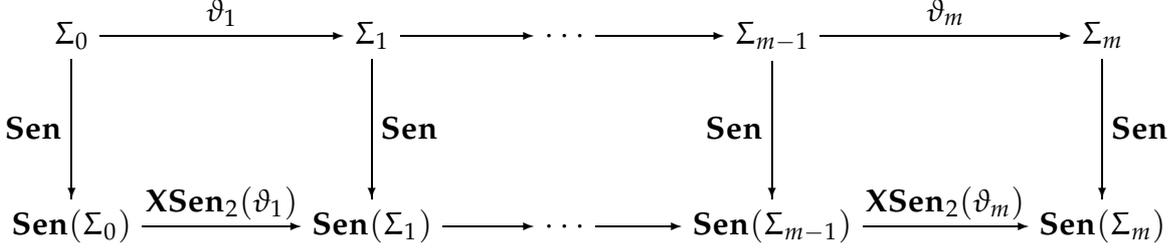


Figure 4.10: Definition of translations

will see, this is enough to define the functors \mathbf{XSen}_2 , \mathbf{XGoal}_2 , and \mathbf{XPrf}_2 completely.

The definition of categories (Definition A.1) requires the arrows to be closed under concatenation. In particular, the collection of arrows of \mathbf{XSig}_2 needs to be closed. Thus, we define \mathbf{XSig}_2 to be the smallest category that includes all arrows from \mathbf{XSig}_1 and Θ and is closed under composition. By construction, each \mathbf{XSig}_2 -arrow ϑ can be represented in the form $\vartheta_m \circ \dots \circ \vartheta_1$, where each ϑ_i is either an \mathbf{XSig}_1 -arrow or a member of Θ . The representation is not necessarily unique, though.

Nevertheless, this provides us with a way to define the value of the extended functors \mathbf{XSen}_2 , \mathbf{XGoal}_2 and \mathbf{XPrf}_2 for an arbitrary \mathbf{XSig}_2 -arrow $\vartheta = \vartheta_m \circ \dots \circ \vartheta_1$. We restrict our attention to the extended sentence functor; the other extended functors are defined similarly. In the diagram given in Figure 4.10 we can assume each ϑ_i to be either an \mathbf{XSig}_1 -arrow or a member of Θ . In the latter case, we have explicitly defined the value of \mathbf{XSen}_2 at the point ϑ . In the former case, $\mathbf{XSen}_2(\vartheta) = \mathbf{XSen}_1(\vartheta)$. Since \mathbf{XSen}_2 is a functor, \mathbf{XSen}_2 is defined at ϑ by

$$\mathbf{XSig}_2(\vartheta) = \mathbf{XSig}_2(\vartheta_m) \circ \dots \circ \mathbf{XSig}_2(\vartheta_1).$$

Of course, \mathbf{XSen}_2 is only well-defined if this fixes a *unique* value for the set $\mathbf{XSen}_2(\vartheta)$. The value is in fact uniquely determined: any different representation of ϑ , i.e. $\vartheta = \vartheta'_k \circ \dots \circ \vartheta'_1$ makes the diagram of Figure 4.11 commute in \mathbf{XSig}_2 . As a consequence, replacing the signatures and signature translations with their value under \mathbf{XSig}_2 also produces a commuting diagram, and thus the value assigned to $\mathbf{XSig}_2(\vartheta)$ this way is unique. By a similar argument, this also defines \mathbf{XGoal}_2 and \mathbf{XPrf}_2 .

It remains to be shown that η and $concl$ are natural transformations for the new functors, i.e. that the diagram in Figure 4.9 commutes for all \mathbf{XSig}_2 -arrows ϑ , \mathbf{XSen}_2 , \mathbf{XGoal}_2 , and \mathbf{XPrf}_2 . Again, this is guaranteed by

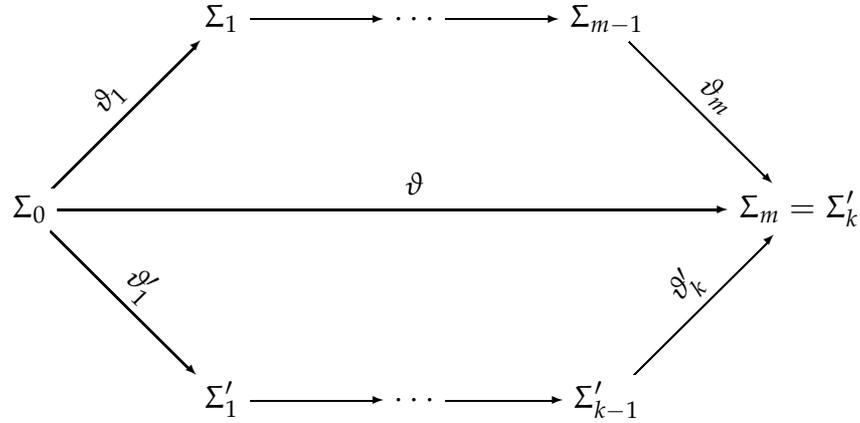


Figure 4.11: Uniqueness of definition of translations

the fact that each ϑ is representable as the concatenation of arrows either from \mathbf{XSig}_1 or Θ .

4.5 Relation to Basic DG-Operations

[Hut00] and [AHMS00] define basic operations on development graphs and describe the effects these operations have on proofs. The difference of two development graphs is expressed by a sequence of basic operations that, when executed on the first graph, produce the second. The effects of these operations on the proof obligation links are then briefly discussed.

Defined operations add or delete links, or add and delete local axioms. These operations are similar to our transformations Tr_{links} and Tr_{axioms} . One difference is that we do not address the question of how the decomposition of theorem links can be avoided by reusing an old decomposition trace; our assumption is that decomposing theorem links is rather cheap in terms of resources. On the other hand, we are interested in two things that are not addressed by the prior work:

1. When an additional assumption becomes visible for an existing open proof, we want to be able to use it in the further construction of this proof: after all the proof is partial and still has open goals, which might depend on the assumption to become available. The prior approach simply keeps the proof but does not address in any way how to make use of additional assumptions.

2. When an additional assumption, which is used in a proof, is withdrawn, the proof can still be kept: the deleted axiom becomes an open goal of the proof. This aspect is not addressed in the basic operations, rather such a proof would be considered invalidated.

There are also technical differences that are due to the fact that [Hut00] defines the signature of a node by a smallest set construction in terms of local signature symbols associated with nodes that are inherited along definitorial links. Thus, adding a link potentially changes the signatures of existing nodes and the morphisms of existing links. This is avoided in our formulation by explicitly giving the signature of a node and requiring link morphisms to be compatible with these signatures.

Another operation in [Hut00] is the replacement of a link morphisms by another. The description of how the old decomposition trace can be reused for a changed one suggests that the same effect can be achieved by adding a new link with the new morphism and removing the link with the old one. Again, we do not think that the difference in efficiency is relevant.

In part, our transformation $Tr_{signatures}$ corresponds to implicit assumptions in the prior work. When a signature symbol is added to a node and this is a monotonic change, the understanding is that the proof is essentially still valid. The question of whether the same proof or a very similar proof is still valid is simply ignored. Additionally, if the proof is a partial proof, it is not quite clear in which way the new signature symbols may be used to close the open goals of the proof. This is all made explicit in our approach. The construction also extends cleanly to the case where the change is not monotonic, so that we can move signature symbols from one node to another one.

Other transformations, namely Tr_{occ} and $Tr_{translate}$ are entirely novel. Tr_{occ} is more fine-grained than adding and removing axioms: we can make use of the knowledge about the changes. Finally, $Tr_{translate}$ changes signatures and axioms together. This has not been addressed at all before.

In fact, we can express the basic operations as development graph transformations. It is thus possible, to integrate both into one system, as indeed our idea is to implement these transformations as an extension to the MAYA/INKA system.

4.6 Summary

We have introduced classes of development graph transformations. These transformations refer to an abstract formulation of the context: specification language, logic, and proof representation. Different instantiations

Chapter 4. Development Graph Transformations

are possible by instantiating the context. We call the abstract formulation of these transformations a framework for this reason. The distinction between framework and instantiation also gives us the opportunity to discuss the idea behind the classes without being lost in the details of a concrete instantiation.

The different classes are summarised in Figure 4.12 together with the effects that a class of development class transformation can possibly have on proof obligations. An instantiation of the framework should ensure that it can support the kinds of effects, i.e. that it can transform proofs accordingly.

Transformation	Def.	Effect: on development graph on proof obligations
Tr_{nodes}	4.1	add/remove isolated nodes (none)
Tr_{links}	4.2	add/remove links add/remove obligation add/remove assumptions
Tr_{axioms}	4.6	add/remove local axioms add/remove obligation add/remove assumptions
Tr_{occ}	4.8	change subformula/-term occurrences apply sentence replacements to assumptions and to conjecture
$Tr_{signatures}$	4.13	adjust signatures map obligation along signature extension map obligation backwards over signature extension
$Tr_{translate}$	4.19	translate development graph translate obligation

Figure 4.12: Development graph transformations

Part III

A Reference Instantiation

Chapter 5

Formal Developments

5.1 Overview

In the preceding chapters, we have described the overall idea of development transformations within a generic framework. The framework is generic with respect to the concrete logic, the concrete specification language, the concrete proof representation, and the concrete translations that are defined for both the logic and the proofs. In this chapter, we will describe a reference instantiation of the given framework for the axiomatic specification of abstract datatypes. This instantiation will then be used to revisit the example scenarios that we have presented as the motivation for our work in Section 2.2.

In the following, we will omit unnecessary formal details wherever possible. In particular, we will not go into details of how first order logic with equality and generatedness-constraints and its proof theory is defined precisely. For the purposes of reference, the definitions we have actually used are given in Appendix C.

5.2 Concrete Logic

In our examples, we use first order logic with equality and generatedness constraints to specify abstract datatypes. First, let us briefly sketch a standard definition of first order logic. Given an algebraic first order signature Σ determining available sorts, functions, and predicates with their profiles (or arities), typed terms are constructed as usual using typed variables and function application. Formulae are constructed using \top and \perp for truth and falsity, predicate applications, equations between terms of the same sort using the symbol $=$, the usual connectives, e.g. \wedge , \vee , \Rightarrow , and \Leftrightarrow , and

Chapter 5. Formal Developments

universal and existential quantification \forall and \exists . For ease of presentation, we assume that sorts, functions, predicates, and variables are pairwise disjoint, and that the names of variables do not matter (implicit α -renaming is possible wherever it is necessary). In practice, this can be realised in many ways, e.g. by using the same set of names in the surface syntax for sorts, functions, and predicates, and disambiguating the names at the few places where there is the possibility of confusion and using higher-order abstract syntax [PE88] or de Bruijn-indices [dB72, dB78] to represent bound variables.

We employ the usual model-theoretic semantics for terms and first order formulae: a Σ -algebra M maps each sort s in Σ to a non-empty set $M(s)$ (the carrier), each function $f : s_1 \times \dots \times s_m \rightarrow s$ in Σ to a total function $M(f)$ from the cartesian product of $M(s_1) \times \dots \times M(s_m)$ to $M(s)$, and each predicate $p : s_1 \times \dots \times s_m \in \Sigma$ to a subset $M(p)$ of $M(s_1) \times \dots \times M(s_m)$ (the extension of the predicate). Given a variable assignment V for M such that for each variable x of type s , $V(x) \in M(s)$ holds, M and V are extended homomorphically to terms and formulae in the usual way: M and V assign carriers to terms and truth values to formulae. We write $M \models_{\Sigma} \varphi$ (read ' φ is satisfied in the algebra M ') iff the closed formula φ is assigned the truth value true. Given a universe of algebras, a signature Σ and a set of closed formulae (i.e. axioms) determine the class of Σ -algebras taken from the chosen universe such that all formulae hold true in each algebra of the class.

First order algebraic signature morphisms from Σ_1 to Σ_2 are defined as usual by symbol mappings that map sort symbols of Σ_1 to sort symbols of Σ_2 (and similarly for function and predicate symbols) and respect the profiles of the symbols (cf. [EM85, Definition 8.1] or [LEW96, Definition 4.1]). As usual, for each signature morphism σ from Σ_1 to Σ_2 , the homomorphic extension of σ to terms maps closed Σ_1 -terms (closed Σ_1 -formulae) to closed Σ_2 -terms (closed Σ_2 -formulae, respectively), and this mapping is associated with σ . It is also standard to extend σ to variables by some injective mapping from variables for Σ_1 to variables for Σ_2 so that σ can also be regarded as a mapping from Σ_1 -terms and -formulae with free variables to Σ_2 -terms and -formulae with free variables. Algebraic signature morphisms can be concatenated by the usual function composition.

We use generatedness constraints to exclude unwanted models and to allow inductive proofs. First, we introduce the standard notion of a generatedness constraint, which we call *simple* generatedness constraint. The notion is suitable for expressing the generatedness of an algebra in a sort by given constructor functions. Satisfaction of simple generatedness constraints is not preserved when they are mapped along signature mor-

phism, however. Therefore, in a second step we will generalise the concept and introduce generatedness constraints that map along morphisms as desired.

Given a first order signature Σ and a sort s in Σ , a simple generatedness constraint for Σ in s is a non-empty set F of Σ -functions with codomain s . A constraint F is satisfied for an algebra M iff the algebra is generated in s by F , i.e. if for each carrier a in $M(s)$ there is a Σ -term t that only uses functions in F and variables of types different from s such that for some variable assignment V , t evaluates to a wrt. M and V , c.f. [LEW96, Definition 3.35]. Since each carrier in $M(s)$ is expressed by a term that only uses constructor functions and variables of sorts different from s , one can then use induction on the structure of the terms.

A simple example is the signature Σ_{List} containing the two sorts `elem` and `list`, and the two constructor functions `nil` : `list` and `cons` : `elem` \times `list` \rightarrow `list`. The generatedness constraint $\{\text{nil}, \text{cons}\}$ ensures that only finite lists are allowed as carriers in $M(\text{list})$, and that the usual structural induction over the length of lists is sound.

As the following example shows, satisfaction as given by (3.1) on page 40 is not preserved in general when simple generatedness constraints are mapped along signature morphisms. Let Σ_1 be the signature containing the sorts `elem1` and `pair1`, and the constructor function `mkpair1` : `elem1` \times `elem1` \rightarrow `pair1`. The Σ_1 -constraint $F_1 = \{\text{mkpair}_1\}$ is satisfiable. Additionally, let us consider the signature Σ_2 containing only the sort `pair2` and the constructor function `mkpair2` : `pair2` \times `pair2` \rightarrow `pair2`. The Σ_2 -constraint $F_2 = \{\text{mkpair}_2\}$ is not satisfiable: domains are required to be non-empty, but there is no finite constructor term without `pair2`-variables to represent any carrier in $A_2(\text{pair}_2)$, where A_2 is a Σ_2 -algebra. Let the signature morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$ map both sorts `elem1` and `pair1` to `pair2`, and the constructor `mkpair1` to `mkpair2`. We consider the Σ_2 -algebra A_2

$$\begin{aligned} A_2(\text{pair}_2) &= \{e\} \\ A_2(\text{mkpair}_2)(e, e) &= e \end{aligned}$$

and its σ -reduct $A_1 = A_2|_\sigma$

$$\begin{aligned} A_1(\text{elem}_1) &= \{e\} \\ A_1(\text{pair}_1) &= \{e\} \\ A_1(\text{mkpair}_1)(e, e) &= e. \end{aligned}$$

Note that A_1 trivially satisfies the constraint F_1 but A_2 does not satisfy $F_2 = \sigma(F_1)$. Thus, the satisfaction condition does not hold for simple generatedness constraints.

Chapter 5. Formal Developments

The following generalised notion of generatedness constraint obeys the satisfaction condition: given a signature Σ and a sort s in Σ , a generatedness constraint (F, σ) for Σ in s consists of a set F of functions and a signature morphism $\sigma : \Sigma' \rightarrow \Sigma$ for some arbitrary signature Σ' . A model satisfies a generatedness constraint (F, σ) if for each carrier a in $M(\sigma(s))$ there is a Σ' -term t that only uses functions in F and variables of types different from s such that for some variable assignment V , the mapped term σt evaluates to a for V and M . Note that this definition collapses to the simple one if $\Sigma = \Sigma'$ and σ is the identity. A constraint (F, σ) for Σ is mapped along a signature morphism $\sigma' : \Sigma \rightarrow \Sigma''$ by composing the two morphisms:

$$\sigma'(F, \sigma) = (F, \sigma' \circ \sigma).$$

As can easily be seen, $\sigma'(F, \sigma)$ is a generatedness constraint for Σ'' if only (F, σ) is a generatedness constraint for Σ , ([BCH⁺04, Proposition 2.13]), and the satisfaction condition holds (C.6 on page 240). As it turns out, the given definition also has the additional benefit of making our notion of correct proofs (introduced in Section 5.4) invariant under signature morphisms without any effort.

We are now in a position to define the concrete institution

$$\mathcal{I}_{\text{FolEqGen}} = (\mathbf{Sig}_{\text{FolEqGen}}, \mathbf{Sen}_{\text{FolEqGen}}, \mathbf{Mod}_{\text{FolEqGen}}, \models_{\text{FolEqGen}})$$

for first order logic with equality and generatedness constraints that we use for the rest of the thesis. In order to avoid excessive use of indices we simply write $\mathcal{I} = (\mathbf{Sig}, \dots)$. There is no potential of confusion of the parameter \mathcal{I} of the framework and the concrete instance of the reference instantiation since we will be working with this fixed institution from here on.

Definition 5.1 (Concrete Institution) *We define the institution for first order logic with equality and generatedness constraints $\mathcal{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ as follows:*

- The category **Sig** has as objects all algebraic signatures. It has as morphisms between two objects Σ_1 and Σ_2 all algebraic signature morphisms σ from Σ_1 to Σ_2 . For each signature Σ , the identity id_Σ is the signature morphism that maps symbols to themselves. Composition of morphisms is the usual composition of algebraic signature morphisms.
- The functor **Sen** : **Sig** \rightarrow **Set** maps each **Sig**-object Σ to the disjoint union of all closed first order formulae over Σ and all generatedness constraints for

$$\begin{array}{l}
\langle \text{formula} \rangle \rightarrow \text{true} \mid \text{false} \\
\quad \mid (\langle \text{predicate name} \rangle \langle \text{term} \rangle^*) \\
\quad \mid (= \langle \text{term} \rangle \langle \text{term} \rangle) \\
\quad \mid (\text{not } \langle \text{formula} \rangle) \\
\quad \mid (\text{and } \langle \text{formula} \rangle^*) \\
\quad \mid (\text{or } \langle \text{formula} \rangle^*) \\
\quad \mid (=> \langle \text{formula} \rangle \langle \text{formula} \rangle) \\
\quad \mid (<=> \langle \text{formula} \rangle \langle \text{formula} \rangle) \\
\quad \mid (\text{all } (\langle \text{variable name} \rangle \langle \text{sort name} \rangle) \langle \text{formula} \rangle) \\
\quad \mid (\text{ex } (\langle \text{variable name} \rangle \langle \text{sort name} \rangle) \langle \text{formula} \rangle) \\
\\
\langle \text{term} \rangle \rightarrow \langle \text{variable name} \rangle \\
\quad \mid \langle \text{operation name} \rangle \\
\quad \mid (\langle \text{operation name} \rangle \langle \text{term} \rangle^*)
\end{array}$$

Figure 5.1: Logic language abstract syntax

Σ . It maps each **Sig**-arrow $\sigma : \Sigma_1 \rightarrow \Sigma_2$ to the function mapping each Σ_1 -formula φ (each generatedness constraint c for Σ_1) to the Σ_2 -formula σA (the generatedness constraint σc for Σ_2 , respectively), i.e. $\mathbf{Sen}(\sigma)(A) = \sigma A$.

- The functor $\mathbf{Mod} : \mathbf{Sig} \rightarrow \mathbf{Set}^{\text{op}}$ maps each object Σ of **Sig** to the set of all Σ -algebras. It maps each **Sig**-arrow $\sigma : \Sigma_1 \rightarrow \Sigma_2$ to the function mapping each member M_2 of $\mathbf{Mod}(\Sigma_2)$ to the σ -reduct $M_2|_{\sigma} \in \mathbf{Mod}(\Sigma_1)$.
- For each signature $\Sigma \in |\mathbf{Sig}|$, $\models_{\Sigma} \subseteq \mathbf{Mod}(\Sigma) \times \mathbf{Sen}(\Sigma)$ is defined by $M \models_{\Sigma} A$ iff M satisfies A , where A is either a closed Σ -formula or a generatedness constraint for Σ .

This defines an institution. The only non-obvious matter here is whether the satisfaction condition (3.1) on page 40 is satisfied, cf. Theorem C.7 on page 240. Together with Definition 3.3 on page 41, this defines development graphs for first order logic.

For the rest of the thesis, we will use the surface syntax described in Figure 5.1, where a suffixed $*$ stands for zero, one, or more occurrences of the preceding phrase. Note that we use prefix notation for everything and that parentheses are mandatory. We abbreviate

$$(\text{all } (x_1 s_1) \cdots (\text{all } (x_m s_m) \varphi(x_1, \dots, x_n)) \cdots)$$

by

Chapter 5. Formal Developments

$$(\text{all } (x_1 s_1) \cdots (x_m s_m) \varphi(x_1, \dots, x_n)),$$

(for $n \geq 1$) and similarly for the existential quantifier. Example formulae are

```
(all (x nat) (not (= 0 (succ x))))
```

```
(all (x nat) (y nat)
  (=> (= (succ x) (succ y))
      (= x y)))
```

```
(all (y nat) (= (+ 0 y) y))
```

```
(all (x nat) (y nat)
  (= (+ (succ x) y)
     (succ (+ x y))))
```

where we assume that `nat` is a sort, and that `0` and `succ` are nullary and unary functions, respectively, over `nat`. The first two state that the functions `0` and `succ` have disjoint ranges and that `succ` is injective. The second two are the canonical defining axioms of addition for natural numbers. This surface syntax is easier to read than the usual mathematical notation for longer formulae because it scales better with formulae extending over several lines: questions of precedence and associativity of operators cannot occur, and the structure of an expression is immediately clear.

5.3 Concrete Specification Language

We have chosen to use a simple specification language, called SSL, for our reference implementation. An SSL specification $\langle spec \rangle$ consists of a sequence of $\langle theory \rangle$ specifications. See Figure 5.2 for an overview of the abstract syntax. As usual, a suffixed $?$ stands for zero or one occurrence, $+$ for one or more, and $*$ for zero, one, or more occurrences of the preceding item.

5.3.1 Specification in the Small

Each theory specifies a local signature: it declares sorts, operations, and predicates together with their profile. For example the following specifies a signature for natural numbers.

5.3. Concrete Specification Language

$\langle spec \rangle$	\rightarrow	$(\langle theory \rangle^*)$
$\langle theory \rangle$	\rightarrow	$(\text{theory } \langle theory \text{ name} \rangle$ $\quad \langle uses \rangle^? \langle satisfies \rangle^? \langle sig \rangle \langle gens \rangle \langle axioms \rangle)$
$\langle uses \rangle$	\rightarrow	$\text{uses } (\langle theory \text{ name} \rangle \langle map \rangle)^*$
$\langle satisfies \rangle$	\rightarrow	$\text{satisfies } (\langle theory \text{ name} \rangle \langle map \rangle)^*$
$\langle map \rangle$	\rightarrow	$\langle point \rangle^*$
$\langle point \rangle$	\rightarrow	$(\langle sort \text{ name} \rangle \langle sort \text{ name} \rangle)$ $\quad $ $\quad (\langle operation \text{ name} \rangle \langle operation \text{ name} \rangle)$ $\quad $ $\quad (\langle predicate \text{ name} \rangle \langle predicate \text{ name} \rangle)$
$\langle sig \rangle$	\rightarrow	$\langle sorts \rangle \langle ops \rangle \langle preds \rangle$
$\langle sorts \rangle$	\rightarrow	$(\text{sort } \langle sort \text{ name} \rangle)^*$
$\langle ops \rangle$	\rightarrow	$(\text{op } \langle operation \text{ name} \rangle \langle domain \rangle \langle codomain \rangle)^*$
$\langle preds \rangle$	\rightarrow	$(\text{prd } \langle predicate \text{ name} \rangle \langle domain \rangle)^*$
$\langle domain \rangle$	\rightarrow	$(\langle sort \text{ name} \rangle^*)$
$\langle codomain \rangle$	\rightarrow	$\langle sort \text{ name} \rangle$
$\langle gens \rangle$	\rightarrow	$(\text{gen } \langle operation \text{ name} \rangle^+)^*$
$\langle axioms \rangle$	\rightarrow	$(\text{axiom } \langle formula \rangle)^*$

Figure 5.2: Specification language abstract syntax

Chapter 5. Formal Developments

```
(sort nat)
(op 0 () nat)
(op succ (nat) nat)
(op + (nat nat) nat)
(prd leq (nat nat))
```

We can allow the same name to be used for different symbols, e.g. naming a sort symbol and a predicate symbol, since there are no ambiguities. We also allow multiple declarations of symbols, provided the declarations are consistent, i.e. all but one declarations are redundant.

Each theory also specifies generatedness constraints and axioms. A generatedness constraint for the natural numbers is written as

```
(gen 0 succ)
```

i.e. `0` and `succ` are the constructors. We do not provide for any special language construct for directly specifying freely generated datatypes. Instead, the necessary axioms can be expressed as a generatedness constraint and additional normal axioms. Axioms are written as closed formulae in the form described in Section 5.2. The axioms, e.g., for making `nat` freely generated (i.e. the constructors `0` and `succ` are injective and have disjoint ranges) read

```
(axiom (all (x nat) (not (= 0 (succ x))))))
```

```
(axiom (all (x nat) (y nat)
            (=> (= (succ x) (succ y))
                (= x y))))
```

and the canonical defining axioms for `+` are:

```
(axiom (all (y nat) (= (+ 0 y) y)))
```

```
(axiom (all (x nat) (y nat)
            (= (+ (succ x) y)
                (succ (+ x y)))))
```

5.3.2 Specification in the Large

Theories are related by *uses* and *satisfies* clauses. The *uses* clause determines a signature. The content of a theory is then interpreted relative to this signature, i.e. the local signature of the theory extends the given background signature. For example, the definition of natural numbers can be

5.3. Concrete Specification Language

presented in two theories. The first theory `nat` defines the natural number and addition:

```
(theory nat
  (sort nat)
  (op 0 () nat)
  (op succ (nat) nat)
  (op plus (nat nat) nat)
  (gen 0 succ)
  ...
  (axiom (all (y nat) (= (plus 0 y) y)))
  ...)
```

The second uses (imports) the first theory, renaming `plus` to `+`, and then extends the signature and states additional axioms.

```
(theory assoc-nat
  (uses (nat (plus +)))
  (prd even (nat))
  ...
  (axiom (all (x nat) (y nat) (z nat)
    (= (+ x (+ y z))
      (+ (+ x y) z))))
  ...)
```

Note that `plus` has been renamed to `+`. All $\langle theory \rangle$ -specifications that are used (also called imported specifications, `nat` in the example) are required to be defined lexically before the referring $\langle theory \rangle$ (`assoc-nat` in the example). The relationship ‘uses’ between $\langle theory \rangle$ -specifications is therefore acyclic. A $\langle theory \rangle$ -specification wrt. the whole specification determines a signature and a set of axioms as follows. The signature symbols defined in the imported theories are renamed according to the respective `uses`-clause, and the resulting symbols are in the signature of the theory. Obviously, this construction is defined only if the mapped signatures of the imported theories can be merged. In this case, the set of axioms of the theory consists of all axioms of the imported theories, mapped according to the `uses`-clauses, and the local axioms stated in the $\langle theory \rangle$ -specification itself.

The `satisfies` clause allows us to state that a theory satisfies another one, i.e. the axioms of the other theory are consequences of the current theory (theory inclusion).

Example 5.2 Consider the theory `nat` with the additional clause

```
(satisfies (assoc-nat (+ plus)))
```

The symbol mapping is interpreted as a mapping from `assoc-nat` to `nat`. The intended semantics is that all axioms in `assoc-nat` mapped according to the symbol mapping are logical consequences of the theory `nat`. ◦

Note that `assoc-nat` uses `nat`, and that `nat` satisfies `assoc-nat`, so the relationship ‘uses or satisfies’ between theories is potentially cyclic. This means that the theory names in a `satisfies` clause may name theories that are specified textually after the reference.

We have intentionally imposed many additional static well-formedness conditions, i.e. restrictions on specifications. For example, the constructors of generatedness constraints in a given theory need to be functions that have been declared in the given theory, rather than imported. Such restrictions have been introduced at will to evoke the problems that deal with more complicated definitions of surface syntax in other specification languages. For instance, generated datatypes are declared in CASL by

```
datatype nat = 0 | succ(nat);
```

which is then understood on the development graph level as the declaration of the sort, the constructors, and the generatedness constraint. Because of this syntax, the constructors are necessarily defined in the same theory as the generatedness constraint. The effect is that the mapping from specifications to development graphs is not surjective, and we cannot rely on finding a specification for any given development graph.

5.3.3 Mapping to Development Graphs

A well-formed SSL specification is mapped to a development graph in the following way. The theories of the specification are inspected in succession, and for each theory specification a development graph node is added to the initially empty graph. Each $\langle \textit{satisfies} \rangle$ -specification is remembered as pending for later handling (in general we cannot handle it now, because the node for the theory that it references may not yet exist). The signatures of the nodes in the $\langle \textit{uses} \rangle$ are retrieved (they already exist because of the acyclicity of the “uses”-relation) and are mapped along the given symbol- $\langle \textit{map} \rangle$ and the definitions are put together yielding an auxiliary signature called background signature. Then the sort, function, and predicate declarations in the theory specification are analysed in turn with respect to the background signature extended by the definitions that have already been

looked at in the current theory, i.e. there is linear visibility. The result is a signature, which is then associated with the node in the development graph. For each used theory, the development graph is extended by a definitorial link from the imported node to the current one, and the link is annotated with a signature morphism from the used node to the current signature that respects the symbol mapping given for the respective used theory. The rest of the theory specification now consists of generatedness constraint and axiom specifications, and these are parsed with respect to the signature of the node. Note that generatedness constraint specifications only mention the constructors, and from them, a generatedness constraint with the identity morphism is constructed. The resulting set of constraints and axioms is associated with the development graph node as the set of local axioms of the node. Finally, when all theories have been handled, each pending satisfies-link induces a postulated link between two existing nodes in the development graph.

5.4 Concrete Proof Representation

We use an analytic sequent calculus, cf. [Gen35], [Fit96], which represents proof states by sequents and proofs by trees, where nodes are sequents. A Σ -sequent, written as $\Gamma \vdash \Delta$, consists of two sets of Σ -formulae. Γ is called the antecedent and Δ the succedent. As usual, we will omit set braces and mix sets and formulae, as in $\Gamma, \varphi \vdash \psi$ which is shorthand for $\Gamma \cup \{\varphi\} \vdash \{\psi\}$. The semantics of a sequent $\Gamma \vdash \Delta$, where Γ and Δ consist of closed formulae, is the same as that of the first order formula $(\bigwedge \Gamma) \Rightarrow (\bigvee \Delta)$, and we say that the sequent holds if the formula evaluates to true. A proof obligation $\Gamma \vdash_{\Sigma} \varphi$ (where φ and all members of Γ are closed formulae) will be represented by the sequent $\Gamma \vdash \varphi$.

Obviously, the definition of sequents and their semantics can be extended canonically to the case where the formulae include free first-order variables. We have chosen, however, to represent Eigenvariables as variables that are free in the formulae, and which are λ -bound at the level of sequents, rather than using free variables in sequents. This has the benefit that the names of Eigenvariables do not matter. This avoids problems with name clashes between Eigenvariables, bound variables, and constants: Eigenvariables and bound variables can be α -renamed uniformly wherever necessary. A sequent that represents the proof state in which $\forall x : s. \varphi(x)$ is to be proven from Γ is written as

$$\Gamma \vdash \forall x : s. \varphi(x) .$$

Chapter 5. Formal Developments

The proof state in which the formula $\varphi(x)$ for an Eigenvariable x is to be proven is written as

$$[x : s] \Gamma \vdash \varphi(x) . \quad (5.1)$$

This is a notational variant of $\lambda x : s. (\Gamma \vdash \varphi(x))$. The scope of the binding of x is the whole sequent, so Γ could refer to x , i.e. as in $[x : s] \Gamma(x) \vdash \varphi(x)$. Due to the implicit α -renaming, there are no clashes with any constants in Γ . Thus, the meaning of a sequent

$$[x_1 : s_1, \dots, x_m : s_m] \Gamma(x_1, \dots, x_m) \vdash \Delta(x_1, \dots, x_m) \quad (5.2)$$

is the meaning of the closed formula

$$\forall x_1 : s_1, \dots, x_m : s_m. (\bigwedge \Gamma(x_1, \dots, x_m)) \Rightarrow (\bigvee \Delta(x_1, \dots, x_m)) .$$

Identity of sequents is defined by equality modulo α -renaming. Sequents with no bound variables, i.e. $[\] \Gamma \vdash \Delta$, are abbreviated by $\Gamma \vdash \Delta$ where this is convenient. $[x_1 : s_1, \dots, x_m : s_m] \Gamma(x_1, \dots, x_m) \vdash \Delta(x_1, \dots, x_m)$ is abbreviated by

$$[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}) .$$

So far, only formulae have been considered, but not generatedness constraints. Generatedness constraints provide induction schemata. Formally, we allow generatedness constraints as members of the antecedent and succedent of sequents, and we provide corresponding induction rules for generatedness constraints occurring in the antecedent. As it will turn out, we can omit generatedness constraints for ease of presentation and assume that every sequent in a proof has the same generatedness constraints. It is thus sufficient to state the available generatedness constraints once for each proof.

We consider partial proofs as trees. A (partial) proof is either a leaf node or a branch. A proof carries a sequent Θ , called its conclusion. A leaf node represents the trivial open proof. A branch represents the application of a proof rule. In addition to the conclusion, a branch carries a justification j, a (where j is the name of the rule and a are arguments specifying the instance of the proof rule) and a sequence $\langle \xi_1, \dots, \xi_n \rangle$ of subproofs, written as

$$\frac{\begin{array}{ccc} \vdots \xi_1 & & \vdots \xi_m \\ \Theta_1 & \cdots & \Theta_m \end{array}}{\Theta} j, a$$

5.4. Concrete Proof Representation

Each proof rule name j has an associated condition that determines whether

$$\frac{\Theta_1 \quad \cdots \quad \Theta_m}{\Theta} j, a$$

is a valid proof rule step. These conditions are defined such that the conclusion Θ follows logically from the conjunction of the premisses $\Theta_1, \dots, \Theta_m$, and thus the calculus is sound. In a proof rule, e.g. in

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}), \dots, \varphi_m(\vec{x}) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \vdash \Delta(\vec{x})} \text{and-1 } \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x})$$

the sets of formulae $\Gamma(\vec{x})$ and $\Delta(\vec{x})$ are called the context, and the formula $\varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x})$ is called the focus formula.

Definition 5.3 (Concrete proof representation) *The concrete goal functor $\mathbf{Goal} : \mathbf{Sig} \rightarrow \mathbf{Set}$ mapping signatures to the set of Σ -goals and morphisms to mappings from goals to goals is defined by:*

- For each $\Sigma \in |\mathbf{Sig}|$, $\mathbf{Goal}(\Sigma)$ is the set of Σ -sequents in (5.1).
- For each \mathbf{Sig} -morphism $\sigma : \Sigma \rightarrow \Sigma'$, $\mathbf{Goal}(\sigma)$ is the mapping from Σ -goals to Σ' -goals defined by

$$\begin{aligned} \mathbf{Goal}(\sigma)([x_1 : s_1, \dots, x_m : s_m] \Gamma(x_1, \dots, x_m) \vdash \Delta(x_1, \dots, x_m)) \\ = [x_1 : \sigma s_1, \dots, x_m : \sigma s_m] \sigma \Gamma(x_1, \dots, x_m) \vdash \sigma \Delta(x_1, \dots, x_m). \end{aligned}$$

We define a natural transformation $\eta : 2^{\mathbf{Sen}} \times \mathbf{Sen} \rightarrow \mathbf{Goal}$:

- For each Σ , η_Σ is the mapping that maps the pair (Γ, φ) to the Σ -sequent $\Gamma \vdash \varphi$.

