Viewpoints In Practice

Explanations Explained

submitted by

Steve Riddle

for the degree of Ph.D.

of the

University of Bath

1997

Summary

Viewpoints, which we define as partial specifications in an appropriate language (not necessarily formal), are used in a number of guises in various facets of software engineering, though their most common use is in requirements elicitation and analysis.

The aim of this work is, first, to demonstrate that viewpoints are used as we claim they are, and can encapsulate information being added in an explanation. Having done this we use some results from the formal theory of refinement to model the way systems are described and explained, not only in the field of requirements but also in disparate areas such as incremental description and tutorial developments given in user manuals. This achieves the major claim of the thesis, which is that explanations can be explained using the concept of viewpoints and the theory of refinement.

To achieve this claim, we begin with a survey of the use of viewpoints in software engineering. This includes summaries of research in a number of fields, not limited to requirements engineering research, and some analysis of their ability to model multiple notations and conflicts. The survey is followed by a review of the theoretical background to refinement and its applicability to the process of amalgamation of viewpoints, which provides a set of criteria for coming up with an appropriate refinement relation, and operations for amalgamation, to model and explain explanations. Having introduced the approach we assess it, first by comparing it with a framework for viewpoint development based on seemingly orthogonal ideas, and then using a number of examples of explanations in which viewpoints can be used to explain the development steps. These examples include an incremental specification of an operation to select the next appropriate element from a queue, and some applications related to denotational semantics.

We conclude that viewpoints can indeed encapsulate such information and that, given a notion of refinement appropriate to the specification language under consideration, the relation between viewpoints can be described in a natural way corresponding to the way explanations are given in the real world.

Acknowledgements

The work described in this thesis was carried out with the support of a grant from the EPSRC. I am grateful also for the supervision and support of Dr Peter J.L.Wallis, advice and feedback from Dr Lindsay Groves, helpful conversations with Dr Mike Ainsworth and Dr Dan Richardson, coffee provided by Nic Doye and Jet Kang, and the love, support, tolerance and helpful suggestions of my wife Sandra.

I am grateful to the Centre for Software Reliability at the Department of Computing Science, University of Newcastle upon Tyne, for time, discussions and use of facilities to complete the thesis.

I am also grateful for the comments of anonymous reviewers of earlier versions of Chapters 2 and 7.

Contents

Chapter 1

Introduction

This thesis concerns information, in the sense not of facts being endlessly produced by television and radio, but of *explanations* — how people explain things to others. The "things" being explained might be pieces of software, larger systems or simply abstract concepts. The perspective of the explanation might be that of a user, a programmer, a designer or a specifier. In any of these cases, what is being explained is a *viewpoint*.

The aims of this thesis are to:

1. Demonstrate that viewpoints are used in everyday explanations, or can be used to advantage in modelling explanations *(Viewpoints in Practice)*

   This is achieved by a literature survey of techniques which use viewpoints (Chapter 2), and by further chapters which model explanations in terms of viewpoints.

2. Show how formal refinement theory can be used to relate these viewpoints together *(Explanations explained)*, and to provide straightforward mathematical properties which must be satisfied for such an explanation to be given a formal approval.

   This is achieved by examining refinement theory (Chapter 3) and using it to develop a theory of refinement between viewpoints. This theory provides the necessary guidance for reasoning about the validity of derived or amalgamated viewpoints.

1.1 What is a Viewpoint

We begin by defining what we mean by a *viewpoint*, a term originally coined by Mullery for the requirements tool CORE [Mul79, Som96] and taken up by a number of researchers for their own models. The survey in Chapter 2 goes into these modelling techniques in more detail, and shows that they each have a different concept of *viewpoint*. To attempt to encompass all these concepts is impossible, but the following definition is general while still being useful:

Definition 1.1 (Viewpoint) *A* viewpoint *is a partial specification, written in a* language *which is not necessarily formal. It may be composed of/with other viewpoints, in a manner dictated either by the language or by an associated development method.*

The absence of a restriction to formal languages allows the following to be a viewpoint:

$\boxed{\text{The clock has a digital display}}$

It is certainly partial in that it does not give a complete specification of the clock — it even leaves unanswered the question "what clock?" which springs to mind. It could however be composed with a more formal viewpoint to provide a more complete specification, in both natural language and mathematical terms, of what it means to have a digital display. This new viewpoint, formed by *amalgamation*, could then be further composed with a viewpoint describing the clock mechanism, and so on until the whole clock is specified. What we have just described is an application of viewpoints to model a bottom-up description.

While we do not insist on a *formal* language, that is to say a language with a defined semantics in which it is possible to reason about the properties of the viewpoint being described, we would at the very least expect that it is possible to decide, of a given sequence of symbols made up of the alphabet of that language, whether or not the sequence constitutes a valid *sentence* in that language — in other words that the language is *decidable*. A viewpoint should consist of *well-formed* sentences from a given language.

1.2 Viewpoints in Practice

We are interested in how effective viewpoints are in describing the way in which people explain things to each other — in particular, but not exclusively, the description of aspects of software systems. We argue that viewpoints are used in practice more widely than has been recognised. For example, composition of Z specifications can be seen as an amalgamation of viewpoints, and incremental development of specifications can be expressed as an amalgamation of new developments with the original idea. Beyond software specifications, an document presenting an explanation of a complex subject using an initial "glossing" description, with subsequent explanations that augment or override information, can be modelled in terms of viewpoints.

1.3 Explanations Explained

The major claim of the thesis is that the use of viewpoints in such incremental developments (explanations) can be modelled and formally verified using the theory of *refinement*, with extensions where necessary. In this way an incremental explanation can be shown to be made up of ordered steps which each refine the previous one. In a real-life example such a process is likely to involve some *back-tracking* so we also address this question with respect to refinement theory to identify when, if ever, such a development can be said to be formally acceptable.

An illustration of a common form of explanation comes in the following outline of chapters, which moves from initial ideas about viewpoints and refinement, through the definitions and theory which are needed, to examples of increasing weight of explanations which can be explained.

1.4 Chapter Outline

1.4.1 A Survey of Viewpoint Techniques

We begin with a survey of a number of approaches to requirements analysis, design, specification development and programming in which viewpoints are used, though not always explicitly. We consider the degree to which each of these approaches exploits viewpoints and the relations between viewpoints.

1.4.2 Amalgamation and Refinement

We introduce key concepts in the formal treatment of combining (amalgamating) viewpoints and of refinement, identify characteristics which must hold for any relation which is required for refinement purposes. We identify a series of refinement orderings which can be of use in different situations, depending on the definition of what makes a correct implementation of a specification.

1.4.3 The Refinement Calculus and Co-Refinement

We discuss a particular refinement approach in the light of the theory introduced previously, and show that the refinement relation used in this calculus can be too strong to allow some sorts of development; these are discussed together with a weaker version, *co-refinement*. Following the ideas of co-refinement to a logical conclusion an even weaker version, *compromise*, is introduced.

1.4.4 Viewpoint-Oriented Software Development

First encountered in the survey chapter, the VOSD framework for integration of methods and tools has some distinct differences in outlook from our approach, notably the highly structured designation of what constitutes a viewpoint and the distributed nature of the framework; viewpoints here are loosely coupled, locally managed objects responsible for directing their own development. We compare the approaches and consider how amenable our more general ideas of refinement and amalgamation are to modelling the VOSD framework.

1.4.5 Viewpoints and Explanations

An explanatory walk through a number of examples in the real world where viewpoints can be used to advantage. The chapter is arranged in an incremental manner, building from tutorial descriptions and user manuals to the writing of documentation with code (literate programming), version control systems and computer support for cooperative working (CSCW), and illustrates the claim of the thesis: that explanations such as these can be modelled, explained and assessed using viewpoints and refinement.

Each of these chapters has provided some ground work for the two larger examples which follow, in which full-sized explanations are explained in terms of refinement.

1.4.6 Incremental Development of an Algebraic Specification

The first case study concerns the incremental development of specifications. In considering an algebraic, rather than state-based specification language, we develop a new version of the co-refinement relation introduced in Chapter 3. This example is adapted from an incremental specification of part of the scheduling system for the Apple Macintosh Toolbox Event Manager [BCG+89].

1.4.7 Denotational Semantics

The semantics of a programming language gives a further example. Denotational semantics is used as a tool for the design and implementation of programming languages. The language concerned is a simple calculator, and we show how a new feature can be added to the calculator. This is achieved by first identifying the ways in which such semantic descriptions can be composed and the appropriate version of a refinement relation to compare the descriptions. As a conclusion we look at how the step from direct to continuation semantics can be modelled in the same way.

1.4.8 Summary and Conclusions

Finally we review where the previous chapters have got us, how the aims set out in this chapter been met, and where we might go from here.

1.5 A Note about Pronouns

Much time and effort has been extended by various worthy writers to avoid apparent sexism in writing; sentences such as "if a writer wants to avoid appearing chauvinistic, he/she should look to his/her pronouns" are laudably gender-free, but clumsy and annoying to read. Making a thesis annoying to read is very probably a Bad Thing.

Various alternatives exist: "it" is pleasingly neutral but best applied only to inanimate objects; "she" could be used always in an effort to restore balance, but smacks of positive discrimination; "one" just sounds silly. I toyed with using "they", but it seems a shame to sacrifice grammar to political correctness.

The solution adopted here is to use a new word, "he", with "his" for the possessive case, to represent both masculine and feminine pronouns. This amalgamation of the old terms should not be confused with the gender-specific "he" and "his" which are now superseded. The author will not be held responsible for any conjectured sexism resulting from such confusion.

Chapter 2

A Survey of Viewpoint Techniques

In this chapter we survey a number of current viewpoint techniques in software engineering, which are applied not only to requirements definition but to the whole software specification process. We take a selection of the most promising work and compare the approaches.

We begin by introducing reasons for the use of viewpoints in software engineering.

2.1 Motivation

The process of producing requirements documents and formal specifications for any medium-to-large system is a complex task: before very much development has taken place the document will have become difficult to maintain. This is due to a number of factors:

- The users of the system may specify contradictory or inconsistent requirements, and are likely to want to change them during the process of development.
- There can be no single "correct" way to structure a formal specification from the elicited requirements; the decomposition of each level will depend on the interpretation given to the information by the analyst concerned.
- There may be a number of analysts producing requirements documents, perhaps in different formalisms, with their own assumptions about the problem and with

different perspectives on it. Communication between these analysts may be less than ideal.

- It is not easy to take in several pages of mathematically correct specification, however much it is accompanied by natural language explanation. In addition, there is a diverse range of specification languages and paradigms which are best suited to different stages of the development process, and to expressing different types of problem.

In fact, even with a carefully co-ordinated software engineering team, in which understandings shared between members of the team are maintained through documentation, misunderstandings and breakdowns of communication can still occur [Eas92].

Breaking the problem up into bite-sized pieces of information, which can be understood, verified and altered simply, is a common technique. Specification languages such as Z [Spi89], with its schema calculus, and CLEAR [BG86], with its parameterised theories, help in expressing a problem in modules and encourage information hiding [Par72b, Par72a]. A key result of studies of modularity is that what makes a good modularisation depends on the point of view of the user of the modularisation; ease of understanding does not necessarily correlate with ease of implementation, and readability may be orthogonal to efficiency of code [FJ90]. The object-oriented methodology OMT [RBP+91] uses object, dynamic and functional models of a system and encourages the use of modules to capture different perspectives of a situation. Tools such as RAISE [Gro92] and the B toolkit [ALN+91] provide support for incremental development of specifications, but it is not a simple task to manage inter-relationships and communications between developing modules.

Central to several of these methodologies is the concept of evolution [Fea89a, Gol83], also termed incremental specification or elaboration. This is a development method in which, starting from a basic initial description, changes are made to the developing specification until a final description is reached: each change is simple enough to be readily understood. In particular, Feather [Fea89a] considers "parallel elaborations" which can be refining or adapting elaborations, to be applied independently and in parallel, to the specification. This is followed by a "recombination" stage in which any glossed-over dependencies between elaborations, and any other causes of interference, can be considered. Several of the methods described in this chapter develop this idea further.

2.2 Viewpoints

The notion of *viewpoints* has been developed by several authors, working generally in the field of requirements engineering but also in the process of formal software specification. The term is derived, as are several of the methods we shall describe, from the requirements definition language CORE [Mul79, Som96] (Section 2.3.1). While this method has achieved popularity in industrial use, it has inherent weaknesses which more recent methods address - see Section 2.3.1

The definition of a viewpoint is subtly different in each approach: for some it is still an external entity interacting with, and partitioning, the system [Mul79, KS92]; for others it is a partial, self-contained specification [Wal92, FKN$^+$92]. In general, however, the idea is that a viewpoint represents partial knowledge of the system: it may be incomplete or inconsistent with other viewpoints during a system's development. Some of the approaches described do not use the term [RW91, ZJ93], but are nonetheless concerned with the development of partial specifications.

A common example is the library problem, [Win88] which considers the differing positions of library users and library staff: users would like to have a wide choice of books, long borrowing times and high limits on the number of books they can take out, while staff will be more concerned with security and stock control. Both parties will wish for a fast check-in/out system. On a different level, the concepts they are considering will differ: a single concept will have different names, e.g. borrowing for a user is lending from the point of view of the staff, or other concepts with the same name will be different, e.g. the concept of a "book" is simply a title and author to the user, while it is also edition, copy number or bar-code for the staff.

2.2.1 Issues

In comparing viewpoint approaches, the following issues need to be considered:

- What is the subject of interest, and where in the development process do viewpoints come in?

  Some methodologies simply use viewpoints to validate the consistency of requirements elicited from users [LF91, RW91]; alternative approaches formalise requirements, or retain the partitioning right through the process of requirements and specification [FKN$^+$92, JFH92].

- How general is the method, and are multiple formalisms supported?

The methods cited range from those imposing a viewpoint definition language on the user [LF91] to those allowing the analyst to work in a number of representation styles and design methodologies as best fits the problem [FKN+92, JFH92]. The advantage of multiple formalism support is that it provides a more realistic model of actual development, in which a system's description will evolve across a number of representation styles of varying formality.

- How are viewpoints formally defined?

  As mentioned above, viewpoints are not defined consistently by all approaches. Some methods have a very loose definition, others have a formal one, and some do not explicitly mention viewpoints at all. We are concerned with the way in which viewpoints are structured and related to each other and what they represent: for example, an agent (stake-holder or person with an interest in the system), a requirement, or a partitioning of the problem.

- How are viewpoints used to specify a system?

  Once a viewpoints structure has been built for a system, and relationships defined, there remains the question of how they are used to provide a specification for the system. Most methods involve integrating or *amalgamating* the viewpoints at some stage, although [FKN+92] proposes a system in which no single, flat specification document need be produced. Inconsistencies between viewpoints must be identified and resolved before integration is possible; not all methods consider the identification of inconsistencies in detail.

Approaches to requirements engineering can be broadly divided into the *technical*, in which a rigorous approach is used to formalise the elicited requirements and develop a specification in a formal language; the *cognitive*, in which knowledge representation techniques are used to model elicited requirements and resolve them by means of heuristics; and the *social*, where the emphasis is on extensive fieldwork to obtain a rich picture of the intended system and its context. The majority of viewpoint approaches in this survey are aids to requirements engineering, and fall into the first two categories: other work which touches on some of the approaches included here are cognitive approaches such as planning [AF89], specification critiquing [FN88] and blackboard systems [LLC91], and technical approaches such as views in object-oriented programming [SS89].

Our comparison of viewpoint approaches is in part motivated by a survey of methodologies for integration of database schemas [BLN86], which considers twelve schema integration methodologies in depth. Schema integration normally takes place in the conceptual design stage, after separate views have been produced in the requirements stage. This results in a global, high-level schema. Reasons for differences between views are equally applicable to partial specifications of program systems — different perspectives, equivalent constructs modelled differently, incompatible design specifications, common concepts or related concepts. Representations of common concepts may be identical, equivalent, compatible (not equivalent but not contradictory) or incompatible: the last two are considered to be conflicts.

Each methodology in the survey can be considered to use some of the following common steps:

- Pre-Integration

  Schemas are identified, relevant information gathered

- Comparison

  Correspondence, conflicts and inter-schema properties are determined. Conflicts are classified as structural conflicts or naming conflicts.

- Conflict resolution

  By reference to designers/users — usually a manual process.

- Merging and Restructuring

  Producing a global schema which is complete, correct, minimal (no redundancies) and understandable.

We will make reference to these steps in the following comparisons, and, since the viewpoint approaches considered are more than integration methodologies, we will also consider the structure of viewpoints and how they are used to develop a specification.

2.2.2 Software Specification Process

The first stage in classifying viewpoint approaches is to consider the stages in the software specification process to which each approach is addressed. Taking our lead from [Som96], we consider the specification process to consist of the following steps;

- Requirements Elicitation

  The system's requirements are defined by negotiation with the potential users, buyers, specifiers and designers. This stage should include a feasibility study.

- System Modelling

  The next three stages provide input for the requirements document, which will form the basis of a binding contract between the buyers and the developer. The System Model details relationships between system components and the environment.

- Requirements Definition

  This stage produces an abstract description of the proposed system, with natural language detailing functional and non-functional requirements (Non-functional requirements are constraints, such as timing and reliability).

- Requirements Specification

  Also termed functional specification, this stage uses techniques such as program description languages to specify each component of the system. Requirements validation will normally be a sub-process of this stage. The specification document is often produced in conjunction with the design, and will form an appendix to the requirements document.

- System Design

  A design is produced, using a variety of available formalisms.

The process is iterative: at each stage, changes may become necessary due to inconsistencies coming to light. For example the requirements specification stage may identify problems, necessitating a change to the requirements definition.

This model of the process is typical, but not set in stone: some of the methods described suggest changes, notably Viewpoint Resolution (Section 2.3.2) which argues that validation should occur as a sub-process of elicitation.

In the following sections we describe the viewpoint approaches. Each section begins with a table classifying and summarising the approach; the classifications correspond to the issues referred to in Section 2.2.1

Context refers to the purpose of the methodology and its place in the development process.

Formalism refers to whether or not the methodology supports multiple formalisms and how the alternative views are reconciled.

Viewpoints identifies how viewpoints (or the equivalent concept) are defined and structured.

Method describes how viewpoints are actually used.

2.3 Viewpoint Approaches

2.3.1 CORE

Context	System Modelling, Requirements Definition
Formalism	Single specification language used at viewpoint level
Viewpoints	Sub-processes of system or external entities: sub-processes are hierarchically structured into functional subsystems
Method	Used to formalise functions and dataflows of processes

CORE (COntrolled Requirements Expression) [Mul79, Mul84] is one of the earliest requirements methods to define the notion of viewpoints. The CORE process is well described in the literature [Mul79, Som96, Sto92, KS92]. Viewpoints are identified in an informal "skull-session" between everyone involved (users, specifiers, buyers), and classified as functional or non-functional. The method provides general guidelines for decomposing the functional viewpoints which are internal to the system (direct viewpoints) into functional sub-systems and analysing their function, input, source, output and destination. It is at this stage that conflicts relating to information flow between viewpoints can be identified, as outputs from one viewpoint can be tied up with inputs to the destination viewpoint.

Further stages analyse the structure of data output by each viewpoint, and identify transactions across the system. Finally, non-functional aspects of the system (constraints) are analysed. The system has inherent weaknesses ([KS92, Sto92]):

- A poorly-defined notion of a viewpoint.
- Difficult to model processes which use different representations.
- Provides only general guidelines for identification and structuring of viewpoints, so it is likely that two analysts on the same problem will come up with radically different viewpoints which cannot be resolved.
- The data-structuring step only analyses output, on the grounds that all output is also input to a destination viewpoint; this means that input from indirect viewpoints (those which are external to the system) will never be analysed.

Extensions to CORE

CORE appears weak in areas such as consistency checking and does not support multiple formalisms. More recently there have been proposals for extensions to CORE to provide for support of specification reuse [Fin88] and to model changes in specification [Sto92].

In the TARA project, Finkelstein [Fin88] proposes using the cross-system transactions identified in the CORE process as the building-blocks for reuse, selecting analogous transactions from a knowledge base to fit a target partial specification, by a variety of methods such as pattern matching and generalisation. After selection a transaction must be restructured to resolve inconsistencies and then "plumbed in" to the partial specification. The author admits that the method employed is not clean, due to the top-down nature of the CORE methodology which is not easy to reconcile with the inherently bottom-up nature of reuse, and concludes that reuse cannot be bolted onto an existing methodology but must be planned for from the start.

Stokes [Sto92] proposes extensions to CORE to allow for changes in specification, and presents Z schemas to formalise the method. He proposes two levels of validity for specifications; evolving and consistent, and introduces the idea of specification continuations as a function from an incomplete (evolving) specification to the minimum consistent specification.

2.3.2 Viewpoint Resolution

Context	Validating requirements as sub-process of elicitation
Formalism	Views are expressed using a given propositional language
Viewpoints	Integration of 3 perspectives of analyst's view of problem
Method	Discrepancies in and between 2 analysts views identified by heuristic analyser

Viewpoint Resolution [LF91] is a methodology for "very early validation" of requirements; the authors believe that validation should occur as a sub-process of the elicitation stage, rather than in requirements specification as in Section 2.2.2. They provide a method for formalising viewpoints and analysing them, using a heuristic analyser and a viewpoint language, VWPl, which is used to express a particular viewpoint, resolve its internal conflicts and then compare it with other viewpoints. The end result is then a consistent requirements document.

In this method, which the authors stress is an aid to fact validation and not a requirements language, a viewpoint is defined as a “standing or mental position used by an individual when examining or observing the overall context of the proposed system.” To identify and classify discrepancies, a process of “view construction” is used to integrate perspectives and hierarchies from a viewpoint. Perspectives are different modelling aspects; the method uses data perspective, actor perspective and process perspective, and the is-a and parts-of hierarchies. Each analyst must express the problem in the VWP1 language using each of the three perspectives; the perspectives are then compared using a heuristic-driven static analyser, and a list of discrepancies is produced. When there are no more discrepancies, the perspectives are integrated into a view.

The view of another analyst can then be compared to that of the first. The whole process aids the analysts in understanding the problem and bringing out any invalid assumptions they may have made.

The static analyser works by finding similar pairs of rules and discrepancies between them. Classification of discrepancies is based on scores resulting from pattern matchers and borrows from the artificial intelligence concept of *analogy*, as in other methods which consider the problem of conflict resolution [LLC91] and specification reuse [Fin88, Mai91]. The analyser classifies discrepancies as due to: wrong information, where there is a contradiction between facts; missing information, where hierarchies are incomplete or some rules or facts are missing; and inconsistency, due to redundancy or a contradiction between a fact and the hierarchy.

Viewpoint resolution is a methodology to help the requirements elicitation process, and thus is not concerned with supporting alternative design methodologies, or incremental specification. The fact that analysts must learn a new language in which to describe their perspectives may be a disadvantage, although the authors claim that no more than two hours training is needed for analysts to be able to use the language.

2.3.3 The Requirements Apprentice

Context	Requirements definition and specification
Formalism	Requirements expressed using given propositional language
Viewpoints	Incremental requirement statements
Method	Takes requirements and reusable components, updates knowledge base, identifies inconsistencies, suggests solution via heuristics, reasoning

The Requirements Apprentice (RA) [RW91] is part of the on-going Programmer's Apprentice project being developed at MIT. Whereas Viewpoint Resolution addresses itself to validation as a subprocess of the requirements elicitation stage, the RA fits into the the Requirements Definition/Specification stages. It is a knowledge-based system which encourages reuse of partial requirements documents, supports evolution of requirements and has facilities for detecting and resolving inconsistencies.

The RA consists of three modules: a "Cliché library" containing a collection of reusable requirements; a hybrid knowledge-representation and reasoning system termed a "Cake" with a layered architecture of propositional logic, algebraic reasoning, frames and plan calculus; and an "Executive" for interaction between analyst and system. Communication is via a formal command language. Requirements are developed by means of a dialogue between analyst and system in which the analyst issues commands regarding requirements and the system checks them for consistency in the knowledge base. Central to the method is the support for informality of requirements; evolving requirements are provided by the analyst in an incremental manner, and information which the executive recognises as omitted can be "pended" until the analyst is ready to provide it or retract other requirements. In addition the library of clichés provides for reuse of requirements and allows some omitted information to be filled in by default or deduction. The "Cake" module makes deductions about the evolving requirements and informs the analyst when a conflict has been identified and what deductions and underlying premises led to this conflict, and gives suggestions for resolving it. Again the analyst can choose to deal with the conflict or ignore it for the time being, as it might be resolved later by further statements.

The Requirements Apprentice does not provide support for multiple formalisms; its concern is with formalising requirements. The end product of the process is a consistent requirements document which is passed onto the next stage in the Programmer's Apprentice project.

2.3.4 Domain Goals

Context	Requirements specification, System specification
Formalism	Specification components linked to goals
Viewpoints	Goals represent attributes of system. Hierarchical structure of subgoals determine increase or decrease in satisfaction of parent goal
Method	Correspondences and conflicts between goals identified by negotiation, analogy, heuristics; resolution to integrate specifications.

Domain goals [Rob89] develop Feather's parallel elaboration model [Fea89a] to allow automation, considering the problem of integrating parallel designs using the general notion of plan integration and negotiation. Domain goals are required attributes of the system, such as "Loan Period" in a library system; this will have a proposed value such as "2 weeks" in one perspective. The goals are linked to specification components which support them. The parallel development method can be used for multi-perspective design, with alternative perspectives of a single system, and also for parallel development of different systems with a common basis. This allows for reuse as in the Requirements Apprentice (Section 2.3.3).