The proof functor $\mathbf{Prf} : \mathbf{Sig} \rightarrow \mathbf{Set}$ is defined by:

- For each signature Σ , the set $\mathbf{Prf}(\Sigma)$ of Σ -proofs is the smallest set that satisfies the following conditions:
 - $\mathbf{Goal}(\Sigma) \subseteq \mathbf{Prf}(\Sigma)$
 - If $\Theta \in \mathbf{Goal}(\Sigma)$, $\xi_1, \dots, \xi_n \in \mathbf{Prf}(\Sigma)$ with the conclusions $\Theta_1, \dots, \Theta_n$, and

$$\frac{\Theta_1 \quad \cdots \quad \Theta_m}{\Theta} j, a$$

is a valid proof rule then

$$\frac{\xi_1 \quad \cdots \quad \xi_m}{\Theta} j, a$$

is a member of $\mathbf{Prf}(\Sigma)$.

Chapter 5. Formal Developments

Proof rules are the standard proof rules for first order sequent calculus with equality and structural induction rules; representative proof rules are given in Figure 5.3.

- For each $\sigma : \Sigma \rightarrow \Sigma'$, $\mathbf{Prf}(\sigma)$ maps the open Σ -proof Θ to $\mathbf{Goal}(\sigma)(\Theta)$, and the branching proof

$$\frac{\xi_1 \quad \cdots \quad \xi_m}{\Theta} j, a$$

to

$$\frac{\mathbf{Prf}(\sigma)(\xi_1) \quad \cdots \quad \mathbf{Prf}(\sigma)(\xi_m)}{\mathbf{Goal}(\sigma)(\Theta)} j, \sigma a \quad (5.3)$$

The natural transformation $\mathit{concl} : \mathbf{Prf} \rightarrow \mathbf{Goal}$ is defined as expected:

- $\mathit{concl}(\Theta) = \Theta$ and $\mathit{concl}(\xi) = \Theta$ where ξ is

$$\frac{\xi_1 \quad \cdots \quad \xi_m}{\Theta} j, a$$

Note that (5.3) presupposes that the result of $\mathbf{Prf}(\sigma)$ is again a valid proof, and in particular that $j, \sigma a$ is a valid proof step. As it turns out, this is indeed the case. We discuss this in detail below after we have given some more details about the concrete proof rule steps. Otherwise, however, it is easy to see that this defines a proof representation according to Definition 3.19 on page 52.

Figure 5.3 only lists some representative calculus rules, all of which are standard. We have chosen to present the rules in a notation that makes the handling of Eigenvariables explicit. This is needed later to explain in detail, how some of the transformations on proofs interact with bound variables. Notice, however, that the rules are completely standard.

In the induction rule $\mathit{ind-r}$, the induction formula for some constructor f_i is abbreviated by $\Phi(\vec{x}, f_i, \sigma, \varphi)$ as follows. We want to prove the formula $\forall x : \sigma s. \varphi(\vec{x}, x)$ by structural induction using the generatedness constraint (F, σ) , where we assume that the codomain of the constructors in F is s . The constraint is applicable, because it constrains σs , which is the sort of the universally quantified variable x . F is a non-empty set of m constructors $f_i : t_{i,1} \times \cdots \times t_{i,n_i} \rightarrow s$ ($1 \leq i \leq m$, $0 \leq k_i$) in some signature. For each such f_i ,

$$\sigma f_i : \sigma t_{i,1} \times \cdots \times \sigma t_{i,k_i} \rightarrow \sigma s$$

5.4. Concrete Proof Representation

$$\begin{array}{c}
\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x})} \text{ weak } \varphi(\vec{x}) \qquad \frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x})} \text{ weak } \varphi(\vec{x}) \\
\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}) \qquad [\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})} \text{ cut } \varphi(\vec{x}) \\
\frac{}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x})} \text{ basic} \\
\frac{}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), t(\vec{x}) = t(\vec{x})} \text{ basic} \\
\frac{}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \top} \text{ basic} \qquad \frac{}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \perp \vdash \Delta(\vec{x})} \text{ basic} \\
\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}), \dots, \varphi_m(\vec{x}) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \vdash \Delta(\vec{x})} \text{ and-l } \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \\
\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \qquad \dots \qquad [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_m(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x})} \text{ and-r } \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \\
\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \forall x : s. \varphi(\vec{x}, x), \varphi(\vec{x}, t(\vec{x})) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \forall x : s. \varphi(\vec{x}, x) \vdash \Delta(\vec{x})} \text{ all-l } \forall x : s. \varphi(\vec{x}, x), t(\vec{x}) \\
\frac{[\vec{x} : \vec{s}, y : s] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}, y)}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \forall x : s. \varphi(\vec{x}, x)} \text{ all-r } \forall x : s. \varphi(\vec{x}, x) \\
\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), t_1(\vec{x}) = t_2(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}, t_2(\vec{x}))}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), t_1(\vec{x}) = t_2(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}, t_1(\vec{x}))} \text{ eqn-l } t_1(\vec{x}) = t_2(\vec{x}), \varphi(\vec{x}, t_1(\vec{x})), \varphi(\vec{x}, t_2(\vec{x})) \\
\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), t_1(\vec{x}) = t_2(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}, t_1(\vec{x}))}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), t_1(\vec{x}) = t_2(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}, t_2(\vec{x}))} \text{ eqn-r } t_1(\vec{x}) = t_2(\vec{x}), \varphi(\vec{x}, t_2(\vec{x})), \varphi(\vec{x}, t_1(\vec{x})) \\
\frac{\dots \quad [\vec{x} : \vec{s}] \Gamma(\vec{x}), (F, s, \sigma) \vdash \Delta(\vec{x}), \Phi(\forall, \vec{x}, f_i, \sigma, \varphi) \quad \dots}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), (F, s, \sigma) \vdash \Delta(\vec{x}), \forall x : \sigma s. \varphi(\vec{x}, x)} \text{ ind-r } \forall x : \sigma s. \varphi(\vec{x}, x), (F, s, \sigma)
\end{array}$$

Figure 5.3: Proof calculus rules. There are other rules, e.g. the usual rules for the missing connectives and quantifiers, and rules for induction and applying equations to formulae in the antecedent. See (5.4) on page 116 for the definition of the induction formula Φ .

Chapter 5. Formal Developments

is a function in Σ . Let $I_i \subseteq \mathbb{N}$ be those indices j for which $t_{i,j} = s$, i.e. the positions in the function profile for which induction hypotheses can be generated. Then we define the induction formula $\Phi(Q, \vec{x}, f_i, \sigma, \varphi)$ by

$$Qy_1 : \sigma t_{i,1}, \dots, y_{k_i} : \sigma t_{i,k_i}. \left(\bigwedge_{j \in I_i} \varphi(\vec{x}, y_j) \right) \Rightarrow \varphi(\vec{x}, (\sigma f_i)(y_1, \dots, y_{k_i})), \quad (5.4)$$

where Q is a quantor (\forall in the case `ind-r`, and \exists for `ind-l`, which is not shown in Figure 5.3). Here, the definition of generatedness constraints on page 103 including a signature morphism is seen to be necessary: It would not be sound to generated induction hypotheses for all $\sigma t_i = \sigma s$.

We turn to the discussion of the fact that Definition 5.3 provides a proof representation, i.e. that the following conditions hold.

- η and *concl* are natural transformations: all sentences occurring in sequences and proofs are mapped uniformly along σ for $\mathbf{Goal}(\sigma)$ and $\mathbf{Prf}(\sigma)$, so this is easy to see.
- Algebraic signature morphisms preserve proof-hood. This property of our proof representation depends on the absence of free variables with the associated implicit α -renaming, and on the definition of generatedness constraints with an explicit signature morphisms. Another important aspect is that arguments to proof rule justifications consist of formulae and terms, rather than, e.g. the number of the focus formula in the antecedent or succedent. Because there are no free variables, the names of signature symbols do not matter for the applicability of the proof rules. This also holds for the induction rule, which is the only place this cannot be seen immediately: according to (5.4), for each induction formula $\Phi(\vec{x}, f_i, \sigma, \varphi)$ for a constraint $\sigma'(F, \sigma) = (F, \sigma' \circ \sigma)$ and constructor $f_i \in F$, we have

$$\Phi(\vec{x}, f_i, \sigma' \circ \sigma, \varphi) = \mathbf{Sen}(\sigma')(\Phi(\vec{x}, f_i, \sigma, \varphi)).$$

Remark 5.4 We have made all available axioms and generatedness constraints part of the sequent, i.e. a proof obligation $\Gamma \vdash_{\Sigma} \varphi$ corresponds to a proof with conclusion $\Gamma \vdash \varphi$. A widely used alternative is to let the proof obligation correspond to the sequent $\vdash \varphi$ and represent proof states as pairs of sequents and the set of available axioms Γ . Axioms can then be inserted by a variant of the cut rule. This has advantages in practice (the number of potentially visible axioms can get very large, and it is

then very beneficial if only explicitly requested axioms are displayed), but is not essentially different from our formulation: if the set of usable axioms is managed explicitly, the theoretical difference is marginal, but our formulation avoids the additional complexity of maintaining the set. If the axioms are managed implicitly, effects of some of the transformations are more difficult to see, so this was not an option for our work.

5.5 Summary

We have presented a concrete formulation of first order logic with equality and generatedness constraints, a corresponding concrete specification language for structured specifications, and a concrete proof representation for the logic. Except for technical differences these are similar to what can be found in existing formal case tools, although we have tried to keep the definitions as simple as possible. We have shown that the concrete definitions constitute an instantiation of the context defined for transformations in our framework.

Chapter 6

Specification Transformations

6.1 Overview

We have now at our disposal a concrete instance of a specification language, a logic, and an associated proof representation. We will use this instance to formulate concrete specification transformations addressing the example scenarios that motivated our work.

For each of these transformations, two questions have to be answered.

- Which changes are allowed on the specification text such that the result is again a (well-formed) specification? For example, if the signature of one theory is changed, the signatures of all other theories that use the theory changes also, and references to the changed symbol need to be adapted. This is a question of the syntax and the static well-formedness conditions of the specification language.
- Given a specification S' that is the result of applying a given specification transformations to S , what is the appropriate development graph transformation that transforms $D = dg(S)$ to $D' = dg(S')$? This is a question of the relationship between specification language and development graphs.

The discussion includes an investigation of what effects we expect the transformations to have on proofs. Specification transformations are related to proof transformations indirectly via development graph transformations and proof obligation transformations. Concrete specification transformations are propagated to development graphs using instances of the the generic development graph transformations that have been introduced in Chapter 4. These in turn propagate to proof obligations and proofs. As a result, we can state which proof transformations are needed

to propagate our specification transformations to proofs. A summary of this association is given graphically at the end of this chapter on page 139.

6.2 Adding and Deleting Elements

We begin with specification transformations that deal with adding or deleting elements like theories, uses or satisfies clauses, signature items, and axioms, but do not change existing ones. Adding such elements to the specification or removing them is mapped straightforwardly to a combination of the development graph transformations Tr_{nodes} , Tr_{links} , $Tr_{signatures}$, and Tr_{axioms} .

We ignore issues about whether, e.g., an axiom that is added is similar to an existing one. If it is, then the similarity would have been represented by applying Tr_{occ} rather than Tr_{axioms} . Since the transformation Tr_{occ} explicitly deals with the similarity of old and new axioms there is no good reason to duplicate the functionality here. There is of course the interesting question of whether the similarity should be detected by the tool or whether the users are required to notice the similarity themselves. However, this question is orthogonal to the question of how the similarity, once it has been noted, is used to keep proofs. Similarly, if an axiom is deleted, there is no point in arguing whether another axiom is similar.

6.2.1 Theories

A theory that contains no use and satisfies clauses, and that is not used in any use or satisfies clauses of any other theory can be added to a specification, provided the name has not been used already for another theory. According to the description of the grammar of SSL specifications, cf. Figure 5.2 on page 107, it is sufficient to add a well-formed expression of the form

$$(\text{theory } \langle name \rangle \langle sig \rangle \langle gens \rangle \langle axioms \rangle)$$

to the list of theories of the specification. Since there are no dependencies between the new theory and the existing parts of the specification, there is no effect on the proof obligations.

Propagation to Development Graphs. The development graph can be adjusted by adding a new node with the given name and signature and sentences, and without any in- or outgoing links. This can be achieved

6.2. Adding and Deleting Elements

by the development graph transformation Tr_{nodes} without any effects on the proofs. Similarly, a node without in- or outgoing links can be deleted without any effects, also using Tr_{nodes} .

6.2.2 Axioms

Adding axioms to or deleting them from a specific theory is straightforward: Simply add the phrase (axiom $\langle formula \rangle$) to the theory, or remove an existing phrase of this form. Axioms do not influence the signatures, so this does not have an influence on the well-formedness of any of the other axioms of the theory, nor does it have an influence on the well-formedness of any other theory.

Semantically, the axioms will potentially be added to or removed from other theories, according to uses and satisfies clauses. This does not change the specification text, however. The theories in which an added axiom is visible is thus determined implicitly and uniformly by the framework, in particular the development graph transformation Tr_{axioms} (cf. Section 4.3.1 on page 72). As described there, the result on proofs is that new proofs may be necessary or old ones superfluous, and that existing proofs will have changed assumptions.

Propagation to Development Graphs. The axiom is added to or removed from the local set of axioms of the corresponding node, while all other nodes' local sets of axioms remain unchanged. The development graph transformation Tr_{axioms} does exactly this. On the level of proof obligations this effects the set of applicable assumptions.

6.2.3 Signature Items

Adding or removing signature items to or from theories in a specification does not have an influence on the structure of the specification as far as theories and use or satisfies clauses are concerned. On the other hand, the signatures of the theory itself and possibly other theories change.

Example 6.1 Consider the trivial SSL specification

```
(theory A (sort s) (op f (s s) s))  
(theory B (uses (A)) (sort t))
```

Chapter 6. Specification Transformations

in which s is visible in the signature of A and B and assume that we want to remove s from B but keep it in A . This is not possible; instead s can be removed from A ,¹ in which case it is also removed from theory B .

In the other direction, assume that we want to add an operation $f : s \rightarrow s$ to theory A of the original specification. This extends the signature of both A and B so that f is visible in both. \circ

The effect is that signatures of theories are extended or restricted. The necessary signature extensions, and whether suitable signature extensions exist at all, can be determined by following the name inheritance rules of the specification language. When an item named i is added to one theory A_1 , it becomes visible in A_1 and it also becomes visible in any other theory A_2 that uses A_1 . In SSL, without an explicit mapping in the uses clause i will be visible in the signature of A_2 under the same name. Suppose that \bar{i} is the name of i in A_2 . There are three cases that we need to consider:

1. There is no item in the signature of A_2 with the name \bar{i} . In this case we add a new item named \bar{i} to the signature and recursively consider all the theories that use n_2 .
2. There is already an item j named \bar{i} in A_2 .
 - (a) j is compatible with i : if it is a function or predicate the arities agree modulo the mapping. In this case there is nothing to do.
 - (b) j is not compatible with i . This is an error and means that the transformation is not applicable to the specification.

As an example, consider the specification

```
(theory A (sort s))
(theory B (uses (A)) (op g (s) s) (op h (s s) s))
```

and assume that we want to add an operation with profile $s \rightarrow s$ to A . If we add `(op f (s) s)` to A , the first case applies since there is no operation f in B . No change to the specification text of B is necessary and f is added to the signature of B . If we add `(op g (s) s)`, case 2(a) applies. Finally, if we try to add `(op h (s) s)`, case 2(b) applies since the profiles $s \rightarrow s$ and $s \times s \rightarrow s$ are not compatible.

Similar effects occur when an item is added to a theory that is the target of a satisfies clause: the new item needs to be mapped to some item in the theory that contains the satisfies clause in order for the statement of satisfaction to be well-formed. Consider the specification

¹In this case, f also needs to be removed, of course.

6.2. Adding and Deleting Elements

```
(theory A (sort s))
(theory B (satisfies (A))
  (sort s) (op g (s) s) (op h (s s) s))
```

Adding an operation with profile $s \rightarrow s$ to A leads to similar observations as in the previous example. For instance, if we add $(\text{op } f \ (s) \ s)$, we also need to add $(\text{op } f \ (s) \ s)$ to B and thereby add f to the signature of B. Note that in this case, the specification text needs to be changed because signatures are not inherited along *satisfies* relationships, according to the rules static well-formedness for SSL.

Since the graph consisting of *uses* and *satisfies* dependencies is potentially circular, we need to follow the links until a fixpoint is reached or case 2(b) above leads to an error. For the simple naming scheme of SSL, the fixpoint or an error is reached in a small, finite number of iterations. However, this need not be the case in general.

Deleting items and propagating their removal over dependencies works in a similar way. Removing i from a theory will also mean that the items it is mapped to by *uses* or reverse *satisfies* clauses will be removed, unless they are declared redundantly or inherited from somewhere else, in which case the formerly redundant declaration will now be their (non-redundant) declaration. Deleting an item also raises the question of what to do with axioms that use the item. Our solution is to forbid this: items can only be removed when they are not used. If they are used, say, in an axiom, the axiom should be removed before the item can be deleted.

Since newly added items are necessarily unused and since only unused items may be deleted, we expect all the existing proofs to be logically unchanged. Of course, proofs can in principle refer to signature items that are not used in the axioms and conjecture of the specification. For instance, a witness term or a cut formula in a proof might use an otherwise unused predicate, or quantify over an otherwise unused sort. If this is the case the transformation fails. Users can then decide whether they want to change the relevant proofs (cf. Section 7.8) and delete the item afterwards or whether they want to keep the item after all.

Propagation to Development Graphs. While propagating added items over dependencies, we collect the information about which item is added to which theory, and which items are additionally mapped along which *uses* and *satisfies* clauses. Since each theory corresponds to a node in the associated development graph, we can use this information directly to compute the new signatures and signature morphisms of the new development after the change. It is easy to see that in the new development

graph, each node either has the same signature as the corresponding node in the original development graph, or that it is a signature extension of the original signature. It is also easy to see that condition (4.10) on page 80 is satisfied. So the association of old and new nodes and the information about which item is added to which node determines a development graph signature adjustment according to Definition 4.13.

Removing signature items works similarly: removing i from A_1 necessitates removing the image of i from all theories that use n_1 or that satisfy A_1 , unless the image is also inherited from another theory or defined explicitly in the respective theory. The result is that nodes in the development graph are either unchanged or are mapped along a signature restriction. This is provided by the development graph transformation $Tr_{signatures}$.

Note that $Tr_{signatures}$ is more flexible and can deal naturally with more complex symbol inheritance schemes that some other specification languages use. It does not require the added or removed symbols to have the same name in different theories because of our liberal definition of what we are willing to accept as a signature extension according to Definition 4.12.

6.2.4 Uses and Satisfies Clauses

Uses or satisfies clauses can be added by inserting *uses* and *satisfies* clauses to the specification of theories such that the specification is well-formed afterwards. This will affect the signatures of existing nodes in the associated development graph, and it will produce new links.

Example 6.2 Consider the SSL specification

```
(theory n (sort s))
(theory m (sort t))
```

which gives rise to a development graph with two nodes and no links, cf. Figure 6.1(a). Adding the clause `(uses (n))` to `m`, i.e. the whole specification then reads

```
(theory n (sort s))
(theory m (uses (n)) (sort t))
```

enlarges the signature of `m` and produces a link with an injection morphism from `n` to `m`, cf. Figure 6.1(b). Adding another *uses* clause by changing `(uses (n))` to

```
(uses (n) (n (s u)))
```

6.2. Adding and Deleting Elements

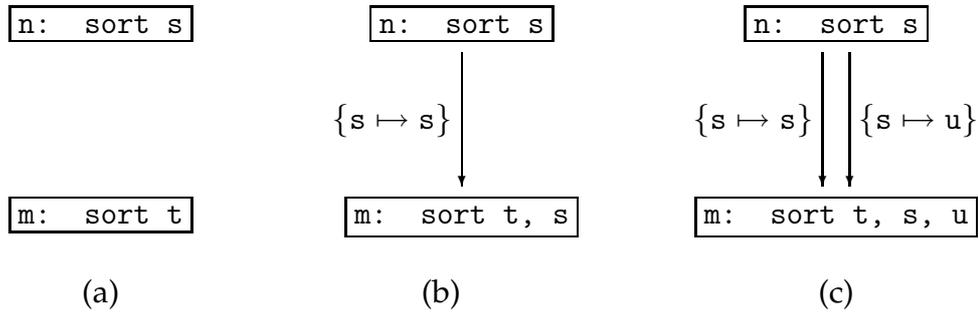


Figure 6.1: Adding links

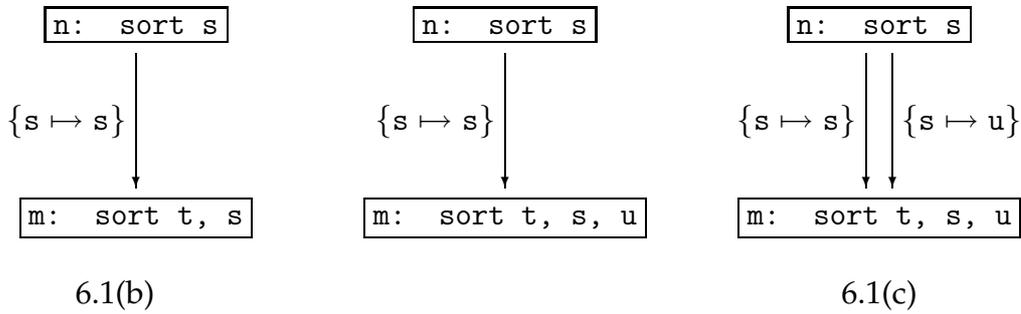


Figure 6.2: Realization of adding links

makes m use n twice with two different mappings, leading to the development graph in Figure 6.1(c). \circ

Adding or deleting dependencies also has the effect of changing the visible axioms of theories and thus the assumptions that are available for some proofs. Additionally, added or deleted dependencies may make existing proof obligations superfluous or add new obligations.

Propagation to Development Graphs. On the level of development graphs, such transformations can be carried out in two steps: first the signatures are adjusted using $Tr_{signatures}$, and second, the link is added.

Example 6.3 The transformation from Figure 6.1(b) to Figure 6.1(c) is carried out on the level of development graphs in the two steps that are visualised in Figure 6.2. \circ

The first step is exactly the same as adding the new items to all theories that inherit them along the new link to be added, i.e. m in the example. We have described this step, i.e. how to find out the development graph signature adjustment for this case, in the preceding section for adding signature

items.

Removing links works in a similar way – just backwards. First, the links are removed using Tr_{links} , and then the signatures are adjusted by removing those signature symbols that are now no longer present due to the missing links. Again, this second step has been described in Section 6.2.3.

In summary, dependencies can be added to a specification or removed from one by using the development graph transformations $Tr_{signatures}$ and Tr_{links} . Proofs are kept over $Tr_{signatures}$, on the other hand Tr_{links} makes obligations obsolete, adds new ones, and changes the available assumptions of some obligations.

6.3 Changing Elements

Specification transformations that change existing signature items or axioms leave the overall structure of the specification unchanged.

6.3.1 Signature Item Names

Changing the name of, say, a sort in a theory from s_1 to s_2 consistently, where we assume that s_2 is not visible in the theory, is straightforward. The name does not matter for the properties of the sort, and all statements involving the sort will all have exactly the same semantical content, and formally there will be an isomorphism between the original and the resulting theory: this is what is meant by “consistently”. However, in existing tools it is hard to change a name consistently, because not all occurrences of the name in the specification refer to the item in question, and not all references to the item in question are necessarily by using the name. Thus going by the specification text is, in general, not going to be satisfactory. The general problem, as it also manifests itself in mainstream programming IDEs, is described in more detail in [Wil05].

Our approach provides a partial solution to the problem: we can express the transformation that renames an item consistently on the level of development graphs using specifically well-behaved development graph translations. This does not solve the question of how to find the new specification text, but it helps in deciding whether a proposed new text in fact is the old text with an item renamed consistently.

In SSL the transformation can be expressed relatively easily on the level of single theories: the syntax is simple enough to let us decide reliably which identifier needs to be changed. We will not go into the details.

6.3. Changing Elements

On the level of uses and satisfies clauses, the transformation is determined in the following way. Assume that we change the name of the item i in theory n_1 . Then each uses or satisfies clause involving n_1 needs to take this into account. When another theory n_2 uses n_1 , it can either map the name of i to another name, or it can simply keep the name by not giving a mapping. In the latter case, we can add a $\langle point \rangle$ to the map of the respective using link so that the changed name is mapped back, in the former case we can simply change the second component.

Example 6.4 If we change s in n to t , and there is a uses clause

$$(\text{uses } (n \ (s \ u)))$$

we simply change that to

$$(\text{uses } (n \ (t \ u)))$$

A clause $(\text{uses } (n))$ is changed to $(\text{uses } (n \ (t \ s)))$.² ◦

If n_1 uses other theories and imports the item the name of which is changed in n_1 , the respective uses clause is also changed. The clauses that describe satisfies dependencies are handled similarly. The net effect is that no other theory is changed except for the theory in which we want to change the name. None of the proofs should be affected logically.

Propagation to Development Graphs. Assume that s_1 is changed to s_2 in the theory corresponding to node n_1 . We provide a copy of the original development graph in which the signature of n_1 is changed to match the renaming of s_1 to s_2 . Figure 6.3 visualises this situation for two additional nodes n_0 and n_2 that are linked to n_1 . The top row of the diagram depicts the signatures of three nodes n_0 , n_1 , and n_2 of the original development graph and the link morphisms for the links l_1 and l_2 between them. The bottom row depicts the corresponding signatures in the new development graph: the changed signature of the node corresponding to n_1 and the two unchanged signatures of the two other nodes. The unchanged signatures in the original and the new development graph are related by the appropriate identity signature morphisms $\text{id}_{\Sigma^{n_0}}$ and $\text{id}_{\Sigma^{n_2}}$. The changed signatures Σ^{n_1} and Σ'^{n_1} are related by a signature morphism σ_{n_1} that renames s_1

²This means that name changes are not propagated along uses and satisfies clauses. However, propagation can be realised by a number of sequential renamings. A transformation to this effect can easily be provided using the more basic transformation we have provided here.

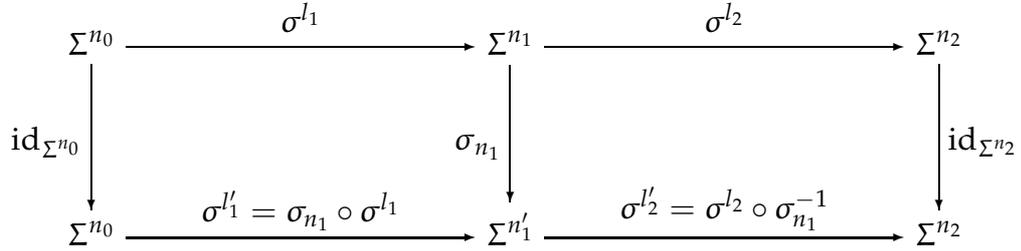


Figure 6.3: Renaming signature items

to s_2 . Note that we have assumed that s_2 is not visible in Σ^{n_1} , so σ_{n_1} is a bijective symbol mapping and the inverse mapping $\sigma_{n_1}^{-1}$ exists. With the definitions

$$\sigma'^{l_1} = \sigma_{n_1} \circ \sigma^{l_1} \circ \text{id}_{\Sigma^{n_0}}^{-1} \quad \text{and} \quad \sigma'^{l_2} = \text{id}_{\Sigma^{n_2}} \circ \sigma^{l_2} \circ \sigma_{n_1}^{-1} \quad (6.1)$$

the diagram in Figure 6.3 commutes. An additional case that needs to be considered is a link from n_1 to n_1 itself. In this case, the identities in (6.1) have to be replaced by σ_{n_1} . All links that do not have n_1 as their source or target node keep their signature morphisms. Together this ensures that (4.16) on page 89 commutes for every link. The inverse of σ_{n_1} is easy to compute in practice for the given construction of σ_{n_1} from the identity $\text{id}_{\Sigma^{n_1}}$.

This defines a development graph translation, where all nodes except for n_1 are mapped along the identity, and n_1 is mapped along the isomorphisms σ_{n_1} – incidentally, this also satisfies the conditions that are required for signature adjustments.

6.3.2 Function and Predicate Arities

As an example for changes to function or predicate arities, we consider a transformation that adds an additional argument to a function. Let f be a function from s to t in the theory n , e.g. the specification includes

```
(theory n
  (uses ...) (satisfies ...)
  ...
  (op f (s) t)
  ...
  (axiom ... (f τ) ...)
  ...)
```

6.3. Changing Elements

For ease of presentation, we assume that f is not a constructor (constructors are discussed in Section 6.3.3).

Changing the domain of f from s to $s \times u$ necessitates

1. changing the declaration of f to $(\text{op } f \ (s \ u) \ t)$,
2. replacing all occurrences of the form $(f \ \tau)$ in subterms of the axioms of theory n by $(f \ \tau' \ \tau_2)$ for some τ_2 of type u , where τ' is obtained from τ by replacing other occurrences $(f \ \dots)$ by $(f \ \dots \ \tau_2)$ recursively), and
3. doing the same procedure for all theories that use the theory n or satisfy n , or that are used or satisfied by n .

In our reference instantiation we have chosen to make the term τ_2 a parameter of the transformation, which has to be provided by the user.

Assume that we have successfully dealt with the first two items of the above list. For the third, we again have to follow the dependencies from n to other theories. In each other theory m that uses n , the function f is mapped to a function with the same or a different name according to the symbol map; assume it is mapped to f' . Similarly, s , t , u , and τ_2 are mapped to some items s' , t' , u' , and term τ'_2 . In order for the specification to be well-defined, we need to add an argument to f' because otherwise the symbol mapping from a binary function f to the unary one f' would not be well-defined. This works exactly as described for f above, except that there is the possibility that m does not include a declaration of f' : when an item is inherited via a *uses* clause, it may be declared redundantly, cf. Section 5.3.1. Similarly, when a theory k satisfies n , then f is mapped to an operation, say f'' , and that operation needs to have an additional argument as well for the *satisfies* clause to be well-formed. Thus, adding an argument is propagated along *uses* and *satisfies* clauses. Again, this involves finding a fixpoint since the graph may be circular.

So far, we have only considered following the dependencies along the direction of the symbol mapping but not backwards. Assume that n uses k with the symbol mapping $g \mapsto f$, and that k declares a unary function g . After the argument has been added to f , this symbol mapping is no longer well-formed: it maps a unary function to a binary one. Thus, we need to add an argument of type u' to g in k , using a term τ'_2 such that the image of u' is u and the image of τ'_2 is τ . Both u' and τ'_2 do not necessarily exist, and if they exist they may not be unique. It is a design decision what to do

if they do not exist or are not unique.³

At any rate, propagating these changes along the dependencies can lead to the case where several functions, not just one, in one theory need to have additional arguments. We characterise this transformation over the whole specification by a set of tuples (n, f, τ) , where n is a theory, f is a function, and τ is a term. The semantics of such a tuple (n, f, τ) is that in node n , an argument of the type of τ is added to the function f . Some of these tuples were given initially, e.g. by the user, and others were added due to the closure conditions of propagating the changes along uses and satisfies clauses which we have described above. We say that the whole set is the transformation determined by the initially given changes.

At this point, there are several design decisions that need to be made.

- If an argument needs to be added to a function f in a theory n as a consequence of propagating the transformation along links (i.e. the respective tuple was not given initially), and if the function is declared redundantly: should this be allowed or should this be an error with a note to the user that the explicitly defined f should also get an additional argument?
- Should the change be propagated along dependencies when this involves going backwards over symbol mappings, provided suitable τ_2 and u' exist? If none exist, an error is raised.
- If yes, what should be done if there is more than one u' and τ_2' with the required image?

However, no matter how these questions are answered, the result of the propagation scheme is a set G of tuples (n, f, τ) describing the changes that are carried out to the nodes of the specification, or an error. However, there is no guarantee that the resulting set is consistent, i.e. that carrying out all the changes results in a well-formed specification. If it does not, the transformation determined by the initial changes is not applicable to the specification, and trying to carry it out will also stop with an error.

Propagation to Development Graphs. Transformations changing existing signature items are realized on the level of development graphs by translations. If the specification transformation represented by the set G of tuples produces a well-formed specification, it can be used to compute a development graph translation such that transforming the specification

³We have chosen to abort the transformation in this case.

6.3. Changing Elements

and translating the development graph commute. For each node n , we collect all pairs (f, τ) for which $(n, f, \tau) \in G$, and this set C_n describes how the node corresponding to the theory needs to be translated.

In order to express the development graph translation we require appropriate **XSig**-arrows. Thus, for all signatures Σ , for each set C of tuples (f, τ) such that f is an operation in Σ , τ is a Σ -term, and C is the graph of a partial function, and all signatures Σ' that are like Φ^n except for each tuple $(f, \tau) \in C$, f has an additional argument of the type τ in Σ' , an **XSig**-arrow $addarg(C) : \Sigma \rightarrow \Sigma'$ is defined.⁴ The intuition is that $addarg(C)$ adds an argument of the type of $\tau = C(f)$ ⁵ to the arity of f in Σ , if f is in the domain of C . Two arrows are identical iff the sets are identical. For $C = \emptyset$, $addarg(C) = id_\Sigma$ is considered as the identity, although this does not matter for the application. In the way described in Section 4.4 on page 94, closure conditions ensure that the resulting **XSig** is an extended category of signatures.

Also, according to Section 4.4, it is enough to define the value of **XSen** for each of the explicitly given translations: $\mathbf{XSen}(addarg(C))$ is the mapping E_C from Σ - to Σ' -formulae that results from recursively replacing subterms of the form $(f \ \tau_1 \ \dots \ \tau_n)$ by $(f \ E_C(\tau_1) \ \dots \ E_C(\tau_n) \ \tau)$:

$$E_C((f \ \tau_1 \ \dots \ \tau_n)) = \begin{cases} (f \ E_C(\tau_1) \ \dots \ E_C(\tau_n) \ \tau) & \text{if } (f, \tau) \in C \\ (f \ E_C(\tau_1) \ \dots \ E_C(\tau_n)) & \text{otherwise.} \end{cases} \quad (6.2)$$

Let $\vartheta_n = addarg(C_n)$, where $C_n = \{(f, \tau) \mid (n, f, \tau) \in G\}$ is the set of tuples that apply to the node $n \in N$.⁶ A development graph translation $(h, (\vartheta_n)_{n \in N})$ and the resulting development graph is now constructed as follows (cf. Figure 6.4):

- The signature Σ^n of each node n is mapped to

$$\Sigma^{h(n)} = addarg(C_n)(\Sigma^n).$$

- ϑ_n is applied to the local axioms of each node n , i.e.

$$\Phi^{h(n)} = \mathbf{XSen}(addarg(C_n))(\Phi^n).$$

⁴Note that for each signature Σ , $addarg(C)$ is defined for arbitrary sets of tuples C satisfying the assumptions, not just for concrete sets C_n relative to a given development graph.

⁵ C is the graph of a partial function so $C(f)$ is defined provided f is in the domain.

⁶For convenience we write n for the theory and its associated node.

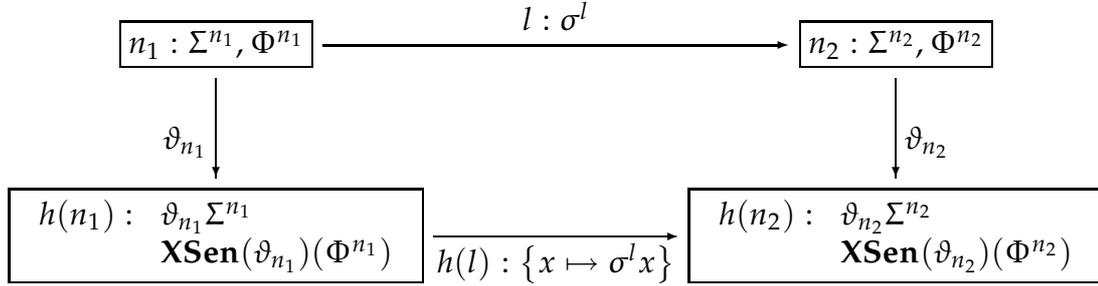


Figure 6.4: Development graph translation for adding arguments

- For each link, the signature morphism is adjusted such that the symbol mappings are unchanged: domain and codomain of the link morphism change, so mapping f to g before and after the change produces two different signature morphisms. When the old signature morphism is σ , we write the new signature morphism as $\{x \mapsto \sigma^l x\}$.

The result is indeed a development graph by construction: this is guaranteed by the condition that C be consistent and applicable to the original specification. The details are tedious and not very interesting, the idea being that consistency over dependencies on the level of specifications is sufficient for consistency over links on the level of development graphs, since each link is derived from a dependency. Furthermore, this ensures that for each link the diagram (4.16) on page 89 commutes as required for development graph translations, and thus our framework propagates the specification transformation to proof obligations.

6.3.3 Generatedness Constraints

Generatedness constraints can be changed in several different ways. We start with changes dealt with by transformations we have already described.