The process of integration of perspectives is achieved via a goal/subgoal structure in which attributes are used to indicate whether an increase in satisfaction of a subgoal will result in an increase or a decrease in satisfaction of the parent goal. Goals have attributes associated with them which are set to zero in the base model and then altered as a result of elaborations. Elaboration linkages between goals and specification components are used to ensure that each goal is supported by a specification component, and each component is justified by a goal.

Integration is a four-step process which corresponds to the steps identified in Section 2.2.1. Correspondences between specification components are first identified, either by tool support for components with the same name or by the analyst for differently-named components with the same meaning. Corresponding components are then structurally compared to identify which ones conflict, and conflicting components of each specification are grouped together based on common domain goals from which they are derived. This stage can be automated to a significant degree.

The next step attempts to remove differences in specifications by compromise or substitution. The domain goals from which conflicts stem are analysed by a variety of (manual) methods; negotiation of attributes, goal substitution, heuristic rules or case-based solutions. The resolution of conflicts is then mapped back to specification level. This should involve simple merging and patching of specifications.

Domain goals tackle the question of resolution of conflicts by calling on negotiation and planning. Tool support is provided by a specification environment, Oz, which allows creation of domain goal graphs, specifications and integrations. Automated support for integration is at an early stage.

2.3.5 Prisma

Context	Requirements specification, System specification
Formalism	Entity relationship models, dataflow diagrams, Petri nets, interpreted centrally in first-order logic
Viewpoints	Representations of system in most appropriate formalism; information in each must correspond
Method	Formalise elicited requirements; switching between views identifies conflicts via heuristics

Prisma [NMS89] is described as a pluralistic knowledge-based system reflecting the fact that specifications do not, in practice, live over a single formalism. Specifications written in different formalisms express particular *views* of a system; each view comprises limited perspectives. The Prisma approach is to capture syntactic, semantic and pragmatic aspects of common methods (entity relationship models, data-flow diagrams, petri nets) by interpreting them into a common logical notation and provide tool support for construction and integration of views.

The Prisma framework generalises the traditional "flat" specification process to a multiple view specification, where at each specification level several parallel views are maintained, with mappings between them such that observable properties of one view should have corresponding properties in another. To solve the problem of translating between views in different formalisms, views are first interpreted in first-order logic, and heuristics are used to help in verifying that properties from one view are properly represented in others.

The intended use of the Prisma system is to formalise the requirements obtained by an analyst as a result of the initial elicitation session with the user. The prototype environment provides an "agenda" mechanism providing advice about tasks which need to be done in view construction and suggestions to eliminate inconsistencies, a "paraphraser" which uses the validation heuristics to generate natural language descriptions of the current view, and a "complementarity checker" which is invoked when the analyst switches between views, informing the analyst of associations between properties which must be observed when a new view is being constructed from a previous one, and allowing partial consistency-checking.

The Prisma prototype environment provides partial answers to some of the theoretical problems of mappings between views; the ideas are generalised by more recent research in the VOSD framework (Section 2.3.6).

2.3.6 ViewPoint Oriented Software Development

Context	Requirements specification, System specification
Formalism	Theoretically unlimited choice of multiple formalisms, also alternative development strategies
Viewpoints	Represent one role of an analyst; define responsibility, development strategy, formalism. Loosely coupled, non-hierarchical structure
Method	Decentralised framework used to manage viewpoints, related only by local checks. Framework remains in place during full development cycle

The ViewPoint Oriented Software Development (VOSD) framework [FKN+92] is the most recent in a number of viewpoint approaches from Imperial College. An earlier approach of interest is Multi-Party specification [FF89], an attempt to model the underlying mechanisms of how people write specifications. Development is viewed as a dialogue between viewpoints, each of which maintains a store of statements to which it is committed, and which is modified by "speech acts", governed by dialogue rules. Using such rules the dialogue progresses by "simple elaboration" or incremental evolution. Once the dialogue is completed the specification is represented by the pool of commitments. While this is an interesting way of looking at the software development process, it has practical limitations as inconsistencies will only come to light through sufficient dialogues.

VOSD considers a viewpoint as a partitioning of a system according to agent, development method and formal representation style; it encapsulates partial knowledge of a system. Viewpoints here are loosely-coupled, locally-managed entities: the framework is decentralised and viewpoints are interrelated only by local checks.

The framework is intended to support concurrent, distributed and cooperative work by multiple agents, each of which might have more than one perspective or responsibility in the system; in addition to supporting differing formalisms (as in Prisma, Section 2.3.5) it supports alternative development strategies (top-down, bottom-up). A viewpoint object contains slots to hold the representation style, domain, specification and work record (encapsulating partial knowledge about the system and domain) and a work plan for the viewpoint; the work plan provides actions for creating new viewpoints from a template, building specifications, and support for checking consistency by in-viewpoint and inter-viewpoint checks. The VOSD method has been used with some success to describe development methods such as JSD and SSADM.

2.3.7 ARIES

Context	Requirements definition and specification, system specification
Formalism	Theoretically unlimited choice of multiple formalisms
Viewpoints	Presentation of knowledge in central base, used to view/alter requirements.
Method	Incremental development and re-use of standard requirements via modularised, highly expressive internal representation

Acquisition of Requirements and Incremental Evolution of Specifications (ARIES) [JFH92] implements ideas from Feather's 1989 papers [Fea89a, Fea89b]. In this system a viewpoint is a presentation of partial knowledge. It utilises a single, highly expressive modularised internal representation for all information about the problem, and a variety of presentations are available to view and alter the information. A hierarchical structure of "folders" is used both to separate and to allow sharing of information between analysts and across projects.

The system supports incremental development via communication between analyst and system in a manner familiar to the human. There is no need to force the analyst to learn another requirements language. The analyst makes changes to the presentation, and these are echoed internally by translating to the internal representation. It also encourages information hiding and reuse of evolutionary changes. Limited validation is available by abstraction and reasoning: there is no support as yet for consistency checking, although it is addressed in [Fea89b].

The system has similarities to the Requirements Apprentice; in this case however the end product is a validated specification in "Re-usable Gist" which is then passed on to an optimiser. It can be most directly compared with the VOSD system (Section 2.3.6): both provide for multiple presentations of emerging specifications, graphical display of viewpoints and have a semantic-net formalism underlying them; however ARIES uses a centralised internal representation of the system being developed, while VOSD uses a framework of loosely-coupled, locally managed viewpoints.

2.3.8 Viewpoint Oriented Analysis

Context	System modelling, Requirements definition
Formalism	Single, shared through all viewpoints
Viewpoints	Service-oriented external entities, sending control information and receiving services and data. Hierarchical structure
Method	Integrate functional requirements and constraints for system by structuring and conflict resolution

Viewpoint Oriented Analysis (VOA) [KS92] is an object-oriented framework for requirements capture and resolution.

The motivation behind VOA is the need seen by the authors to integrate functional and non-functional requirements (constraints, such as timing and reliability); CORE and Viewpoint Resolution see viewpoints as simple sources or sinks of information, or sub-system processes. In contrast, viewpoints in this framework are "service-oriented" entities, external to the system but interacting with it, receiving services from the system and providing control information and data to it. They are divided into active (those which initiate services) and passive (information sinks).

A four-step process is identified: viewpoint identification, structuring and decomposition of viewpoints, information collection, and information reconciliation. Viewpoints are structured such that sub-viewpoints are more specialised and inherit services, attributes, control information and constraints from their parents. The information collection stage prepares a more detailed specification of each viewpoint in terms of control information and data generated; this provides a basis for completeness checking. The information reconciliation stage ensures that all services are provided, identifying omissions and conflicts in provisions of services across viewpoints.

The method is a requirements resolution technique; tool support is being developed for graphical manipulation of viewpoints.

2.3.9 Alternative Viewpoints Hierarchy

Context	Requirements definition
Formalism	Viewpoints share common formalism
Viewpoints	Self-consistent representations of an agent's knowledge; hierarchical structure of sub-viewpoints inherit attributes from parent
Method	Sub-viewpoints used to represent conflicting alternatives, to bring out any assumptions made and force negotiation by analysts

Easterbrook [Eas92, EN94] proposes that in order to clarify the representation of conflicts, common ground must be established between viewpoints. He notes that the common ground is explicit in many of the methodologies: for example in incremental specification methods the common ground is the initial specification, and in Robinson's Domain Goals (Section 2.3.4) it is the shared domain model. The VOSD framework (Section 2.3.6), by contrast, makes no assumptions about common ground.

In this object-oriented methodology, a viewpoint is a self-consistent description of an area of knowledge with an identifiable originator: a viewpoint is created to represent the knowledge of each person the analyst comes into contact with. If conflict is detected in a viewpoint, the piece of information which caused the conflict is placed in a descendant viewpoint and the negation of that information is placed in another. Both inherit the characteristics of the original viewpoint. A hierarchy of viewpoints evolves as distinctions between viewpoints become clear. The method is not intended to aid the requirements elicitation process, but is a domain-modelling environment for representing and manipulating elicited knowledge. Tool support is provided by a prototype, *Analyser*, which allows the expression of viewpoints in a number of formalisms (first-order predicate logic, data-flow diagrams, state transition diagrams) and provides a graphical representation of the evolving viewpoints hierarchy.

Detection of conflicts is achieved by a special viewpoint for each representation scheme which includes routines for conflict detection and an inference engine. All viewpoints using that scheme inherit from the special viewpoint. Conflict detection is limited: it is spotted only by explicit contradictions, and conflicts caused by use of different terminology (synonyms) will not be caught. There are currently no heuristics for conflict detection.

2.3.10 Multi-Paradigm Specification

Context	Requirements definition and specification, system specification
Formalism	Widest possible range of specification languages, design methodologies
Viewpoints	Partial specifications in any formalism, semantics interpreted in typed internal representation (predicate logic)
Method	Specifications composed by union of specificands in semantic domain. Limited consistency checking

Jackson and Zave [JZ93, ZJ93] use an approach to composition of partial specifications which is similar in motivation to the VOSD and ARIES frameworks: here the emphasis is not on tool support but on extending the range of specification techniques that can be considered.

The technique addresses itself to the diversity of specification techniques — not only the common algebraic, state-based and set-theory languages, but less formal techniques such as data-flow diagrams and decision trees, and problem-specific programming paradigms such as functional or object-oriented languages. It involves translating the semantics of a range of specification techniques into first-order predicate logic. This provides an extremely general common semantic domain, as types and other structures can be expressed as predicates — in contrast to the common internal representation used in the ARIES project which is high-level and strongly typed. The semantic domain consists of distinct individuals and a finite set of predicates. The semantics of the composition of partial specifications is then the set of all members of the semantic domain satisfying all the partial specifications. A set of partial specifications is consistent if their composition is non-empty.

In practice it is infeasible to translate a large number of partial specifications into predicate logic; the result is likely to be a number of large, incomprehensible formulae. The authors conclude that practical consistency checking must come at the same conceptual level as the specification language.

The aim of the technique is that it should enable a specifier to construct specifications by combining partial specifications written in the language best suited to expressing and analysing the properties of a system. Further work includes extending the domain to include control, and investigating the implications for reuse.

2.3.11 Viewpoint Amalgamation: The MFD Model

Context	System specification
Formalism	Generic model; instantiated for single formalism
Viewpoints	Partial specifications; no structure imposed
Method	Models amalgamation of two viewpoints by successive refinement steps until the viewpoints are consistent

The MFD (Modular Formal Description) model [Wal92, ACGW93] is intended to exploit an intuitive understanding of what makes a good modularisation and the way people write specifications in practice, with the aim of making formal descriptions easier to understand. It is a generic model, independent of any particular descriptive formalism, and thus can be applied to a number of problem domains: examples are incremental development and integration of multiple perspectives. The MFD model provides a conceptual framework for relating different aspects of modularisation. The motivation behind it is the need identified for a general, unconstrained understanding, abstracting away from specifics of particular specification languages.

The model is used to describe the amalgamation of viewpoints, considered in this case to be partial specifications. Amalgamation of the viewpoints is regarded as necessary in order to check for consistency across the whole specification, and so to be able to implement the specification. This contrasts strongly with the ideas behind methodologies such as VOSD, which regard consistency checking as an issue to be addressed by inter-viewpoint checks and amalgamation of viewpoints as optional.

The amalgamation process combines formal descriptions, producing an amalgamated description and an amalgamation trail; the latter is a record of all changes made during the amalgamation. The rationale behind this is to ensure the process is reversible, and so enable a different modularisation of a system for implementation purposes. In addition any changes can be processed by re-winding the amalgamation of a particular viewpoint, altering or replacing it, and then re-amalgamating.

The amalgamation process is given a formal basis by the theory of *co-refinement* [ARW96] (Chapter 3). This is a weaker version of *data refinement*, in which variables can be added to a specification and refinement laws are shown to hold subject to a restriction on the values of the added variables. By this process viewpoints can be refined repeatedly until they are consistent. The model has been applied to amalgamation of viewpoint specifications written in Z [ACGW93]; a formal definition of viewpoint amalgamation in terms of co-refinement is presented in [Ain95].