- Entire generatedness constraints can be added or removed. This works similar to adding and removing axioms. On the level of development graphs, generatedness constraints are logical sentences, so the development graphs transformation Tr_{axioms} already covers these specification transformations.
- Single constructors can be changed, e.g., by adding or removing an additional argument to one of the constructors. Adding an argument

to the constructor is analogous to adding an argument to the function as described in Section 6.3.2. Note, that on the level of specifications, development graphs, and proof obligations, it does not matter whether or not the function is a constructor: a generatedness clause reads

```
(gen ... f ...)
```

no matter how many arguments the constructor f has, so there is no need to change the clause when an argument is added. Similarly, generatedness constraints in development graphs and proof obligations refer to the constructor function and thus the change to the generatedness constraint is effected implicitly.

Adding or deleting constructors is a transformation that is not dealt with by another transformation. Consider a theory n , in which a generatedness constraint is changed by making a non-constructor function become a constructor.

Example 6.5 The theory includes an operation declaration for `chmod` and a generatedness clause, e.g.

```
(op chmod (subject filename) command)
(gen write read create remove)
```

and the generatedness clause is then changed to

```
(gen write read create remove chmod)
```

meaning that the operation `chmod` is now also a constructor for the type `command`. ◦

Note that, contrary to changing the arity of an operation, adding a constructor does not change the visible signatures. It implicitly changes the induction schema that is encoded in the institution as a generatedness constraint, i.e. a sentence. However, inheritance and propagation of sentences is afforded by the development graph mechanism, so we do not need to be concerned with the propagation here explicitly.

Propagation to Development Graphs. The transformation is propagated to development graphs using the transformation Tr_{occ} as described in the following section: all nodes' signatures and axioms, and all links' morphisms are kept, except that in node n the respective generatedness constraint is replaced by the new one.

6.3.4 Formula and Term Occurrences

Formula or term occurrences are changed frequently while working on a specification.

Example 6.6 In the scenario concerned with fault-tolerant systems (cf. Section 2.2.2 on page 22), an additional slot was added to the representation of processes: each process is either up or down. Before this change, the state of a process was modelled as a generated datatype process with the constructor⁷

```
mk-process : option × option × option × option → process .
```

Afterwards the constructor had the profile

```
mk-process : option × option × option × option × boolean → process .
```

Since the sort `process` is intended to be freely generated by the constructor `mk-process`, the specification includes an axiom to the effect that two processes are equal iff all their slots are equal, i.e.

```
(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
      (bb2 option) (ib2 option) (db2 option) (dm2 bag)
      (<=> (= (mk-process bb1 ib1 db1 dm1)
              (mk-process bb2 ib2 db2 dm2))
           (and (= bb1 bb2)
                 (= ib1 ib2)
                 (= db1 db2)
                 (= dm1 dm2)))) .
```

A new parameter is added to the constructor in several steps. *First*, after adding an additional argument to `mk-process`, the axiom reads

```
(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
      (bb2 option) (ib2 option) (db2 option) (dm2 bag)
      (<=> (= (mk-process bb1 ib1 db1 dm1 boolean-dummy)
              (mk-process bb2 ib2 db2 dm2 boolean-dummy))
           (and (= bb1 bb2)
                 (= ib1 ib2)
                 (= db1 db2)
                 (= dm1 dm2)))) .
```

⁷The details about the slots of `mk-process` are immaterial here, cf. Chapter 8.

6.3. Changing Elements

where the constant `boolean-dummy` is the parameter of the transformation (cf. (6.2) on page 131) that is inserted to make `mk-process-terms` well-defined. *Second*, two universal quantifications are added to the axioms:

```
(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
      (up1 boolean)
      (bb2 option) (ib2 option) (db2 option) (dm2 bag)
      (up2 boolean)
      (<=> (= (mk-process bb1 ib1 db1 dm1 boolean-dummy)
              (mk-process bb2 ib2 db2 dm2 boolean-dummy))
           (and (= bb1 bb2)
                 (= ib1 ib2)
                 (= db1 db2)
                 (= dm1 dm2)))) .
```

Third, the first occurrence of `boolean-dummy` is replaced by `up1` and the second occurrence by `up2`, which yields

```
(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
      (up1 boolean)
      (bb2 option) (ib2 option) (db2 option) (dm2 bag)
      (up2 boolean)
      (<=> (= (mk-process bb1 ib1 db1 dm1 up1)
              (mk-process bb2 ib2 db2 dm2 up2))
           (and (= bb1 bb2)
                 (= ib1 ib2)
                 (= db1 db2)
                 (= dm1 dm2)))) .
```

Note that the boxed subformula includes a conjunct for each of the old slots, but not for the newly added one. This is corrected, *fourth*, by replacing the boxed part of the axiom by

```
(and (= bb1 bb2)
      (= ib1 ib2)
      (= db1 db2)
      (= dm1 dm2)
      (= up1 up2))
```

thus adding a new clause for the new slot. The effect is to replace the original axiom by a second one that is similar in some way: here a subformula occurrence is replaced by another subformula that shares propositions with the original occurrence.

Chapter 6. Specification Transformations

Similarly, subterm occurrences were changed in the course of the case study. The axiom for the post-condition of one of the state transitions, `b-1p`, originally specified that the processes are unchanged except for process `i`, which has its first slot set to `none`:

```
(all (s state) (i nat) (s1 state)
  (<=> (post s (b-1p i) s1)
    (= (procs s1)
      (update (procs s) i
        (mk-process none
          (bb (aref (procs s) i))
          (db (aref (procs s) i))
          (delivered (aref (procs s) i)))))))
```

Note that all other slots of the process are unchanged: `(aref (procs s) i)` is the process with index `i` before the state transition, and `bb`, `db`, and `delivered` are the selectors for the respective slots. After the additional argument has been added to `mk-process`, the axiom also includes a term `boolean-dummy`. The axiom thus reads

```
(all (s state) (i nat) (s1 state)
  (<=> (post s (b-1p i) s1)
    (= (procs s1)
      (update (procs s) i
        (mk-process none
          (bb (aref (procs s) i))
          (db (aref (procs s) i))
          (delivered (aref (procs s) i))
          boolean-dummy))))))
```

Because `b1-p` is assumed to change only the first slot and leave the others unchanged, the user replaces the boxed occurrence of `boolean-dummy` by the term

```
(is-up (aref (procs s) i))
```

where `is-up` is specified as the selector function for the added slot. Thus the resulting axiom is

```

(all (s state) (i nat) (s1 state)
  (<=> (post s (b-1p i) s1)
    (= (procs s1)
      (update (procs s) i
        (mk-process none
          (bb (aref (procs s) i))
          (db (aref (procs s) i))
          (delivered (aref (procs s) i))
          (is-up (aref (procs s) i)))))))

```

Such changes affect the proofs in which the respective axiom is used. According to the rules of the specification language, the changed axiom is possibly inherited by other theories and thus possibly used by many proofs.

For propagating changes to axioms through the specification, it does not matter how exactly the old and the changed axiom differ, and how they are similar. We can thus ignore the matter here and investigate what happens if an arbitrary axiom φ is replaced by another axiom ψ . We will discuss the issue of similarity and difference in detail when we present the corresponding proof transformations in Section 7.7.

Propagation to Development Graphs. Assume that a subterm or subformula occurrence in an axiom φ in theory n is changed so that the resulting new axiom is ψ . This is propagated to development graphs using the development graph transformation Tr_{occ} . It is easy to see that $q_n = \{(\varphi, \psi)\}$ is a sentence replacement for Φ^n , i.e. the local axioms of the node corresponding to the theory n . Additionally, $q_n(\Phi^n)$ is the set of local axioms specified for the theory n with φ replaced by ψ . Thus, applying Tr_{occ} with an appropriate development graph bijection and empty sentence replacements for all nodes n' except for $n' = n$ will transform the development graphs in a way that ensures that the new graph corresponds to the changed specification. The inheritance of axioms along dependencies is taken care of automatically by the development graph translation.

6.4 Summary

In this chapter we have studied concrete specification transformations for the specification language that we have introduced in Section 5.3. Some transformations on specifications have been described in detail. For each

Chapter 6. Specification Transformations

transformation, its effects on the specification have been described. Further, we have discussed how each specification transformation can be propagated to development graphs: there is a sequence of development graph transformations from those described in Chapter 4 that transform the development graph in a way that corresponds to the change of the specifications. A summary of this relationship is given in Figure 6.5.

Specification transformations are thus linked to proof obligation transformations via development graph transformations. This lets us say for a certain specification transformation which transformations on proof obligations are needed to propagate it to proof obligations. A sketch of the relationship between specification and proof obligation transformations is provided by Figure 6.6.

Transformation	Sect.	Effect on specification Development graph transformation
theories	6.2.1	add/delete isolated theories Tr_{nodes} : add/delete isolated nodes
axioms	6.2.2	add/delete axioms Tr_{axioms} : add/remove local axioms
dependencies	6.2.4	add/delete using/satisfies clauses $Tr_{signatures}$: adjust signatures Tr_{links} : add/delete links
signature items	6.2.3	add/delete signature items $Tr_{signatures}$: adjust signatures
	6.3.1	rename signature items $Tr_{translate}$: map along isomorphism
	6.3.2	change function and predicate arities $Tr_{translate}$: map along translation
	6.3.3	change constructor arities $Tr_{translate}$: map along translation Tr_{occ} : change generatedness constraints
induction schemata	6.3.3	add/remove generatedness constraints Tr_{axioms} : add/delete generatedness constraints
	6.3.3	add/remove constructor Tr_{occ} : change generatedness constraint
occurrences	6.3.4	change occurrence Tr_{occ} : change occurrence

Figure 6.5: Specification transformations

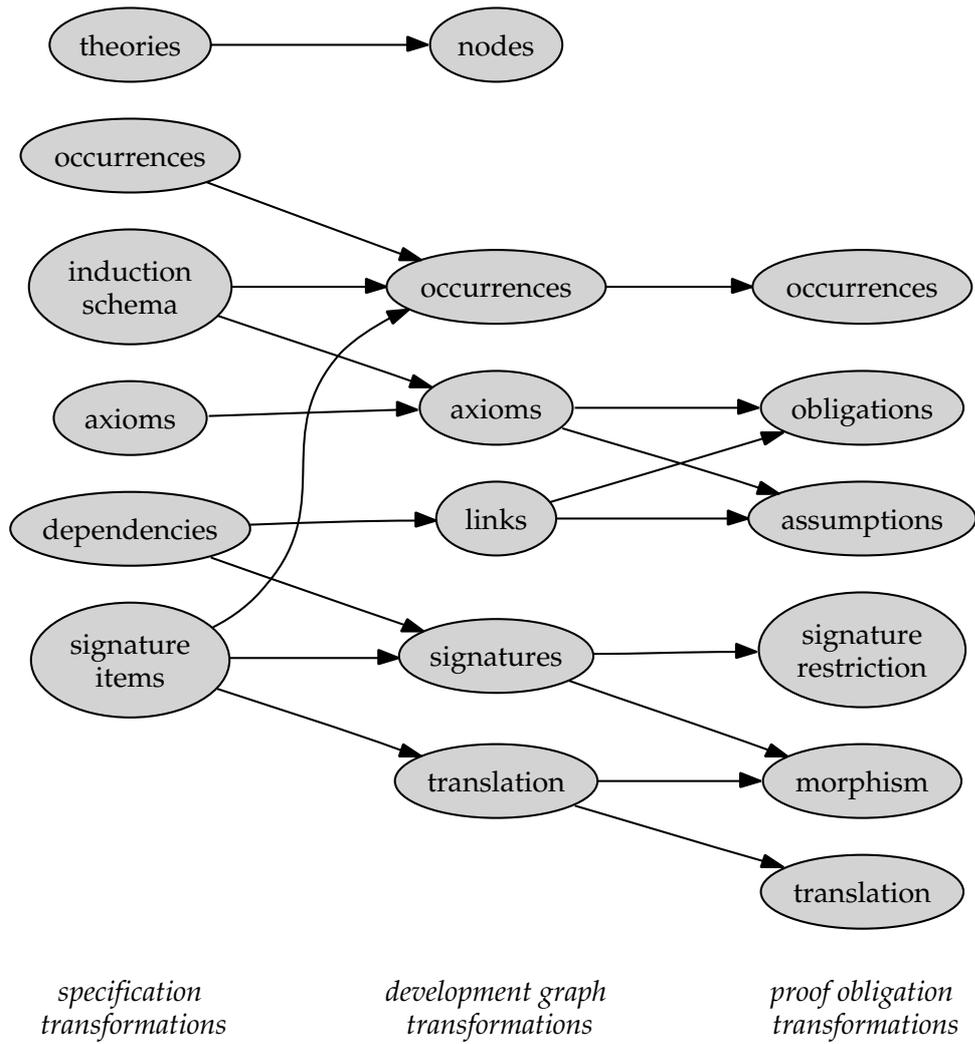


Figure 6.6: Dependency between concrete transformations

Chapter 7

Proof Transformations

7.1 Overview

Specification transformations trigger development graph transformations, and these in turn trigger proof obligation transformations. The relationship between specification and development graph transformations has been described for our concrete instance in the preceding chapter; the relationship between the development graph and proof obligation transformations is fixed by the framework given in Chapter 4. Thus, for the concrete specification transformations that we have introduced in Chapter 6 we have the following transformations on proof obligations (cf. Figure 6.6):

- adding and deleting assumptions
- mapping obligations along signature morphisms
- restricting the signature
- translating obligations
- changing occurrences

This chapter discusses how to transform entire proofs when their conclusion is transformed by one of these proof obligation transformations. As it turns out, further proof transformations that do not change the conclusions of the proofs are needed. These are not triggered by specification transformations; instead they are applied to a given proof by the user directly.

7.2 General Pattern of Proof Transformations

Proof transformations are defined recursively according to the structure of the proof tree. Slightly simplified, a goal transformation is applied to the conclusion of a proof tree; the subproofs are transformed accordingly, and then a new proof is assembled from the new conclusion and the new subproofs. Obviously, this describes a mapping over proofs. The transformation applied to subproofs is not necessarily from the same class of transformations as the one applied to the whole proof in general. Thus, the transformations on subproofs sometimes refer to a different class of transformations and the presentation of the transformations is spread over the chapter. We refrain from presenting this entire transformation formally: we have found that doing so does not provide any benefit but makes understanding the ideas of the transformations very tedious.

The starting point for a transformation is an arbitrary existing proof ξ , called the original proof in the following, with its conclusion:

$$\begin{array}{c} \vdots \xi \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}) . \end{array}$$

Then, we describe how the transformation changes the conclusion, yielding

$$[\vec{x}' : \vec{s}'] \Gamma'(\vec{x}') \vdash \Delta'(\vec{x}') .$$

For any *leaf* $[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})$ the transformed proof is the leaf (\vec{x}) consisting of the new conclusion $[\vec{x}' : \vec{s}'] \Gamma'(\vec{x}') \vdash \Delta'(\vec{x}')$. For *branches*, ξ has the form

$$\frac{\begin{array}{c} \vdots \xi_1 \\ [\vec{x}_1 : \vec{s}_1] \Gamma_1(\vec{x}_1) \vdash \Delta_1(\vec{x}_1) \end{array} \quad \cdots \quad \begin{array}{c} \vdots \xi_k \\ [\vec{x}_k : \vec{s}_k] \Gamma_k(\vec{x}_k) \vdash \Delta_k(\vec{x}_k) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})} \quad j, a \quad (7.1)$$

Depending on the reasoning step j, a of the original proof, the old conclusion, and the transformation, the first reasoning step is adapted for the new proof. This is either a matter of transforming the rule justification arguments a , or it may be much more involved such as constructing an intermediate proof. In any case, the result of transforming the first reasoning step can be represented by an intermediate subproof ξ_0 , yielding a number of open goals:

$$\begin{array}{c} [\vec{x}'_1 : \vec{s}'_1] \Gamma'_1(\vec{x}'_1) \vdash \Delta'_1(\vec{x}'_1) \quad \cdots \quad [\vec{x}'_m : \vec{s}'_m] \Gamma'_m(\vec{x}'_m) \vdash \Delta'_m(\vec{x}'_m) \\ \vdots \xi_0 \\ [\vec{x}' : \vec{s}'] \Gamma'(\vec{x}') \vdash \Delta'(\vec{x}') \end{array}$$

7.3. Adding and Deleting Assumptions

For each of the open leaves $[\vec{x}'_j : \vec{s}'_j] \Gamma'_j(\vec{x}'_j) \vdash \Delta'_j(\vec{x}'_j)$ (for $1 \leq j \leq m$) of ξ_0 we consider the following cases:

- There is an old proof that we want to reuse (and maybe adapt) for the open leaf: We describe how the new premiss results from one of the premisses $[\vec{x}_i : \vec{s}_i] \Gamma_i(\vec{x}_i) \vdash \Delta_i(\vec{x}_i)$ (for some i with $1 \leq i \leq k$) of the original proof ξ . I.e. we provide the old premiss and describe a proof transformation that transforms an arbitrary proof ξ_i with conclusion $[\vec{x}_i : \vec{s}_i] \Gamma_i(\vec{x}_i) \vdash \Delta_i(\vec{x}_i)$ into a proof ξ'_j with conclusion $[\vec{x}'_j : \vec{s}'_j] \Gamma'_j(\vec{x}'_j) \vdash \Delta'_j(\vec{x}'_j)$.
- There is no proof that we want to reuse for the open leaf: We leave the new premiss open as a new open goal in the resulting transformed proof.

For each j for which we leave the goal open in the resulting proof, let ξ'_j be the trivial open proof for the conclusion $[\vec{x}'_j : \vec{s}'_j] \Gamma'_j(\vec{x}'_j) \vdash \Delta'_j(\vec{x}'_j)$. Then the result of transforming the proof (7.1) with the given transformation is

$$\begin{array}{c}
 \vdots \xi'_1 \\
 [\vec{x}'_1 : \vec{s}'_1] \Gamma'_1(\vec{x}'_1) \vdash \Delta'_1(\vec{x}'_1) \quad \dots \quad [\vec{x}'_m : \vec{s}'_m] \Gamma'_m(\vec{x}'_m) \vdash \Delta'_m(\vec{x}'_m) \\
 \vdots \xi'_m \\
 \vdots \xi_0 \\
 [\vec{x}' : \vec{s}'] \Gamma'(\vec{x}') \vdash \Delta'(\vec{x}')
 \end{array}$$

This is a valid proof object if only the intermediate proof object ξ_0 is a valid proof and the recursive transformations on the subproofs produce valid proof objects. We assert that the result is indeed a valid proof object, arguing that the latter condition is the induction hypothesis and the former can be ensured in the usual way by only using valid inference rules. We will present the construction of the intermediate proof in a way that makes clear that this is indeed the case.

7.3 Adding and Deleting Assumptions

Adding an assumption to the conclusion of a proof is simple from a semantic point of view: the assumption weakens the proposition, i.e. the new conclusion is implied logically by the old one. The new assumption can be ignored. On the other hand, the whole point of adding an assumption to a proof is to be able to use it in the proof. This is easy, too: all valid proof steps are still valid proof steps if the same formula is added to the

Chapter 7. Proof Transformations

antecedent of the conclusion and each premiss. As an example, consider the rule and-r:

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \quad \cdots \quad [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_m(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \wedge \cdots \wedge \varphi_m(\vec{x})}$$

becomes

$$\frac{[\vec{x} : \vec{s}] \gamma(\vec{x}), \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \quad \cdots \quad [\vec{x} : \vec{s}] \gamma(\vec{x}), \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_m(\vec{x})}{[\vec{x} : \vec{s}] \gamma(\vec{x}), \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \wedge \cdots \wedge \varphi_m(\vec{x})}$$

with unchanged justification “and-r $\varphi_1(\vec{x}) \wedge \cdots \wedge \varphi_m(\vec{x})$ ”. This holds for each of the rules of the calculus that we have chosen. As can easily be seen, each open goal of the original proof will have the additional formula in the antecedent so that it can be used in any subproof.

The same reasoning applies to adding a formula to the succedent of the conclusion, and it also applies for adding several formulae to the antecedent and/or the succedent.

In other proof representations, even for the same calculus, this need not be as easy to formulate. It is important how the justifications are represented. For instance, the and-r-rule needs to determine the focus formula that is to be decomposed. In our proof representation, the formula is represented in the justification arguments by the formula itself. The formula is invariant under adding formulae to the conclusion and premisses, so the original justifications can be kept. This is in contrast to proof representations in which the focus formula is determined by its position in the sequent (this is the case for the theorem proving component of the VSE system and for PVS) or in which it is determined heuristically (which is the case, e.g. for Isabelle’s proof scripts). The latter does not reliably support the transformation because adding a formula potentially changes the search space of the heuristic and thus different premisses may be generated, the former supports the transformation but more work is needed because formula positions possibly shift when new formulae are added.

Deleting a formula from the antecedent or the succedent is more difficult because this strengthens the proposition. Obviously, a context formula, i.e. a formula that is not used in a rule, can be deleted from each proof rule by deleting it from the conclusion and the premisses, i.e. the example for adding $\gamma(\vec{x})$ to the rule and-r above can be read backwards as removing the formula. When the formula to be removed is not a context formula, it plays some rôle for the applicability of the proof rule, and some patch is required to make the proof rule be valid in general. Assume that

7.3. Adding and Deleting Assumptions

we want to delete $\varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x})$ from the succedent in

$$\frac{\begin{array}{c} \vdots \xi_1 \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \end{array} \quad \dots \quad \begin{array}{c} \vdots \xi_m \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_m(\vec{x}) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x})}$$

This leaves us with the new conclusion $[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})$. Depending on why the conjunct is deleted, there are several possibilities to proceed.

- If the conjunction is not needed at all, we can simply remove the first proof step, remove the conjunct φ_i from one of the premisses, transform the original subproof ξ_i by removing the conjunct from the whole proof with result ξ'_i , and use ξ'_i as subproof:

$$[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}) \quad \begin{array}{c} \vdots \xi'_i \end{array}$$

The user can choose which premiss and proof to use.

- If the conjuncts are used, yet the focus formula is withdrawn as an assumption, we would expect the conjuncts to be derivable from some other assumptions, either existing ones or ones that will be introduced later. A scenario for this is when axioms are withdrawn and a different axiomatisation is introduced.

This is achieved by introducing the conjunct with a cut-rule and then carrying on:

$$(7.2) \quad \frac{\begin{array}{c} \vdots \xi \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_n(\vec{x}) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})} \text{ cut } \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_n(\vec{x})$$

with the new open goal

$$[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_n(\vec{x}) \vdash \Delta(\vec{x}). \quad (7.2)$$

We have chosen to provide the second possibility only, because it seemed to be needed more often in our examples. Note that the result of the first can be achieved nevertheless in our instantiation: First, replace the assumption $\varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x})$ by $\varphi_i(\vec{x})$ (cf. Section 7.7), and then remove the assumption $\varphi_i(\vec{x})$.

7.4 Mapping Proofs

Given a proof ξ and a signature morphism σ , $\mathbf{Prf}(\sigma)(\xi)$ is a proof for $\mathbf{Goal}(\sigma)(\mathit{concl}(\xi))$ according to Figure 3.4 in the definition of a proof representation in Section 3.5. In our concrete proof representation $\mathbf{Prf}(\sigma)(\xi)$ has the same structure as ξ , and thus the same open goals and rule justifications (both mapped along σ , of course).

In the specification transformations of the concrete instance, proof obligations are mapped along signature morphisms as the result of extending signatures or renaming signature items. For the latter, signature items are renamed in goals and rule arguments; for the former, goals and rule arguments are injected into the extended language.

7.5 Restricting the Signature

In our concrete instance, a signature restriction from Σ to Σ' is a symbol mapping $\sigma : \Sigma' \rightarrow \Sigma$ that is injective. The restriction of σ to its range is thus trivially bijective. So there exists an inverse symbol mapping from the range of the mapping to its domain. Since symbol mappings only map finitely many symbols to other symbols, both the inverse $\hat{\sigma}$ and its domain $\hat{\Sigma} \subseteq \Sigma$ are easy to determine.

Now let us assume that we want to restrict the signature of the Σ -proof ξ along σ . We determine the σ -range $\hat{\Sigma}$ of Σ . If $\xi \in \mathbf{Prf}(\hat{\Sigma})$ then $\mathbf{Prf}(\hat{\sigma})(\xi) \in \mathbf{Prf}(\Sigma')$ is the transformed proof. Otherwise, the proof uses symbols that would be removed by the restriction and the transformation is not applicable.

7.6 Translating Proofs

In our reference instantiation, the only proof obligation translations that we have described explicitly are concerned with adding arguments to function and predicate arities. We will concentrate on function arities in the following, changing predicates arities works out similarly. Recall that if C is a partial function from functions in Σ to terms in Σ , $\vartheta = \mathit{addarg}(C) : \Sigma \rightarrow \Sigma$ is the translation that adds an argument to the arity of all functions in the domain of C . The sentence translation $\mathbf{XSen}(\mathit{addarg}(C))$ adds the term $C(f)$ as an additional argument to all terms constructed with f (cf. E_C as defined by (6.2) on page 131).

Similarly, we define $\mathbf{XGoal}(addarg(C))$ to be the translation that replaces the terms constructed with f in the sentences of a proof goal according to E_C . As a consequence, mapping formulae and goals along $addarg(C)$ and constructing proof goals from the formulae commutes.

Finally, we define $\mathbf{XPrf}(addarg(C))$ to map a proof

$$\frac{\begin{array}{ccc} \vdots \xi_1 & & \vdots \xi_m \\ \Theta_1 & \cdots & \Theta_m \end{array}}{\Theta} j, a \quad (7.3)$$

to

$$\frac{\begin{array}{ccc} \vdots \mathbf{XPrf}(\vartheta)(\xi_1) & & \vdots \mathbf{XPrf}(\vartheta)(\xi_m) \\ \mathbf{XGoal}(\vartheta)(\Theta_1) & \cdots & \mathbf{XGoal}(\vartheta)(\Theta_m) \end{array}}{\mathbf{XGoal}(\vartheta)(\Theta)} j, a' \quad (7.4)$$

for $\vartheta = addarg(C)$, where $a' = E_C(a)$ is the sequence of justification arguments, i.e. terms or formulae, that results from replacing f -subterms according to C . As a consequence, translating proof goals and proofs and taking the conclusion commutes. Thus, the diagram given in Figure 4.9 on page 92 commutes as required for an extended proof representation.

In the statement of $\mathbf{XPrf}(addarg(C))$ we claim implicitly that the result is a valid proof. Under the assumption that none of the functions in C is a constructor, this is indeed the case for the concrete proof representation that we have chosen. The detailed proof is very tedious, though. The main idea is that the applicability of proof rules is based on unification of terms and formulae, and that unification is invariant under adding the same term $C(f)$ to all f -terms. Adding arguments to constructors or removing them from constructors additionally changes the associated induction schema, and requires further patches, cf. Section 7.7.3.

Another interesting translation is concerned with removing an argument from a function arity, i.e. going from $f : t_1 \times \cdots \times t_{m-1} \times t_m \rightarrow t$ to $f : t_1 \times \cdots \times t_{m-1} \rightarrow t$ and removing the last argument from any term constructed with f . There are two differences to $addarg$. *First*, two terms or formulae may unify after the argument has been removed but fail to unify before. An example is the pair of formulae $(p (f a b))$ and $(p (f a c))$, which both become $(p (f a))$. If these two formulae appear both in, say, the antecedent of the conclusion of a proof then after the transformation of the goal these formulae coincide. This needs to be propagated to the premisses. *Second*, the argument term that is removed may have been essential for the applicability of an equation. An example is when

the equation $(= b (g c))$ is applied to $(p (f a b))$ to yield $(p (f a (g c)))$, and then the second argument of f is removed. The term that is to be rewritten does not exist anymore, and thus the rewriting is redundant. Our proof representation is set up such that this degenerate case of rewriting is already allowed and no new reasoning steps need to be introduced. The consequence is that the proof transformation of removing arguments can be expressed by (7.3) and (7.4) for an appropriate signature translation ϑ , and thus can be formulated as a translation of the extended proof representation.

In both cases, the proof $\mathbf{XPrf}(\vartheta)(\xi)$ that results from adding an argument to a function or predicate or removing one has the same structure as the original proof ξ , and conclusion and all open goals of the new proof result from conclusion and corresponding open goal of the original proof by mapping the goal along ϑ , with the exception of patches due to changing induction schemata.

7.7 Changing Occurrences

Changing subterm or -formula occurrences in the specification is propagated to proof obligations according to Section 4.3.2 with the aid of sentence replacements. Recall that a sentence replacement q for a set Γ of Σ -sentences is a relation $q \subseteq \Gamma \times \mathbf{Sen}(\Sigma)$. A pair $(\varphi, \psi) \in q$ means that $\varphi \in \Gamma$ is to be replaced by ψ . Also recall that the relation q over sentences can be interpreted as a function over sets of sentences (cf. Definition 4.7):

$$q(\Gamma) = (\Gamma \setminus \{\varphi \mid (\varphi, \varphi') \in q\}) \cup \{\varphi' \mid (\varphi, \varphi') \in q\} .$$

According to Theorem 4.11, each proof obligation

$$\Phi_D^{n_2} \models_{\Sigma^{n_2}} \mathbf{Sen}(\sigma)(\varphi)$$

is changed by Tr_{occ} to another proof obligation

$$l(\Phi_D^{n_2}) \models_{\Sigma^{n_2}} r(\mathbf{Sen}(\sigma)(\varphi)) .$$

for two sentence replacements l and r determined by Tr_{occ} .¹ The proof for the proof obligation needs to be changed accordingly.

In the following, we discuss how such replacements are applied to proofs, i.e. how they are propagated up through proof trees. First the general scheme (cf. Section 7.7.1) and then special cases (cf. Section 7.7.2) are

¹In terms of Theorem 4.11, the replacement l is $q_{n_2}^*$ in (4.6), and r is the relation $\{(\mathbf{Sen}(\sigma)(\varphi), \mathbf{Sen}(\sigma)(\varphi'))\}$ with φ taken from (4.5) and φ' from (4.6).

presented. In both cases, we distinguish those situations in which the replacements do not essentially influence the proof rule that is applied, and those in which the rules are no longer applicable after the replacement. For the former, the rule is applied and the replacements are inherited by the premisses and the rule arguments. For the latter, some patching is needed before the replacements can be propagated. Formally, there is always a patch. It cannot be adequate in all situations, though.

7.7.1 Replacing Occurrences

We start out with a proof ξ that has the conclusion $\Gamma \vdash \Delta$ and in which we want to replace occurrences of formulae in the antecedent and in the succedent. The conclusion $\Gamma \vdash \Delta$ of the original proof ξ is thus transformed into $l(\Gamma) \vdash r(\Delta)$ for two given sentence replacements l and r . We propagate these changes over the whole proof. In the course we introduce Eigenvariables, and the changes will need to honour them. Therefore we generalise the problem description to proof obligations and replacements which refer to Eigenvariables. If we make these explicit in the sentence replacements then the original conclusion

$$[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})$$

is changed to

$$[\vec{x} : \vec{s}] l(\vec{x})(\Gamma(\vec{x})) \vdash r(\vec{x})(\Delta(\vec{x})) .$$

If the original proof ξ is a leaf we are done. Otherwise we have to deal with the question of how to propagate the sentence replacements $l(\vec{x})$ and $r(\vec{x})$ over the first rule step so that we can transform the subproofs for the premisses recursively.

Of course, how a single change can be propagated from the conclusion to the premisses depends on the proof rule that was applied and on whether the change applies to the context or a focus formula. Additionally, we need to take into account how the effects play together for several changes. One main problem with this is the large number of resulting cases. These cannot simply be treated one by one in practice, not even for our simple, lean calculus. The number of special cases can be reduced by separating the changes that apply to the context (cf. Section 7.7.1.1) and those that apply to the focus formulae (cf. Section 7.7.1.2), and then combining the effects (cf. Section 7.7.1.3).

7.7.1.1 Context

Propagating changes over rules is easy for context formulae: since the context of reasoning steps is unchanged in the premisses, the changes can be applied to the context of the premisses. As an example consider a proof starting with the application of an and-1 rule

$$\frac{\begin{array}{c} \vdots \xi_1 \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}), \dots, \varphi_m(\vec{x}) \vdash \Delta(\vec{x}) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \vdash \Delta(\vec{x})} \text{ and-1 } \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \quad (7.5)$$

which we want to transform by applying two sentence replacements $l(\vec{x})$ and $r(\vec{x})$ to the context of the conclusion. Applying the sentence replacements to the context of the conclusion yields

$$[\vec{x} : \vec{s}] l(\vec{x})(\Gamma(\vec{x})), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \vdash r(\vec{x})(\Delta(\vec{x}))$$

Applying the same sentence replacements to the context of the premiss yields a correct application of the and-1-rule:

$$\frac{[\vec{x} : \vec{s}] l(\vec{x})(\Gamma(\vec{x})), \varphi_1(\vec{x}), \dots, \varphi_m(\vec{x}) \vdash r(\vec{x})(\Delta(\vec{x}))}{[\vec{x} : \vec{s}] l(\vec{x})(\Gamma(\vec{x})), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \vdash r(\vec{x})(\Delta(\vec{x}))}$$

with justification and-1 $\varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x})$. This is obvious since both $l(\vec{x})(\Gamma(\vec{x}))$ and $r(\vec{x})(\Delta(\vec{x}))$ are sets of formulae, say $\Gamma'(\vec{x})$ and $\Delta'(\vec{x})$, and thus the new reasoning step is an instance of the and-1-rule:

$$\frac{[\vec{x} : \vec{s}] \Gamma'(\vec{x}), \varphi_1(\vec{x}), \dots, \varphi_m(\vec{x}) \vdash \Delta'(\vec{x})}{[\vec{x} : \vec{s}] \Gamma'(\vec{x}), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \vdash \Delta'(\vec{x})}$$

The problem of applying the sentence replacements $l(\vec{x})$ and $r(\vec{x})$ to the context in the proof has been reduced to the problem of applying the sentence replacement to the context of the conclusion and the problem of applying the sentence replacement to the context of the subproof ξ_1 ; if we write ξ'_1 for the result of the latter, the resulting transformed proof is

$$\frac{\begin{array}{c} \vdots \xi'_1 \\ [\vec{x} : \vec{s}] l(\vec{x})(\Gamma(\vec{x})), \varphi_1(\vec{x}), \dots, \varphi_m(\vec{x}) \vdash r(\vec{x})(\Delta(\vec{x})) \end{array}}{[\vec{x} : \vec{s}] l(\vec{x})(\Gamma(\vec{x})), \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_m(\vec{x}) \vdash r(\vec{x})(\Delta(\vec{x}))}$$

Note that $l(\vec{x})$ does not apply to the whole antecedent of the conclusion or premiss, but to the context $\Gamma(\vec{x})$ only, cf. Section 7.7.1.3.

7.7.1.2 Focus Formulae

Changes to focus formulae are relevant for the question of whether the bottom-most proof rule is applicable to the changed conclusion. We distinguish two cases: either the focus formula is changed in a way such that the rule remains applicable (*deep focus*), in general with slight changes, or the rule is not applicable as is (*immediate focus*). These are investigated separately.

In the problem formulation above, we have implied that the focus formula is replaced by a single formula. However, recall that sentence replacements are binary relations, and thus the focus formula φ may be in relation to several formulae, e.g. as in $f(\{\varphi\}) = \{\psi_1, \psi_2\}$ for $f = \{(\varphi, \psi_1), (\varphi, \psi_2)\}$. As we will see in Section 7.7.1.3, for focus formulae it is enough to consider the case, where f is a partial function on sentences, i.e. the result of applying the sentence replacement f to the singleton $\{\varphi\}$ is again a singleton. Therefore, we assume that the sentence replacements for the focus formulae presented in the following are right-unique.²

Deep Focus. For most rules, only the outermost structure of the focus formula matters for the applicability of the proof rule. If changes to the focus formula keep this structure intact, the rule is still applicable. We use the slogan *deep focus* to characterise this situation. For instance, assume that the bottom-most proof rule is *and-r*. The only focus formula is the conjunction in the succedent, thus $\varphi_1(\vec{x}) \wedge \cdots \wedge \varphi_m(\vec{x}) = \varphi(\vec{x})$. Recall the assumption that $f(\vec{x})$ is a sentence replacement for the focus formula $\varphi(\vec{x})$, and that $f(\vec{x})(\{\varphi(\vec{x})\})$ is a singleton. By our assumptions, the replacement $f(\vec{x})$ for the focus does not change the outer structure, and thus the single member of the singleton $\{\psi(\vec{x})\} = f(\vec{x})(\{\varphi(\vec{x})\})$ is a conjunction with m conjuncts, i.e. it has the form

$$\psi(\vec{x}) = \psi_1(\vec{x}) \wedge \cdots \wedge \psi_m(\vec{x}) .$$

Let m sentence replacements $f_i(\vec{x})$ for the singleton sets $\{\varphi_i(\vec{x})\}$ ($1 \leq i \leq m$) be defined by

$$f_i(\vec{x}) = \{(\varphi_i(\vec{x}), \psi_i(\vec{x}))\} \quad (1 \leq i \leq m) . \quad (7.6)$$

²Theorem 7.2 shows that this is a useful assumption; Remark 7.4 discusses the practical consequences.