The MFD model is in its infancy; in particular no tool support exists and only a small number of formalisms have been considered. In motivation it is comparable with the multi-paradigm approach and the VOSD and ARIES frameworks; the wish is for a way to describe what happens when different viewpoints resulting from multiple perspectives or from an incremental change to a specification need to be amalgamated.

2.4 Summary and Conclusions

The viewpoints originated by CORE are a springboard for a wealth of different approaches, and there is wide variation in the definitions, structure and use of viewpoints - even amongst those approaches which do not use the term explicitly. While most uses of viewpoints are in the field of requirements engineering, others involve specification and coding. In later chapters we consider other paradigms in software engineering, in which the viewpoints idea can be used to formally explain the relation between constituent parts of the development.

We return to the VOSD framework of Finkelstein et al [FKN$^+$92] in Chapter 5, where we consider the issues raised in attempting to model the distributed framework with refinement ideas developed in Chapters 3 and 4.

Viewpoints are emerging as an important contribution to the requirements analysis and formal specification processes. Recent developments include a meeting of the BCS Requirements Engineering Specialist Group on the subject of "Viewpoints in Requirements Engineering" (Edinburgh, October 1995) and a special "viewpoints" issue of the IEE/BCS Software Engineering Journal [SEJ96], in which articles on a number of techniques featured in this chapter appear. Ongoing research also indicates that the viewpoints paradigm can also be applied to areas such as safety analysis [KS94].

This chapter has illustrated the wide range and use of viewpoints in several guises, with varying degrees of formality. In the next chapter we consider how the use of viewpoints can be formalised by considering the notion of a *refinement relation*.

Chapter 3

Amalgamation and Refinement

In this chapter we consider the use of a paradigm from formal methods research for relating viewpoints together. This provides some background for later chapters in which we consider applications of viewpoints and refinement to model and explain explanations.

3.1 Introduction

We have established (Chapter 2) that viewpoints are used, sometimes implicitly, in a number of documented approaches to software development. However few of these approaches deal with viewpoints, and the relationships between them, in any formal way. Recent work at the University of Bath is concerned with this issue; in particular Ainsworth [Ain95] (see section 4.5).

In this chapter we begin with a discussion of what we mean by amalgamation of viewpoints. We then consider *refinement* and examine the problems which arise from trying to apply strict refinement theory to the amalgamation process. This leads us to consider some solutions to those problems.

3.2 Amalgamation

We seek to justify the use of viewpoints to structure and explain systems, by calling on theoretical refinement work. This will enable us to prove whether or not desirable properties of each constituent viewpoint are held by their amalgamation.

Amalgamation can be described simply as the composition of viewpoints in the most appropriate way in order to produce a viewpoint which is related to each of its constituent viewpoints. Just as we gave (Definition 1.1) a very general definition of what a viewpoint is, so it is hard to give a much more precise definition of amalgamation without being overly prescriptive. From that definition we know our viewpoints have the following properties:-

- They are described in an identified, *decidable* language
- Operations for composition of viewpoints are provided, as part of the language or in an associated development method.

Examples of composition operators are sequential composition in a programming language, conjunction of logical formulæ, and concatenation of natural language sentences.

3.2.1 Reasoning about Amalgamations

In Chapter 1, we presented an example in which a natural language viewpoint could be merged with a more formal specification. This illustrates the expressiveness of the general approach, but we lose the ability to say anything very much about the relation between the amalgamation and its constituent parts. We could quite easily amalgamate the natural language sentence "The clock has a digital display" with a formal model of a steam-engine. What tells us that this would be unhelpful is the assumption that the natural language sentence has a *meaning* that is not consistent with that of the formal model.

So if we intend reasoning about the relations between viewpoints and their amalgamations we need to make a further stipulation:

- The language of the viewpoint has a defined *semantics*.

The semantics of the language will enable us to relate the meanings of the viewpoints. This does not prevent us from amalgamating viewpoints in the manner described above. We should simply be aware that some amalgamations will remain unverified, unless we have a natural-language parser at our disposal (which would inevitably impose some restrictions on the range of sentences which could be recognised).

Because of the semantics issue it is advisable only to attempt to reason between viewpoints whose meanings can be related - as we saw in Chapter 2's survey, some techniques ([NMS89, JZ93, JFH92]) achieve this by presenting a common semantic model into which different formalisms are mapped, while others ([KS92, Mul79]) insist on a common formalism.

In talking about the amalgamation process we use stages identified in the approach of Wallis [Wal92], introduced in the survey (Section 2.3.11). We divide amalgamation into separate stages of *Coalescence planning* and *Coalescence*. The former applies to the process of comparing viewpoints for naming conflicts and working out the changes that need to be made to each. The output of this process is a *Coalescence Plan*. Coalescence is then the process of making these changes, via a series of correctness-preserving steps, and composing the viewpoints, using the composition operations given in the notation used by the viewpoints. The product of this composition is called the *amalgamation* of the viewpoints. A by-product of the amalgamation process is the *Amalgamation Trail*, which records a history of the amalgamation. Figure 3-1 illustrates this process: boxes signify items produced or input to the process, circles are processes and directed arcs lead between the two.

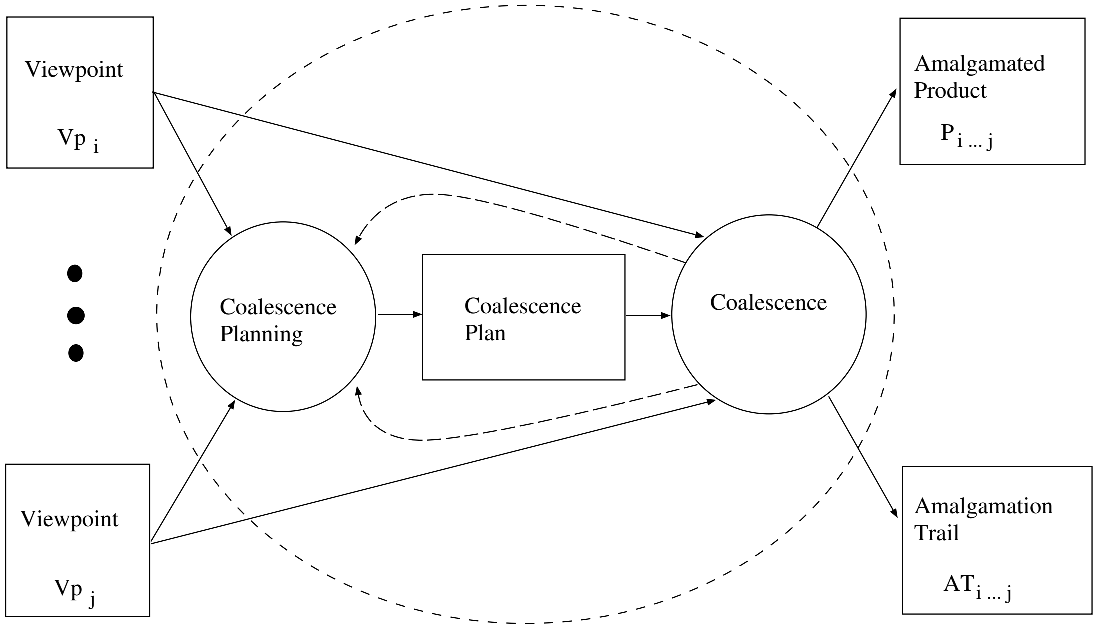

Figure 3-1: MFD model process

We will refer to these steps later in the current chapter when we consider applications of refinement to the amalgamation process, and in Chapter 5 where we compare our approach to that of ViewPoint Oriented Software Development.

3.3 Refinement

We now consider the concept of a refinement relation between viewpoints. We begin with a discussion of the intuitive notion of an "improvement", to motivate a more formal definition of refinement.

3.3.1 Improvement

Most of the time, we have little difficulty in "real life" in identifying when something is an improvement over something else; a modem which transfers data at a rate of 28.8 Kilobytes per second is surely better than one which works at 14.4. However, implicit in this comparison is the idea that the speed requirement is more important than others such as price and reliability; a modem which operates at twice the speed but which costs twice as much and is half as reliable isn't much of an improvement. A lesson here is that requirements need to be made as explicit as possible — if we have been explicit in setting out our requirements including stipulations on cost and reliability, there is clearly no improvement if these requirements are not met in the final product. If the basis on which comparison is being made is not specified precisely, it makes no sense to talk about an improvement (though this would not stop a salesman from trying to do so).

Part of making the basis of comparison clear includes identifying the perspective from which the comparison is being made; the above example may be an improvement for the customer if price is reduced and reliability is increased; however for the provider of the product, the opposite may be the case (though in reality other economic factors would have to be taken into account, dictated by the laws of supply and demand, among others).

The following short tale will illustrate some of the issues.

The Broken Photocopier

Office photocopiers are justly famous for duplicating the content of sheets of paper of varying sizes cleanly, quickly and accurately — unless the human operator is in a hurry, or the task is of great importance, in which case the photocopier will usually create a paper jam.

Such a product is, inevitably, subject to relentless enhancement and one such is the addition of a stapler. Amalgamate a photocopier with a stapler and what you get is a photocopier which, having produced a pile, or several sorted piles, of output, shuffles the piles so that the corners all match up and staples the top left corner of each pile[1]. So far, so good.

However, in being merged with the photocopier the stapler picks up one of the photocopier's more annoying habits; the tendency to jam. The photocopier behaves as it always has when a jam occurs; it ceases to work until the jam is cleared. Thus, an otherwise perfectly functioning photocopier is out of action because of an errant staple which requires an engineer to come out and fix. No matter that the frantic human operator doesn't even *want* the paper stapled.

As a result of customer feedback, the photocopier is subject to a further improvement; the "I don't want a stapler" button. With great pomp and ceremony the manufacturers announce the enhancement: if you have a problem with the stapler, simply press this button.

What is of interest to us in this sorry tale is that the initial change appeared to provide an improvement, but on more detailed investigation it didn't. This is because the old version could be relied on to work as long as certain conditions were maintained such as power, correct use, paper tray not empty, paper not jammed. In the new version, these conditions could all hold and the photocopier could still not work. We would like to be able to rely on an improved version being able to *produce any result consistent with our requirements for the original one.*

However, if we can "stabilise" the stapler in some way, we may be able to identify an improvement as far as "everything else" goes. This is effectively what occurs in the next iteration; a mechanism is found for disabling the stapler, and an improvement can be observed.

3.3.2 Formalising Improvement

This discussion has lead us to the point of being able to say that, as long as it is explicit what is being compared, we should have no trouble in being able to recognise an improvement when we see one. Being explicit about what is being compared involves identifying how we can be sure that the final product is *correct* with respect to its *specification*; in the case of the above examples this would involve demonstrating that each of the properties stipulated in the original requirements are held by the final product. We use specification here to mean an agreement between supplier and customer which

[1] In answer to the question "Do such things exist in real life", yes, they do: this thesis has been photocopied on one.

sets out the required properties to be exhibited by the product.

If we define a relation $\leq$ such that $A \leq B$ means that A can do (at least) everything B can do, then we have a relation that is reflexive (A is as good as A) and transitive (B is at least as good as A, and C is at least as good as B, so C is at least as good as A). It therefore defines a *pre-order* over the domain of A and B (modems, photocopiers, what have you). We would also insist that the relation preserves *correctness*, so that if there is some initial specification S which is satisfied by A, and $A \leq B$, then S must also be satisfied by B.

We will call a relation which has these properties a *refinement* relation, defined below. We will speak in terms of viewpoints, which may as we have seen be any kind of "product", or fragment of a product, which we may be interested in.

Definition 3.1 (Refinement) *For two viewpoints A and B, B* refines *A iff B can be used in place of A and behave in the same way as A. This is written $A \sqsubseteq B$.*

We have observed that there are some developments of viewpoints which, while intuitively appearing to "refine" previous versions, do not maintain all the properties held by those versions. This may arise, for example, when some property is thought better of and removed from the next iteration.

3.4 Refinement Properties

While the method of generating and verifying a refinement may be particular to a specific refinement process, we identify some properties which should apply to any refinement method, whatever the nature of the actual viewpoint being refined.

3.4.1 Preserving Correctness

We cannot describe a viewpoint simply as "correct"; it must be correct with respect to something. In terms of specifications, a program will be correct with respect to a specification if the program will always terminate when the specification says it should, and every possible final state of the program will satisfy the specification. Suppose a program P which is correct with respect to an initial specification S. If we derive a further program P' from P then for P' to refine P, P' must also be correct with respect to S. Similarly, if we derive a specification S' from S, if we require that S' refine S

then any program which is correct with respect to S' must also be correct with respect to S.

3.4.2 Pre-order

As outlined above the refinement relation should be transitive and reflexive. Transitivity enables us to make refinements in a stepwise, iterative manner, allowing us from n steps of the form $P_i \sqsubseteq P_{i+1}$ to conclude that $P_0 \sqsubseteq P_i$. Reflexivity gives us a base case for the process.

Having a relation which is a pre-order and preserves correctness allows us to assemble our viewpoints by *Vertical Composition* [ST95] — another name for stepwise refinement, as described above. A complementary way to build viewpoints comes from *Horizontal Composition*, illustrated in Figure 3-2. This allows development to proceed by decomposing the viewpoint into modules, and refining just some of those modules while keeping the rest constant. Reflexivity ensures that “keeping the rest constant” still provides a refinement.

$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[d] & + & C \\ B' & & \end{tikzcd}$$

Figure 3-2: Horizontal Composition: A is decomposed into B and C, and B is then refined to B'. We then have $A \sqsubseteq B' + C$

However, we do need to show some care over the choice of operations used to compose our viewpoints — these are the operations used by the relevant language of the viewpoints concerned.

3.4.3 Monotonicity of Operators

The process of development outlined above depends on the fact that, if $F(P)$ is a viewpoint which contains a sub-viewpoint P, and we derive some Q such that $P \sqsubseteq Q$, we have $F(P) \sqsubseteq F(Q)$. This in turn relies on the operations used to compose $F(P)$ being *monotonic with respect to refinement.*

Definition 3.2 (Monotonicity) *A function f is monotonic with respect to an ordering $\leq$ if, whenever $X \leq Y$, $f(X) \leq f(Y)$.*

Monotonicity is a property that needs to be considered, especially in the case of parameterisation and abstraction [Mor88b]. Programmers are taught the dangers of call-by-name procedure calls, and that call-by-value is the safer method; this is because of the potential loss of monotonicity due to the possibility of distinct variable names in a procedure taking on the same actual parameter (aliasing).

3.5 Example Refinement Relations

By way of illustration we now list some refinement relations which satisfy the properties given in Section 3.4.

3.5.1 Equality

If $A = B$ we can certainly be sure that B is an acceptable replacement for A. It trivially has the required properties of pre-order and correctness preservation. Equality is the weakest form of refinement and isn't really of much practical use for development!

3.5.2 Equivalence

If $A \equiv B$ we have a more interesting ordering. The difference between this and the last case is that A and B may structurally be quite different; but looked on as black boxes, A and B operate over the same state space and take the same input to the same output. B can thus replace A in any context and the properties hold as required. While this may not seem any better than equality it is in fact used as the ordering in some algebraic specifications and functional program developments, in which the function domains are fixed. For example the two functions

$$twice : \mathbb{N} \to \mathbb{N}$$
$$twice(x) = x + x$$

$$double : \mathbb{N} \to \mathbb{N}$$
$$double(x) = x * 2$$

are equivalent, but not equal.

The orderings in both these cases are equivalence orderings, a special case of pre-ordering.

3.5.3 Domain Extension

The next step in the ordering of orderings is to step out of equivalence into pre-order. We define a conservative extension as follows:

Definition 3.3 (Conservative Extension) *For two functions f and g, g is a conservative extension of f iff* $\operatorname{dom} f \subseteq \operatorname{dom} g$ *and* $g(x) = f(x)$ *for all* $x \in \operatorname{dom} f$.

Our earlier function *double*, defined over the natural numbers, can be refined by extending its domain to the integers:

$$doubleInt : \mathbb{Z} \to \mathbb{Z}$$
$$doubleInt(x) = x * 2$$

The new function will always provide an answer compatible with the old one when given a non-negative integer.

The conservative extension is certainly reflexive. Transitivity comes from the relation $\operatorname{dom} f \subseteq \operatorname{dom} g$ in the definition. If we define correctness for functions simply in terms of mapping inputs to outputs then this ordering will preserve correctness: however, we may wish to ensure that our function is *undefined* for certain inputs - for example the function *div*, defined as

$$div : \mathbb{Z} \times \mathbb{Z}^+ \to \mathbb{Z}$$

$$div(x, y) = x/y.$$

We would prefer that any function which refines this function be undefined for $y = 0$; a new improved div^+ which gave a value to $div^+(x, 0)$ would be a novelty, but would not be correct.

3.5.4 Range Restriction

At first sight restricting the range of a viewpoint may not seem much of an improvement. Consider however the function *sqrt*:

$$sqrt : \mathbb{R}^+ \to \mathbb{R}$$
$$sqrt(x) = \{y : y^2 = x\}$$

If we now come up with a function *posSqrt* which will always return the positive square root, we have restricted the range to $\mathbb{R}^+$ (and removed non-determinism into the bargain). Since a positive square root is a square root, our new function can be said to be an improvement over the old one. We define range restriction:

Definition 3.4 (Range restriction) *For two functions f and g, g is a range restriction of f iff* $\operatorname{ran} g \subseteq \operatorname{ran} f$ *and*

$$\forall\, y \in \operatorname{ran} g.\, \exists\, x \in \operatorname{dom} f \bullet y = f(x) \wedge y = g(x)$$

The definition ensures that anything produced by g could also have been produced by f from the same input.

Such a relation is still a pre-order, transitivity coming as a result of the subset relation. Preservation of correctness again depends on the definition of correctness; if we specified just that the result of our square root function should return a result in the range $\{-\infty \ldots \infty\}$, the *posSqrt* function will be correct (since anything in the range $\{0 \ldots \infty\}$ is certainly in the larger range). If however we required that it was equally likely for a negative square root to be returned as for a positive one, *posSqrt* will not be correct. A random-number generator would not be regarded as correct if the number between 0 and 1 that it returned was always 0.673, and anyone tossing a coin would be annoyed to find it always came down heads.

The above refinement relations have dealt with examples expressed as functions, for the sake of simplicity. Our viewpoints may be functions; or they may be state-based specifications, predicates or anything with a defined semantics. The semantics allow us to characterise a viewpoint as a function over a state-space or between truth values.

A further point to note about the above relations is that there is an implication ordering on them: *equality* $\Rightarrow$ *equivalence* $\Rightarrow$ *extension*, and *equality* $\Rightarrow$ *equivalence* $\Rightarrow$ *restriction*. Restriction and extension are not comparable in the ordering.

3.5.5 Transformation

The next kind of refinement is one often used in program development from specifications. We may need, for reasons of implementation, to change from one data representation to another which is more concrete. The chosen representation will be subject to restrictions imposed by factors such as the target language and the size of the data to be dealt with.

In this case we would need to relate the new representation to the old by means of an *abstraction*. In terms of functions again, we would define a transformation as:

Definition 3.5 (Transformation) *For two functions f and g and abstractions abs_d, abs_r, g is a transformation of f iff:*

$$\forall x \in \operatorname{dom} f \,.\, \exists x' \in \operatorname{dom} g \bullet x = abs_d(x') \wedge f(x) = abs_r(g(x'))$$

In the definition abs_d and abs_r are abstraction functions on the domains and ranges respectively of the functions f and g. These abstraction functions must be surjective, so that all of the domain of f is represented in the domain of g.

Such a relation will be reflexive if the abstractions are simply identity transformations. By composing the abstraction functions between two successive transformations we may obtain a composite transformation, so providing the transitive property. Correctness, however, may appear to have gone out of the window.

However, the use of such a transformation is to change from one particular viewpoint model to another; correctness then should be defined as correctness of the model with respect to the initial specification. So preservation of correctness would be to ensure that the new model is also correct with respect to that initial specification.

This kind of transformation is called *data refinement* and is widely used in practice. The complement to data refinement is *algorithmic* refinement, which is concerned with making the viewpoint "more algorithmic" by reducing non-determinism and removing non-executable statements. A particular refinement method will use a combination of these techniques to derive a program from its specification.

All of the relations listed above have been shown to be valid as a refinement relation.

3.6 Operations for Specification Development

In the foregoing sections we have discussed the necessary properties of a refinement relation and have mentioned the need for specification building operations. In this section we will review these operations, in preparation for the operations we will be introducing for particular approaches to specification (of varying formality) identified in later chapters.

Choice of specification building operations involves a trade-off between expressive power and ease of understanding [ST95]. At the simplest level, operations can be limited to *enrichment* to add details to a specification and *hiding* to remove details. Even with these limited operations there is a reasonable degree of expressive power.

3.6.1 Z Schema Calculus

The Z schema calculus provides a methodology for the structuring of Z schemas. A schema typically has two parts, and may also be given a name:

name

$x, x', y : \mathbb{N}$

$x' > x + y$

The first section (the *declaration* part) includes type declarations, the second (the *predicate* part) a specification relating the variables in the schema. Various notational conventions are used, such as in this case where x' refers to the value of x in the state after the operation represented by the schema, and x to its value in the state before the operation. A further convention is to prepend Δ to a name to include both the before and after states.

An enrichment operator is provided by *schema inclusion*. Given a schema named *existing_schema*, we can define an enriched one as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{enriched\_schema}} \\
\hline
\textit{existing\_schema} \\
\textit{further declarations} \\
\hline
\textit{new predicates} \\
\hline
\end{array}
$$

This is equivalent to explicitly re-specifying the declarations and predicates from the existing schema.

Hiding is also provided:

$$Schema2 \mathrel{\widehat=} Schema1 \setminus (\Delta Hidden)$$

has the effect of removing before-state and after-state variables of *Hidden* from the declaration of the schema, existentially quantifying them in the predicates part. Hiding is used in the operation known as *promotion* [Woo89], where a schema defined on an individual entity is generalised to become defined on a complete system. The convention is to signify a framing schema by prepending the symbol Φ.

In addition to these operations the calculus provides more general operations for combining schemas: schema *conjunction* and schema *disjunction*. Schemas are conjoined by assembling the declaration parts and forming the conjunctions of the predicates, and disjoined by assembling the declarations and forming disjunctions of the predicates. Before this operation the schemas must be "normalised", so that the declarations have only type declarations. In the case of a schema such as

$$
\begin{array}{|l}
\multicolumn{1}{l}{Ex1} \\
\hline
x : 1 \mathrel{..} 10 \\
y : \mathbb{N} \\
\hline
x \times y < 100 \\
\hline
\end{array}
$$

normalisation takes the range declaration into the predicate:

$$
\begin{array}{|l}
\hline
\textit{Ex1norm} \\
x, y : \mathbb{N} \\
\hline
1 \leq x \leq 10 \\
x \times y < 100 \\
\hline
\end{array}
$$

The normalised schemas can then be combined.

Finally, a schema can be *negated* by negating the predicate after normalisation — thus the predicate in the above example would be $(x < 1 \vee x > 10) \vee x \times y \geq 100$.

3.6.2 VDM-SL

VDM (Vienna Development Method) [Jon90] has some similarities with Z [HJN93]. Development in VDM proceeds via data reification and operation decomposition; structuring operations are presented in the VDM specification language (VDM-SL[Daw91]). A VDM document is a list of definition blocks or a module list — specifications can be built using the "flat" language of definition blocks, but for any large specification modules are necessary if there is to be any chance of the document being understood.

A VDM module also consists of two parts:

module *name*
    interface part
    definition part
end *name*

The definition part defines entities of the module, while the interface part identifies entities imported from other modules (offering enrichment), entities exported from the module (to be made available to other modules for importing) and parameters and instantiations in the case of parameterised modules.

These constructs are more limited than those provided by the Z schema calculus, as what VDM-SL provides is a language for structuring a VDM document. The operations provided by the Z schema calculus are more geared towards the *development* of the specification. Other methods such as B [ALN$^+$91] and CLEAR [BG86] also provide operations for specification structuring. In addition extensions to VDM and Z have been proposed which provide further techniques for building large specifications.

3.7 Summary and Conclusions

The previous sections have illustrated the use of specification construction operations in the two most widely-known model-based specification languages. Each of these has a standard operation of enrichment.

In later chapters we will present some specification developments in which various specification building operations are used. For example, Chapter 7 uses a simple algebraic specification language which does not commit itself to any type declarations but only equational axioms (for didactic purposes). The basic operations we use are function composition, application and abstraction, and the refinement relation is conservative extension. Chapter 8 deals with the realm of denotational semantics and identify operations more clearly identifiable as enrichments. In each of these chapters, we will identify the operations and definition of refinement being used, and show that the necessary properties hold:

- Refinement relation is pre-order and preserves correctness
- Operations on specifications are monotonic

In this chapter we have discussed

- The concept of amalgamation of viewpoints.
- Background and motivation for refinement relations.
- Necessary properties of refinement relations and specification development operations to ensure correct developments.

This chapter has provided the background information and foundations which are necessary before we can undertake a further investigation of the use of viewpoints and refinement for modelling and explaining explanations in the chapters which follow.

Chapter 4

The Refinement Calculus and Co-Refinement

In this chapter we present an example paradigm for refinement of software specifications which illustrates some of the different kinds of refinement dealt with in the previous chapter. This leads on to particular refinement relations that are especially suited to explaining explanations.