Chapter 7. Proof Transformations

Given an original proof

$$\frac{\begin{array}{c} \vdots \xi_1 \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \end{array} \quad \cdots \quad \begin{array}{c} \vdots \xi_m \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_m(\vec{x}) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi_1(\vec{x}) \wedge \cdots \wedge \varphi_m(\vec{x})}$$

with justification “and-r $\varphi_1(\vec{x}) \wedge \cdots \wedge \varphi_m(\vec{x})$ ” we can write the transformed first proof step as³

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f_1(\vec{x})(\varphi_1(\vec{x})) \quad \cdots \quad [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f_m(\vec{x})(\varphi_m(\vec{x}))}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f(\vec{x})(\varphi_1(\vec{x}) \wedge \cdots \wedge \varphi_m(\vec{x}))}$$

Using the definitions (7.6), this reduces to

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \psi_1(\vec{x}) \quad \cdots \quad [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \psi_m(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \psi_1(\vec{x}) \wedge \cdots \wedge \psi_m(\vec{x})}$$

which is again a valid instance of the and-r-rule, namely with justification “and-r $\psi_1(\vec{x}) \wedge \cdots \wedge \psi_m(\vec{x})$ ”. Obviously, similar schematic rules can be written for the other *structural* and *propositional* calculus rules.

Again, we have reduced the question of how to apply the sentence replacement to a formula in the proof to the question of how to apply the replacement to a formula in the conclusion and how to apply other replacements f_i to the subproofs ξ_i . If the respective result of the latter is ξ'_i , the transformed proof is

$$\frac{\begin{array}{c} \vdots \xi'_1 \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f_1(\vec{x})(\varphi_1(\vec{x})) \end{array} \quad \cdots \quad \begin{array}{c} \vdots \xi'_m \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f_m(\vec{x})(\varphi_m(\vec{x})) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f(\vec{x})(\varphi_1(\vec{x}) \wedge \cdots \wedge \varphi_m(\vec{x}))}$$

For *predicate calculus* rules the propagation works similar but has to deal with the scope of Eigenvariables and the fact that terms are substituted into the focus formulae. As an example, we consider the all-r rule. Assume that

$$\frac{\begin{array}{c} \vdots \xi_1 \\ [\vec{x} : \vec{s}, y : s] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}, y) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \forall y : s. \varphi(\vec{x}, y)} \text{ all-r}$$

is a correct proof that is to be transformed according to the sentence replacement $f(\vec{x})$ for the focus formula. Since we assume that the relevant

³From hereon, when we apply a sentence replacement f to a singleton set $\{\varphi\}$, we sometimes write $f(\varphi)$ as an abbreviation for $f(\{\varphi\})$.

7.7. Changing Occurrences

formula structure is unchanged, the new focus formula $f(\vec{x})(\forall y : s. \varphi(\vec{x}, y))$ can be written as $\forall y : s. \psi(\vec{x}, y)$, and thus with the definition

$$f_1(\vec{x}, y) = \{(\varphi(\vec{x}, y), \psi(\vec{x}, y))\}$$

the resulting proof is

$$\frac{[\vec{x} : \vec{s}, y : s] \Gamma(\vec{x}) \vdash \overset{\xi'_1}{\Delta(\vec{x}), f_1(\vec{x})(\varphi(\vec{x}, y))}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f(\vec{x})(\forall h : s. \varphi(\vec{x}, h))}$$

with justification “all-r $f(\vec{x})(\forall y : s. \psi(\vec{x}, y))$ ”, and for ξ'_1 being the proof that results from ξ_1 by propagating $f_1(\vec{x}, y)$ up through the subproof ξ .

For the ex-r rule the witness term can be handled similarly. An additional complication results from the fact that the quantified formula is retained in addition to the instantiated one. This is necessary for reasons of relative completeness of the sequent calculus: in general, several instantiations of one formula are required. In an original proof with first proof step

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \exists x : s. \varphi(\vec{x}, x), \varphi(\vec{x}, t(\vec{x}))}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \exists x : s. \varphi(\vec{x}, x)}$$

and with justification “ex-r $\exists x : s. \varphi(\vec{x}, x), t(\vec{x})$ ”, we can write the replacement focus formula⁴ $f(\vec{x})(\exists x : s. \varphi(\vec{x}, x))$ as $\{\exists x : s. \psi(\vec{x}, x)\}$, and with the definition

$$f_1(\vec{x}) = \{(\varphi(\vec{x}, t(\vec{x})), \psi(\vec{x}, t(\vec{x})))\}$$

the new proof reads

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f(\vec{x})(\exists x : s. \varphi(\vec{x}, x)), f_1(\vec{x})(\varphi(\vec{x}, t(\vec{x})))}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f(\vec{x})(\exists x : s. \varphi(\vec{x}, x))}$$

with justification “ex-r $f(\vec{x})(\exists x : s. \varphi(\vec{x}, x)), t(\vec{x})$ ”. Induction rules work similarly: each of the occurrences of φ in the induction formula (5.4) needs to be replaced by ψ .

For the remaining rules it is more difficult to decide whether a given replacement leaves the rule applicable. Consider any of the *axiom* rules, e.g.

$$\overline{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x})} \text{ basic}$$

⁴Recall that we assumed that the sentence replacement on focus formulae map the singleton consisting of the focus formula to another singleton set.

Chapter 7. Proof Transformations

The applicability of the rule depends on the two occurrences of $\varphi(\vec{x})$ in the antecedent and the succedent to be occurrences of the same formula. If we replace $\varphi(\vec{x})$ by $\psi(\vec{x})$ the original rule will only be applicable if they are the same formulae. It is not enough to keep the outermost formula structure: the whole formula is required to be the same. On the other hand, if both occurrences are changed then the rule is still applicable. Checking whether this is the case is not much cheaper than checking for the applicability of the basic rule in the first place. Thus, a proof consisting of an application of a basic rule is transformed by transforming the conclusion and then checking whether any of the basic rules is applicable. If it is then the proof is closed, otherwise the conclusion is left as an open goal.

Finally, dealing with the *equality rules* is the most involved. The equivalent of retaining the relevant outer structure has been taken in our reference instantiation to be the following conditions, for the rule⁵

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), t_1(\vec{x}) = t_2(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}, t_2(\vec{x}))}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), t_1(\vec{x}) = t_2(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}, t_1(\vec{x}))}$$

with justification “eqn-1 $t_1(\vec{x}) = t_2(\vec{x}), \varphi(\vec{x}, t_1(\vec{x})), \varphi(\vec{x}, t_2(\vec{x}))$ ” that applies the equation from left to right:

- The equation that is applied as a rewrite rule retains the outer structure, i.e. remains an equation:

$$f_1(\vec{x})(t_1(\vec{x}) = t_2(\vec{x})) = \{t'_1(\vec{x}) = t'_2(\vec{x})\} .$$

- The subterm occurrence that is rewritten is changed in the same way as the left hand side of the equation:

$$f_2(\vec{x})(\varphi(\vec{x}, t_1(\vec{x}))) = \{\psi(\vec{x}, t'_1(\vec{x}))\} .$$

Thus, the new rule

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), t'_1(\vec{x}) = t'_2(\vec{x}) \vdash \Delta(\vec{x}), \psi(\vec{x}, t'_2(\vec{x}))}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), t'_1(\vec{x}) = t'_2(\vec{x}) \vdash \Delta(\vec{x}), \psi(\vec{x}, t'_1(\vec{x}))}$$

can, using the definition

$$f_3(\vec{x}) = \{(\varphi(\vec{x}, t_2(\vec{x})), \psi(\vec{x}, t'_2(\vec{x})))\}$$

⁵Note that equality rules have two focus formulae, and thus we give two replacements f_1 and f_2 for the conclusion, one for each focus formula.

be written as

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), f_1(\vec{x})(t_1(\vec{x}) = t_2(\vec{x})) \vdash \Delta(\vec{x}), f_3(\vec{x})(\varphi(\vec{x}, t_2(\vec{x})))}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), f_1(\vec{x})(t_1(\vec{x}) = t_2(\vec{x})) \vdash \Delta(\vec{x}), f_2(\vec{x})(\varphi(\vec{x}, t_1(\vec{x})))}$$

with justification

$$\text{eqn-1 } f_1(\vec{x})(t_1(\vec{x}) = t_2(\vec{x})), f_2(\vec{x})(\varphi(\vec{x}, t_1(\vec{x}))), f_3(\vec{x})(\varphi(\vec{x}, t_2(\vec{x})))$$

It is obvious that this is a correct proof rule instance, although the fact is hidden in all the book-keeping details. The other equational rules are treated analogously.

Immediate Focus. If none of the cases that we have called *deep focus* are applicable, one of the focus formulae is changed in a way that prevents the first proof rule from being applied to the conclusion of the proof. I.e. the bottom-most rule of the original proof is no longer applicable after the replacement has been carried out on the conclusion of the proof. We categorise these by *immediate focus*: the most important example is changing the outermost formula structure of the focus formula in propositional or quantifier calculus rules. For example, if the focus formula of an and-r rule is changed such that the result is not a conjunction, then the rule is no longer applicable. What should be done is dependent to a certain extent on the rule that was applied in the original proof (the and-r rule here), but even more on the way in which the replacements have changed the focus formulae. Support tailored to specific changes is the subject of Section 7.7.2 below. If none of the special cases described there is applicable, a uniform way of handling this is to give up at this point. If a focus formula φ is changed to an arbitrary other formula ψ then there is not much that we can do to patch the proof. The best bet seems to leave the resulting new sequent as an open goal so that the user can work on it later. However, this would throw away the old proof above the point in the proof at which we give up. This can be avoided by relating the old and new focus formula by a cut rule. I.e. if our original proof was

$$\frac{\vdots \xi}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x})}$$

with focus formula $\varphi(\vec{x})$, and $f(\vec{x})(\varphi(\vec{x})) = \{\psi(\vec{x})\}$, the resulting proof is

$$(7.8) \quad \frac{\vdots \xi' \quad [\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x}), f(\vec{x})(\varphi(\vec{x}))}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f(\vec{x})(\varphi(\vec{x}))} \quad (7.7)$$

Chapter 7. Proof Transformations

with premiss

$$[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f(\vec{x})(\varphi(\vec{x})), \varphi(\vec{x}) \quad (7.8)$$

and justification “gap-r $\varphi(\vec{x}), f(\vec{x})(\varphi(\vec{x}))$ ”, where ξ' results from ξ by adding the formula $f(\vec{x})(\varphi(\vec{x}))$ to the succedent. The second premiss is an open goal to the effect that the old focus formula implies the new one in the given context. Either this goal needs to be closed, or the proof needs to be changed again later. Note that with this construction, the subproof ξ' is part of the new proof and is not thrown away. The user can take it as the basis for, e.g., a heuristic proof replay.

Remark 7.1 Although this step is an application of the cut rule, we have found it helpful to distinguish cut rules that have been introduced in this way (gap-l and gap-r) from the ones that the user has explicitly chosen (cut). This has the benefit that artefacts of transformations can be distinguished from genuine applications of the cut rule. In particular, gap steps should be silently removed from proofs when the two formulae in its justification, i.e. $\varphi(\vec{x})$ and $f(\vec{x})(\varphi(\vec{x}))$ in the justification of (7.7), are made equal by a subsequent transformation. As far as propagation of replacements over gap-l and gap-r rules are concerned, the first argument acts as a genuine cut formula and can be changed by the user (cf. Section 7.8) whereas the second argument acts as a focus formula similarly to propositional and quantifier rules, i.e. the justification is changed when the respective formula in the conclusion is replaced.

7.7.1.3 Focus and Context

Let us now consider how sentence replacements that change both the focus and the context propagate over entire proofs. Thus, let ξ be a proof with conclusion

$$\Theta = [\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}) \quad (7.9)$$

and let two sentence replacements $l(\vec{x})$ for $\Gamma(\vec{x})$ and $r(\vec{x})$ for $\Delta(\vec{x})$ be given. Assume that we want to apply these to ξ so that the conclusion becomes $[\vec{x} : \vec{s}] l(\vec{x})(\Gamma(\vec{x})) \vdash r(\vec{x})(\Delta(\vec{x}))$. The previous sections describe how to apply replacements to the context and to the focus formulae. In general, the replacements $l(\vec{x})$ and $r(\vec{x})$, however, apply to both context and focus.

We divide the sentence replacements into separate replacements that apply to the context or the focus, propagate both independently, and then recombine the resulting sentence replacements for the premisses. Figure 7.1 sketches the idea.

7.7. Changing Occurrences

$$\begin{array}{c}
 \dots \quad \frac{\Gamma_1, \varphi_i \vdash \Delta \quad \dots}{\Gamma_1, \varphi \vdash \Delta} \quad \dots \quad \xrightarrow{l_r} \\
 \\
 \frac{\dots \quad ? \quad \dots}{l(\Gamma_1, \varphi) \vdash r(\Delta)} \\
 \downarrow \text{Theorem 7.2} \\
 \frac{\dots \quad ? \quad \dots}{l_1(\Gamma_1), f(\varphi) \vdash r(\Delta)} \\
 \downarrow \text{Section 7.7.1.1} \\
 \frac{\dots \quad l_1(\Gamma_1), ? \vdash r(\Delta) \quad \dots}{l_1(\Gamma_1), f(\varphi) \vdash r(\Delta)} \\
 \downarrow \text{Section 7.7.1.2} \\
 \frac{\dots \quad l_1(\Gamma_1), f_i(\varphi_i) \vdash r(\Delta) \quad \dots}{l_1(\Gamma_1), f(\varphi) \vdash r(\Delta)} \\
 \downarrow \text{Theorem 7.3} \\
 \frac{\dots \quad l'_i(\Gamma_1, \varphi_i) \vdash r(\Delta) \quad \dots}{l_1(\Gamma_1), f(\varphi) \vdash r(\Delta)} \\
 \downarrow \text{Theorem 7.2 backwards} \\
 \frac{\dots \quad l'_i(\Gamma_1, \varphi_i) \vdash r(\Delta) \quad \dots}{l(\Gamma_1, \varphi) \vdash r(\Delta)} \quad \dots \quad \vdots \quad \xi'
 \end{array}$$

Figure 7.1: Sentence replacement for context and focus

Chapter 7. Proof Transformations

Let $\Gamma(\vec{x}) = \Gamma_1(\vec{x}), \varphi(\vec{x})$, where $\Gamma(\vec{x})$ is the context and $\varphi(\vec{x})$ is the focus formula. In a first step we divide $l(\vec{x})$ (or $r(\vec{x})$) into separate replacements $l_1(\vec{x})$ (or $r_1(\vec{x})$, respectively) for the context and $f(\vec{x})$ for the focus formula (the case of equational rules where we have two focus formulae is similar). I.e. we write (7.9) in the form

$$[\vec{x} : \vec{s}] \Gamma_1(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x})$$

for a rule that has exactly one focus formula $\varphi(\vec{x})$ in the antecedent (again, the case where the focus formula is in the succedent is similar). Since we have divided $l(\vec{x})$ into $l_1(\vec{x})$ for the context and $f(\vec{x})$ for the focus, we require

$$\begin{aligned} [\vec{x} : \vec{s}] l(\vec{x})(\Gamma(\vec{x})) \vdash r(\vec{x})(\Delta(\vec{x})) = \\ [\vec{x} : \vec{s}] l_1(\vec{x})(\Gamma_1(\vec{x})), f(\vec{x})(\varphi(\vec{x})) \vdash r(\vec{x})(\Delta(\vec{x})) \end{aligned} \quad (7.10)$$

Suitable replacements $l_1(\vec{x})$ and $f(\vec{x})$ indeed exist, as we will see in Theorem 7.2.

Section 7.7.1.2 describes how $f(\vec{x})$ can be propagated over the first proof step in ξ , if only $l_1(\vec{x})$ and $r(\vec{x})$ are the identity sentence replacements, i.e. $l_1(\vec{x}) = \emptyset$ and $r_1(\vec{x}) = \emptyset$. We reduce the more complicated case where $l_1(\vec{x})$ and $r(\vec{x})$ are not identities to the simpler case where they are, using the results of Section 7.7.1.1. For the and-1-rule, e.g., the original first proof step in ξ has the form

$$\frac{[\vec{x} : \vec{s}] \Gamma_1(\vec{x}), \varphi_1(\vec{x}), \varphi_2(\vec{x}) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma_1(\vec{x}), \varphi_1(\vec{x}) \wedge \varphi_2(\vec{x}) \vdash \Delta(\vec{x})}$$

This is transformed into another valid proof step if we replace $\Gamma_1(\vec{x})$ and $\Delta(\vec{x})$ by different sets of formulae $\Gamma'_1(\vec{x})$ and $\Delta'(\vec{x})$; in particular for

$$\begin{aligned} \Gamma'_1(\vec{x}) &= l_1(\vec{x})(\Gamma_1(\vec{x})) \\ \Delta'(\vec{x}) &= r(\vec{x})(\Delta(\vec{x})) \end{aligned}$$

we get the valid proof step

$$\frac{[\vec{x} : \vec{s}] l_1(\vec{x})(\Gamma_1(\vec{x})), \varphi_1(\vec{x}), \varphi_2(\vec{x}) \vdash r(\vec{x})(\Delta(\vec{x}))}{[\vec{x} : \vec{s}] l_1(\vec{x})(\Gamma_1(\vec{x})), \varphi_1(\vec{x}) \wedge \varphi_2(\vec{x}) \vdash r(\vec{x})(\Delta(\vec{x}))}$$

We can now apply the sentence replacement $f(\vec{x})$ to the focus formula of the conclusion and propagate the replacement over the proof step according to Section 7.7.1.2. As described there, a precondition is that $f(\vec{x})(\varphi(\vec{x}))$

7.7. Changing Occurrences

is a singleton set. We will see in Theorem 7.2 that the decomposition of $l(\vec{x})$ into $l_1(\vec{x})$ and $f(\vec{x})$ ensures that $f(\vec{x})$ has this property.

For the case labelled *deep focus*, the result of propagating the sentence replacement is

$$\frac{[\vec{x} : \vec{s}] l_1(\vec{x})(\Gamma_1(\vec{x})), f_1(\vec{x})(\varphi_1(\vec{x})), f_2(\vec{x})(\varphi_2(\vec{x})) \vdash r(\vec{x})(\Delta(\vec{x}))}{[\vec{x} : \vec{s}] l_1(\vec{x})(\Gamma_1(\vec{x})), f(\vec{x})(\varphi_1(\vec{x}) \wedge \varphi_2(\vec{x})) \vdash r(\vec{x})(\Delta(\vec{x}))}$$

(for $f_1(\vec{x})$ and $f_2(\vec{x})$ as defined in (7.6) for the specific rule). According to (7.10), the conclusion can be written as

$$[\vec{x} : \vec{s}] l(\vec{x})(\underbrace{\Gamma_1(\vec{x}) \cup \{\varphi_1(\vec{x}) \wedge \varphi_2(\vec{x})\}}_{\Gamma(\vec{x})}) \vdash r(\vec{x})(\Delta(\vec{x}))$$

The remaining question now is: can we write the premiss in the form

$$[\vec{x} : \vec{s}] l'(\vec{x})(\Gamma_1(\vec{x}) \cup \{\varphi_1(\vec{x}), \varphi_2(\vec{x})\}) \vdash r'(\vec{x})(\Delta(\vec{x}))$$

so that we can apply $l'(\vec{x})$ and $r'(\vec{x})$ to the subproof ξ_1 above the premiss recursively? The answer is yes, as Theorem 7.3 shows.

The other case, *immediate focus*, is similar: the sentence replacements for the conclusion are decomposed, cf. Theorem 7.2 so that independent replacements apply to the original focus and context. The replacement that applies to the context is handled exactly as described for the case *deep focus* above. The replacement that applies to the focus formula is then also propagated over the first proof step. The result is a number of premisses, some of which are related to premisses before the focus sentence replacement was applied by sentence replacements. These premisses are then expressed in the form of two sentence replacements that apply to antecedent and succedent and that transform the original premiss into the new one, cf. Theorem 7.3.

It remains to provide theorems to decompose and compose sentence replacements as needed above.

Theorem 7.2 Let Γ be a set of formulae, let l be a sentence replacement for Γ , and let $\varphi \in \Gamma$. Then there is a set $\Gamma_1 \subseteq \Gamma$ and sentence replacements l_1 for Γ_1 and f for $\{\varphi\}$ such that $l(\Gamma) = l_1(\Gamma_1) \cup f(\varphi)$ and f is a partial function over sentences, i.e. $f(\{\varphi\}) = \{\psi\}$.

Proof of 7.2 For a sentence replacement l for Γ , let l_Γ^* be the relation

$$l_\Gamma^* = l \cup \{(\varphi, \varphi) \mid \varphi \in \Gamma \text{ and } \varphi \notin \pi_1 l\} .$$

Chapter 7. Proof Transformations

(where the projection π_1 of a binary relation to its first domain is defined by $\pi_1(l) = \{\varphi \mid (\varphi, \psi) \in l\}$). It is easy to see that l_Γ^* is a sentence replacement for Γ , that

$$l(\Gamma) = l_\Gamma^*(\Gamma) = \pi_2(l_\Gamma^*), \quad (7.11)$$

(where π_2 is defined similarly to π_1 by $\pi_2(l) = \{\psi \mid (\varphi, \psi) \in l\}$), and that for each $\varphi \in \Gamma$, there is a ψ such that $(\varphi, \psi) \in l_\Gamma^*$.

Let ψ be a formula such that $(\varphi, \psi) \in l_\Gamma^*$. Such a ψ exists, but it need not be unique in general. Define

$$\begin{aligned} f &= \{(\varphi, \psi)\} \\ l_1 &= l_\Gamma^* \setminus f \\ \Gamma_1 &= \pi_1(l_\Gamma^*). \end{aligned}$$

Note that it is contingent whether $\varphi \in \Gamma_1$.

We can check that Γ_1 , l_1 and f have the postulated properties:

- $\Gamma_1 \subseteq \Gamma$ trivially.
- l_1 is a sentence replacement for Γ_1 trivially.
- f is a sentence replacement for $\{\varphi\}$ trivially.
- $f(\{\varphi\}) \cup l_1(\Gamma_1) = l(\Gamma)$: We have

$$f(\{\varphi\}) = \{\psi\} \quad (7.12)$$

and

$$l_1(\Gamma_1) = (l_\Gamma^* \setminus f)(\Gamma_1) = (\Gamma_1 \setminus \pi_1(l_\Gamma^* \setminus f)) \cup \pi_2(l_\Gamma^* \setminus f) \quad (7.13)$$

by (4.3) on page 75. If l_Γ^* maps φ to exactly one formula ψ then $\pi_1(l_\Gamma^* \setminus f) = \pi_1(l_\Gamma^*) \setminus \{\varphi\}$ (and $\varphi \notin \Gamma_1$), otherwise $\pi_1(l_\Gamma^* \setminus f) = \pi_1(l_\Gamma^*)$. In both cases, $(\Gamma_1 \setminus \pi_1(l_\Gamma^* \setminus f)) = \emptyset$, and thus (7.13) reduces to

$$l_1(\Gamma_1) = \pi_2(l_\Gamma^* \setminus f). \quad (7.14)$$

Similarly,

$$\pi_2(l_\Gamma^* \setminus f) = \begin{cases} \pi_2(l_\Gamma^*) \setminus \{\psi\} & \text{if } \psi \notin \pi_2(l_1) \\ \pi_2(l_\Gamma^*) & \text{otherwise.} \end{cases}$$

Note that in both cases $\psi \in \pi_2(l_\Gamma^*)$, so that in either case we have

$$\{\psi\} \cup \pi_2(l_\Gamma^* \setminus f) = \pi_2(l_\Gamma^*). \quad (7.15)$$

7.7. Changing Occurrences

Putting all this together, we get

$$\begin{aligned}
 f(\{\varphi\}) \cup l_1(\Gamma_1) &= \{\psi\} \cup \pi_2(l_\Gamma^* \setminus f) && \text{by (7.12) and (7.14)} \\
 &= \pi_2(l_\Gamma^*) && \text{by (7.15)} \\
 &= l(\Gamma) && \text{by (7.11)}
 \end{aligned}$$

as required. □

Theorem 7.3 Let l_i be a sentence replacement for Γ_i ($i = 1, 2$). Then there is a sentence replacement l for $\Gamma = \Gamma_1 \cup \Gamma_2$ such that $l(\Gamma) = l_1(\Gamma_1) \cup l_2(\Gamma_2)$.

Proof of 7.3 Let $l = l_{1,\Gamma_1}^* \cup l_{2,\Gamma_2}^*$.

- $\pi_1(l) = \pi_1(l_{1,\Gamma_1}^*) \cup \pi_1(l_{2,\Gamma_2}^*) = \Gamma_1 \cup \Gamma_2 = \Gamma$, thus l is a sentence replacement for Γ .
- $l(\Gamma) = l_1(\Gamma_1) \cup l_2(\Gamma_2)$ because of

$$\begin{aligned}
 l(\Gamma) &= (\Gamma \setminus \pi_1(l)) \cup \pi_2(l) && \text{by (4.3)} \\
 &= (\Gamma \setminus \pi_1(l_{1,\Gamma_1}^* \cup l_{2,\Gamma_2}^*)) \cup \pi_2(l_{1,\Gamma_1}^* \cup l_{2,\Gamma_2}^*) \\
 &= (\Gamma \setminus (\underbrace{\pi_1(l_{1,\Gamma_1}^*)}_{\Gamma_1} \cup \underbrace{\pi_1(l_{2,\Gamma_2}^*)}_{\Gamma_2})) \cup (\underbrace{\pi_2(l_{1,\Gamma_1}^*)}_{l_{1,\Gamma_1}^*(\Gamma_1)} \cup \underbrace{\pi_2(l_{2,\Gamma_2}^*)}_{l_{2,\Gamma_2}^*(\Gamma_2)}) \\
 &= l_1(\Gamma_1) \cup l_2(\Gamma_2)
 \end{aligned}$$

□

Remark 7.4 The proofs above are more subtle than their informal description would have made one believe. The reason is that the theorems need to deal with the fact that unrelated formulae can be mapped to the same formula by a sentence replacement. Although this seems to be a purely technical issue, it is the symptom of the following interesting phenomenon.

There are two styles of presenting sequent calculi: in one, antecedent and succedent are defined as sets, in the other as sequences (or lists) of formulae. As far as the logic is concerned the order of the formulae and their multiplicity does not matter at all. Thus, users typically do not distinguish between different copies of the same formula: copies of the same formula usually cannot be distinguished once they are printed, anyway. Thus, using sets is the most natural choice.

As far as proof analysis is concerned, however, it is often necessary to know where a particular formula comes from: e.g. one is interested in whether the proof is still valid if a particular axiom is withdrawn or changed. After all the same formula might be available from several places, so changing one does not necessarily affect formulae derived from the other. It is interesting to know at least whether a particular formula is still available if it is changed in one place. This is made explicit in [FH94] and [Sch98]. Whether the proof needs to be changed depends on whether the changing axiom is used or the one that remains unchanged. However, since users do not distinguish different copies of formulae, it is arbitrary which formula was used and thus, in practice, the detailed information is not as useful as expected.

Separating and combining sentence replacements via the construction of l_{Γ}^* from l in the theorems above is a way to simulate keeping the multiplicities of formulae where they matter: when a formula that occurs multiple times is changed in one place, we make sure it is mapped by the respective replacement to the new formula and to itself. As long as formulae are not changed, however, the presentation above does not track the multiplicity of formulae and thus the presentation using sets is appropriate.

A consequence is that a focus formula can be replaced by more than one distinct formulae. In this situation, one of these formulae needs to be chosen as the replacement for the focus formula, cf. the choice of ψ in the Proof of 7.2. It turns out that this is a border case that does not turn up in practice very often. Nevertheless, the situation needs to be handled when it occurs. Since it occurs rarely, it is acceptable to ask the user when it does. If this turns out to be a problem, some other solution needs to be worked out.

7.7.2 Special Cases

We have described a uniform way of handling the case in which the focus formula is changed and the original proof rule is no longer applicable: giving up gracefully (*immediate focus*). In many cases this is exactly what should be done, as we have explained in the preceding section. In many situations, however, this is not the best we can do.

7.7.2.1 Propositional Replacement

For example, when a formula is weakened, e.g. from $A \wedge B$ to $(A \wedge B) \vee C$, the new disjunct needs to be taken into consideration, but the reasoning

7.7. Changing Occurrences

for the conjunction is still useful. When we look at a proof ξ of the form

$$\frac{\begin{array}{c} \vdots \xi_1 \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}), \varphi_2(\vec{x}) \vdash \Delta(\vec{x}) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}) \wedge \varphi_2(\vec{x}) \vdash \Delta(\vec{x})} \text{and-1} \dots$$

and then carry out the replacement described above (for ease of exposition we ignore the replacements on the context), the conclusion is

$$[\vec{x} : \vec{s}] \Gamma(\vec{x}), (\varphi_1(\vec{x}) \wedge \varphi_2(\vec{x})) \vee \varphi_3(\vec{x}) \vdash \Delta(\vec{x})$$

and the and-1 rule is not applicable. However, we know that the conclusion has been wrapped inside the disjunction, so after getting rid of the disjunction, the old subproof ξ_1 is reusable:

$$\frac{\begin{array}{c} \vdots \xi_1 \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}), \varphi_2(\vec{x}) \vdash \Delta(\vec{x}) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}) \wedge \varphi_2(\vec{x}) \vdash \Delta(\vec{x})} \text{and-1} \dots \quad \frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_3(\vec{x}) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), (\varphi_1(\vec{x}) \wedge \varphi_2(\vec{x})) \vee \varphi_3(\vec{x}) \vdash \Delta(\vec{x})} \text{or-1} \dots$$

When instead of a disjunction a conjunction is wrapped around in the antecedent, the transformed proof becomes

$$\frac{\begin{array}{c} \vdots \xi'_1 \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}), \varphi_2(\vec{x}), \varphi_3(\vec{x}) \vdash \Delta(\vec{x}) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi_1(\vec{x}) \wedge \varphi_2(\vec{x}), \varphi_3(\vec{x}) \vdash \Delta(\vec{x})} \text{and-1} \dots}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), (\varphi_1(\vec{x}) \wedge \varphi_2(\vec{x})) \wedge \varphi_3(\vec{x}) \vdash \Delta(\vec{x})} \text{and-1} \dots$$

where ξ_1 is reusable after adding the new formula $\varphi_3(\vec{x})$ to the antecedent. Both examples also work backwards, i.e. the conjunction is unwrapped from the disjunction and conjunction and the proof steps corresponding to the connectives that have been eliminated can be left out from the proof.

These examples can be generalised to the other connectives, and it is also possible to generalise it even further to arbitrary subformulae that are built from two sets of subformulae only using connectives. The two sets have some subformulae in common, and whenever a rule was applied to one of these common formulae in the original proof, the transformed proof decomposes any connectives around the common subformula and then reuses the original proof. Conversely, if in the original proof a rule is applied to decompose a connective which is not part of the new formula, the rule application is left out in the transformed proof.

7.7.2.2 Wrapping and Unwrapping Quantifiers

Another special case of replacing subformulae is concerned with quantifiers: a formula $\varphi(\vec{x})$ is wrapped inside a quantor, e.g. $\forall y : s. \varphi(\vec{x})$, or a quantor is removed, e.g. from $\forall y : s. \varphi(\vec{x})$. Note that in both cases the bound variable does *not* occur in the scope of the quantor. This makes transforming the proof really simple: from

$$\frac{\begin{array}{c} \vdots \xi_1 \\ [\vec{x} : \vec{s}] \Gamma(\vec{x}), \forall y : s. \varphi(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x}) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \forall y : s. \varphi(\vec{x}) \vdash \Delta(\vec{x})}$$

to

$$[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x})$$

or back for existentially acting formulae, and from

$$\frac{\begin{array}{c} \vdots \xi_1 \\ [\vec{x} : \vec{s}, y : s] \Gamma(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x}) \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \exists y : s. \varphi(\vec{x}) \vdash \Delta(\vec{x})}$$

to

$$[\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x})$$

or back for universally acting formulae. The price we pay for the simplicity of the transformations is that we can only introduce or remove quantors for variables that are not used in the proof. Almost always it is therefore necessary to apply other transformations, e.g. replace occurrence, to use the introduced variables or to make the variables unused which we want to remove.

7.7.3 Induction Schemata

As we have pointed out in Section 6.3.3, we need to consider several types of changes to generatedness constraints. Since we consider generatedness constraints to be sentences, *adding or deleting a generatedness constraints* is handled by the proof transformation that deals with adding and deleting assumptions. *Adding or deleting constructors* amounts to replacing a sentence (the generatedness constraint), and when the constraint is used the

7.7. Changing Occurrences

premisses can be adjusted straightforwardly: the original proof

$$\frac{[\vec{x} : \vec{s}] \vdash \Phi(\vec{x}, f_1, \sigma, \varphi) \quad \dots \quad [\vec{x} : \vec{s}] \vdash \Phi(\vec{x}, f_m, \sigma, \varphi)}{[\vec{x} : \vec{s}] \vdash \forall x : s. \varphi(\vec{x}, x)}$$

becomes

$$\frac{[\vec{x} : \vec{s}] \vdash \Phi(\vec{x}, f_1, \sigma, \varphi) \quad \dots \quad [\vec{x} : \vec{s}] \vdash \Phi(\vec{x}, f_m, \sigma, \varphi) \quad [\vec{x} : \vec{s}] \vdash \Phi(\vec{x}, f_{m+1}, \sigma, \varphi)}{[\vec{x} : \vec{s}] \vdash \forall x : s. \varphi(\vec{x}, x)}$$

when a new constructor is added, and similarly backwards when the constructor is removed.

Finally, *changing the arity of a constructor* is handled as part of changing the arity of the constructor as a function. For $f : s \times t_1 \times s \rightarrow s$ and $s \neq t_1$, the induction formula reads (cf. (5.4) on page 116)

$$\forall y_1 : s, y_2 : t_1, y_3 : s. (\varphi(\vec{x}, y_1) \wedge \varphi(\vec{x}, y_3)) \Rightarrow \varphi(\vec{x}, f(y_1, y_2, y_3)). \quad (7.16)$$

(where we have assumed that the constraint morphism is the identity for ease of presentation). Adding a fourth argument of type t_2 to f without consideration for the status of f as constructor will change (7.16) to

$$\forall y_1 : s, y_2 : t_1, y_3 : s. (\varphi(\vec{x}, y_1) \wedge \varphi(\vec{x}, y_3)) \Rightarrow \varphi(\vec{x}, f(y_1, y_2, y_3, \tau)) \quad (7.17)$$

where τ is the dummy term that is inserted as the new argument by the sentence translation *addarg*. Of course, this is not an induction formula, because the fourth argument to the constructor should be an all-quantified variable. We achieve this in several steps using transformations that we have described before. First we wrap a variable y_4 of type t_2 around the body of (7.17):

$$\forall y_1 : s, y_2 : t_1, y_3 : s, y_4 : t_2. (\varphi(\vec{x}, y_1) \wedge \varphi(\vec{x}, y_3)) \Rightarrow \varphi(\vec{x}, f(y_1, y_2, y_3, \tau))$$

and then replace the term τ by y_4 :

$$\forall y_1 : s, y_2 : t_1, y_3 : s, y_4 : t_2. (\varphi(\vec{x}, y_1) \wedge \varphi(\vec{x}, y_3)) \Rightarrow \varphi(\vec{x}, f(y_1, y_2, y_3, y_4)).$$

Chapter 7. Proof Transformations

Finally, in case t_2 is the induction sort, i.e. $t_2 = s$, we add an additional induction hypothesis:

$$\forall y_1 : s, y_2 : t_1, y_3 : s, y_4 : t_2. \\ (\varphi(\vec{x}, y_1) \wedge \varphi(\vec{x}, y_3) \wedge \varphi(\vec{x}, y_4)) \Rightarrow \varphi(\vec{x}, f(y_1, y_2, y_3, y_4)).$$

The result is the induction formula corresponding to the new arity of the constructor.

Removing an argument is easier: when the argument is removed from (7.16) without consideration for the status as constructor, the only change left is to remove the superfluous quantor in the induction formula.