4.1 Introduction

The paradigm of stepwise refinement [Wir71, Dij72] for the development of software is the basis from which more theoretical ideas have been developed, via Dijkstra's weakest precondition semantics [Dij76, Gri81] and Hoare's work on data representation [Hoa72], to the refinement calculi of Morgan [Mor94] and Back [Bac88], and the use of refinement methods in specification languages like Z [PST91] and VDM [Jon90].

The motivation behind the development of these approaches comes from the observation that, given a specification of a large system, verification of an implementation with respect to that specification is not a straightforward task; testing can never been exhaustive (although the specification can be used to generate test cases automatically). A formal proof that the implementation satisfies the specification is infeasible for a large program, due to the size and nature of proofs involved. An alternative approach is to use the paradigm of "divide and conquer" to develop the program incrementally from the specification, and prove each of these smaller steps to be correct.

The version of the refinement calculus we discuss here is Morgan's; in fact there are three major varieties of the refinement calculus, the earliest being Back's [Bac78]. Morris [Mor87] has also developed an approach. While each of these is a distinct method they are strongly related, and each makes a starting point from Dijkstra's language of guarded commands ([Dij76]).

4.2 The Guarded Command Language

The idea behind Dijkstra's guarded command language was to present a "toy" language to use for didactic purposes. It has three simple statements:

- *skip*, the most simple, which does nothing
- *abort*, which, even worse, doesn't even do nothing (it will not terminate)
- assignment $a := E$, assigning the value of the expression E to a.

These primitive statements can be composed via operators:

- Sequential composition (S1 ; S2)
- Alternation (**if**...**fi**)
- Repetition (**do**...**od**)

Alternation and repetition are expressed in terms of guarded commands $B_i \rightarrow S_i$. A guard B is a boolean value, and S is the program fragment executed if B is *true*.

A program fragment expressed in this language begins in a state satisfying a predicate (the *precondition*) and, if it terminates, will establish another predicate (the *postcondition*).

Dijkstra's concern was with characterising the semantics of a program S by identifying it as a *predicate transformer*, that is a rule for deriving, for any postcondition R the *weakest precondition* $wp(S, R)$ defined as follows:

Definition 4.1 (Weakest Precondition *wp*) *For a program S and postcondition R, $wp(S, R)$ is the weakest precondition sufficient to ensure that, if begun in a state satisfying $wp(S, R)$, S will terminate in a state satisfying R.*

A specification of program S can now be written $pre \Rightarrow wp(S, post)$, meaning that if executed in a state satisfying pre, S will terminate in a state satisfying $post$.

Dijkstra identified four conditions which should by held by the predicate transformer of any program fragment S.

1. **Law of the Excluded Miracle**. $wp(S, false) = false$. Since there are no states satisfying $false$, there can be no initial state from which execution of S will terminate and establish $false$.
2. **Monotonicity**. For any two postconditions Q, R such that $Q \Rightarrow R$, $wp(S, Q) \Rightarrow wp(S, R)$.
3. **$\wedge$-Distribution**. For any two postconditions Q, R we have $(wp(S, Q) \wedge wp(S, R)) = wp(S, Q \wedge R)$.
4. **$\vee$-Distribution**. For Q, R as before $(wp(S, Q) \vee wp(S, R)) \Rightarrow wp(S, Q \vee R)$.

The final law is not an equality like the third, since the implication does not necessarily hold in the opposite direction: there is a non-deterministic choice between Q and R.

The predicate transformers for the guarded command language are constructed so that they adhere to these rules: the primitives we mentioned above have their semantics characterised as:

$$\begin{aligned} \forall R. wp(skip, R) &= R \\ \forall R. wp(abort, R) &= false \\ \forall R. wp(x := E, R) &= R[x \backslash E] \end{aligned}$$

The notation $R[x \backslash E]$ means that all occurrences of x in R are replaced by E.

There is no S such that $wp(S, R) = true$ for all postconditions R, as that would violate the first condition (Law of the Excluded Miracle) above.

The composition operators (;, **if** ... **fi**, **do** ... **od**) also have their semantics characterised in terms of weakest preconditions. For example:

$$\forall R. wp(S_1;\ S_2, R) = wp(S_1, wp(S_2, R))$$

By structural induction each of the operators can be shown to have the four properties above; this includes showing the monotonicity of the predicate transformers, just as we discussed in Section 3.4.3.

4.3 Specifications and Programs

Morgan adds to the guarded command language the specification statement [Mor88c], which enables executable code to be derived piecewise from an initial abstract specification without the need for a translation step between languages.

Definition 4.2 (Specification Statement) *A specification statement, written*

$$\mathbf{w}\!:\left[\mathbf{pre}\,,\,\mathbf{post}\right]$$

denotes an abstract program which, begun in a state satisfying the predicate **pre** *is guaranteed to terminate in a state satisfying the predicate* **post***, changing at most the values of variables in the frame,* **w***.*

The addition of the specification statement blurs the distinction between specifications and programs; a program can contain specification statements and so may not be executable. A program which contains only code is a "concrete" program, while one which has no code is an "abstract" program (which we will continue to call a specification).

The specification statement, having been added to the language, must also have its semantics defined:

$$\forall R.wp(w\!:\left[pre\,,\,post\right],R) \mathrel{\widehat{=}} pre \wedge (\forall w.post \Rightarrow R)$$

Both specifications and programs are characterised as predicate transformers in this calculus; in this way they are given a common semantics, as we discussed in Section 3.2.1.

Refinement in this calculus is then defined as:

Definition 4.3 (Refinement) *For programs P and Q, $P \sqsubseteq Q$ iff for all predicates R, $wp(P,R) \Rightarrow wp(Q,R)$*

A consequence of this definition is that Q may terminate for preconditions that P would not terminate for; and that it may be more deterministic. The properties we stipulated earlier will hold; reflexivity and transitivity come as a result of the properties of the implication.

As we noted in Section 3.4, correctness will be preserved as long as the definition of correctness permits termination for initial states outside the precondition of the specification.

4.4 Refinement Calculus Laws

Refinement proceeds in this calculus by a number of laws which have been proved to be sound; these include:

$$\texttt{If}\ \ pre \Rightarrow pre'\ \ \texttt{then}$$
$$w\!: \left[pre\ , post \right] \sqsubseteq w\!: \left[pre'\ , post \right] \qquad \text{weaken precondition}$$

$$\texttt{If}\ \ post' \Rightarrow post\ \ \texttt{then}$$
$$w\!: \left[pre\ , post \right] \sqsubseteq w\!: \left[pre\ , post' \right] \qquad \text{strengthen postcondition}$$

These two laws correspond to domain extension and range restriction in Sections 3.5.3 and 3.5.4. Other laws provide for algorithmic and data refinement.

4.4.1 Miracles

The extensions made by Morgan to Dijkstra's language are not limited to specifications; he also introduces local variables and allows *conjunction* of programs. Local variables preserve each of the properties outlined by Dijkstra, but conjunction, defined as the weakest program that refines all the programs being conjoined, loses the property of $\wedge$-distribution.

The specification statement also breaks the law of the Excluded Miracle, by allowing the specification $[true, false]$, which can establish *false* for any initial state. Such a specification is called a *miracle*. Thus the only property left of Dijkstra's original list is monotonicity.

Miracles cannot be implemented, of course; their use will lead to *infeasible* code. However, a comparison can be made with complex numbers; they are of use in a derivation, but cannot be a part of a solution in the real domain. Similarly miracles have their uses. A guarded command is actually a miracle; in code it should only be permitted as part of an alternation or repetition, but the form $B \rightarrow S$ is of use in data refinement.

4.4.2 Data Refinement

Data refinement in the refinement calculus uses an abstraction function as discussed in Section 3.5.5. In fact there are alternative approaches to data refinement and the abstraction can be represented as a predicate transformer, or as a predicate called a *coupling invariant*.

One method for data refinement is the *auxiliary variables* approach [Mor88a], in which concrete variables are added to the specification, related by the coupling invariant to the abstract ones, and then using refinement laws the superseded abstract variables are removed. With abstract variables a, concrete variables c and coupling invariant CI, specification S_A is data-refined to specification S_C iff for all postconditions R not containing c,

$$(\exists\, a \bullet CI \wedge wp(S_A, R)) \Rightarrow wp(S_C, (\exists\, a \bullet CI \wedge R))$$

We then write $S_A \preceq_{CI} S_B$.

We have shown that the refinement calculus deals with the concerns we have identified in Section 3.4. The next step is to consider situations to which normal refinement rules may not apply.

4.5 Co-refinement

Where two or more viewpoint specifications are to be amalgamated, as described in Section 3.2), the resulting amalgamated viewpoint will contain variables from each of the component viewpoints. If the variables are common to all viewpoints then the relationship between the viewpoints should be straightforward, at least in terms of the data; however there may be variables in the amalgamation which are present in only some of the constituent viewpoints.

We then run into difficulties if we wish to describe the relationship between each of the initial specifications and their amalgamation, as we need to consider the effect of these "extra" variables; we cannot simply ignore the extra variables as they may restrict the behaviour of the amalgamation, hence disqualifying it as a refinement.

Co-refinement, a weaker version of refinement, has been proposed by Ainsworth [Ain95, AW94] to deal with situations of this kind. A co-refinement holds between specifications A and B if there are some circumstances under which refinement appears to hold between them; i.e. there is some state in which the additional variables take values which allow a refinement relation to hold. The work on co-refinement described here is expressed in terms of Morgan's specification statements — we derive alternative forms in later chapters.

To simplify the definition of co-refinement the concept of an *implicit signature* of a specification is first introduced.

Definition 4.4 (Implicit signature) *[War93] The implicit signature of specification A, denoted* sig A*, is the set of variables used in the specification.*

Co-refinement is then defined (in terms of specification statements) as follows:

Definition 4.5 (Co-refinement) *Specification A is co-refined by specification B, written* $A \mathrel{\underline{\sqsubseteq}} B$*, iff*

$$\exists(\text{sig } B - \text{sig } A) \bullet A \sqsubseteq B$$

This is the general form of co-refinement, which due to the extra variables may impose constraints on the refinement. A special case of co-refinement is *non-restrictive* co-refinement, which occurs when a refinement relation holds for all values of the additional variables.

Co-refinement can be seen as a weakened form of data-refinement. The coupling invariant in this context is also referred to as an *eyepiece*: if we return to the analogy of a customer who has to be satisfied, we can say that he will be satisfied by an amalgamation if we use the eyepiece to show that the amalgamation satisfies his original viewpoint – the customer looks at the amalgamation "through" the eyepiece.

4.5.1 Links and Restrictions

In two viewpoints A and B, which are amalgamated to form C, it may be that different variables from A and B are mapped to the same variable in C. This is referred to as a *link*. In the converse case, constraints imposed on the extra variables mean that a refinement holds only for certain values of these variables: this is a *restriction*. The coupling invariant relates both of these concepts, and is the conjunction of a linking predicate and a restricting predicate.

4.5.2 Co-refinement Properties

Note that $A \sqsubseteq B \Rightarrow A \mathrel{\overset{\circ}{\sqsubseteq}} B$. Algorithmic refinement can be seen as a special case of co-refinement, in which the signatures are equal.

We have noted desirable properties of a refinement relation (Section 3.4). These do not all hold for co-refinement:

- Co-refinement is a pre-order. It is reflexive (any specification trivially refines, and hence co-refines, itself) and transitive (if $A \mathrel{\overset{\circ}{\sqsubseteq}} B$ and $B \mathrel{\overset{\circ}{\sqsubseteq}} C$, $A \mathrel{\overset{\circ}{\sqsubseteq}} C$ follows by definition), though this will involve identifying the links and restrictions on C by combining the coupling invariants at each step.
- Correctness is preserved by co-refinement in the same way as it is for data refinement.
- Horizontal composition of co-refinements (described in Section 3.4) will be more problematic, due to the potential loss of monotonicity in adding variables to a viewpoint which may interfere with other variables. *Augmented specifications*, introduced below, are intended to deal with this problem.

An *augmented specification* is defined as follows:

Definition 4.6 (Augmented Specification) *An augmented specification is a tuple* $(S, \mathcal{L}, \mathcal{R})$ *made up of a specification* S*, a predicate describing links* $\mathcal{L}$*, and a predicate describing restrictions* $\mathcal{R}$*.*

The augmented specification is a specification which keeps a record of its coupling invariant (links and restrictions). Initially such a specification will have the predicate *true* for its links and restrictions; subsequent amalgamations will lead to a gathering of links and restrictions, so that at each amalgamation step a new link predicate and restriction predicate are derived and added to the augmented specification by logical conjunction.