7.8 Auxiliary Transformations

The proof transformations described so far are needed to propagate specification transformations to proofs. Additionally, other ways to change proofs are needed: proofs change while the specification stays unchanged. This is obvious for applying proof rules, pruning subproofs, or applying heuristics, i.e. traditional activities when working on proofs. In our setting, additional transformations can be provided that allow proofs to be transformed rather than created anew. These are particularly useful in situations that result from other transformations.

Our proof transformations consider the proof calculus to be used analytically: a reasoning step is a function from conclusion and justification to a sequence of premisses. The transformations given earlier change the conclusion, and this change is propagated to the justification and then the premisses and subproofs. When, e.g., a focus formula changes then the argument of the justification also changes so that the rule now applies to the changed focus formula. However, there are justification arguments that do not occur in the conclusion and that can be chosen: cut formulae and witness terms. Of course these are chosen by the user or heuristics such that the proof works, and frequently heuristics determine them using unification. These constraints are not encoded in the proof rules, and they may turn out to be mistaken (cf. attempts to postpone the commitment to a particular witness term by using logical variables (meta-variables) that can be instantiated). Therefore, we allow the user to revise these decisions and provide transformations that change cut formulae in cut and gap-1/gap-r rules and witness terms in all-1/ex-r rules.

7.8. Auxiliary Transformations

For a given proof of the form

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), \varphi(\vec{x}) \quad \begin{array}{c} \vdots \xi_1 \\ \vdots \xi_2 \end{array} \quad [\vec{x} : \vec{s}] \Gamma(\vec{x}), \varphi(\vec{x}) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})} \text{ cut } \varphi(\vec{x})$$

we can change the cut formula $\varphi(\vec{x})$ to $\psi(\vec{x})$, by replacing the justification argument and both occurrences of $\varphi(\vec{x})$ in the premisses by $\psi(\vec{x})$. Let $f(\vec{x}) = \{(\varphi(\vec{x}), \psi(\vec{x}))\}$ and assume that it is to be applied to the cut formula. The new proof can then be written as

$$(7.18) \quad \frac{\begin{array}{c} \vdots \xi'_1 \\ \vdots \xi'_2 \end{array} \quad [\vec{x} : \vec{s}] \Gamma(\vec{x}), f(\vec{x})(\varphi(\vec{x})) \vdash \Delta(\vec{x})}{[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x})}$$

with

$$[\vec{x} : \vec{s}] \Gamma(\vec{x}) \vdash \Delta(\vec{x}), f(\vec{x})(\varphi(\vec{x})) \quad (7.18)$$

and justification “cut $f(\vec{x})(\varphi(\vec{x}))$ ”, where ξ'_i results from ξ_i by propagating the change to the subproofs recursively.

Similarly, when changing the witness terms $t(\vec{x})$ to $t'(\vec{x})$, the proof

$$\frac{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \forall x : s. \varphi(\vec{x}, x), \varphi(\vec{x}, t(\vec{x})) \vdash \Delta(\vec{x}) \quad \begin{array}{c} \vdots \xi_1 \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \forall x : s. \varphi(\vec{x}, x) \vdash \Delta(\vec{x})} \text{ all-1 } \forall x : s. \varphi(\vec{x}, x), t(\vec{x})$$

is transformed into

$$\frac{[\vec{x} : \vec{s}] (l(\vec{x}) \cup c(\vec{x}))(\Gamma(\vec{x}), \forall x : s. \varphi(\vec{x}, x), \varphi(\vec{x}, t(\vec{x}))) \vdash r(\vec{x})(\Delta(\vec{x})) \quad \begin{array}{c} \vdots \xi'_1 \end{array}}{[\vec{x} : \vec{s}] \Gamma(\vec{x}), \forall x : s. \varphi(\vec{x}, x) \vdash \Delta(\vec{x})}$$

with justification

$$\text{“all-1 } \forall x : s. \varphi(\vec{x}, x), t'(\vec{x})\text{”, } c(\vec{x}) = \{(\varphi(\vec{x}, t(\vec{x})), \varphi(\vec{x}, t'(\vec{x})))\}$$

and ξ'_i the proof that results from applying the replacements to the subproofs recursively.

In both cases, replacing the cut formula and witness term is combined with replacements in the context as described in Section 7.7.1.3.

7.9 Summary

We have presented concrete proof transformations. For each proof obligation transformation that results from propagating any of our concrete specification transformations through the development graphs (cf. Figure 6.6 on page 140), a corresponding proof transformations was introduced. We can thus propagate all those specification transformations to the proofs. Additionally, transformations for changing proofs without changing their conclusion have been introduced.

Chapter 8

Mechanising Transformations

8.1 Overview

We have implemented a proof-of-concept prototype of our transformational framework and the concrete instantiation that we have developed in this thesis. The implementation comprises the representation of development graphs over the logic described in Section 5.2, a translator from SSL described in Section 5.3 to development graphs, a representation of sequent calculus proofs according to Section 5.4 together with a simple command-line interface to construct proofs, a representation of formal developments together with a checker for whether specification and proofs form a well-formed development, specification, development graph, and proof transformations, a command line based interface to manage formal developments and to apply transformations, and debugging facilities that present the history of a development as a set of interactive, hyperlinked documents.

The prototype was tested using a reconstruction of the case study on fault-tolerant systems given in Section 2.2.2 in order to assess whether our transformations are useful in practice. To this end the specification and proofs were built incrementally, e.g. axioms, theories, and links were added, and the system was transformed into a fault tolerant system, e.g. additional slots in state representations and additional state transitions were added, and axioms were changed.

8.2 Original Specification

For a reconstruction of the case study given in Section 2.2.2, a system of concurrent processes is specified. The specification consists of an abstract

datatype that represents the relevant state space of processes. A global state is specified as an abstract finite array of processes and communication channels between processes. The behaviour of the system as a whole is specified as a state-transition system: possible state-transitions are described by so called commands, and these are given by the terms of a freely generated datatype. An interpreter for these commands is specified using pre- and postconditions that specify in which states a given command is applicable (or enabled), and what the relationship between original and resulting state is in this case. The system is specified incrementally, e.g. commands are added and transitions for existing commands are added or changed, in a way that resembles how such specifications are constructed step by step in practice.

The concrete state transitions are according to the non-deterministic algorithm given in [MG00]. The intention is that the behaviour of the system complies with a given protocol for reliably broadcasting messages. The authors define reliability as a safety property: no message is delivered to a process unless it has been broadcast, and each broadcast message is delivered at most once. This property is specified as a predicate over states: each process maintains a multiset (or bag) of messages that have been delivered to it, and the global state contains the set of messages that have been broadcast; the predicate is true iff for each process in the state its delivered messages are a subset of the broadcast messages, and if no message is delivered more than once. This is specified formally as the predicate *safe*:¹

```
(axiom (all (s state)
  (<=> (safe s)
    (all (i nat)
      (and
        (=> (< i size)
          (subsetq
            (delivered
              (aref (procs s) i))
            (broadcast s)))
          (=> (< i size)
            (nodups
              (delivered
                (aref (procs s) i))))))))))
```

(8.1)

The proposition that this property logically follows from the algorithm

¹More details about the case study are given in Appendix D.

8.2. Original Specification

that the processes carry out is postulated in the following way. The specification is presented in a modular fashion, resulting in the development graph given in Figure 8.1,² The specification includes the theory system, which specifies the behaviour of the system by a constant `initial-state` representing the initial state and a predicate `trans*` that is true of a state `s`, a sequence of commands `tr`, and a second state `s'` iff execution of `tr` is enabled in `s` and a possible result is `s'`. The theory reliability specifies that each state reachable from `initial-state` satisfies the safety property:

$$\begin{aligned} &(\text{axiom } (\text{all } (\text{tr } \text{trace}) (\text{s } \text{state}) && (8.2) \\ & \quad (\Rightarrow (\text{trans* } \text{initial-state } \text{tr } \text{s}) (\text{safe } \text{s})))) \end{aligned}$$

The specification of system contains a `satisfies`-clause

`(satisfies (reliability))`

resulting in the postulated link from reliability to system. The development graph calculus reduces this global link to a local one, and the resulting proof obligation is that (8.2) is a logical consequence of the behaviour of the system. Partial proofs have been constructed for this and other proof obligations. The proof tree for (8.2) is printed in Figure 8.2. Each node stands for a proof goal, and links relate the conclusion (below) of a proof rule application with their premisses (on top).

The overall structure of the proof is an induction over the length of `tr`, leading to a binary branch in the tree. The base case is finished off by the left subtree. The step case is then split into six subproofs by a case distinction over the first command of `tr`, i.e. by an application of the induction rule over a universally quantified variable of sort `command`. The type `command` is defined by

$$\begin{aligned} &(\text{sort } \text{command}) \\ &(\text{op } \text{b-1 } (\text{nat } \text{message}) \text{command}) \\ &(\text{op } \text{b-1p } (\text{nat}) \text{command}) \\ &(\text{op } \text{b-2 } (\text{nat } \text{nat } \text{message}) \text{command}) && (8.3) \\ &(\text{op } \text{b-3 } (\text{nat}) \text{command}) \\ &(\text{op } \text{b-4 } (\text{nat}) \text{command}) \\ &(\text{op } \text{b-5 } (\text{nat}) \text{command}) \\ &(\text{gen } \text{b-1 } \text{b-1p } \text{b-2 } \text{b-3 } \text{b-4 } \text{b-5}) \end{aligned}$$

²In the development graph solid arrows represent definitorial links, dashed ones are postulated links. Bold dashed arrows correspond to the global postulated links that were specified, and the grey dashed ones represent the local proof obligation links left after applying the development graph calculus.

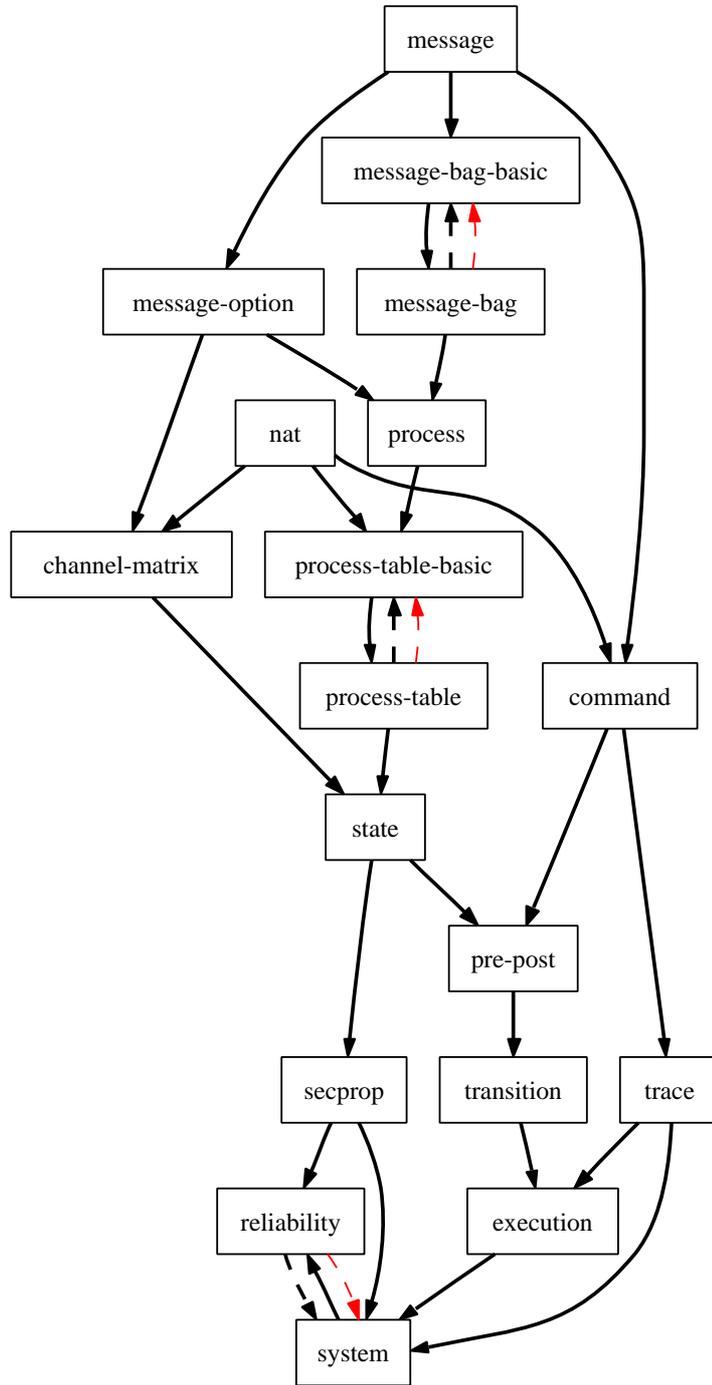


Figure 8.1: Initial development graph

8.2. *Original Specification*

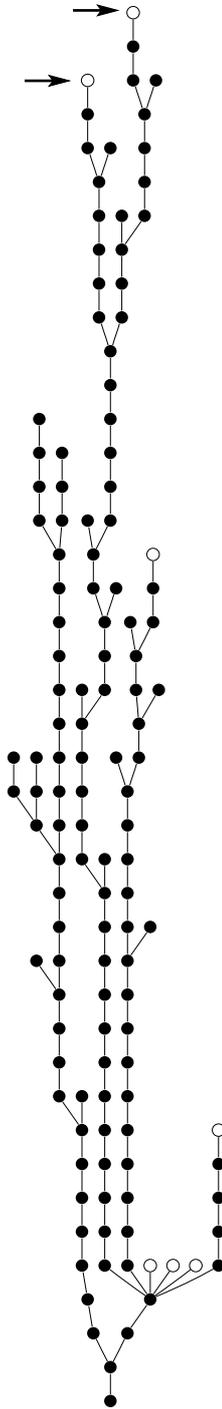


Figure 8.2: Initial proof

so the induction according to the generatedness constraint produces six subproofs, one for each of the given constructors. The proofs for the first, second, and last command (b-1, b-1p, and b-5, respectively) have been attempted. For the first command, the proof has two open goals, cf. the arrows in the figure. These goals cannot be closed because the specification of the pre- and postconditions for the first command, i.e. b-1, are needed. However, these are missing from the original specification: recall that the specification is created incrementally.

8.3 Missing Axioms

We zoom in on the first open goal, i.e. the left one of the two marked goals in Figure 8.2. Here, a command of the form (b-1 n m) (for some natural number n and some message m) is executed in some state, and the resulting state is s . The subgoal that remains to be shown here has the form

$$\Gamma \vdash (\text{subsetq } (\text{delivered } (\text{aref } (\text{procs } s) i)) (\text{broadcast } s)) \quad (8.4)$$

which requires that the messages which have been delivered to the process number i in the process table (procs s) of the state s have all been broadcast before. The antecedent Γ contains amongst others the following formulae:

$$\begin{aligned} &(\text{post } s-3 \text{ (b-1 nat message) } s) \\ &(\text{pre } s-3 \text{ (b-1 nat message)}) \\ &(< i \text{ size}) \\ &(\Rightarrow (< i \text{ size}) \\ &\quad (\text{subsetq } (\text{delivered } (\text{aref } (\text{procs } s-3) i)) \\ &\quad \quad (\text{broadcast } s-3))) \end{aligned}$$

The Eigenvariables $s-3$ is the state in which the command is executed, and s stands for the state after the execution of the command (b-1 nat message), where nat and message are also Eigenvariables. Here, further information about the pre- and postconditions are required. As mentioned earlier, these are not yet specified. We will add them to the specification in the first step, i.e. we will carry out the transformation to add axioms and add

$$\begin{aligned} &(\text{all } (s \text{ state}) (i \text{ nat}) (m \text{ message}) \\ &\quad (<=> (\text{pre } s \text{ (b-1 } i \text{ } m)) \\ &\quad \quad (\text{and } (< i \text{ size}))) \end{aligned}$$

8.3. Missing Axioms

```
(not (member m (broadcast s)))  
(none-p (bb (aref (procs s) i))))))
```

and

```
(all (s state) (i nat) (m message) (s1 state)  
  (<=> (post s (b-1 i m) s1)  
    (and (= (procs s1)  
      (update (procs s) i  
        (mk-process  
          (some m)  
          (ib (aref (procs s) i))  
          (db (aref (procs s) i))  
          (delivered (aref (procs s)  
            i))))))  
      (= (broadcast s1)  
        (adjoin m (broadcast s)))))))
```

to the node `pre-post`.

The only proof obligation link at which the change is visible is the one between `system` and `reliability`. All other proof obligations like the ones resulting, e.g., from the postulated link between `process-table-basic` and `process-table` are not affected at all.

All proof obligations arising from the affected link are transformed to include additional assumptions. Which assumptions need to be added is computed from the axioms that were added to the original theory `pre-post` and the paths along which they can be inherited in the theory of the node `system`. The axioms are mapped along the path morphism and are then added to the conclusions of the proof. Note that in this case there is only one postulated path from `pre-post` to `system`; otherwise the axioms would have been mapped along different paths, and potentially several instances of the axioms would have been added.

Adding the axioms to the proofs means that for each proof obligation its proof tree is transformed by adding the new axioms to the antecedent of the conclusion and then propagating it over all proof rule applications towards the leaves of the proof tree. This has been described in detail in Section 7.3.

After this transformation is carried out, the proof goal has two additional axioms in the antecedent; these axioms are available in addition to the other formulae that were already members of the antecedent. They can thus be used to finish the goal. After some traditional proof work the

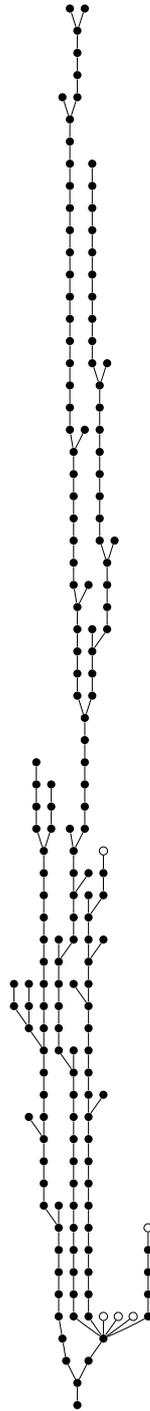


Figure 8.3: Proof with completed subproof for $b-1$

subproof is completed, and at this point the proof tree looks as shown in Figure 8.3.

Note that the transformation is carried out on all proofs that are associated with the all affected links, and that the result is that the assumptions are added to all leaves and can thus be used in all open goals that are still to be closed. The user does not have to be concerned with which proofs need to be replayed or reused and which do not.

8.4 Missing Theory

The authors of [MG00] transformed a specification such that it is fault tolerant. To that end they changed the specification so that the system state maintained additional information about each process: namely whether it was running or whether it was crashed. Before we can carry out this transformation we need to add a new theory that provides the additional datastructures that we use to represent the information.

We introduce a theory `boolean` and specify an enumeration `boolean` with two constructors, `tt` and `ff`.³ This involves adding a node to the development graph. The resulting development graph is shown in Figure 8.4. Sort `boolean`, the constructors, the generatedness constraint, and an axiom stating that the sort is additionally freely constructed is added to the new node:

```
(sort boolean)
(op ff () boolean)
(op tt () boolean)
(gen boolean tt ff)
(axiom (not (= tt ff)))
```

Since the new node is isolated, these changes are not visible anywhere else. The new datatype is supposed to be used in the definition of processes, and thus in `process` an additional `uses`-clause is added leading to a definitorial link from `boolean` to `process`. The development graph changes to the one shown in Figure 8.5. All these items and axioms are now visible in `process` and all theories below. Note that the new axiom also becomes visible in the node `reliability`, i.e. in a place where it is supposed to be

³Since theories and signature items inhabit different name spaces, the overloading of `boolean` is allowed and intentional. The representation of formal developments and transformations works on the internal datastructures and thus does not depend on the names of the elements it operates on, either.

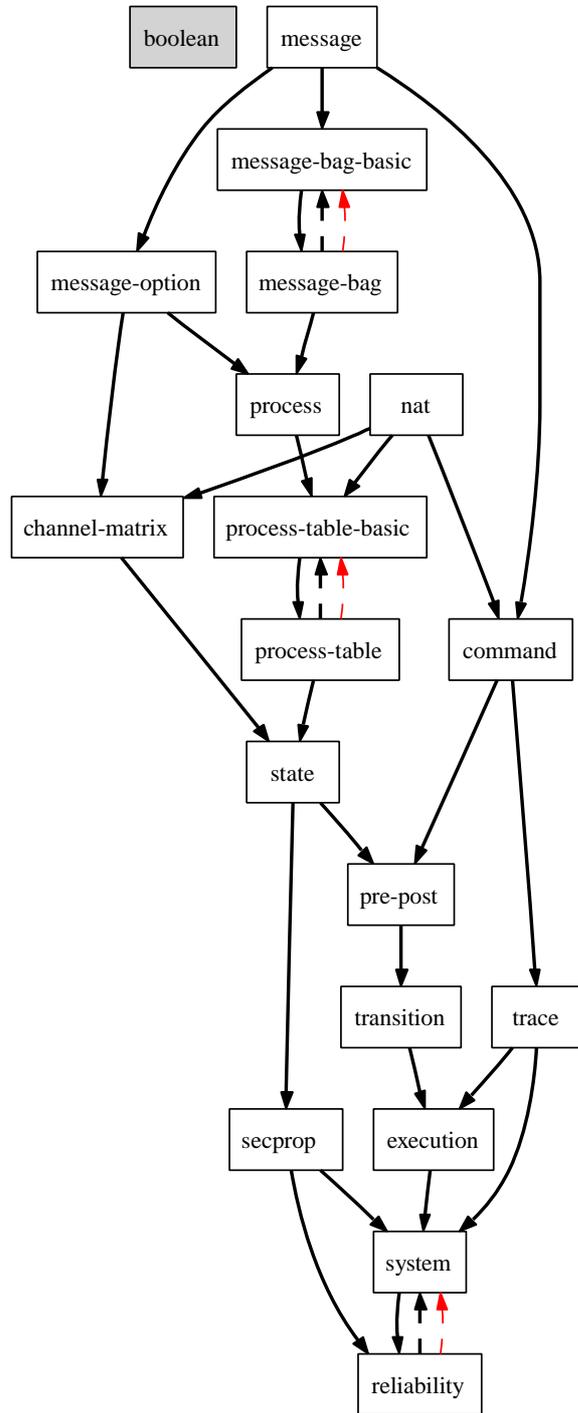


Figure 8.4: Development graph with boolean

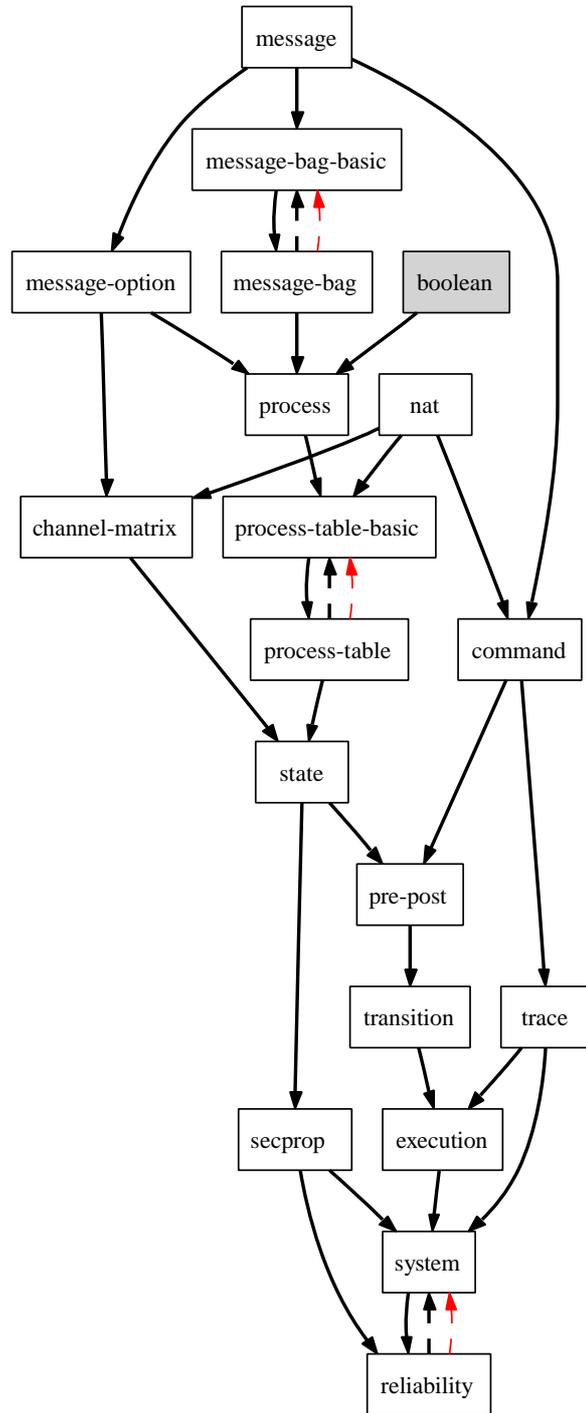


Figure 8.5: Development graph with added link

a logical consequence of the visible assumptions. However, the decomposition of theorem links with the development graph calculus makes sure that no proof obligation for the axiom is generated: `boolean` is also visible in the node system, from which the axiom would have to be proven.

8.5 Missing Slot

We now add a selector function `is-up`

```
(op is-up (process) boolean)
```

to the theory `boolean` that is supposed to evaluate to `tt` iff the process it is applied to is still running. We have decided to store the information about whether a process is up in the process data structure (rather than in an additional table). Processes are represented by a generated sort `process` which has a single constructor `mk-process` and the four selectors `bb`, `ib`, `db`, and `delivered`.

```
(op mk-process (option option option bag) process)
(gen process mk-process)
(op bb (process) option)
(op ib (process) option)
(op db (process) option)
(op delivered (process) bag)
```

(8.5)

There are the usual selector axioms, e.g.

```
(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
      (= bb1
         (bb (mk-process bb1 ib1 db1 dm1))))
```

(8.6)

or

```
(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
      (bb2 option) (ib2 option) (db2 option) (dm2 bag)
      (<=>
        (= (mk-process bb1 ib1 db1 dm1)
           (mk-process bb2 ib2 db2 dm2))
        (and (= bb1 bb2)
              (= ib1 ib2)
              (= db1 db2)
              (= dm1 dm2))))
```

(8.7)

Adding a new slot to store the data about whether the process is up or down amounts to adding an argument to the constructor function `mk-process`: the function declaration in (8.5) is changed to

```
(op mk-process (option option option bag boolean) process) .
```

All occurrences of terms constructed with `mk-process` have an additional argument. Before adding this argument, we created a new dummy constant of sort `boolean` named `boolean-dummy` and gave it to the transformation as the term to be used as the additional argument. The two axioms (8.6) and (8.7) given above are changed to

```
(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
      (= bb1
         (bb (mk-process bb1 ib1 db1 dm1 boolean-dummy)))) (8.8)
```

and

```
(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
      (bb2 option) (ib2 option) (db2 option) (dm2 bag)
      (<=>
        (= (mk-process bb1 ib1 db1 dm1 boolean-dummy)
           (mk-process bb2 ib2 db2 dm2 boolean-dummy))
        (and (= bb1 bb2)
              (= ib1 ib2)
              (= db1 db2)
              (= dm1 dm2))))))
```

These changes are also applied to the proofs without any changes to the proof structure by the transformation rule `translation` with argument `addarg(C)` (where C is the partial function that maps the constructor function `mk-process` to the boolean-valued term `boolean-dummy`); the details are described in Section 7.6 on page 146.

In order to make full use of the result of this transformation, in the course of further development the axiom (8.8) is changed to

```
(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
      (up1 boolean)
      (= bb1
         (bb (mk-process bb1 ib1 db1 dm1 up1))))
```

so that the additional argument is also universally quantified. This has no essential effect on the proofs since wherever the axiom was used before, the new universally quantified variable can be instantiated to whatever term is necessary for the resulting equation to be applicable. This necessitates introducing `all-1` rules before each proof rule application that looks into the subformula

```
(= bb1
  (bb (mk-process bb1 ib1 db1 dm1 boolean-dummy)))
```

in the original proof.

Note that all these changes are also visible for the proof obligations associated with the link between `process-table-basic` and `process-table`. However, those proofs do not deal with the slots of processes, so they are only changed in the context of proof rules, and the changes can be ignored. Looking at the *structure of the development graph* is not sufficient to say that these proofs are unchanged; rather the *proofs themselves* need to be inspected in order to find out. This is done implicitly when the proofs are transformed.

8.6 Missing Action

The additional slot of sort `boolean` is set to `tt` in the initial state and inherited by all other commands. It will only ever set to `ff` when a process crashes. To this end an additional action `crash` is introduced by adding a function

```
(op crash (nat) command)
```

to the node `command`, and then turning this function into a constructor of the sort `command`. Recall that `command` has been defined in (8.3) as

```
(sort command)
(op b-1 (nat message) command)
(op b-1p (nat) command)
(op b-2 (nat nat message) command)
(op b-3 (nat) command)
(op b-4 (nat) command)
(op b-5 (nat) command)
(gen b-1 b-1p b-2 b-3 b-4 b-5)
```

Adding `crash` as a constructor results in the definition

```
(sort command)
(op b-1 (nat message) command)
(op b-1p (nat) command)
(op b-2 (nat nat message) command)
(op b-3 (nat) command)
(op b-4 (nat) command)
```

8.6. Missing Action

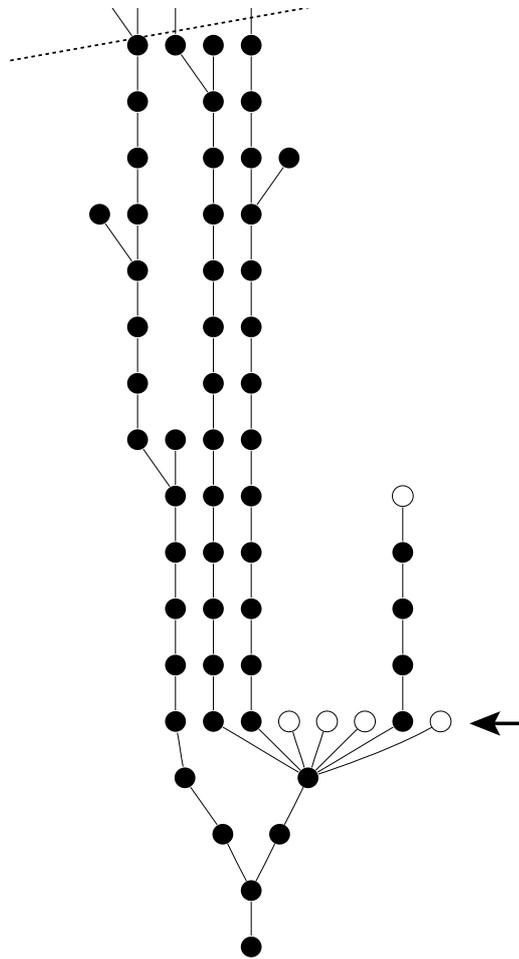


Figure 8.6: Proof with crash action

```
(op b-5 (nat) command)
(op crash (nat) command)
(gen b-1 b-1p b-2 b-3 b-4 b-5 crash)
```

The proof tree resulting from this transformation is given in Figure 8.6. As one would have expected there is a new open goal for the additional case corresponding to the crash action. The rest of the proof remains unchanged.

8.7 Stronger Precondition

Finally we have to strengthen the precondition of the first clause of (8.1) to restrict the property to running processes. This is done by replacing the boxed occurrence in the axiom

```
(axiom (all (s state)
  (<=> (safe s)
    (all (i nat)
      (and
        (=> (< i size)
          (subseteq
            (delivered
              (aref (procs s) i))
            (broadcast s)))
          (=> (< i size)
            (nodups
              (delivered
                (aref (procs s) i))))))))))
```

(8.9)

in `secprop` by

```
(and (< i size) (= tt (is-up (aref (procs s) i))))
```

so that the result is

```
(axiom (all (s state)
  (<=> (safe s)
    (all (i nat)
      (and
        (=> (and (< i size)
          (= tt
            (is-up (aref (procs s) i))))
          (subseteq
            (delivered (aref (procs s) i))
            (broadcast s)))
          (=> (< i size)
            (nodups
              (delivered
                (aref (procs s) i))))))))))
```

This is afforded by the transformation wrapping an existing subformula occurrence (namely the boxed occurrence above) into a connective: the boxed subformula occurrence \bullet is replaced by

8.7. Stronger Precondition

(and • (= tt (is-up (aref (procs s) i))))

This is a special case of the transformation occurrence described in Section 6.3.4 for specifications and in Section 7.7 for proofs. Applying the proof transformation (given by the propagation of the specification transformation over development graphs as defined in Section 4.3.2) to the proof given in Figure 8.6 results in the proof depicted on the right in Figure 8.7.

Relative to the proof in Figure 8.6, most of the proof rules in Figure 8.7 are changed straightforwardly by replacing the occurrence of ($< i \text{ size}$) by (and ($< i \text{ size}$) (= tt (is-up (aref (procs s) i)))) in the context of the rules. There are three places where the transformation has to do more work. These are marked by the arrows (a) to (c) in Figure 8.7 and are described in turn below.

(a) The first marked goal, part of the subproof for the base case, was an application of the basic rule that proved the sequent

$$\Gamma_1, (< i \text{ size}) \vdash \Delta_1, (< i \text{ size}). \quad (8.10)$$

The formula in the succedent results from the specification of the initial state as follows. In the initial state, each the process in the process table has the form

(mk-process none none none empty tt)

The processes are stored in the table indexed by $0, 1, \dots, \text{size} - 1$. The axiom representing this formally reads

(axiom (all (i nat)
 (\Rightarrow ($< i \text{ size}$)
 (= (aref (procs initial-state) i)
 (mk-process none none none empty tt))))))

This axiom is an assumption in the proof. In the proof rule application two steps below the node marked (a), the implication in the axiom is split, i.e. the sequent below the rule reads

$$\begin{array}{l} \Gamma_2, \\ (< i \text{ size}), \\ (\mathbf{\Rightarrow} (< i \text{ size}) \\ \quad (= (aref (procs \text{initial-state}) i) \\ \quad \quad (\text{mk-process none none none empty tt}))) \\ \hline \Delta_2 \end{array} \quad (8.11)$$

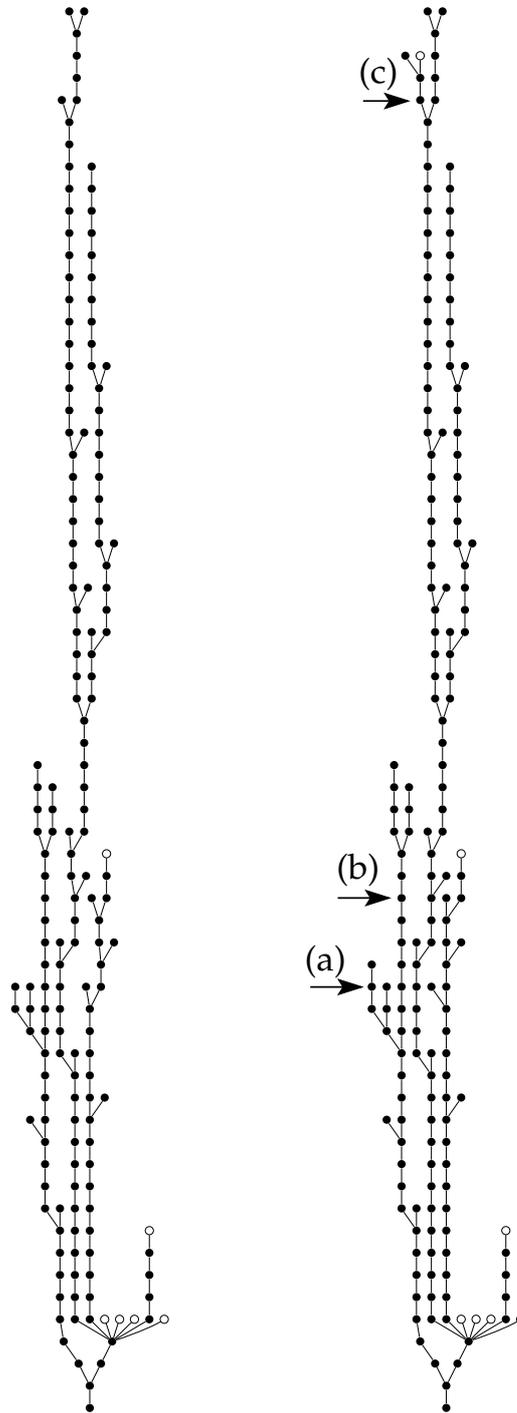


Figure 8.7: Left: proof of Figure 8.6 reprinted in full. Right: proof with weakened property

8.7. Stronger Precondition

$$\frac{\begin{array}{c} (8.10) \\ \vdots \\ (8.12) \end{array} \quad \begin{array}{c} \vdots \\ (8.13) \end{array}}{(8.11)}$$

Figure 8.8: Part of the original proof. Sequent (8.10) is the node marked by (a) in Figure 8.7.

Applying the rule \Rightarrow -r to (8.11) results in the two goals

$$\frac{\Gamma_2, (< i \text{ size})}{\Delta_2, (< i \text{ size})} \quad (8.12)$$

and

$$\frac{\begin{array}{l} \Gamma_2, \\ (< i \text{ size}), \\ (= (\text{aref} (\text{procs} \text{ initial-state}) i) \\ \quad (\text{mk-process} \text{ none} \text{ none} \text{ none} \text{ empty} \text{ tt})) \end{array}}{\Delta_2} \quad (8.13)$$

Figure 8.8 sketches the relevant part of the proof. Replacing $(< i \text{ size})$ by

$(\text{and} (< i \text{ size}) (= \text{tt} (\text{is-up} (\text{aref} (\text{procs} \text{ s}) i))))$

in the axiom (8.9) has the following effect on the proof sketched in Figure 8.8. Goal (8.11) becomes

$$\frac{\begin{array}{l} \Gamma_2, \\ (< i \text{ size}), \\ (\Rightarrow (\text{and} (< i \text{ size}) \\ \quad (= \text{tt} \\ \quad \quad (\text{is-up} (\text{aref} (\text{procs} \text{ s}) i)))) \\ (= (\text{aref} (\text{procs} \text{ initial-state}) i) \\ \quad (\text{mk-process} \text{ none} \text{ none} \text{ none} \text{ empty} \text{ tt})) \end{array}}{\Delta_2} \quad (8.11')$$

Chapter 8. Mechanising Transformations

As described in Section 7.7, the rule $\Rightarrow\text{-r}$ is applied to the changed focus formula in (8.11'), resulting in the two goals

$$\frac{\Gamma_2, \text{(and (< i size) (= tt (is-up (aref (procs s) i))))}}{\Delta_2, (< i size)} \quad (8.12')$$

and

$$\frac{\Gamma_2, (< i size), \text{(= (aref (procs initial-state) i) (mk-process none none none empty tt))}}{\Delta_2} \quad (8.13')$$

Note that (8.13') is unchanged from (8.13) and thus the proof above this goal can be reused directly without further transformations. For (8.12'), the formula in the antecedent is the subformula that we have changed, so instead of the goal (8.12)

$(< i size) \dots \vdash (< i size) \dots$

we now have

$$\frac{\text{(and (< i size) (= tt (is-up (aref (procs s) i))))} \dots}{(< i size) \dots}$$

and the original rule (basic) requires the formula structure of the formula occurrence that is changed. According to Section 7.7.1.2, this is *immediate focus* and the definition of the transformation requires that the added connective is decomposed, yielding

$$\frac{(< i size) \text{(= tt (is-up (aref (procs s) i))))} \dots}{(< i size) \dots} \quad (8.14)$$

and the proof is then carried on, i.e. by applying the original rules. In this case, the original rule is the basic rule, which is applicable again now, so the proof is again closed. The resulting proof is sketched in Figure 8.9. Note that (8.10') is not the same as (8.10): the new conjunct $(= \text{tt} \text{(is-up (aref (procs s) i))))$ is part of the context of (8.10').

8.7. Stronger Precondition

$$\frac{\frac{(8.10')}{(8.14)} \quad \begin{array}{c} \vdots \\ (8.12') \end{array} \quad \begin{array}{c} \vdots \\ (8.13) \end{array}}{(8.11')}$$

Figure 8.9: Part of the new proof for (a)

(b) The second marked goal, also in the part of the base case, exhibits another interesting aspect of the transformation. In the original proof, the goal (b) contains the equation

```
(= (aref (procs initial-state) i)
    (mk-process none none none empty tt))
```

as an assumption and an instance of the axiom (8.9): the sequent reads

$$\frac{\Gamma, \text{ (= (aref (procs initial-state) i) (mk-process none none none empty tt))}}{\Delta, \text{ (and (=> } \boxed{(< i size)} \text{ (subsetq (delivered (aref (procs initial-state) i)) (broadcast initial-state)))) (=> (< i size) (nodups (delivered (aref (procs initial-state) i))))))} \quad (8.15)$$

The proof step in the original proof is an application of an equational proof rule: the underlined subterm occurrence of `(aref (procs initial-state) i)` is replaced by the right hand side of the equation, i.e. `(mk-process none`

Chapter 8. Mechanising Transformations

none none empty tt), resulting in the following goal:

$$\begin{array}{l}
 \Gamma, \\
 (= (\text{aref} (\text{procs initial-state}) i) \\
 \quad (\text{mk-process none none none empty tt})) \\
 \hline
 \Delta, \\
 (\text{and} (\Rightarrow \boxed{(< i size)} \\
 \quad (\text{subsetq} \\
 \quad \quad (\text{delivered} \\
 \quad \quad \quad (\text{mk-process none none none empty tt})) \\
 \quad \quad \quad (\text{broadcast initial-state}))) \\
 \quad (\Rightarrow (< i size) \\
 \quad \quad (\text{nodups} \\
 \quad \quad \quad (\text{delivered} (\text{aref} (\text{procs initial-state}) i))))))
 \end{array} \tag{8.16}$$

The transformation alters the axiom (8.9); as a consequence the boxed sub-term occurrence in (8.15) is replaced, and (8.15) reads

$$\begin{array}{l}
 \Gamma, \\
 (= (\text{aref} (\text{procs initial-state}) i) \\
 \quad (\text{mk-process none none none empty tt})) \\
 \hline
 \Delta, \\
 (\text{and} (\Rightarrow \boxed{(\text{and} (< i size) (= tt (\text{is-up} (\text{aref} (\text{procs s}) i))))}) \\
 \quad (\text{subsetq} \\
 \quad \quad (\text{delivered} (\text{aref} (\text{procs initial-state}) i)) \\
 \quad \quad \quad (\text{broadcast initial-state}))) \\
 \quad (\Rightarrow (< i size) \\
 \quad \quad (\text{nodups} \\
 \quad \quad \quad (\text{delivered} (\text{aref} (\text{procs initial-state}) i))))))
 \end{array}$$

8.7. Stronger Precondition

Note that the underlined subterm is unchanged. So the equation can be applied, resulting in

$$\begin{array}{l}
 \Gamma, \\
 (= (\text{aref } (\text{procs } \text{initial-state}) \text{ i}) \\
 \quad (\text{mk-process none none none empty tt})) \\
 \hline
 \Delta, \\
 (\text{and } (=> \boxed{(\text{and } (< \text{ i size}) (= \text{ tt } (\text{is-up } (\text{aref } (\text{procs } \text{ s}) \text{ i}))))}) \\
 \quad (\text{subsetq} \\
 \quad \quad (\text{delivered} \\
 \quad \quad \quad (\text{mk-process none none none empty tt})) \\
 \quad \quad (\text{broadcast initial-state}))) \\
 \quad (=> (< \text{ i size}) \\
 \quad \quad (\text{nodups} \\
 \quad \quad \quad (\text{delivered } (\text{aref } (\text{procs } \text{initial-state}) \text{ i}))))))
 \end{array}$$

instead of (8.16). After this proof step, the rest of the proof can be transformed recursively.

Note that the focus formula to which the equation had been applied was changed by the proof transformation. The proof representation and the rules for the proof transformation, however, ensure that the equation is applied to the correct formula after the transformation. This is possible since the change between old and new formulae is explicitly represented (as sentence replacements). The sentence replacements are then adapted so that the rest of the proof can be transformed recursively.

(c) Finally, the part of the proof marked (c) is in the step case for the first command b-1. In the original proof, the induction hypothesis was

$$\begin{array}{l}
 (=> (< \text{ i size}) \\
 \quad (\text{subsetq } (\text{delivered } (\text{aref } (\text{procs } \text{ s-3}) \text{ i})) \\
 \quad \quad (\text{broadcast } \text{ s-3}))) \qquad (8.17)
 \end{array}$$

and the conjecture

$$\begin{array}{l}
 (=> (< \text{ i size}) \\
 \quad (\text{subsetq } (\text{delivered } (\text{aref } (\text{procs } \text{ s}) \text{ i})) \\
 \quad \quad (\text{broadcast } \text{ s})))
 \end{array}$$

Chapter 8. Mechanising Transformations

where s is the state after the command has been executed and $s-3$ the state before. Eliminating the implication in the succedent results in the sequent

$$\begin{array}{l}
 \Gamma, \\
 (\Rightarrow (< i \text{ size}) \\
 \quad (\text{subsetq} (\text{delivered} (\text{aref} (\text{procs } s-3) i)) \\
 \quad \quad (\text{broadcast } s-3))), \\
 (< i \text{ size}) \\
 \hline
 \Delta \\
 (\text{subsetq} (\text{delivered} (\text{aref} (\text{procs } s) i)) \\
 \quad (\text{broadcast } s))
 \end{array} \tag{8.18}$$

With the aid of the pre- and postconditions for $b-1$, which are members of Γ , the conjecture can be rewritten to

$$\begin{array}{l}
 (\text{subsetq} (\text{delivered} (\text{aref} (\text{procs } s-3) i)) \\
 \quad (\text{adjoin message} (\text{broadcast } s-3)))
 \end{array} \tag{8.19}$$

Γ also includes the following lemma about bags:

$$\begin{array}{l}
 (\text{all } (s-1 \text{ bag}) (s-2 \text{ bag}) (m \text{ message}) \\
 \quad (\Rightarrow (\text{subsetq } s-1 \text{ } s-2) \\
 \quad \quad (\text{subsetq } s-1 (\text{adjoin } m \text{ } s-2))))
 \end{array}$$

This means that the new conjecture (8.19) is a consequence of the induction hypothesis (8.17) if only $(< i \text{ size})$ can be shown. This is easy since $(< i \text{ size})$ is an assumption:

$$\begin{array}{l}
 \Gamma_1, (< i \text{ size}) \\
 \hline
 \Delta_1, (< i \text{ size})
 \end{array} \tag{8.20}$$

The transformation changes the occurrence of $(< i \text{ size})$ in both the conjecture and the induction hypothesis. In the induction hypothesis it is changed to

$$(\text{and } (< i \text{ size}) (= \text{tt } (\text{is-up } (\text{aref } (\text{procs } s-3) i))))$$

and in the conjecture to

$$(\text{and } (< i \text{ size}) (= \text{tt } (\text{is-up } (\text{aref } (\text{procs } s) i))))$$

8.7. Stronger Precondition

After eliminating the implication in the conjecture, the sequent (8.18) is thus changed to

$$\begin{array}{l}
 \Gamma, \\
 (\Rightarrow (\text{and} (< i \text{ size}) (= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s-3) i)))) \\
 (\text{subseteq} (\text{delivered} (\text{aref} (\text{procs } s-3) i)) \\
 (\text{broadcast } s-3)), \\
 (\text{and} (< i \text{ size}) (= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s) i)))) \quad (8.18') \\
 \hline
 \Delta \\
 (\text{subseteq} (\text{delivered} (\text{aref} (\text{procs } s) i)) \\
 (\text{broadcast } s))
 \end{array}$$

Note that the conjecture

$$(\text{subseteq} (\text{delivered} (\text{aref} (\text{procs } s) i)) \\
 (\text{broadcast } s))$$

is unchanged in (8.18'). It can thus be rewritten using the pre- and post-conditions of b-1 as before, resulting in

$$(\text{subseteq} (\text{delivered} (\text{aref} (\text{procs } s-3) i)) \\
 (\text{adjoin message} (\text{broadcast } s-3))) \quad (8.19')$$

Again note that the rewritten conjecture is unchanged from (8.19). Also, the lemma about bags can be instantiated and applied as before, to the effect that the unchanged conjecture (8.19') is a consequence of the induction hypothesis

$$(\Rightarrow (\text{and} (< i \text{ size}) (= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s-3) i)))) \\
 (\text{subseteq} (\text{delivered} (\text{aref} (\text{procs } s-3) i)) \\
 (\text{broadcast } s-3))) \quad (8.17')$$

if only $(\text{and} (< i \text{ size}) (= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s-3) i))))$ can be shown. The goal corresponding to (8.20) is thus changed by the transformation into

$$\begin{array}{l}
 \Gamma_1, \\
 (\text{and} (< i \text{ size}) (= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s) i)))) \\
 \hline
 \Delta_1, \\
 (\text{and} (< i \text{ size}) (= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s-3) i)))) \quad (8.20')
 \end{array}$$

In the original proof, the rule basic was applied to finish the proof. The applicability of the rule depends on the focus formulae, and so this is

the case *immediate focus* of the transformation occurrence. Both formulae are decomposed according to the special case *propositional replacement* described in Section 7.7.2 on page 162. This results in the two new subgoals

$$\frac{\begin{array}{l} \Gamma_1, \\ (< i \text{ size}), \\ (= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s) i))) \end{array}}{\text{-----}} \quad (8.21)$$

$$\begin{array}{l} \Delta_1, \\ (< i \text{ size}) \end{array}$$

and

$$\frac{\begin{array}{l} \Gamma_1, \\ (< i \text{ size}), \\ (= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s) i))) \end{array}}{\text{-----}} \quad (8.22)$$

$$\begin{array}{l} \Delta_1, \\ (= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s-3) i))) \end{array}$$

The original remaining proof (consisting of the application of the basic rule to the formula $(< i \text{ size})$) is transformed to close the first of the resulting subgoals (8.21). The other subgoal (8.22), however, requires us to prove that a process that is up in the state s after the command $b-1$ has been taken was also up in the state $s-3$ before the command was taken. This is an entirely new subproof. It is relatively easy to construct, but the transformation simply leaves the goal open. This is as well, since no part of the old proof has dealt with whether processes stay up, and so this is something that the user has to do, using any proof automation that is available.

Note that in the presentation of the case study in [MG00, p. 483], the authors note that the subproof for each of the commands has an additional subproof that has to do with the weakened property, whereas the rest of the proofs was essentially unchanged. The “unchanged” proofs had to be constructed based on the old proofs manually in the reference, whereas with our transformations they can be kept. Our new open goal (8.22) is the base for the new subproof that we need to construct according to [MG00].

8.8 Summary

We have presented a reconstruction of a case study [MG00] that we have mechanised using our prototypical implementation. The specification of

8.8. Summary

a state transition system is constructed incrementally using development transformations, and partial proofs are constructed. These proofs are transformed when the specification is extended or corrected. After that, the specified system is changed to be fault tolerant by adding features to the specification and changing existing parts of the specification of the system and its properties. The case study thus deals with many of the scenarios from Chapter 2 that have been the motivation for our work.

The transformations carried out in the case study deal with adding whole theories and axioms, adding arguments to functions and constructor functions, adding constructors to generated datatypes, and changing subterm and subformula occurrences. The transformations allow to make the changes that are necessary to construct the specification incrementally. More importantly, specification transformations extend to proofs in a manner that keeps most of the original proofs where this makes sense. When new open goals appear as the result of a transformation they are genuine new open goals. Examples are the new open goal in the case distinction for a newly added command (cf. Section 8.6), or an open goal corresponding to the fundamentally new subproof that the authors of the case study found to be necessary (cf. Section 8.7 (c)).

The results are very encouraging. Obviously, more case studies are desirable, cf. Section 10.2, to further assess the usefulness of the transformational approach.

Part IV

Related Work and Conclusions

Chapter 9

Related Work

9.1 Management of Change

Our approach has been conceived as an aid for handling changes in the context of systems that use an *invent and verify* approach to formal software development. In this context, an alternative approach is to anticipate change by modelling systems in a way that isolates aspects which are likely to change. This has been pursued with specification constructs, [GB92], [DGS93], [FM97], [Hut00], [DM03], [KBM04] to list just a few examples, and methodologies, e.g. [RSW97], [AF02].

The idea invariably is to provide explicit means to structure the system (or its description) into separate elements such that changes can be made to one element only, without any effect on others. Our experience, however, is that there are always changes which cut across the boundaries set up by these structures. The reason is that, usually, there is not a single structuring dimension, and changes, in particular ones that deal with omissions and bugs, usually cut across boundaries of several of the dimensions. However, using the explicit structure of specifications is important in practice, and we have taken care that our techniques are complementary to this approach rather than a replacement. We have integrated our work with one of the structuring mechanisms, described in [Hut00], which is used extensively in our group. Our transformations can be seen as basic operations that are defined to alter development graphs in addition to the ones that have been defined in the earlier work. Some of our transformations are more fine-grained than the ones provided by prior work, some of our transformations are entirely novel in this context. This relationship has been discussed in Section 4.5.

In [Gro02], [Gro00] specifications and programs are expressed using

the same wide-spectrum language, and changes to a specification are expressed by special combinators applied to the specification. Examples for combinators are sequential composition and program conjunction. These combinators can also be applied to the program. Rules describe how a combinator can then be propagated into the original program structure. The result is a program that satisfies the changed specification. This approach is restricted to changes that can be expressed by the given combinators, which rules out, e.g., changes to existing signature items.

In work on “evolving specifications” [PS01], [ASP02], [PS02], the term “evolving” does not refer to the evolution of specifications as the development progresses, rather it refers to the evolution of the state of a system as the execution progresses. The idea there is that the specification describing the current state evolves over time of the execution of the system, and thus describes the possible state transitions. Changes to the specification as part of the development process are not considered at all.

9.2 Proof Reuse and Replay

Another technique to cope with the evolutionary nature of a formal software development process is the replay of proofs or proof plans on changed specifications. Proof scripts that have been assembled when creating the original proof can be replayed, cf. e.g. [Cur95]. In this case, however, proof scripts are patched and changed manually, whereas our approach transforms the proofs without manual intervention.

In contrast to a heuristic replay of proofs on changed proof obligations, e.g. [KW94], [Ko197], [RS93], [MW97], [Sch98, MS98], [BK04], where the tactics or proof plans of a source proof are mapped from source to target problem and are replayed on the target goal, our approach allows transformations to produce proofs that could not have been produced by replaying the tactics. This allows us to change the overall proof structure in a reliable and predictable way.

In this respect, our approach to the transformation of proofs was inspired by and is similar to the work in [FH94] in that an explicit proof is transformed, not by replaying the tactic steps but by changing proofs explicitly. However, we additionally consider transformations that change existing signature items and proof rules, i.e. inductions schemes. Also, we support patches for the case in which a formula is, e.g., wrapped into another one, in which case the technique described in [FH94] would throw away the subproof which uses the changed formula.

These techniques are orthogonal to our work in the sense that they

9.3. Correctness-Preserving Transformations

address different problems and can be used in addition to our transformations, e.g. to replay a specific proof by a heuristics after a change has been made for which it is expected that a specific heuristics works better than the transformation did. They can also be used as building blocks for the support that we provide, e.g. a proof reuse heuristics can be used to propagate a particular specification transformation to proofs. In fact, our proof transformations for changing occurrences can be seen as an extension of the proof reuse technique presented in [FH94].

[Boi04] describes an approach to proof reuse that is very similar to some of the proof transformations we have presented in [SH02] and Chapter 7. The author also proposes to use preconceived transformations to change specifications and proof terms. The transformations deal with inductive types exclusively, e.g. adding a constructor, or adding an argument to a constructor, and form a subset of the transformations we have considered in [SH02] and this thesis. Most notably, transformations to change occurrences of terms and formulae – which we have found to be necessary in practice also in the context of changing induction schemes, cf. Section 6.3.3 – are missing. Transformations are formulated as explicit operations on definition and proof terms in Coq’s type theory; they correspond to our definition of sentence and proof translations $\mathbf{XSen}(addarg(C))$ and $\mathbf{XPrf}(addarg(C))$, however without exploiting the relationship between translations and morphisms. Because [Boi04] is restricted to transformations that correspond to a clean semantic transformation, the correctness of these transformations can be proven. As with other techniques for proof reuse, an account of how the technique would be integrated with structured specifications is not provided.

9.3 Correctness-Preserving Transformations

There is a large amount of prior work on correctness-preserving transformations, e.g. [Par90], [HKB93], [BB96], [JL04a], [JL04b]. The general idea is to develop correct programs starting from an initial specification by successive applications of correctness-preserving transformations. By construction, each transformation enforces a certain relation between the two specifications it relates, e.g. a notion of refinement or implication. These relations are usually transitive, and so the resulting program is in relation to the original specification. Thus, by construction the program satisfies (or refines) its original specification. Each correctness-preserving transformation may include side conditions that need to be checked for the transformation to be applicable. Usually, the transformation rules and

their side conditions are represented in a framework such that the correctness of each of the transformations together with its side conditions can be proven in the framework itself.

The approach makes powerful use of the intuition that changes to a specification can be seen as directed attempts to keep some property, represented by the refinement relation, while changing other properties. For example, transforming a recursive program into an equivalent one that is tail-recursive keeps the functional specification of the program but leads to a supposedly more efficient program. Another example is to transform a program into one that has more functionality such that the more functional program is a refinement of the original one.

This approach differs from ours in several ways. The most important aspect is to consider the motivation and applicability of the work. We have argued in Chapter 1 that specifications need to be changed frequently because they are faulty or ill-conceived. In this situation the intention of the developer is to remove a bug. Thus, producing an equivalent specification or one that is a refinement of the faulty one is not going to be helpful. Since usually there is more than one problem with real-life specifications, the specification will in most cases still be faulty after a certain corrective step has been taken – just a little bit less so. It is therefore difficult or impossible to formulate exactly which logical property should be preserved by the transformations. It has thus been noted that in software maintenance, correctness-preserving transformations are most helpful in refactoring the program before the corrective action is taken, where the corrective change itself is not correctness-preserving [BB96]. Our transformations do not preserve correctness but are intended to support the changes that developers have made in the past in such situations. Of course, they need to produce consistent states of formal developments, but the relevant notion of consistency, i.e. well-formedness as described in Section 3.6 on page 55, is not a property in the realm of the logic we are interested in. As a consequence, we do not represent our transformations as logical rules. This has the benefit that we are not restricted by the logic for formulating our transformations. Where we use this freedom, we incur the disadvantage that our transformations are less elegant on the semantic side, and the propagation of transformations to proofs is operationally motivated.

Another difference between our work and correctness-preserving transformations is that a program derived from a specification using correctness-preserving transformations can be checked to be correct with respect to the specification when the whole history of the derivation is known. For our setup, every new development that results from the preceding one by a transformation is self-contained. This is important for mainly two reasons.

First, in the context of software evaluation according to, e.g., the Common Criteria [CC99] a checkable form of the state of a development is a prerequisite for the evaluation process. Second, since our transformations themselves are not necessarily part of the trusted kernel of the support tool, it is necessary to be able to check whether the resulting development has the required property of being well-formed.

On a technical level, producing self-contained developments requires us to provide proof transformations that correspond to a given specification transformation. Using the approach of correctness-preserving transformations this can be avoided. It is enough to show that a semantic property is preserved by the transformation without consideration for how arbitrary proofs of the property are transformed over specification transformations. When correctness-preserving transformations are proven correct in the same logic in which the correctness property is stated and in which their side-conditions are discharged it should be possible to construct a self-contained proof from the history of transformations by plugging the proofs together, although this is usually not done.

9.4 Advanced Programming IDEs

In programming, there is a general interest in advanced integrated development environments (IDEs) for programming. Instead of editing flat program text, for which the semantics is provided later by the compiler in an opaque way, the developer works with an environment that “understands” some aspects of the programs. In an object-oriented language it is, e.g., possible to browse a class diagram that results from the code, or navigate in a call graph, i.e. get an overview over which methods call the method that is inspected and changed at the moment. There is now also a growing number of tools that support restructuring or refactoring [Opd92], [MT04], i.e. changing the program representation systematically without altering its external behaviour, cf. the discussion of correctness-preserving transformations in Section 9.3. Example refactoring steps are renaming a method consistently, i.e. its declaration, definition, and all call sites, or moving a method from one class to another one in class-based languages. These functionalities rely on the fact that the environment understands the structure of the artefact (syntax and static semantics) and the relevant aspects of the dynamic semantics.

This raises the question of how the structure and the semantics are worked out by the environment. One possibility is to infer the relevant information from the artefact itself. In the examples above, this means that

Chapter 9. Related Work

the program analyses the program code using its own parser and static analyser and computes the class hierarchy and call graph itself. This necessitates duplicating the functionality dealing with the relevant aspects of the input language. Because of this disadvantage and the risk of implementing two subtly different definitions, a better approach is to extract the information directly from the compiler or runtime system. This necessitates opening up the relevant internals of the compiler and exposing enough of the runtime system, e.g. by introspective facilities. This has mainly been done in the past in environments for dynamic and reflective languages like Common Lisp or Smalltalk. Many old ideas are currently rediscovered in this context, cf. Wilson [Wil05]. We have chosen to follow this approach and provide additional interfaces to the case tool and theorem prover, which our implementation of the transformations is able to use.

Usually, IDEs provide a way to (re-) compile and execute the program after it has been changed using the compiler and runtime environment. Since the compilation process is completely mechanised and no user input is necessary this is guaranteed to work without user interaction: the whole program can be compiled from scratch and then started. For efficiency reasons, however, only changed files and those that really need recompiling are actually compiled if possible. To this end it is necessary to extract whether a file depends on a changed one in a way that necessitates recompiling it but it is not necessary to work out how the dependent file has changed: the file itself is recompiled from scratch. If user action is required for the equivalent of the compilation as is the case in our work, more information about the nature of the dependency should be known to further restrict the work that has to be done from scratch.

Cynthia [WBBL99], [WBB02], an advanced IDE for a subset of Standard ML (SML, a functional programming language), is an example that takes steps in this direction. It guarantees that the edited set of SML function definitions is well-typed, and each function is total and terminating according to a given sufficient condition. These properties are weaker than the ones we have considered and are chosen such that all necessary proofs can be constructed fully automatically. Cynthia represents the program by a synthesis proof in Martin-Löf Type Theory that includes subproofs for the termination conditions. These subproofs are found using a heuristic algorithm. Changes to the program are carried out indirectly by manipulating and transforming the synthesis proof and redisplaying the corresponding program. It is possible to distinguish two classes of transformations that Cynthia supports. The *first* deals with structural changes like adding a constructor to a datatype definition. This necessitates adding

another defining clause whenever a case distinction over the datatype is made in the program. The term to use for this case is left open and can be replaced by another term later. Since these unspecified terms are represented by open goals in the synthesis proof, function definitions may temporarily violate the well-typedness and termination conditions. This is not possible in our work, because well-formedness conditions are checked statically rather than by a type-correctness derivation in type theory. Thus only complete and type-correct terms are allowed in our setting. The *second* class deals with replacing one term occurrence by another term. This is only possible if the new term is type-correct for the context of the old term occurrence and if it is smaller than the surrounding context according to a termination ordering. This latter condition ensures that the function is terminating and is checked mechanically. Cynthia includes a graphical user interface and has been used and experimentally evaluated in the context of education of computer science students [WC00].

9.5 Requirements Traceability

Requirements traceability, e.g. [GF94], [RJ01], [vKP02] is concerned with the question of how to acquire and maintain a trace from requirements along their manifestations in development artefacts at different levels of abstraction. This is a key technique for software maintenance, because the information about relationships between different artefacts that is provided by these traces allows to assess the impact and scope of planned changes, cf. [GHM98], [RJ01], [vK01], [vK02]. A planned change can be propagated along the links that relate dependent items in the artefacts, and thus a work plan of necessary follow-up changes can be acquired. The dependency relationships and the ways in which changes propagate over them is application dependent.

Propagated changes can be kept in a work plan and be handled by the user step by step, and some implied changes might even be carried out automatically, cf. [GHM98]. The consequence is, however, that inconsistencies can stay in the artefacts for some time; it is acceptable that they are repaired only after several steps have been taken. As soon as the development is in a consistent state again, dependent artefacts (like binaries) can be created, but not in the intermediate stages. This is in contrast to our setting where the dependent artefacts include proofs, which cannot be created from scratch without a prohibitive amount of effort. It is, thus, not acceptable for us to allow inconsistent intermediate states. Without the need to maintain proofs, techniques used for requirements traceability

Chapter 9. Related Work

could directly be used to manage dependent parts of specifications and the associations between the parts. Our main motivation, however, was to deal with proofs.

Similarly to the way in which dependency relationships between specification entities are represented as links in development graphs in our work, domain specific artefacts and dependencies are often represented using abstract entities and relations that make the dependencies explicit, e.g. “components and relationships” in [GHM98], “logical entities and relations” in [vK02], or “suggestion carrying DAGs” in [SBR04].

There are many issues that have been investigated in the context of requirements traceability, which could be investigated for our work. Examples include the uniform representation and propagation of changes, and an adequate presentation of implied changes and presentation of the change history to the user, cf. [GHM98]. We have not addressed these issues.

Chapter 10

Conclusions and Outlook

10.1 Conclusions

Experience shows that developing formal specifications is an error-prone task – just like any other formal or informal software development activity. The consequence is that one needs to be prepared for change: thinking about a system and formulating tentative specifications and properties is an iterative and evolutionary process. On a high level of abstraction, proof work provides valuable and reliable feedback. An indispensable aspect of formal methods in practice, therefore, is how unavoidable changes are handled gracefully. In this thesis we contributed methods, techniques, and a prototypical implementation of a tool that help to mechanise the management of change in this situation.

We investigated the problem in the context of axiomatic specification and verification of systems and their properties. In this context, the main problem is that proofs require user interaction, and that in general these proofs get invalidated when the system description or the postulated properties change. Substantial effort spent for constructing formal proofs may then be lost. We aim at adjusting these invalidated proofs automatically, according to the changes that were made to the system or problem description.

In this thesis, we have proposed an approach that propagates changed specifications to changed proof obligations, and then uses the knowledge about the concrete change to patch the proofs. Technically, a development step is carried out by transforming the whole state of the formal development using preconceived transformations. These transformations relate possibly incomplete and erroneous states of the same formal development and are operationally motivated.

Chapter 10. Conclusions and Outlook

Based on numerous case studies and projects carried out by our group, we identified scenarios and associated changes to specifications (i) that occurred frequently, (ii) that were not addressed sufficiently by existing support methods and techniques, and (iii) which the developers would have known how to patch systematically. These changes have been the motivation for our work. We classified the changes that appeared in practice into the following categories: adding and removing axioms, replacing subformulae and subterms of axioms, adding and removing signature symbols, changing existing signature symbols, and changing induction schemata. The changes and the associated patches are orthogonal to the structuring of specifications. For each of these changes, corresponding patches to the proofs had to be carried out manually in the case studies, and we have described how these can be mechanised.

In order to be able to deal with these kinds of changes and associated proof patches in practice we have investigated how they manifest themselves in the context of structured specifications, their translation into development graphs, and the extraction of proof obligation from the resulting graph. We have achieved an abstraction of the setting that is to a large extent independent of the concrete specification language and proof calculus and representation. It was thus possible to work out the details of how changes propagate through the development graph machinery, from structured specifications to proofs, as a framework that can be instantiated for different choices of specification language and proof representation. Formulating these aspects separately has the benefit that a clear distinction of different classes of transformations is explicitly possible. These classes coincide only partially with the classes that were identified in the motivating scenarios.

We have described an instantiation of the framework, which is simple enough to act as a proof of concept instantiation, yet still shares the crucial difficulties found in real-life specification languages and proof calculi. Having separated out most of the theoretic issues into the framework, the exposition of the instantiation can be provided relatively informally and the transformations can be described completely operationally. Using this instantiation, we revisited the example scenarios that motivated our work and showed how they are handled using our approach. The crucial parts of the instantiation have been successfully tested in a prototypical implementation. In particular the central aspect of propagating transformations through development graphs has been tested thoroughly.

Using the methods and techniques that we have developed and integrated in this thesis, it is possible to start with formal proof work earlier in the formal development process with the expectation that most of

the proof work only needs to be done once rather than several times after changes. Similarly, it is possible to carry out minor changes and corrections late without the risk of losing a large portion of proof work. In practice, this often makes the crucial difference.

10.2 Further Work

Many interesting problems have presented themselves while we worked out the details of our approach. Of course we could not solve all of them. Further work is certainly needed to extend our techniques and adapt them to existing formal case tools; some more detailed items are given below. Also, more case studies would certainly help to assess further the usefulness of the approach and its potential over and above the scenarios that we have envisioned it for.

Improvements of Existing Instantiation. The instantiation we have presented in this thesis works best for changes that exhibit a close correspondence between the lexical part of the specification that is changed and the portion of the proof that needs to be changed accordingly. It does not work as well when equations are heavily applied. The reason seems to be that the proof calculus we used does not explicitly link the place of the term that is replaced when applying an equation with a part of the proof: terms can be rewritten under connectives. We would expect that further investigations in this direction reveal further conditions on transformations that would give better support for transforming the proofs. We suspect that hierarchical reasoning (cf. [Aut03]) would be well-suited to formulate the transformations cleanly. Note that this is a question of which changes to represent specially when replacing occurrences and how to handle them when transforming proofs. The propagation through development graphs is not affected at all.

Different Instantiations. The question of how to control and trigger the appropriate specification transformations has only partially been solved by the example specification transformations given in Chapter 6. In particular, complicated structuring mechanisms in the specification language warrant further attention. An idea that seems to be worth exploring starts out from a *calculus* for translating structured specifications into development graphs. This calculus should then be enriched so that explicitly represented changes are propagated through the derivation of the develop-

ment graph, similarly to how changes are propagated through the derivation of proof obligation links in [Hut00] and [AHMS00].

Similarly, more complicated translations from specifications in the small to development graphs deserve further work and experimentation. An example is the translation of procedures in specifications into one big temporal formula, implemented, e.g., in the VSE system [AHL⁺00]. Changes to the program lead to systematic changes spread over the whole proof obligations. It seems possible and well worthwhile to take advantage of this systematic relationship to implement powerful transformations. A thorough investigation is needed, however, to see which transformations are useful in practice.

Other proof calculi also deserve further study. In particular it would be interesting to see whether calculi for hierarchical reasoning cf. [Aut03] would make propagating local changes of parts of formulae and terms to parts of proofs easier.

Finally, the idea of development graphs has been applied to the management of change in informal documents [KBHL⁺03], and it should be interesting to see whether meaningful transformations can be given in that context to adapt parts of a document that are affected after a change. It is not clear to us at the moment what kind of transformations would be needed, though.

Extending the Framework. The formulation of the framework in Part II relied on a concrete definition of development graphs with two kinds of global and local links. Development graphs have been extended to allow for different institutions in different nodes of the same graph (heterogeneous development graphs [Mos02]) and to allow for different links (e.g. hiding links [MAH01]). Also, lately development graphs have been formulated and implemented in MAYA in a way that allows for axioms and lemmata in nodes, i.e. each node is separated into two conceptual nodes, one of which uses the other while the latter satisfies the former. It would be interesting to see how the framework extends to these different formulations of development graphs.

Further Integration. We have integrated transformations into the architecture of existing tools for formal software development. This means that transformations can be used in addition to any other activity supported by the respective tools, e.g. proof automation or heuristic replay. However, tighter integration that allows transformations to be used by other activities or to use other activities is not possible as is. This could be very advan-

tageous, and it would be interesting to see whether, e.g., heuristics could be applied to close new open goals that result from patching proofs, or whether proof planning and difference reduction techniques like, e.g., rippling [BSvH⁺93], [BBHI05], equalising terms [Hut97], or critics [Ire96], can help to decide how to patch proofs best after changes. In particular, guiding heuristics may provide real benefit with the replacement of terms, which our instance does not yet handle completely satisfactorily, as discussed above.

Another interesting area worthy of investigation is the combination of correctness preserving transformations, cf. Section 9.3 on page 201, and transformations like the ones we have presented in this thesis. Thus, a change could be applied to, say, the original requirements specification and would then be propagated towards the implementation over the correctness preserving transformations that were applied to derive it.

User Interaction. Interactive theorem proving relies to a large extent upon appropriate user interfaces that support easy manipulation of the development artefacts. In fact, having a rudimentary user interface in our proof of concept implementation turned out to be a noticeable problem when working with realistically sized case studies. Advanced user interface functionalities that we felt would have helped are, e.g., the following.

When a transformation has been carried out, all the open goals and the changed parts on the level of development graphs, proof structure, and proof goals are highlighted and the differences between the state before and after the transformation can be navigated. In particular, all open proof goals can be tracked down easily.

Maybe some of the specification transformations can be recognised mechanically by looking at the old and new specification using an algorithm similar to difference unification [BW93], the difference computation for CASL specifications in [AHMS02], or other techniques known from software merging [Men02], and then inspecting the result for patterns corresponding to known transformations. This would allow transformations to be used exactly like any other basic operation on development graphs in the way in which MAYA uses them.