Augmented co-refinement is then defined using augmented specifications:

Definition 4.7 (Augmented Co-refinement) $(A, \mathcal{L}_A, \mathcal{R}_A) \mathrel{\tilde{\overset{\circ}{\sqsubseteq}}} (B, \mathcal{L}_B, \mathcal{R}_B)$ *iff*

$$
\begin{array}{ll}
 & (A \preceq_{(\mathcal{L}_{AB} \wedge \mathcal{R}_{AB})} B) \\
\wedge & (\mathcal{L}_B = \mathcal{L}_A \wedge \mathcal{L}_{AB}) \\
\wedge & (\mathcal{R}_B = \mathcal{R}_A \wedge \mathcal{R}_{AB})
\end{array}
$$

where $\mathcal{L}_{AB}$ and $\mathcal{R}_{AB}$ are the links and restrictions, respectively, on this particular co-refinement between A and B.

The intention of the augmented specification is to ensure that the amalgamation is consistent with the rest of the specification.

4.6 Compromising Correctness

The refinement relations identified in Chapter 3 can cope with straightforward developments; however there are inevitably stages in any development when a development may not be correctness-preserving. We have already identified situations where correctness is not preserved by a refinement, but in some cases the refined version is still of some use: Ainsworth's co-refinement addresses this situation.

Outside the realm of programs and specifications it is common to have to come to some compromise over the product. We would like to be able to characterise this kind of relation, in order to use the relation in cases where conflicts arise between viewpoints. Such a relation carries on where co-refinement leaves off: if the restrictions imposed by an amalgamation cannot be satisfied, we have a conflict, and the compromise is that action which must be taken to satisfy the restriction. In the realm of explanations on which we are concentrating in the present work, we may have a situation in which an initial description glosses over certain facts which must later be explained properly with a phrase like "we said **A** before, but in actual fact **B**" — and **B** turns out to be incompatible with **A**.

In such situations we would proceed by first identifying how the restriction might be satisfied — that is, which part of the original viewpoint must be altered. We would then create a new viewpoint from the old with the problematic feature removed. For example, suppose we have a viewpoint specification (expressed as a specification statement) for the real roots of a quadratic equation:

$$\mathbf{A} \mathrel{\widehat{=}} x, y\colon \left[\; b^2 \geq 4ac\,,\; \begin{array}{l} ax^2 + bx + c = 0 \\ ay^2 + by + c = 0 \\ (b^2 > 4ac) \Rightarrow x \neq y \end{array} \right]$$

If developers then decide that only one root can be provided, a new specification with all references to y removed would be produced,

$$\mathbf{B} := x\text{:}\ \left[b^2 \geq 4ac\ ,\ ax^2 + bx + c = 0 \right]$$

In normal development terms we have a backwards step in going from **A** to **B**; we will call this *backward refinement.* The step from A to B is a compromise because of the reduction in functionality provided.

In a larger example A might be composed of specifications $A_1, A_2 \ldots A_n$, and B composed of $B_1, B_2 \ldots B_n$. If we have $\forall\, i : 1 \leq i \leq n . A_i \sqsubseteq B_i$, then the composition rules give us $A \sqsubseteq B$. This simplification assumes A and B are both structured into ordered modules in the same manner.

A compromise identifies the case where there is at least one $j : 1 \leq j \leq n$ such that $A_j \not\sqsubseteq B_j$. A measurement of acceptability of the compromise can be made based on how many parts are lost and their relative importance. Clearly we are no longer talking about a refinement here; at best we have a refinement of some sub-specification of A, in other words a viewpoint formed from A which is refined by B. More generally this viewpoint will be co-refined by B. The following then identifies a compromise relation:-

Definition 4.8 (Compromise) *For specifications A and B composed of modules $A_i : 1 \leq i \leq n$ and $B_i : 1 \leq i \leq m$, B is a compromise of A (written $A \supseteq B$) if there is a non-empty set $V \subset \{1, \ldots, n\}$ such that the composition of modules $\bigcup A_j : j \in V$ is co-refined by B.*

In the case of our quadratic equation example, the viewpoint $\bigcup A_j$ is the specification $x\text{:}\ \left[b^2 \geq 4ac\ ,\ ax^2 + bx + c = 0 \right]$ which is co-refined by B (as it is equal to B).

This relation is reflexive, and symmetric but not transitive, and does not preserve correctness so (as we would expect) is a non-starter for formal development. It is not a pre-order, and neither is it an equivalence relation because of the absence of transitivity. However it can combine with refinement in the following ways:-

- $A \sqsubseteq B \wedge B \supseteq C \Rightarrow A \supseteq C$
- $A \supseteq B \wedge B \sqsubseteq C \Rightarrow A \supseteq C$

Thus a development in which there is one compromise is characterised as a compromise. The benefit of the compromise relation is in characterising a development in which compromise takes place as *not* preserving correctness, and in providing another generalisation of refinement (co-refinement is the special case where $V = \{1, \ldots, n\}$).

4.6.1 Backtracking

A related situation occurs in a development where three stages A, B, C are related ($A \sqsubseteq B \sqsubseteq C$), and it is then required to change B. This may be because B is a product released to the public and C is the next release under development, and a bug-fix has to be applied to B. The bug-fix release B' will be a compromise of B, as at the very least some part of B's functionality will still hold true in B'. But what is the relation between B' and C?

By symmetry we have $B' \supseteq B$, and by the relation to refinement identified above we have $B' \subseteq C$. The next step in the development would then be to merge the bug-fixes into C to produce a C' that is

- a compromise of C, $C \supseteq C'$
- a refinement of B', $B' \sqsubseteq C'$

The actual merging activity to produce C' would consist of amalgamating the changes between B and B' with C. In practice a configuration management system such as RCS will deal with this situation: in Section 6.4 we consider how viewpoints can model this case.

4.7 Summary

This chapter has provided an overview of a particular method for stepwise refinement of specifications, and shown how it can be adapted, and the corresponding refinement relation weakened, to deal with situations which arise in viewpoint amalgamation. Subsequent chapters will deal with situations in which an explanation is taking place, and show how this can be modelled as a series of viewpoint amalgamations in which a refinement process is taking place.

Chapter 5

ViewPoint Oriented Software Development

In this chapter we address the ViewPoints framework [NKF94], and related research [HN95, LFKN95] introduced in Chapter 2's survey, as an example of an approach that deals with relationships between viewpoints, and assess the usefulness of the refinement theory introduced in Chapters 3 and 4 to model these relationships.

We will endeavour to be consistent in the use of terminology here, using "ViewPoint" to refer to the specific object used in the framework, and reserving "viewpoint" for the more general use.

5.1 Introduction

To recap (Section 2.3.6), ViewPoints provide a loosely-coupled, locally managed framework of distributable objects which encapsulate partial knowledge of a system or domain. ViewPoint-Oriented Software Development (VOSD) is designed primarily for use in requirements engineering, to develop requirements elicited from multiple perspectives. It can be divided into two broad stages, *Method Design* and *Method Use*.

5.1.1 Method Design

Method design is the design or reuse of *templates*. A template is a special kind of ViewPoint in which only the first two of five "slots" are defined:

1. *Style* — the notation used for the ViewPoint's specification.

2. *Work Plan* — a description of the ViewPoint's development process, the actions that may be used in the development.

   Actions fall into categories of assembly actions, in-ViewPoint check actions (ensuring syntactic correctness of a ViewPoint), inter-ViewPoint check actions (to check consistency between ViewPoints), and trigger actions, to create a new instantiated ViewPoint from a template.

The *Method* is then defined as a collection of templates with relations between them, which must be satisfied for consistency. A collection of templates thus implements a specific development method.

5.1.2 Method Use

Method use is the instantiation of the templates as ViewPoints. The remaining slots in a ViewPoint are:

3. *Domain* — the ViewPoint's area of interest in the overall system.

4. *Specification* — of the Viewpoint domain in the *Style* notation, developed as per the *Work Plan*.

5. *Work Record* — history and current state of development of the ViewPoint specification (actions performed from the *Work Plan*), to enable requirements traceability.

ViewPoint relations are instantiated (in-ViewPoint and inter-ViewPoint) from the template relations. The check actions defined in a template are instantiated as rules: in-Viewpoint rules can be classed as syntactic checks, and inter-ViewPoint rules would typically describe equivalence relations between corresponding elements and can be used passively to confirm consistency between ViewPoints, or actively for interchange and transformation of data between ViewPoints. In this way a domain-specific ViewPoint is instantiated from a generic template.

A requirements specification will typically consist of a number of potentially overlapping ViewPoint specifications, which need not, in the course of the development, be consistent with each other; a central tenet of the ViewPoints approach is that inconsistency is not only inevitable in a multiple perspectives development, it is actively to be encouraged so as to elicit more information about the system; inconsistency encourages ViewPoint owners to communicate and negotiate to resolve their differences.

5.1.3 ViewPoint Development

The actions and process model included in the Work Plan of a ViewPoint may be used to drive the development of a distributed system. Development for each ViewPoint progresses by recourse to its inter-ViewPoint rules, once the in-ViewPoint rules have been satisfied to show that the ViewPoint is well-formed. The invocation of a rule, expressed in general terms as a relation $\mathcal{R}$ between source and destination ViewPoint VP_S, VP_D,

$$\forall\, VP_S\, \exists\, VP_D \bullet VP_S\ \mathcal{R}\ VP_D$$

results first in a check for the existence of such a VP_D. Trigger actions are used to create an appropriate instantiation of a template if necessary. The rule is then applied, and either succeeds or fails; failure leads to actions to deal with the inconsistency implied by the failure. The inconsistency is either resolved or left pending. This application of rules implies some transfer of information between source and destination ViewPoint; the rules are used not merely for static comparison but also to drive communication between ViewPoints.

The distributed nature of the framework means that, in general, consistency checks are not controlled by some central controlling ViewPoint, though a "global ViewPoint" can be defined by the Method Designer to ensure consistency across all ViewPoints if necessary. Instead the checks are driven by each ViewPoint, and inter-ViewPoint communication proceeds by asynchronous message-passing.

5.2 ViewPoint Integration and the Amalgamation Process

There seems to be no immediate parallel to our notion of *amalgamation* in this decentralised framework, as the whole point of the framework is to maintain ViewPoints as distributed objects. The closest notion is *integration*, the process of comparing ViewPoints for consistency; two ViewPoints are deemed integrated if they are consistent, but no "amalgamated" ViewPoint need be formed. In some cases only a partial integration will be possible, as not all rules may be satisfied.

Our model of viewpoint amalgamation does however include stages analogous to those in the process identified above; our first stage is that of Coalescence Planning, where the viewpoints are compared for clashes and commonalities, producing a Coalescence Plan as output. This corresponds to the identification of inter-ViewPoint rules to be invoked. The history listed in the Work Record is analogous to the Amalgamation

Trail (Section 3.2).

5.3 ViewPoints and Refinement

There are points in the process where an implicit refinement takes place: again we divide this into method design and method use stages.

- Method Design
  - Template development will feature incremental modifications, and refinements such as we have been discussing here will hold between versions of the templates.
- Method Use
  - Instantiation of ViewPoints is a refinement, as detail is added to a template.
  - In the subsequent development of ViewPoints a refinement will hold at each stage as in the method design stage.
  - Consistency checking, via inter-ViewPoint rules. The failure of a rule can be seen as identifying a restriction on the refinement relation that should exist between the ViewPoints; if the restriction were satisfied, the rule would not fail.

In addition, if an integrated ViewPoint *is* constructed, as the ViewPoint owners or Method designer may decide to do, we can represent the construction of an integrated ViewPoint by identifying a *co-refinement* (Section 4.5) between the integration and each constituent ViewPoint. The nature of the co-refinement depends on the impact of "foreign elements" in the integration (those elements external to a particular constituent ViewPoint). A *non-restrictive* co-refinement will result if the foreign elements do not have any impact on the original ViewPoint, and a *restrictive* co-refinement will apply if new rules are imposed on a ViewPoint by the integration. In the case of partial integration, there may be a restrictive co-refinement for some ViewPoints, and no refinement at all for those ViewPoints which remain inconsistent (this would be modelled as a restriction that could not be satisfied). This would not be a valid amalgamation in terms of our model; in the VOSD framework it implies further action must be taken to deal with the inconsistency.

5.3.1 Phone Example

For example, in [EN95] consistency checking in an evolving specification is illustrated; relations between two ViewPoints of a telephone are used to identify and resolve inconsistencies. The ViewPoints are described using state transition diagrams. Initial ViewPoints from the perspective of a caller (Ann) and a callee (Bob) have some states in common, such as "idle" and "connected". There are also inconsistencies, due largely to a different concept of "connected" in each ViewPoint; Ann has a transition "replace receiver" between her "connected" and "idle" states, whereas Bob does not. An inter-ViewPoint rule enforcing this condition is written:

$$
\begin{aligned}
&R_1 : \forall\, VP_D(STD, D_S) \\
&\quad VP_S.transition(X, Y) \wedge VP_D.state(X) \wedge state(Y) \Rightarrow VP_D.transition(X, Y)
\end{aligned} \tag{5.1}
$$

This rule says that for all destination ViewPoints containing state transition diagrams with the same domain as the source ViewPoint (in this case the domain is "Telephone"), if the source ViewPoint has a transition between two states, both of which appear in the destination ViewPoint, then the transition must also appear in the destination ViewPoint. In our terminology, there is a co-refinement between source and destination ViewPoint in which the links include the states and transitions common to both ViewPoints.