As a vision, after changing the definition of a function, the system could come back and ask whether the user wanted to add a new argument to the function, and if yes, go off and trigger the corresponding transformation, then change the definition, and finally provide the user with a list of terms and places at which a dummy term has been introduced and further changes are likely necessary.

Chapter 10. Conclusions and Outlook

Some of these issues are research and engineering questions, academically interesting in their own right. Some, however, are questions of making the techniques easier to use in practice, and whether or not they should be tackled depends on economic considerations more than on academic ones. We are confident that the techniques we have presented in this thesis have the potential to change the way in which formal methods are used in industrial practice, but evidently, further work and effort is needed to realise their full potential.

References

- [AF02] Lus Andrade and Jos Luiz Fiadeiro. Coordination primitives for evolving event-based systems. In Hutter et al. [HBLL02], pages 31–41.
- [AH02] Serge Autexier and Dieter Hutter. Maintenance of formal software developments by stratified verification. In *Proceedings 9th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 2514 of *LNAI*, pages 36–52, 2002.
- [AHL⁺00] S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: Formal methods meet industrial needs. *Int. Journal on Software Tools for Technology Transfer*, 3(1), 2000.
- [AHMS99] Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. System description: Inka 5.0 - a logic voyager. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1672 of *LNAI*, pages 207–211. Springer, 1999.
- [AHMS00] Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. Towards an evolutionary formal software-development using CASL. In Didier Bert, Christine Choppy, and Peter Mosses, editors, *Recent Trends in Algebraic Development Techniques, Proceedings of the 14th International Workshop on Algebraic Development Techniques (WADT'99)*, volume 1827 of *LNCS*, pages 73–88. Springer, 2000.
- [AHMS02] Serge Autexier, Dieter Hutter, Till Mossakowski, and Axel Schairer. The development graph manager MAYA (system description). In Kirchner and Ringeissen [KR02], pages 495–501.

REFERENCES

- [ASP02] Matthias Anlauff, Doug Smith, and Dusko Pavlovic. Composition and refinement of evolving specification (invited talk). In Hutter et al. [HBLL02], pages 31–41.
- [Aut03] Serge Autexier. *Hierarchical Contextual Reasoning*. PhD thesis, Universität des Saarlandes, Saarbrücken, 2003.
- [Bal00] Helmut Balzert. *Lehrbuch der Software-Technik, Band 1*. Spektrum Akademischer Verlag, second edition, 2000.
- [BB96] Tim Bull and Keith Bennett. A report on the durham program transformations workshop. *ACM SIGSOFT Software Engineering Notes*, 21(4):51–53, 1996.
- [BBHI05] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Number 56 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2005.
- [BCH⁺04] Hubert Baumeister, Maura Cerioli, Anne Haxthausen, Till Mossakowski, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. Part III: CASL semantics. In Mosses [Mos04], pages 113–271.
- [Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
- [BK04] Bernhard Beckert and Vladimir Klebanov. Proof reuse for deductive program verification. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 77–86, September 2004.
- [BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [BN04] Stefan Berghofer and Tobias Nipkow. Random testing in isabelle/hol. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, September 2004.
- [Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

REFERENCES

- [Boi04] Olivier Boite. Proof reuse with extended inductive types. In *Proceedings of TPHOLs 2004*, volume 3223 of *LNCS*, pages 50–65. Springer, 2004.
- [BSvH⁺93] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.
- [BW93] David Basin and Toby Walsh. Difference unification. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 116–122. Morgan Kaufmann, 1993.
- [BW96] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall, London, second edition, 1996.
- [CC99] Common criteria for information technology security evaluation (CC), 1999. Also ISO/IEC 15408: IT – Security techniques – Evaluation criteria for IT security.
- [Cur95] P. Curzon. The importance of proof maintenance and reengineering. In *Proc. Int. Workshop on Higher Order Logic Theorem Proving and Its Applications*, 1995.
- [dB72] N. G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. *Indagationes Mathematicae*, 34(5):381–392, 1972.
- [dB78] N. G. de Bruijn. Lambda-calculus notation with namefree formulas involving symbols that represent reference transforming mappings. *Indagationes Mathematicae*, 40:348–356, 1978.
- [DGS93] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularisation. In *Papers presented at the Second Annual Workshop on Logical Environments*, pages 83–130, New York, 1993. Cambridge University Press.
- [DM03] Francisco Durán and José Meseguer. Structured theories and institutions. *Theoretical Computer Science*, 309(1):357–380, December 2003.
- [EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1*. Springer, 1985.

REFERENCES

- [FH94] A. Felty and D. Howe. Generalization and reuse of tactic proofs. In *Proc. Int. Conf. Logic Programming and Automated Reasoning (LPAR)*, 1994.
- [Fit96] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, second edition, 1996.
- [FM97] J. Fiadeiro and T. S. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28(2–3):111–138, 1997.
- [Gär99] F. C. Gärtner. Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification. *Journal of Universal Computer Science*, 5(10):668–692, 1999.
- [GB84] J. A. Goguen and R. M. Burstall. Introducing institutions. In E. Clarke and Dexter Kozen, editors, *Proceedings of the Workshop on Logics of Programs*, volume 164 of *LNCS*, pages 221–256. Springer, 1984.
- [GB92] Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, January 1992.
- [Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. Reprinted in English as [Gen69].
- [Gen69] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North Holland, Amsterdam, 1969.
- [GF94] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the IEEE International Conference on Requirements Engineering (ICRE 94)*, pages 94–101, 1994.
- [GHM98] John Grundy, John Hosking, and Rick Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11):960–981, 1998.

REFERENCES

- [Gro00] Lindsay Groves. A formal approach to program modification. In *Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, pages 274–281. IEEE Computer Society, 2000.
- [Gro02] Lindsay Groves. A formal approach to program evolution. In Hutter et al. [HBLL02], pages 31–41.
- [HBLL02] Dieter Hutter, David Basin, Peter Lindsay, and Christoph Lüth, editors. *Proceedings of the First Workshop on Evolutionary Formal Software Development*, July 2002.
- [Hig02] Jim Highsmith. *Agile Software Development Ecosystems*. Addison Wesley, 2002.
- [HKB93] B. Hoffmann and B. Krieg-Brückner, editors. *Program Development by Specification and Transformation*. Springer, Berlin, 1993.
- [HMSS05] Dieter Hutter, Heiko Mantel, Ina Schaefer, and Axel Schairer. Security of multiagent systems: A case study on comparison shopping. *Journal of Applied Logic*, 2005. Special Issue on Logic-Based Agent Verification. Forthcoming.
- [HS01] Dieter Hutter and Axel Schairer. Towards an evolutionary formal software development (short paper). In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)* [IEE01], pages 417–420.
- [Hut97] Dieter Hutter. Equalising terms by difference reduction techniques. In H. Kirchner and B. Gramlich, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction*, 1997.
- [Hut00] Dieter Hutter. Management of change in verification systems. In *Proceedings 15th IEEE International Conference on Automated Software Engineering*, pages 23–34. IEEE Computer Society, 2000.
- [IEE01] IEEE Computer Society. *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, November 2001.
- [Ire96] Andrew Ireland. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

REFERENCES

- [JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
- [JL04a] Einar Broch Johnsen and Christoph Lüth. Abstracting transformations for refinement. *Nordic Journal of Computing*, 10:316–336, 2004.
- [JL04b] Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *International Conference on Theorem Proving in Higher-Order Logics TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 152–167. Springer, September 2004.
- [KBHL⁺03] Bernd Krieg-Brückner, Dieter Hutter, Christoph Lüth, Erica Melis, Arnd Pötsch-Heffter, Markus Roggenbach, Jan-Georg Smaus, and Martin Wirsing. Towards multimedia instruction in safe and secure systems. In *Recent Trends in Algebraic Development Techniques (WADT-02)*, volume 2755 of *LNCS*. Springer, 2003.
- [KBM04] Bernd Krieg-Brückner and Peter D. Mosses. Part I: CASL summary. In Mosses [Mos04].
- [Kol97] Thomas Kolbe. *Optimizing Proof Search by Machine Learning Techniques*. PhD thesis, Fachbereich Informatik der Technischen Hochschule Darmstadt, 1997. Published as a manuscript by Shaker, Aachen.
- [KR02] Hélène Kirchner and Christophe Ringeissen, editors. *Proceedings of the 9th International Conference on Algebraic Methodology And Software Technology (AMAST 2002)*, volume 2422 of *LNCS*. Springer, 2002.
- [KW94] Thomas Kolbe and Christoph Walther. Reusing proofs. In A. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 80–84, 1994.
- [Läm04] Ralf Lämmel. Coupled software transformations (extended abstract). In *First International Workshop on Software Transformations*, November 2004. Available online at <http://banff.cs.queensu.ca/set2004/>.

REFERENCES

- [Lar04] Craig Larman. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley, 2004.
- [LEW96] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. Wiley and Teubner, 1996.
- [LKB02] Craig Larman, Philippe Kruchten, and Kurt Bitner. How to fail with the rational unified process: Seven steps to pain and suffering. Available online at <http://www.aanpo.org/articles/>, 2002.
- [LS86] J. Lambek and P. J. Scott. *Introduction to higher order categorial logic*, volume 7 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1986.
- [LUV00] Bruno Langenstein, Markus Ullmann, and Roland Vogt. The use of formal methods for trusted digital signature devices. In *Proc. 13th Intern. FLAIRS Conf.* AAAI Press, 2000.
- [MAH01] Till Mossakowski, Serge Autexier, and Dieter Hutter. Extending development graphs with hiding. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *LNCS*, pages 269–283. Springer, 2001.
- [Man00] Heiko Mantel. Unwinding possibilistic security properties. In *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, LNCS 1895, pages 238–254, Toulouse, France, October 4-6 2000. Springer.
- [Man03] Heiko Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, Universität des Saarlandes, 2003.
- [McC96] Steve McConnell. *Rapid Development*. Microsoft Press, 1996.
- [Men02] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.
- [MG00] H. Mantel and F. C. Gärtner. A case study in the mechanical verification of fault tolerance. *Journal of Experimental & Theoretical Artificial Intelligence*, 12:473–487, 2000.

REFERENCES

- [MHAH04] Till Mossakowski, Piotr Hoffman, Serge Autexier, and Dieter Hutter. Part IV: CASL logic. In Mosses [Mos04], pages 273–359.
- [Mos02] Till Mossakowski. Heterogeneous development graphs and heterogeneous borrowing. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of LNCS, pages 326–341. Springer, 2002.
- [Mos04] Peter D. Mosses, editor. *CASL Reference Manual – The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of LNCS. Springer, 2004.
- [MS98] Erica Melis and Axel Schairer. Similarities and reuse of proofs in formal software verification. In Barry Smyth and Pádraig Cunningham, editors, *Advances in Case-Based Reasoning, Proceedings of the Fourth European Workshop on Case Based Reasoning (EWCBR 98)*, volume 1488 of LNAI, pages 76–87. Springer, 1998.
- [MS05] Heiko Mantel and Axel Schairer. Exploiting generic aspects of security models in formal developments. In Dieter Hutter and Werner Stephan, editors, *Essays in Honor of Jörg H. Siekmann on the Occasion of his 60th Birthday*, volume 2605 of LNAI. Springer, 2005.
- [MSK⁺01] Heiko Mantel, Axel Schairer, Matthias Kabatnik, Michael Kreuzer, and Alf Zugenmaier. Using information flow control to evaluate access protection of location information in mobile communication networks. Technical Report 159, Institut für Informatik, Universität Freiburg, August 2001.
- [MT04] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
- [MW97] E. Melis and J. Whittle. External analogy in inductive theorem proving. In *KI-97: Advances in Artificial Intelligence*, pages 111–122. Springer, 1997.
- [Opd92] Willian F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

REFERENCES

- [Par90] Helmut A. Partsch. *Specification and Transformation of Programs. A Formal Approach to Software Development*. Springer, 1990.
- [Pau00] Larry Paulson. Isabelle User Mailing List, 26 October, 19 December 2000. Available from <http://www.cl.cam.ac.uk/users/lcp/archive>.
- [PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 199–208. ACM, 1988.
- [Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, Massachusetts, 1991.
- [PS01] Dusko Pavlovic and Douglas R. Smith. Composition and refinement of behavioral specifications. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001) [IEE01]*, pages 157–165.
- [PS02] Dusko Pavlovic and Douglas R. Smith. Guarded transitions in evolving specifications. In Kirchner and Ringeissen [KR02], pages 411–425.
- [RJ01] Bala Ramesh and Matthias Jarke. Towards reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1), 2001.
- [Roy70] W. W. Royce. Managing the development of large-scale software: Concepts and techniques. In *Proceedings, Wescon*, August 1970. Reprinted as [Roy87].
- [Roy87] W. W. Royce. Managing the development of large-scale software: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, Monterey, California, USA, 1987. ACM Press.
- [RS93] Wolfgang Reif and Kurt Stenzel. Reuse of proofs in software verification. In R. Shyamasundar, editor, *Foundation of Software Technology and Theoretical Computer Science*, volume 761 of LNCS, Berlin, 1993. Springer.

REFERENCES

- [RSB00] Georg Rock, Werner Stephan, and Michael Brodski. Modeling, specification and verification of an emergency closing system. In *The 13th International FLAIRS Conference, Special Track on Verification, Validation and System Certification*, Orlando, Florida, 2000.
- [RSW97] Georg Rock, Werner Stephan, and Andreas Wolpers. Tool support for the compositional development distributed systems. In Adam Wolisz and Axel Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch*, volume 315 of *GMD-Studien*, pages 89–98, Berlin, 1997. GMD-Forschungszentrum Informationstechnik GmbH.
- [RSW⁺99] Georg Rock, Werner Stephan, Andreas Wolpers, Michael Balsler, Wolfgang Reif, and Stefan Scheer. Structured formal development in VSE II: The Robertino case study. In Francesca Saglietti and Wolfgang Goerigk, editors, *Sicherheit und Zuverlässigkeit software-basierter Systeme, Bericht ISTec-A-367*, pages 138–152, 1999.
- [Rus01] J. Rushby. Security requirements specifications: How and what? In *Symposium on Requirements Engineering for Information Security (SREIS)*, 2001.
- [SAH01] Axel Schairer, Serge Autexier, and Dieter Hutter. A pragmatic approach to reuse in tactical theorem proving. In Maria Paola Bonacina and Bernhard Gramlich, editors, *Proceedings of the 4th International Workshop on Strategies in Automated Deduction (STRATEGIES 2001)*, pages 75–86, 2001.
- [SBRS04] Jan Scheffczyk, Uwe M. Borghoff, Peter Rödiger, and Lothar Schmitz. Managing inconsistent repositories via prioritized repairs. In E. V. Munson and J.-Y. Vion-Dury, editors, *Proceedings of the ACM Symposium on Document Engineering*, pages 137–146, 2004.
- [Sch98] Axel Schairer. A technique for reusing proofs in software verification. Diplomarbeit, FB 14 (Informatik) der Universität des Saarlandes und Institut A für Mechanik der Universität Stuttgart, Saarbrücken/Stuttgart, March 1998.

REFERENCES

- [Sch02] Axel Schairer. Proof transformations for reusing proofs after changing subformulae of verification conditions. In Hutter et al. [HBLL02], pages 31–41.
- [Sch03] Ina Schaefer. Information flow control for multiagent systems – a case study on comparison shopping. Diplomarbeit, Universität Rostock, Fachbereich Informatik, August 2003. Also available as [Sch04].
- [Sch04] Ina Schaefer. Information flow control for multiagent systems – a case study on comparison shopping. Technical Report RR-04-01, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI) GmbH, Saarbrücken, January 2004.
- [SH02] Axel Schairer and Dieter Hutter. Proof transformations for evolutionary formal software development. In Kirchner and Ringeissen [KR02], pages 441–456.
- [Som95] Ian Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1995.
- [SR03] Matt Stephens and Doug Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Apress, 2003.
- [SRS⁺00] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger, Vernon Austel, and David Toll. Verification of a formal security model for multiapplicative smart cards. In Frédéric Cuppens, Yves Deswarte, Dieter Gollmann, and Michael Waidner, editors, *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, volume 1895 of *LNCS*, pages 17–36. Springer, 2000.
- [SRS⁺02] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger, Vernon Austel, and David Toll. Verified formal security models for multiapplicative smart cards. *Journal of Computer Security*, 10(4):339–367, 2002.
- [vK01] Antje von Knethen. A trace model for system requirements changes on embedded systems. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 17–26, 2001.

REFERENCES

- [vK02] Antje von Knethen. Automatic change support based on a trace model. In *Proceedings of the Traceability Workshop*, Edinburgh, 2002.
- [vKP02] Antje von Knethen and Barbara Paech. A survey on tracing approaches in practice and research. IESE-Report 095.01/E, Fraunhofer IESE, January 2002.
- [Voe01] Jeroen Voeten. On the fundamental limitations of transformational design. *ACM Transactions on Design and Automation of Electronic Systems*, 6(4):533–552, October 2001.
- [WBB02] John Whittle, Alan Bundy, and Richard Boulton. Proofs-as-programs as a framework for the design of an analogy-based ml editor. *Formal Aspects of Computing*, 13(3–5):403–421, 2002.
- [WBBL99] J. Whittle, A. Bundy, R. Boulton, and H. Lowe. An ML editor based on proofs-as-programs. In *Proc. Int. Conf. Automated Software Engineering (ASE)*, 1999.
- [WC00] Jon Whittle and Andrew Cumming. Evaluating environments for functional programming. *International Journal of Human Computer Studies*, 52(5):847–878, 2000.
- [Wil05] Gregory V. Wilson. Extensible programming for the 21st century. *ACM Queue*, 2(9), January 2005.

Index

- $2^{\mathbf{F}}$, 233
- \models_{Σ} , 40
- Σ , 40
- Θ , 52
- $|\mathbf{A}|$, 231
- $\mathbf{F} \times \mathbf{G}$, 234
- \circ , 231
- $\#_A(n)$, 48
- $n_1 \xrightarrow{\sigma} n_2$, 41
- Φ_D^n , 43
- $n \xRightarrow{\sigma} n'$, 45
- id_A , 231
- σ^n , 41
- $n_1 \xrightarrow{\sigma} n_2$, 41
- Φ^n , 41
- Σ^n , 41
- $n \xrightarrow{\sigma} n'$, 45
- $C \downarrow_{N,A}$, 49
- σ , 40
- ϑ , 88
- η_{Σ} , 52
 - concrete, 113
- axioms
 - global, 43
- A , 41
- abstract datatype, 14, 24
- adequateness, 33
- algebra, 102, 238
- algebraic signature morphism, 102
- algebraic specification, 24
- antecedent, 111
- arrows, 231
- axiomatic specification, 14
- basic development graph operations,
 - 96
- basic specifications, 49
- C , 41
- calculus rules, 115
- category, 231
- completed development, 55
- completeness, 33, 63
- composition operator, 231
- concl*, 52
 - concrete, 113
- conclusion, 45
- concrete institution, 104
- consequence morphism, 41
- consequence relations, 41
- consistency, 19
- context, 113
- context formula, 113
- coupled transformations, 60
- covered proof obligations, 54
- D , 41
- de Bruijn-indices, 102
- decomposition rule, *see* development
 - graph calculus
- definitorial global path, 45
- definitorial links, 41
- definitorial path, 45
- depth
 - of link, 48
 - of node, 48, 77, 83, 90

INDEX

- dev*, 55
- development
 - state of, 55
 - well-formed, 55
- development graph, 41
- development graph calculus, 46, 68–71
- development graph transformation, 58, 60
 - Tr_{axioms} , 72
 - Tr_{links} , 66
 - Tr_{nodes} , 65
 - Tr_{occ} , 76
 - $Tr_{signatures}$, 80
 - $Tr_{translate}$, 88
- development graph translation, 88, 94
- development graphs
 - for first order logic, 105
- development transformation, 60
- $dg(S)$, 50
- E_C , 131
- equality, 237
- extended category of signatures, 88
- extended goal functor, 92
- extended proof functor, 92
- extended proof representation, 92, 94
- extended sentence functor, 88, 94
- fault assumptions, 22
- first order logic, 101, 237
- focus formula, 113
- FolEqGen, 104, 237
- formal development, 55
- formulae, 101, 237
- function, 237
- functor, 231
- functor lifted to product, 234
- functor lifted to sets, 233
- generatedness constraint, 102, 239
- global changes, 76
- global path, 45
- Goal**, 52
 - concrete, 113
- Σ -goals, 52
 - concrete, 113
- graph isomorphism, 68
- h , 68
- \bar{h} , 70
- higher-order abstract syntax, 102
- \mathcal{I} , 40, 104
- identity arrow, 231
- implied link, 44
- induction formula, 116
- information flow properties, 20
- institution, 39, 40
- justification, 112
- l , 148
- links, 42
- local axioms, 42
- local signature symbols, 97
- logic, 39
- logical consequence, 40
- management of change, 10
- Mod**, 40
- Σ -models, 40
- model-theoretic semantics, 102
- models, 7
- morphism, 231
- moving axioms, 72
- N , 41
- natural deduction, 52
- natural transformation, 232
- nodes, 42
- non-dependency, 23
- normal form, 49

- objects, 231
- Obl**, 53
- obligation links, 49
- obligation transformation, 58, 60
- $obl(D)$, 51
- P , 55
- path, 45
- postulated links, 41
- predicate, 237
- predicates, 24
- premiss, 45
- Prf**, 52
 - concrete, 113
- proof obligations, 51
 - of specification, 50
- proof representation, 52
 - concrete, 113
- proof rule, 113
- proof transformation, 58
- proof tree, 112
- Σ -proofs, 52
 - concrete, 113
- q , 75
- q_n^* , 76
- r , 148
- Σ -reduct, 40
- reduct theorem, 239
- respecting morphisms, 69
- S , 55
- Spec*, 50
- satisfaction condition, 40, 240
- satisfaction relation, 40
- satisfied development graph, 44, 50
- satisfied specification, 50
- satisfies clause, 109
- Sen**, 40
- sentence replacement, 75
- sentence translation, 88
- Σ -sentences, 40
- sequent, 111
- sequent calculus, 52, 111, 240
- Sig**, 40
- signature, 40
 - algebraic, 237
- signature extension, 79
- signature morphism, 40
 - algebraic, 40, 238
- signature restriction, 79
- signature symbols, 97, 237
- signature translation, 88
- simple generatedness constraint, 102
- sort, 237
- specification in the large, 49
- specification in the small, 49
- specification language, 50
- specification transformation, 58
- SSL, 106
- subsumption rule, *see* development
 - graph calculus
- succedent, 111
- surface syntax
 - formulae, 105
 - specifications, 106
- terms, 101, 237
- theory, 42
- transformation, 58
- transformation instance, 58
- transformation rule, 58
- translation, 88
- truth value, 238
- typed formulae, 237
- typed terms, 237
- uses clause, 108
- variable, 237
- variable assignment, 238
- variable context, 237
- verified development, 55

INDEX

verified development graph, 51

verified specification, 50

well-formedness conditions, 49, 110

XGoal, 92

XOb1, 92

XPrf, 92

XSen, 88

XSig, 88

Part V
Appendix

Appendix A

Category, Functor, Natural Transformation

For ease of reference, we provide short definitions of category, functor, and natural transformation (cf. [Pie91], Definitions 1.1.1, 2.1.1, and 2.3.1, or [BW96], Paragraphs 2.1.3, 3.1.1, and 4.2.10) in the notation that we use in the main part of the thesis.

Definition A.1 (Category) *A category \mathbf{A} consists of a collection of objects $|\mathbf{A}|$, a collection of arrows (or morphisms) between objects, a composition operator \circ on arrows, and an identity arrow $\text{id}_A : A \rightarrow A$ for each object A , such that the following conditions all hold:*

- *If $f : A \rightarrow B$ and $g : B \rightarrow C$ are arrows from A to B and from B to C , respectively, then $g \circ f : A \rightarrow C$ is an arrow from A to C .*
- *If additionally $h : C \rightarrow D$ is an arrow then $h \circ (g \circ f) = (h \circ g) \circ f$.*
- *If $f : A \rightarrow B$ is an arrow, then $\text{id}_B \circ f = f$ and $f \circ \text{id}_A = f$.*

Definition A.2 (Functor) *Let \mathbf{C} and \mathbf{D} be categories. A functor $\mathbf{F} : \mathbf{C} \rightarrow \mathbf{D}$ is a map taking each \mathbf{C} -object A to a \mathbf{D} -object $\mathbf{F}(A)$ and each \mathbf{C} -arrow $f : A \rightarrow B$ to a \mathbf{D} -arrow $\mathbf{F}(f) : \mathbf{F}(A) \rightarrow \mathbf{F}(B)$, such that for all \mathbf{C} -objects A and composable \mathbf{C} -arrows f and g ,*

- $\mathbf{F}(\text{id}_A) = \text{id}_{\mathbf{F}(A)}$ and
- $\mathbf{F}(g \circ f) = \mathbf{F}(g) \circ \mathbf{F}(f)$

are satisfied.

Appendix A. Category, Functor, Natural Transformation

Definition A.3 (Natural transformation) Let \mathbf{C} and \mathbf{D} be categories and let $\mathbf{F} : \mathbf{C} \rightarrow \mathbf{D}$ and $\mathbf{G} : \mathbf{C} \rightarrow \mathbf{D}$ be functors. A natural transformation η from \mathbf{F} to \mathbf{G} is a function that assigns to every \mathbf{C} -object A a \mathbf{D} -arrow $\eta_A : \mathbf{F}(A) \rightarrow \mathbf{G}(A)$ such that for any \mathbf{C} -arrow $f : A \rightarrow B$ the diagram

$$\begin{array}{ccc} \mathbf{F}(A) & \xrightarrow{\eta_A} & \mathbf{G}(A) \\ \mathbf{F}(f) \downarrow & & \downarrow \mathbf{G}(f) \\ \mathbf{F}(B) & \xrightarrow{\eta_B} & \mathbf{G}(B) \end{array}$$

commutes in \mathbf{D} , i.e. that $\mathbf{G}(f) \circ \eta_A = \eta_B \circ \mathbf{F}(f)$.

Appendix B

Functors Lifted to Sets and Tuples

A functor $\mathbf{F} : \mathbf{A} \rightarrow \mathbf{Set}$ assigns a set to each $a \in |\mathbf{A}|$. For example, $\mathbf{Sen}(\Sigma)$ is the set of all sentences over the signature Σ , so $\varphi \in \mathbf{Sen}(\Sigma)$ expresses that φ is a Σ -sentence. We are interested in the associated functor that maps each signature Σ to the powerset of the set of all sentences over Σ , written $2^{\mathbf{Sen}}$, so $\Gamma \in 2^{\mathbf{Sen}(\Sigma)}$ expresses that Γ is a set of Σ -sentences. The benefit of formulating this as a functor is that we can naturally express the canonical relationship between translating sentences and between uniformly translating sets of sentences pointwise, e.g. in Definition 3.19 on page 52.

Definition B.1 (Functor lifted to sets) *Given a functor $\mathbf{F} : \mathbf{A} \rightarrow \mathbf{Set}$, the functor \mathbf{F} lifted to sets, written as $2^{\mathbf{F}} : \mathbf{A} \rightarrow \mathbf{Set}$, is defined by*

$$\begin{aligned} 2^{\mathbf{F}}(a) &= 2^{\mathbf{F}(a)} = \{A \mid A \subseteq \mathbf{F}(a)\} && \text{for } a \in |\mathbf{A}| \\ 2^{\mathbf{F}}(f)(A) &= \mathbf{F}(f)(A) = \{\mathbf{F}(f)(a) \mid a \in A\} && \text{for } \mathbf{A}\text{-arrow } f \end{aligned}$$

where we write 2^A for the powerset of A .

Theorem B.2 $2^{\mathbf{F}}$ is indeed a functor from \mathbf{A} to \mathbf{Set} .

Proof of B.2 We have to show the following.

1. $a \in |\mathbf{A}|$ implies $2^{\mathbf{F}}(a) \in |\mathbf{Set}|$ trivially.
2. f is an \mathbf{A} -arrow implies $2^{\mathbf{F}}(f) : |\mathbf{Set}| \rightarrow |\mathbf{Set}|$ trivially.

Appendix B. Functors Lifted to Sets and Tuples

3. $2^{\mathbf{F}}(\text{id}_a) = \text{id}_{2^{\mathbf{F}}(a)}$: The left hand side is defined for $A \subseteq \mathbf{F}(a)$, and we have

$$\begin{aligned} 2^{\mathbf{F}}(\text{id}_a)(A) &= \{\mathbf{F}(\text{id}_a)(a') \mid a' \in A\} \\ &= \{\text{id}_{\mathbf{F}(a)}(a') \mid a' \in A\} && \mathbf{F} \text{ is functor} \\ &= \{a' \mid a' \in A\} && \text{id}_{\mathbf{F}(a)} \text{ is identity over } \mathbf{F}(a) \\ &= A, \end{aligned}$$

so $2^{\mathbf{F}}(\text{id}_a)$ is the identity function on $2^{\mathbf{F}}(a)$ as required.

4. $2^{\mathbf{F}}(g \circ f) = 2^{\mathbf{F}}(g) \circ 2^{\mathbf{F}}(f)$: The left hand side is defined for $A \subseteq \mathbf{F}(a)$, and we have

$$\begin{aligned} 2^{\mathbf{F}}(g \circ f)(A) &= \{\mathbf{F}(g \circ f)(a) \mid a \in A\} \\ &= \{\mathbf{F}(g)(\mathbf{F}(f)(a)) \mid a \in A\} \\ &= \{\mathbf{F}(g)(b) \mid b \in \{\mathbf{F}(f)(a) \mid a \in A\}\} \\ &= 2^{\mathbf{F}}(g)(2^{\mathbf{F}}(f)(A)) \\ &= (2^{\mathbf{F}}(g) \circ 2^{\mathbf{F}}(f))(A) \end{aligned}$$

as required. □

Similarly, we define \times for functors.

Definition B.3 (Functor lifted to cartesian product) *Given two functors $\mathbf{F} : \mathbf{A} \rightarrow \mathbf{Set}$ and $\mathbf{G} : \mathbf{A} \rightarrow \mathbf{Set}$, the functor $\mathbf{F} \times \mathbf{G} : \mathbf{A} \rightarrow \mathbf{Set}$ is defined by*

$$\begin{aligned} (\mathbf{F} \times \mathbf{G})(a) &= \mathbf{F}(a) \times \mathbf{G}(a) && \text{for } a \in |\mathbf{A}| \\ (\mathbf{F} \times \mathbf{G})(f)(\langle a_1, b_2 \rangle) &= \langle \mathbf{F}(f)(a_1), \mathbf{G}(f)(b_2) \rangle && \text{for } \mathbf{A}\text{-arrow } f. \end{aligned}$$

Theorem B.4 $\mathbf{F} \times \mathbf{G}$ is indeed a functor from \mathbf{A} to \mathbf{Set} .

Proof of B.4 Again, we have to show the following:

1. $a \in |\mathbf{A}|$ implies $(\mathbf{F} \times \mathbf{G})(a) \in |\mathbf{Set}|$ trivially.
2. f is an \mathbf{A} -arrow implies $(\mathbf{F} \times \mathbf{G})(f) : |\mathbf{Set}| \rightarrow |\mathbf{Set}|$ trivially.

3. $(\mathbf{F} \times \mathbf{G})(\text{id}_a) = \text{id}_{(\mathbf{F} \times \mathbf{G})(a)}$: The left hand side is defined for $\langle b_1, b_2 \rangle \in (\mathbf{F} \times \mathbf{G})(a) = \mathbf{F}(a) \times \mathbf{G}(a)$, i.e. $b_1 \in \mathbf{F}(a)$ and $b_2 \in \mathbf{G}(a)$. Then

$$\begin{aligned}
 (\mathbf{F} \times \mathbf{G})(\text{id}_a)(\langle b_1, b_2 \rangle) &= \langle \mathbf{F}(\text{id}_a)(b_1), \mathbf{G}(\text{id}_a)(b_2) \rangle \\
 &= \langle \text{id}_{\mathbf{F}(a)}(b_1), \text{id}_{\mathbf{G}(a)}(b_2) \rangle \\
 &= \langle b_1, b_2 \rangle \\
 &= \text{id}_{\mathbf{F}(a) \times \mathbf{G}(a)}(\langle b_1, b_2 \rangle) \\
 &= \text{id}_{(\mathbf{F} \times \mathbf{G})(a)}(\langle b_1, b_2 \rangle)
 \end{aligned}$$

as required.

4. $(\mathbf{F} \times \mathbf{G})(g \circ f) = (\mathbf{F} \times \mathbf{G})(g) \circ (\mathbf{F} \times \mathbf{G})(f)$: The left hand side is defined for $\langle b_1, b_2 \rangle \in (\mathbf{F} \times \mathbf{G})(a) = \mathbf{F}(a) \times \mathbf{G}(a)$, i.e. $b_1 \in \mathbf{F}(a)$ and $b_2 \in \mathbf{G}(a)$. Then

$$\begin{aligned}
 (\mathbf{F} \times \mathbf{G})(g \circ f)(\langle b_1, b_2 \rangle) &= \langle \mathbf{F}(g \circ f)(b_1), \mathbf{G}(g \circ f)(b_2) \rangle \\
 &= \langle (\mathbf{F}(g) \circ \mathbf{F}(f))(b_1), (\mathbf{G}(g) \circ \mathbf{G}(f))(b_2) \rangle \\
 &= \langle \mathbf{F}(g)(\mathbf{F}(f)(b_1)), \mathbf{G}(g)(\mathbf{G}(f)(b_2)) \rangle \\
 &= (\mathbf{F} \times \mathbf{G})(g)(\langle \mathbf{F}(f)(b_1), \mathbf{G}(f)(b_2) \rangle) \\
 &= (\mathbf{F} \times \mathbf{G})(g)((\mathbf{F} \times \mathbf{G})(f)(\langle b_1, b_2 \rangle)) \\
 &= ((\mathbf{F} \times \mathbf{G})(g) \circ (\mathbf{F} \times \mathbf{G})(f))(\langle b_1, b_2 \rangle)
 \end{aligned}$$

as required.

□

Appendix C

Details of the Definition of FolEqGen

For reference purposes, we provide further details for the definition of the institution in Chapter 5. The intention is that the body of the thesis is accessible without referring to these details.

Definition C.1 (First order logic with equality) *Let $\mathcal{S}, \mathcal{F}, \mathcal{P}, \mathcal{V}$ be countably infinite, pairwise disjoint sets of sort, function, predicate, and variable names. An algebraic first order signature (or signature for short) Σ is determined by finite sets $\Sigma_S \subset \mathcal{S}, \Sigma_F \subset \mathcal{F}, \Sigma_P \subset \mathcal{P}$ of sort, function, and predicate symbols, and by a mapping from function symbols in Σ_F to a finite, non-empty sequence of sort symbols from Σ_S , and from predicate symbols in Σ_P to finite, possibly empty sequences of sort symbols from Σ_S . We write $s \in \Sigma$ for ' $s \in \Sigma_S$ ', $(f : s_1 \times \dots \times s_n \rightarrow s) \in \Sigma$ for ' $f \in \Sigma_F$ and Σ maps f to the sequence $\langle s_1, \dots, s_n, s \rangle$ ', and $(p : s_1 \times \dots \times s_n) \in \Sigma$ for ' $p \in \Sigma_P$ and Σ maps p to the sequence $\langle s_1, \dots, s_n \rangle$ '.*

A variable context C is a partial, finite mapping from \mathcal{V} to Σ_S . $C[x : s]$ is the variable context that is like C except x is mapped to s . The set of terms of sort s wrt. Σ and C for each $s \in \Sigma$ is the smallest set such that the following conditions are satisfied.

- $C(x) = s$ implies x is a term of sort s .
- $(f : s_1 \times \dots \times s_n \rightarrow s) \in \Sigma$ and t_i is a term of sort s_i (for $1 \leq i \leq n$) implies $f(t_1, \dots, t_n)$ is a term of sort s .

The set of formulae wrt. Σ and C is defined to be the smallest set satisfying the following conditions.

- \top and \perp are formulae.

Appendix C. Details of the Definition of FolEqGen

- If t_1 and t_2 are terms of sort s , then $t_1 = t_2$ is a formula.
- If φ and ψ are formulae, so is $\neg\varphi$ and $\varphi \wedge \psi$, and similarly for all other connectives.
- If φ is a formula wrt. $C[x : s]$, then $\forall x : s. \varphi$ and $\exists x : s. \varphi$ are formulae wrt. C .

Let M be a Σ -algebra. A variable assignment X for M and a variable context C is a mapping from variable symbols in C into the respective carrier, i.e. X is such that $C(x) = s$ implies $X(x) \in M(s)$. $X[x/v]$ is the variable assignment that is like X except that it assigns v to x . The valuation of a Σ -term t wrt. a Σ -algebra M and a variable assignment X for M , written as $M_X(t)$, is defined by

$$\begin{aligned} M_X(x) &= X(x) \\ M_X(f(t_1, \dots, t_n)) &= M(f)(M_X(t_1), \dots, M_X(t_n)) . \end{aligned}$$

The truth value of a formula φ wrt. a Σ -algebra M and a variable assignment X , written as $M_X(\varphi)$ is defined by

$$\begin{aligned} M_X(\top) &\text{ is true} \\ M_X(\perp) &\text{ is false} \\ M_X(t_1 = t_2) &\text{ is true iff } M_X(t_1) = M_X(t_2) \\ M_X(\neg\varphi) &\text{ is true iff } M_X(\varphi) \text{ is false} \\ M_X(\varphi \wedge \psi) &\text{ is true iff } M_X(\varphi) \text{ is true and } M_X(\psi) \text{ is true} \\ &\text{(and similarly for the other connectives)} \\ M_X(\forall x : s. \varphi) &\text{ is true iff } M_{X[x/v]}(\varphi) \text{ is true for all } v \in M(s) \\ M_X(\exists x : s. \varphi) &\text{ is true iff } M_{X[x/v]}(\varphi) \text{ is true for some } v \in M(s) . \end{aligned}$$

For each closed Σ -formula φ and Σ -algebra M we write $M(\varphi)$ for the truth value of φ in M . For each closed Σ -formula φ and Σ -algebra M , we say that φ holds in M iff $M(\varphi)$ is true, or that M satisfies φ , and write $M \models_{\Sigma} \varphi$.

Definition C.2 (Signature morphism) Let Σ and Σ' be two signatures. An algebraic signature morphism (or signature morphism for short) $\sigma : \Sigma \rightarrow \Sigma'$ from Σ to Σ' is a mapping from sort (function, predicate) symbols in Σ to sort (function, predicate, respectively) symbols in Σ' such that function and predicate profiles are preserved, i.e. if $\sigma f = f'$ and $(f : s_1 \times \dots \times s_n \rightarrow s) \in \Sigma$ then $f' : s'_1 \times \dots \times s'_m \rightarrow s' \in \Sigma'$, $n = m$, $\sigma s_i = s'_i$ for $1 \leq i \leq n$, and $\sigma s = s'$, and similarly for predicates. A signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ maps a variable context C for Σ to the variable context σC for Σ' such that $\sigma(C(x)) = \sigma(C(x))$

for every x in C . We also write σ for the homomorphic extension of σ to terms, i.e.

$$\begin{aligned}\sigma x &= x \\ \sigma(f(t_1, \dots, t_n)) &= (\sigma f)(\sigma t_1, \dots, \sigma t_n)\end{aligned}$$

and formulae, i.e.

$$\begin{aligned}\sigma \top &= \top \\ \sigma \perp &= \perp \\ \sigma(t_1 = t_2) &= (\sigma t_1) = (\sigma t_2) \\ \sigma(\neg \varphi) &= \neg(\sigma \varphi) \\ \sigma(\varphi \circ \psi) &= (\sigma \varphi) \circ (\sigma \psi) \\ \sigma(\forall x : s. \varphi) &= \forall x : (\sigma s). (\sigma \varphi) \\ \sigma(\exists x : s. \varphi) &= \exists x : (\sigma s). (\sigma \varphi).\end{aligned}$$

Note that variables are unchanged.

The following proposition is called *reduct theorem*, e.g. [LEW96, Theorem 4.12].

Proposition C.3 (Reduct theorem) For all signature morphisms $\sigma : \Sigma \rightarrow \Sigma'$, all Σ' -algebras M' , and all closed Σ -formulae φ , M' satisfies $\sigma \varphi$ iff $M'|_\sigma$ satisfies φ (where $M'|_\sigma$ is the σ -reduct of M' , i.e. $M'|_\sigma(s) = M'(\sigma s)$ and similarly for functions and predicates).

Generatedness constraints consist of the constructor functions, the sort that is generated, and a signature morphisms. This allows a simple definition of generatedness that is invariant under translation of constraints and reducts of models (cf. [BCH⁺04, pp. 143]).

Definition C.4 (Generatedness constraint) A generatedness constraint c for signature Σ' from signature Σ is a tuple $c = (F, s, \sigma)$ such that $\sigma : \Sigma \rightarrow \Sigma'$ is a signature morphism, F is a non-empty set of function symbols from Σ with the same codomain s (i.e. for each $f \in F$, $(f : \dots \rightarrow s) \in \Sigma$). A Σ' -algebra M' satisfies the generatedness constraint (F, s, σ) iff for every value $v \in M'(\sigma s)$ there is a term t wrt. Σ that consists of functions from F and variables of types different from s , and a variable assignment X' for Σ' such that $M'_{X'}(\sigma t) = v$. We write $M \models_\Sigma c$ iff M satisfies the constraint c .

A generatedness constraint (F, s, σ) for Σ' from Σ is mapped along a signature morphism $\sigma' : \Sigma' \rightarrow \Sigma''$ by $\sigma'(F, s, \sigma) = (F, s, \sigma' \circ \sigma)$.

Appendix C. Details of the Definition of FolEqGen

Proposition C.5 If c is a generatedness constraint for Σ' from Σ and $\sigma' : \Sigma' \rightarrow \Sigma''$ is an algebraic signature morphism, then $\sigma'c$ is a generatedness constraint for Σ'' from Σ .

The equivalent of the reduct theorem for generatedness constraints is as follows.

Proposition C.6 For all signature morphisms $\sigma : \Sigma \rightarrow \Sigma'$, Σ' -algebras M' , and generatedness constraints c for Σ' , M' satisfies σc iff $M'|_\sigma$ satisfies c .

The proof is fairly simple. The relationship between M' and $\sigma(\sigma_0 t)$ on the one side and $M'|_\sigma$ and $\sigma_0 t$ (where σ_0 is the third component of the constraint and t is the term for a given carrier) on the other is given by the definition of the reduct operation together with the fact that variable assignments for an algebra are also variable assignments for the reduct of the algebra (this amounts to a variant of the substitution theorem, cf. [LEW96, Theorem 2.41]; note that in our definition $\sigma x = x$ for variables) and *vice versa*, and that the valuation of a term is invariant under translation and reducts. Additionally, neither the set of carriers for which we require witness terms nor the signature from which the witness terms are drawn changes.

Theorem C.7 satisfaction condition Satisfaction of formulae and generatedness constraints is compatible with reducts of algebras and translation of formulae and constraints along signature morphisms. I.e., let $\sigma : \Sigma \rightarrow \Sigma'$ be a signature morphism and M' a Σ' -algebra. Then

$$\begin{aligned} M' \models_{\Sigma'} \sigma\varphi & \text{ iff } M|_\sigma \models_{\Sigma} \varphi \text{ for each closed } \Sigma\text{-formula } \varphi \text{ and} \\ M' \models_{\Sigma'} \sigma c & \text{ iff } M|_\sigma \models_{\Sigma} c \text{ for each generatedness constraint } c \text{ for } \Sigma. \end{aligned}$$

A correct, but obviously incomplete, calculus for this logic can be given by the usual sequent calculus rules for first order logic, rules for equality, and an induction rule. This means that positive generatedness constraints can only contribute to a proof in the axiom rule. Thus, a constraint cannot be proved from another one. Sometimes, more flexibility should be provided by giving rules to derive one constraint from others. Also, the concrete form of the induction rule is relatively inflexible: in practice it is usually more convenient, if induction about eigenvariables is possible, i.e. the choice about whether to carry out an induction or not can be delayed.

Appendix D

Details of the Case Study

We have looked at examples in a proof-of-concept implementation and have presented some of them in Chapter 8. Here we provide some further details.

D.1 Development Trace

The implementation is controlled through a command line interface. This makes it easy to store transformation sequences as scripts but makes controlling the system difficult for users. For practical applications, a better interface is needed. A web-based read-only interface has been provided to inspect the states of the development. Each transformation makes a copy of the current state of the development and stores it away so that it is possible to inspect the whole history. Figure D.1 shows part of the history of the example that we have presented in Chapter 8. The text for each numbered item is the input that was typed at the command line, except for traditional work on proofs alone, which is just captured by the overall result, cf. the item “Work on proof ...”. For each item there is an overview containing the development graph, cf. Figure D.2, a detailed listing of the contents of nodes, cf. Figure D.3, in turn consisting of the inherited and locally defined signature symbols and axioms, and finally the proofs (D.4).

D.2 Developments

An excerpt from the proof obligation that we concentrate on in Chapter 8 initially looks as in Figure D.5 (the formula below the line is the succedent,

Appendix D. Details of the Case Study

```
(mk-process (some m) (ib (aref (procs s) i))
  (db (aref (procs s) i))
  (delivered (aref (procs s) i))))
(= (broadcast s1) (adjoin m (broadcast s))))))

4. Work on proof for (all (tr trace) (s state)
  (=> (trans* initial-state tr s) (safe s))) (link (system
  satisfies
  (reliability))).

5. (dt-add-node 'boolean)

6. (dt-add-sort 'boolean 'boolean)

7. (dt-add-op 'boolean 'tt 'nil 'boolean)

8. (dt-add-op 'boolean 'ff 'nil 'boolean)

9. (dt-add-gen 'boolean '(tt ff))

10. (dt-add-axiom 'boolean '(not (= tt ff)))

11. (dt-add-link 'boolean 'process)

12. (dt-add-op 'process 'is-up '(process) 'boolean)

13. (dt-add-argument 'process 'mk-process 'boolean 'tt)

14. (dt-add-axiom 'process
  '(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
  (up1 boolean) (= up1 (is-up (mk-process bb1 ib1 db1 dm1 up1))))

15. (dt-wrap-binding 'process
  '(all (bb1 option) (ib1 option) (db1 option) (dm1 bag)
  ($ (= bb1 (bb (mk-process bb1 ib1 db1 dm1 tt))))
  'all
  'up1
  'boolean)
```

Figure D.1: History of example

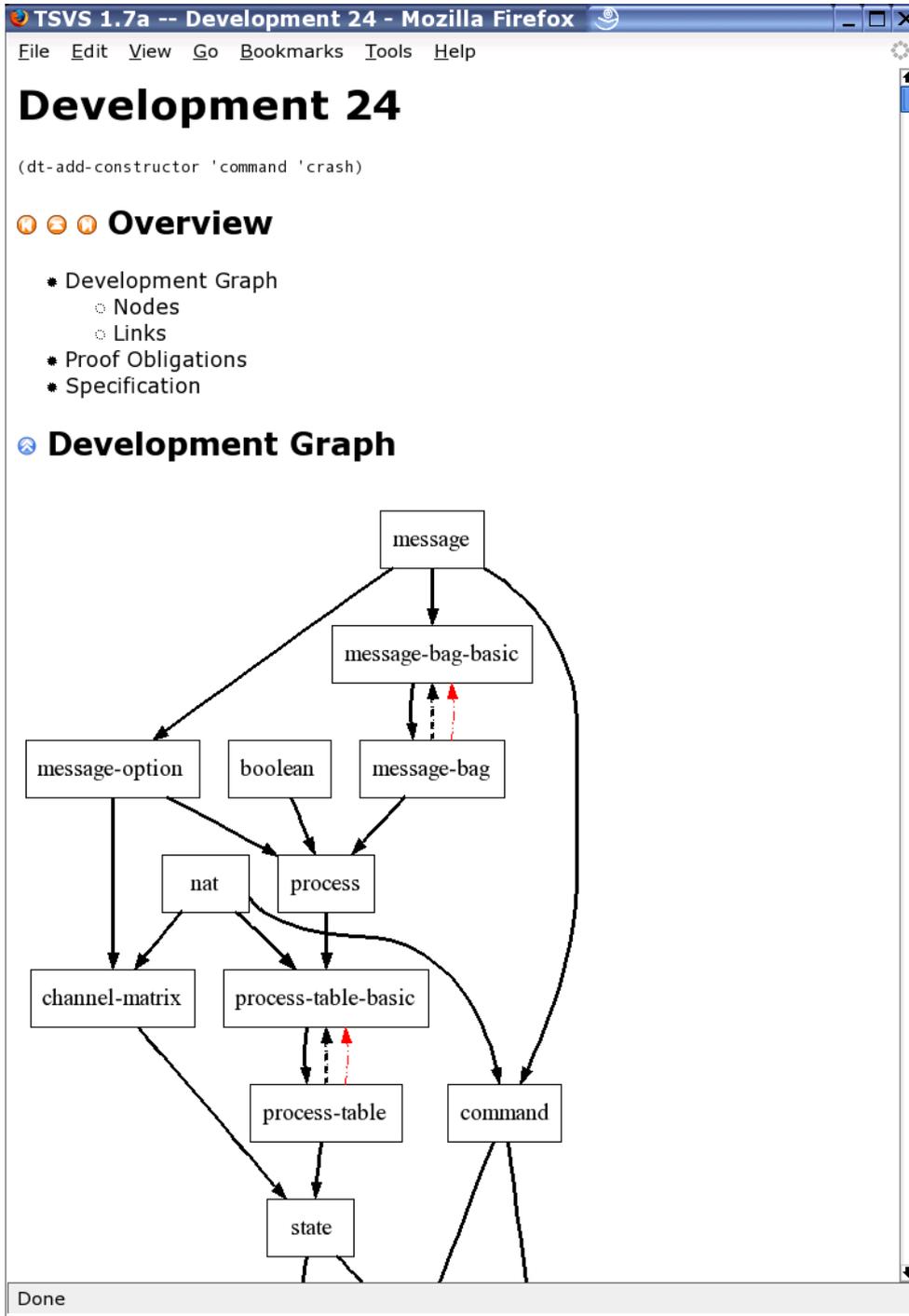


Figure D.2: Development graph

Appendix D. Details of the Case Study



Figure D.3: Contents of nodes

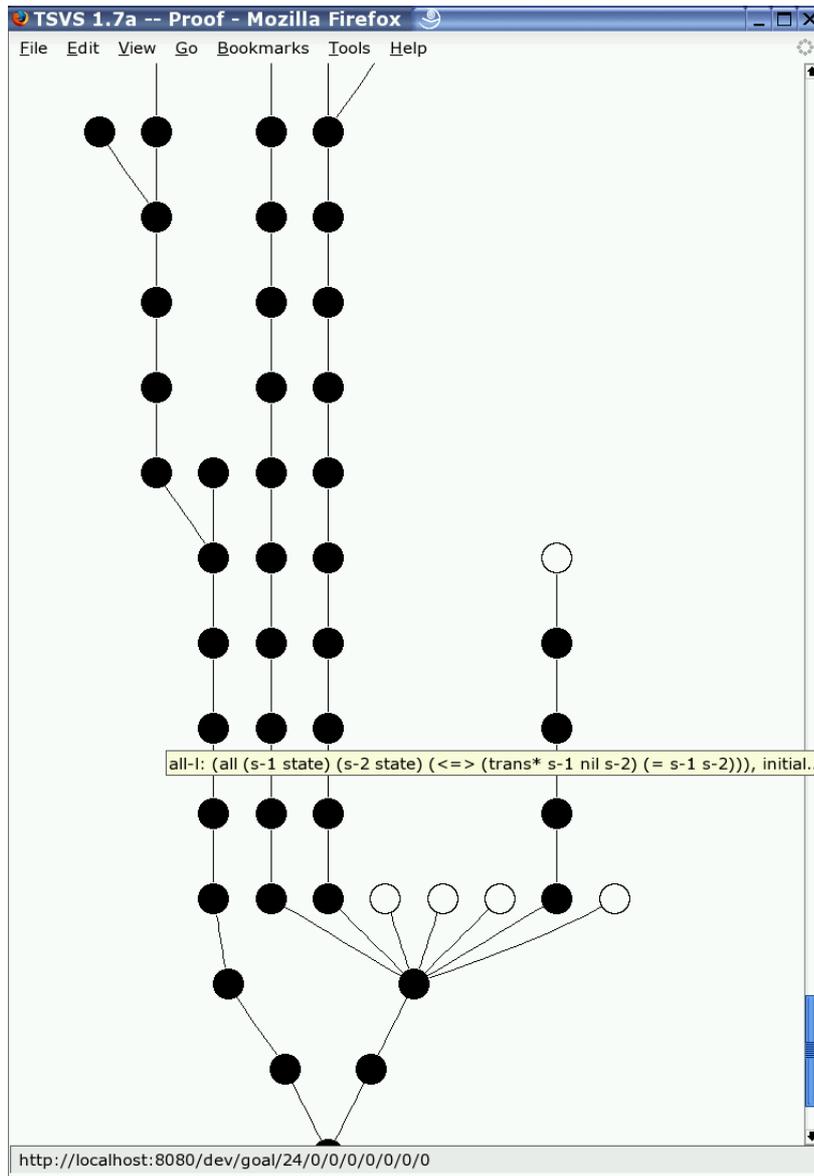
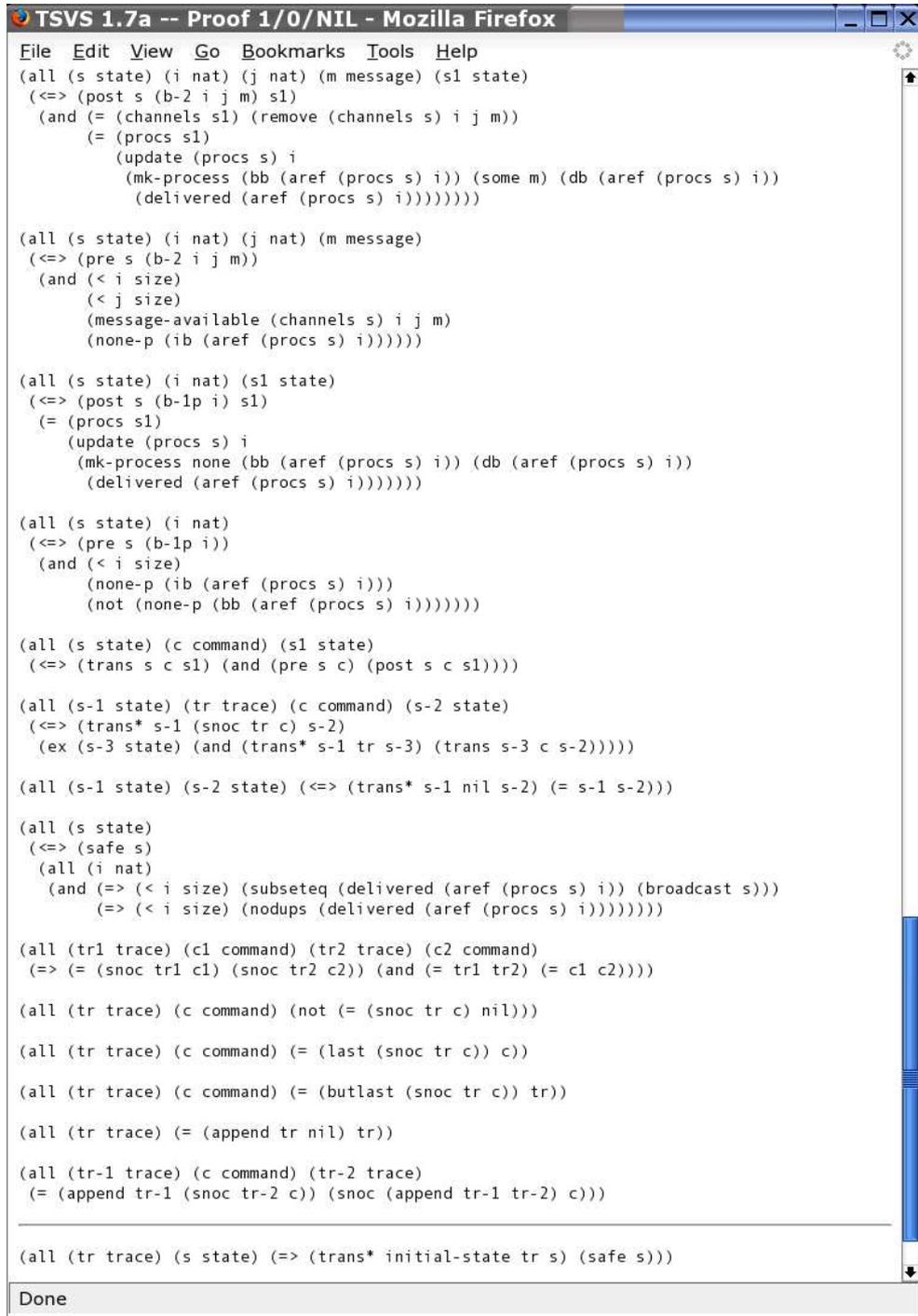


Figure D.4: Proof tree

Appendix D. Details of the Case Study



```
File Edit View Go Bookmarks Tools Help
(all (s state) (i nat) (j nat) (m message) (s1 state)
 (<=> (post s (b-2 i j m) s1)
  (and (= (channels s1) (remove (channels s) i j m))
    (= (procs s1)
      (update (procs s) i
        (mk-process (bb (aref (procs s) i)) (some m) (db (aref (procs s) i))
          (delivered (aref (procs s) i))))))))))

(all (s state) (i nat) (j nat) (m message)
 (<=> (pre s (b-2 i j m))
  (and (< i size)
    (< j size)
    (message-available (channels s) i j m)
    (none-p (ib (aref (procs s) i))))))

(all (s state) (i nat) (s1 state)
 (<=> (post s (b-1p i) s1)
  (= (procs s1)
    (update (procs s) i
      (mk-process none (bb (aref (procs s) i)) (db (aref (procs s) i))
        (delivered (aref (procs s) i)))))))

(all (s state) (i nat)
 (<=> (pre s (b-1p i))
  (and (< i size)
    (none-p (ib (aref (procs s) i)))
    (not (none-p (bb (aref (procs s) i)))))))

(all (s state) (c command) (s1 state)
 (<=> (trans s c s1) (and (pre s c) (post s c s1))))

(all (s-1 state) (tr trace) (c command) (s-2 state)
 (<=> (trans* s-1 (snoc tr c) s-2)
  (ex (s-3 state) (and (trans* s-1 tr s-3) (trans s-3 c s-2)))))

(all (s-1 state) (s-2 state) (<=> (trans* s-1 nil s-2) (= s-1 s-2)))

(all (s state)
 (<=> (safe s)
  (all (i nat)
    (and (=> (< i size) (subsetq (delivered (aref (procs s) i)) (broadcast s)))
      (=> (< i size) (nodups (delivered (aref (procs s) i))))))))

(all (tr1 trace) (c1 command) (tr2 trace) (c2 command)
 (=> (= (snoc tr1 c1) (snoc tr2 c2)) (and (= tr1 tr2) (= c1 c2))))

(all (tr trace) (c command) (not (= (snoc tr c) nil)))

(all (tr trace) (c command) (= (last (snoc tr c)) c))

(all (tr trace) (c command) (= (butlast (snoc tr c)) tr))

(all (tr trace) (= (append tr nil) tr))

(all (tr-1 trace) (c command) (tr-2 trace)
 (= (append tr-1 (snoc tr-2 c)) (snoc (append tr-1 tr-2) c)))

(all (tr trace) (s state) (=> (trans* initial-state tr s) (safe s)))

Done
```

Figure D.5: Proof obligation

and the others above are some assumptions available in the antecedent). For reference purposes the whole original specification text is given below on page .

D.3 Transformations

Part of the first marked open goal from Section 8.3 that is missing some axioms is depicted in Figure D.6. Going two steps into the future in another window yields Figure D.7, where the front window contains the two added axioms that are missing from the older goal in the upper-left window.

Finally, Figure D.8 presents the goal that leads to the open goal when strengthening the precondition of the overall conjecture in Section 8.7(c). While the older goal proves ($< i \text{ size}$) in one step, after the transformation the formula from the induction hypothesis is strengthened, and the proof includes some additional steps and an additional open goal that postulates ($= \text{tt} (\text{is-up} (\text{aref} (\text{procs } s-3) i))$).

D.4 Original Specification Text

```
;;; -----
((theory nat)
 (sort nat)
 (op 0 () nat)
 (op succ (nat) nat)
 (gen 0 succ)
 (prd <= (nat nat))
 (prd < (nat nat)))

;;; -----
((theory message)
 (sort message))

;;; -----
((theory message-option uses (message))
 (sort option)
 (op none () option)
 (op some (message) option)
 (axiom (all (m1 message) (m2 message)
             (=> (= (some m1) (some m2))
                 (= m1 m2)))))
```

Appendix D. Details of the Case Study

The screenshot displays the TSVS 1.7a proof environment. On the left, a sidebar shows the current goal: `(post s-3 (b-1 nat message) s)`. Below this, a list of variables is provided: `i: nat`, `s-3: state`, `s: state`, `message: message`, `nat: nat`, and `trace: trace`. The main area contains a state transition diagram with nodes represented by circles (white for initial state, black for subsequent states) and edges representing transitions. A blue arrow points from the goal text in the sidebar to the top-left node of the diagram.

```

(post s-3 (b-1 nat message) s)
(pre s-3 (b-1 nat message))

(< i size)
(=> (< i size) (nodups (delivered (aref (procs s-3) i))))
(=> (< i size) (subseteq (delivered (aref (procs s-3) i)) (broadcast s-3)))

(all (i nat)
  (=> (< i size)
    (= (aref (procs initial-state) i) (mk-process none none none empty))))

(= (channels initial-state) empty-matrix)

(all (s-1 bag) (s-2 bag) (m message)
  (=> (subseteq s-1 s-2) (subseteq s-1 (adjoin m s-2))))

(all (s bag) (subsetq empty s))
  
```

Figure D.6: Goal with missing axioms

D.4. Original Specification Text

```
(prd none-p (option))
(axiom (none-p none))
(axiom (all (m message) (not (none-p (some m))))))
(gen none some)
(axiom (all (m message)
            (not (= none (some m))))))
(op get (option) message)
(axiom (all (m message)
            (= (get (some m)) m))))

;;; -----
((theory message-bag-basic uses (message)
   satisfies (message-bag))
 (sort bag)
 (op empty () bag)
 (op adjoin (message bag) bag)
 (axiom (all (m1 message) (b1 bag) (m2 message) (b2 bag)
            (=> (= (adjoin m1 b1) (adjoin m2 b2))
                (and (= m1 m2) (= b1 b2))))))
 (gen empty adjoin)
 (prd member (message bag))
 (axiom (all (m message) (not (member m empty))))
 (axiom (all (m1 message) (m2 message) (s bag)
            (<=> (member m1 (adjoin m2 s))
                (or (= m1 m2) (member m1 s))))))
 (prd subseteq (bag bag))
 (axiom (all (s1 bag) (s2 bag)
            (<=> (subseteq s1 s2)
                (all (m message)
                    (=> (member m s1)
                        (member m s2))))))
 (prd nodups (bag))
 (axiom (nodups empty))
 (axiom (all (m message) (s bag)
            (<=> (nodups (adjoin m s))
                (and (not (member m s))
                    (nodups s))))))

;;; -----
((theory message-bag uses (message-bag-basic))
 (axiom (all (m message)
            (all (s bag)
                (not (= empty (adjoin m s))))))
 (axiom (all (s-1 bag) (s-2 bag) (m message)
            (=> (subseteq s-1 s-2)
                (subseteq s-1 (adjoin m s-2))))))
 (axiom (all (s bag) (subseteq empty s))))
```

Appendix D. Details of the Case Study

```
((theory command uses (nat message))
 (sort command)
 (op b-1 (nat message) command)
 (op b-1p (nat) command)
 (op b-2 (nat nat message) command)
 (op b-3 (nat) command)
 (op b-4 (nat) command)
 (op b-5 (nat) command)
 (gen b-1 b-1p b-2 b-3 b-4 b-5))

;;; -----
((theory trace uses (command))
 (sort trace)
 (op nil () trace)
 (op snoc (trace command) trace)
 (axiom (all (tr1 trace) (c1 command)
             (tr2 trace) (c2 command)
             (=> (= (snoc tr1 c1)
                   (snoc tr2 c2))
                 (and (= tr1 tr2) (= c1 c2))))))
 (gen nil snoc)
 (axiom (all (tr trace) (c command)
             (not (= (snoc tr c) nil))))
 (op last (trace) command)
 (axiom (all (tr trace) (c command)
             (= (last (snoc tr c)) c)))
 (op butlast (trace) trace)
 (axiom (all (tr trace) (c command)
             (= (butlast (snoc tr c)) tr)))
 (op append (trace trace) trace)
 (axiom (all (tr trace)
             (= (append tr nil) tr)))
 (axiom (all (tr-1 trace) (c command) (tr-2 trace)
             (= (append tr-1 (snoc tr-2 c))
               (snoc (append tr-1 tr-2) c))))))

;;; -----
((theory process uses (message-option message-bag))
 (sort process)
 (op mk-process (option option option bag) process)
 (gen mk-process)
 (axiom (all (bb1 option) (ib1 option) (db1 option)
             (dm1 bag)
             (bb2 option) (ib2 option) (db2 option)
             (dm2 bag)
             (<=> (= (mk-process bb1 ib1 db1 dm1))
```

D.4. Original Specification Text

```

      (mk-process bb2 ib2 db2 dm2))
      (and (= bb1 bb2)
            (= ib1 ib2)
            (= db1 db2)
            (= dm1 dm2))))))
(op bb (process) option)
(axiom (all (bb1 option) (ib1 option) (db1 option)
            (dm1 bag)
            (= bb1 (bb (mk-process bb1 ib1 db1 dm1))))))
(op ib (process) option)
(axiom (all (bb1 option) (ib1 option) (db1 option)
            (dm1 bag)
            (= ib1 (ib (mk-process bb1 ib1 db1 dm1))))))
(op db (process) option)
(axiom (all (bb1 option) (ib1 option) (db1 option)
            (dm1 bag)
            (= db1 (db (mk-process bb1 ib1 db1 dm1))))))
(op delivered (process) bag)
(axiom (all (bb1 option) (ib1 option) (db1 option)
            (dm1 bag)
            (= dm1
              (delivered (mk-process bb1 ib1 db1 dm1))))))

;;; -----
((theory process-table-basic
  uses (process nat)
  satisfies (process-table))
 (sort table)
 (op empty-table () table)
 (op update (table nat process) table)
 (op size () nat)
 (op aref (table nat) process)
 (gen empty-table update)
 (axiom (all (i nat) (j nat) (t table) (p process)
             (=> (= i j) (= (aref (update t i p) j) p))))
 (axiom (all (i nat) (j nat) (t table) (p process)
             (=> (not (= i j))
                 (= (aref (update t i p) j)
                    (aref t j))))))

;;; -----
((theory process-table uses (process-table-basic))
 (axiom (all (t table) (n nat) (m nat) (a option)
            (b option) (c option)
            (= (delivered
                (aref
                 (update

```

Appendix D. Details of the Case Study

```

      t
      m
      (mk-process
        a b c (delivered (aref t m))))
      n))
      (delivered (aref t n))))))

;;; -----
((theory channel-matrix
  uses (nat message-option))
 (sort channel-matrix)
 (prd message-available (channel-matrix nat nat message))
 (op empty-matrix () channel-matrix)
 (op add (channel-matrix nat nat message) channel-matrix)
 (op remove (channel-matrix nat nat message) channel-matrix)

 (axiom (all (i nat) (j nat) (m message)
             (not (message-available empty-matrix i j m))))
 (axiom (all (c channel-matrix) (i nat) (j nat) (m message)
             (k nat) (l nat) (n message)
             (<=> (message-available (add c i j m)
                                     k l n)
                  (or (and (= i k) (= j l) (= m n))
                      (message-available c k l n)))))

 (axiom (all (c channel-matrix) (i nat) (j nat) (m message)
             (k nat) (l nat) (n message)
             (<=> (message-available (remove c i j m)
                                     k l n)
                  (and (not (and (= i k) (= j l) (= m n)))
                      (message-available c k l n)))))

;;; -----
((theory state uses (process-table channel-matrix))
 (sort state)
 (op mk-state (table bag channel-matrix) state)
 (axiom (all (t1 table) (b1 bag) (m1 channel-matrix)
             (t2 table) (b2 bag) (m2 channel-matrix)
             (<=> (= (mk-state t1 b1 m1)
                    (mk-state t2 b2 m2))
                  (and (= t1 t2)
                      (= b1 b2)
                      (= m1 m2)))))

 (op procs (state) table)
 (axiom (all (t table) (b bag) (m channel-matrix)
             (= (procs (mk-state t b m)) t)))

 (op broadcast (state) bag)
```

D.4. Original Specification Text

```
(axiom (all (t table) (b bag) (m channel-matrix)
           (= (broadcast (mk-state t b m)) b)))
(op channels (state) channel-matrix)
(axiom (all (t table) (b bag) (m channel-matrix)
           (= (channels (mk-state t b m)) m))))

;;; -----
((theory pre-post uses (state command))
 (prd pre (state command))
 (prd post (state command state))

;; b-1
(axiom (all (s state) (i nat) (m message)
           (<=> (pre s (b-1 i m))
                (and (< i size)
                     (not (member m (broadcast s)))
                     (none-p (bb (aref (procs s) i)))))))

(axiom (all (s state) (i nat) (m message) (s1 state)
           (<=> (post s (b-1 i m) s1)
                (and
                 (= (procs s1)
                    (update
                     (procs s)
                     i
                     (mk-process
                      (some m)
                      (ib (aref (procs s) i))
                      (db (aref (procs s) i))
                      (delivered (aref (procs s)
                                      i))))))
                 (= (broadcast s1)
                    (adjoin m (broadcast s)))))))

;; b-1p
(axiom (all (s state) (i nat)
           (<=> (pre s (b-1p i))
                (and (< i size)
                     (none-p (ib (aref (procs s) i)))
                     (not (none-p (bb (aref (procs s)
                                           i)))))))

(axiom (all (s state) (i nat) (s1 state)
           (<=> (post s (b-1p i) s1)
                (= (procs s1)
                   (update
                    (procs s) i
```

Appendix D. Details of the Case Study

```
(mk-process (none)
            (bb (aref (procs s) i))
            (db (aref (procs s) i))
            (delivered (aref (procs s)
                              i))))))

;; b-2

(axiom (all (s state) (i nat) (j nat) (m message)
           (<=> (pre s (b-2 i j m))
               (and (< i size)
                    (< j size)
                    (message-available
                     (channels s) i j m)
                    (none-p (ib (aref (procs s) i)))))))

(axiom (all (s state) (i nat) (j nat) (m message)
           (s1 state)
           (<=> (post s (b-2 i j m) s1)
               (and (= (channels s1)
                       (remove (channels s) i j m))
                    (= (procs s1)
                       (update
                        (procs s) i
                        (mk-process
                         (bb (aref (procs s) i))
                         (some m)
                         (db (aref (procs s) i))
                         (delivered (aref (procs s)
                                           i))))))))))

;; b-3, b-4 b-5 not specified yet

)

;;; -----
((theory transition uses (pre-post))
 (prd trans (state command state))
 (axiom (all (s state) (c command) (s1 state)
            (<=> (trans s c s1)
                (and (pre s c)
                     (post s c s1))))))

;;; -----
((theory execution uses (transition trace))
 (prd trans* (state trace state))
 (axiom (all (s-1 state) (s-2 state)
```

D.4. Original Specification Text

```
(=<=> (trans* s-1 nil s-2) (= s-1 s-2))))
(axiom (all (s-1 state) (tr trace) (c command) (s-2 state)
  (<=> (trans* s-1 (snoc tr c) s-2)
    (ex (s-3 state)
      (and (trans* s-1 tr s-3)
        (trans s-3 c s-2)))))))

;;; -----
((theory secprop uses (state))
  (prd safe (state))
  (axiom (all (s state)
    (<=> (safe s)
      (all (i nat)
        (and
          (=> (< i size)
            (subsetq
              (delivered (aref (procs s) i))
              (broadcast s)))
          (=> (< i size)
            (nodups
              (delivered
                (aref (procs s) i)))))))))))

;;; -----
((theory system uses (trace execution secprop)
  satisfies (reliability))
  (op initial-state () state)
  (prd invariant (state))
  (axiom (all (i nat)
    (=> (< i size)
      (= (aref (procs initial-state) i)
        (mk-process none none none empty))))))
  (axiom (= (broadcast initial-state) empty))
  (axiom (= (channels initial-state) (empty-matrix))))

;;; -----
((theory reliability uses (system secprop))
  (axiom (all (tr trace) (s state)
    (=> (trans* initial-state tr s) (safe s))))
  (axiom (all (s1 state) (s2 state)
    (=> (and (ex (tr trace)
      (trans* initial-state tr s1))
      (invariant s1)
      (ex (c command) (trans s1 c s2))
      (invariant s2))))))
```