Applying rule 5.1 in this case gives rise to a predicate *missing*, which is recorded in the development history for the source and destination ViewPoint; it is then up to the developers to decide how (and if) to resolve it. This can be modelled as an (unsatisfied) restriction on the co-refinement; only if the predicate *missing* was negated would there be some circumstances in which the co-refinement could be satisfied.

The main difference in our use of links and restrictions is that in our method the retrieve relations are defined between the amalgamated viewpoint and the constituents, rather than between each ViewPoint as here. This is because of the decentralised approach of the VOSD framework; it is the constituent ViewPoints which are driving the consistency checks and "deciding" which other ViewPoints they should be consistent with.

Once the ViewPoints have been found to be consistent, an integration might be formed by union of the state transition diagrams; we can model this process using co-refinement by considering the retrieve relations identified above; using "Phone" for the integrated ViewPoint, assuming a transition *other_party_hangs_up* in the Phone ViewPoint, the retrieve relation for the Phone would include

$$(other\_party\_hangs\_up_P = callee\_replaces\_receiver_A) \wedge$$
$$(other\_party\_hangs\_up_P = caller\_replaces\_receiver_B)$$

We use subscripts A, B and P for the ViewPoints of Ann, Bob and the amalgamated Phone, respectively. This identifies a link, and there will be links for each state and transition that is brought into the integrated ViewPoint relating it to its source.

Applicability restrictions should not occur for ViewPoints that are consistent, but are likely to arise when consistency checks are being carried out. Restrictions on the correctness of an amalgamated ViewPoint may arise even with consistent ViewPoints; for example, we can identify a restriction between Ann's ViewPoint and the integrated Phone by observing that the correctness of the Phone with respect to Ann's ViewPoint depends on the Phone only being used to make calls, not to receive them. This leads us to identify a state of the Phone which does satisfy Ann's ViewPoint, namely one in which the Phone is restricted to outgoing calls only.

5.4 Reasoning with Inconsistent ViewPoints

We have stressed that the idea of the ViewPoints framework is to allow development of ViewPoints that may be inconsistent, in order to encourage better understanding of the requirements. Consistency checks are thus performed only at particular "checkpoint" stages in the development, or between certain ViewPoints which may be considered more tightly coupled than others. The ViewPoints framework uses classical logic to detect inconsistencies [FGH+93] and use them to motivate further action. The analogy presented is of a database in which some inconsistencies that arise are useful — such as in a tax-payer's records — because it shows the tax inspector that he needs to investigate.

Consistency is checked between the specifications of two ViewPoints by translating the specification knowledge of each into classical logic, and adding classical-logic-translated versions of the inter-ViewPoint rules. Comparison of the resulting logical formulae, commences using a logical theorem prover and the Closed World Assumption (CWA), a concept from database theory. The CWA allows, given the absence of a fact α from a list of facts and their consequences, the conclusion $\neg\, \alpha$. Any inconsistencies arising at this stage give rise to the invocation of meta-rules to cope with the inconsistency — though not, necessarily, to resolve it. Possible actions include ignoring, circumventing (ignoring temporarily) or removing the inconsistency.

Problems arise however in attempting to reason about ViewPoints which are inconsistent; use of classical logic breaks down if both α and $\neg\,\alpha$ hold in a logical system, as rules of inference mean that anything at all can then be trivially deduced. One alternative being developed [HN95] is based on *Quasi-classical (QC) logic* [BH95], in which trivial formulae cannot be derived; the logic is weaker than classical logic, and only in the "final step" permits disjunction introduction. In QC logic a query (classical formula) follows from a set of assumptions exactly when there is a derivation of a conjunctive normal form[1] of the query from the assumptions using QC deduction rules. These rules are a subset of those which hold in classical logic; for example, introduction and elimination of negation, conjunct elimination, resolution, distribution and De Morgan's laws all hold, but laws which can lead to trivial derivations in the presence of inconsistency do not.

Reasoning with inconsistent ViewPoints can be achieved via *labelled* QC logic; unique labels are attached to each item of information, and propagate to deduced consequences by combining the labels of the premises. In this way, when an inconsistency arises the sources of the inconsistency are easier to locate from the labels.

5.4.1 Banking System Example

The example given in [HN95] is of two ViewPoints in a banking system. They disagree over the association between a cashier and a terminal: one has a predicate

$$has\_exactly\_one(Cashier, Terminal)$$

and the other has

$$has\_exactly\_two(Cashier, Terminal)$$

Inter-ViewPoint rules would include the following:

$$\forall X, Y\ \ has\_exactly\_one(X, Y) \Leftrightarrow \neg\, has\_exactly\_two(X, Y)$$

which would lead to an inconsistency in this case; but non-trivial implications following from these relations include

[1] a conjunction whose conjuncts are a disjunction of one or more literals.

$$\forall X, Y \; has\_exactly\_one(X, Y) \Rightarrow has\_one\_or\_more(X, Y)$$
$$\forall X, Y \; has\_exactly\_two(X, Y) \Rightarrow has\_one\_or\_more(X, Y)$$

so a non-trivial consequence, $has\_one\_or\_more(Cashier, Terminal)$, follows from the inconsistent ViewPoints.

Looking at these predicates in refinement terms, we can see that

$$\begin{aligned} \forall X, Y \; has\_one\_or\_more(X, Y) &\sqsubseteq \forall (X, Y) has\_exactly\_one(X, Y) \\ \text{and } \forall X, Y \; has\_one\_or\_more(X, Y) &\sqsubseteq \forall (X, Y) has\_exactly\_two(X, Y) \end{aligned}$$

since anything satisfying $has\_exactly\_one$ will certainly satisfy $has\_one\_or\_more$, and similarly with $has\_exactly\_two$. This "backward" step corresponds to the ideas discussed in Section 4.6. Backtracking is likely to be inevitable in such a development in the course of "negotiation" between ViewPoints.

Another way to look at this is as an amalgamation of the two predicates $has\_exactly\_one$ and $has\_exactly\_two$ to form $has\_one\_or\_more$; however there is no refinement to be observed from the predicates to their amalgamation. We could argue for a vacuous co-refinement here: that $has\_exactly\_one \; \underline{\underline{\sqsubseteq}}$ $has\_one\_or\_more$ with restriction $has\_exactly\_one$. This is comparable to the following in terms of co-refinement of specification statements (introduced in section 4.2):

$$\begin{aligned} \mathbf{A} &\mathrel{\widehat{=}} x_1\text{:}\left[\, true \, , \; x_1 = 1 \,\right] \\ \mathbf{B} &\mathrel{\widehat{=}} x_2\text{:}\left[\, true \, , \; x_2 \geq 1 \,\right] \end{aligned}$$

We follow the general practice of adding numeric subscripts to those variables (x in this case) shared between the viewpoints being compared. We can identify a co-refinement between **A** and **B** with a link $x_2 = x_1$ and restriction $x_2 = 1$.

Thus we can identify a similarity in approach between the use of logical reasoning in the ViewPoints framework and results in general viewpoint amalgamation; the identification of the generalised predicate $has\_one\_or\_more$ is produced by a process similar to the MFD concept of coalescence, and identification of links and restrictions in the amalgamation give a formal verification of a co-refinement. Indeed, such a formal verification can be used to back up the results of a derivation in QC-logic.

5.5 Summary and Conclusions

Finkelstein et al [FKN$^+$92, NFK94] can be said to have the most structured idea of viewpoints, as discussed in Chapter 2. They have developed a versatile framework for viewpoints, and their work on relationships between ViewPoints deals with many of the issues raised in comparing multiple perspectives.

While the decentralised ethos of the ViewPoints approach appears to be at odds with our approach to viewpoint amalgamation, we have identified some parallels and some areas where formal ideas about refinement and co-refinement can be used to some advantage. This ties in with some of the further research being done in the ViewPoints framework to use a category-theoretical basis [FM95] to formalise the approach, representing rules and relations in terms of functors mapping between objects and morphisms of different formalisms.

Integration of ViewPoints, which is achieved by verification of consistency by application of inter-ViewPoint rules, can be modelled as a process of coalescence leading optionally to amalgamation — though ViewPoints are described as integrated when they are consistent and need not be "physically" amalgamated.

The advantage of using our ideas about amalgamation and refinement to model the process that goes on in the ViewPoints framework is that we are able to give some guidance about when ViewPoints can be said to be correct with respect to an initial ViewPoint. A ViewPoint is *correct* with respect to another if all of the inter-ViewPoint rules are satisfied for the ViewPoints. Developments are permitted which break the correctness, but eventually the resultant inconsistencies must be dealt with: only when the correctness rules again apply can we say that a ViewPoint refines another. In this way the framework provides enough expressiveness to allow developers to explore alternative strategies: our approach additionally specifies which strategies provide a valid refinement.

Chapter 6

Viewpoints and Explanations

This chapter can be seen as an incremental description of the way viewpoints and refinement can be used to model incremental descriptions (or, explain explanations). The chapters which follow treat examples on the theme of explanations in more detail: here we introduce some of the concepts involved in explanations and address some of the issues arising from the use of viewpoints and refinement to model them.

We begin with a look at how first-year undergraduates at a University might be taught to use an editor (Section 6.1); we then build on the tutorial idea by extending the description to user manuals (Section 6.2).

Documentation in general be improved by use of the paradigm of *literate programming* (Section 6.3), which itself advocates an incremental approach. This paradigm is particularly useful for providing a mechanism for specifying variants of a system, although a more general mechanism is provided by *revision control systems* (RCS) (Section 6.4).

RCS is, in turn, a useful tool for configuration management, although it does not provide a framework for resolution of conflict (Section 6.5).

Having given an overview of the chapter, we "refine" it below by fleshing out the details. Such a presentation of material is a common form of explanation, and can be called a refinement in a similar sense to the way a table of contents is refined by the rest of the document: an adding of detail and a reduction of undeveloped sections. Indeed, some simple document management systems provide a way of viewing a document by presenting first the table of contents, then performing an in-line expansion of selected sections.

6.1 Text Editor Tutorial

```
In the EMACS editor all text that you type will be entered into the document
that you are creating.  So that the computer can differentiate between text to
be entered and instructions to do something, two special keys are available.
One is the CONTROL key labelled 'CTRL' and the other is the ESCAPE key.

The CONTROL key is like a special SHIFT key in as much as it should be held
down while another key is hit.  The ESCAPE key on the other hand is typed
before the command character is typed.

The convention used here is that if the control key should be used with the
letter a, say, then it will be shown as CTRL-a , if an escape sequence is
required it is shown as ESC a.


STARTING THE EDITOR

    emacs name          This starts the editor.
                        If the file 'name' exists the first 22 lines
                          of the file will be displayed.
                        If the file does not exist then it is created
                          and the first 22 empty lines are displayed.

CURSOR CONTROL KEYS

        CTRL-f          move cursor right one character (Forward)

        CTRL-b          move cursor left one character (Back)

        CTRL-p          move cursor up one line  (Previous)

        CTRL-n          move cursor down one line  (Next)
```

Figure 6-1: Excerpt from emacs tutorial file

First year students at the University of Bath School of Mathematical Sciences tend to fall into one of two main streams: those with some computing experience and those with none. When it comes to teaching them how to use the University's computing facilities, they are placed into groups of mixed abilities and given a tutorial introduction to the machines. This includes how to log on, basic file concepts in the UNIX operating system, and how to edit files. What the students learn here should set them up to be able to use computers in later years with confidence, in different working environments to those provided by the University; thus the emphasis is on grasping the key concepts before getting involved with specifics. For this reason the editor taught for some years in these tutorials was `vi`, it being the standard editor for Unix systems. However, it can be quite obtuse to learn for beginners, and now the editor taught is `GNU emacs`[1] as it is very widely used and more friendly to the beginner.

6.1.1 The Editor GNU emacs

The students are each given a text file in their home directory which contains an emacs tutorial, and told to run emacs on the file by typing `emacs emacs.ex` in a shell window. An excerpt from this file is presented in figure 6-1.

The file takes a typical approach in tutorial exercises by presenting the information in stages; if we asked the student what they know about the editor at various stages as they progress through the text[2], we might progress from "Nothing" initially through "You use CTRL-f to move the cursor forwards and CTRL-D to delete" and on to the other cursor control keys. At each stage we can think of the domain of emacs-related knowledge in the student's mind being extended. To do this we should choose a language in which to represent the knowledge that has been developed; for example a semantic description or logical predicate. Such a naive representation is, of course, treading on the toes of research in human-computer interaction, knowledge representation and intelligent tutoring systems.

The viewpoint formed in whatever modelling language we choose will have some gaps in it: for example, what happens when the cursor moves over the end of a line etc, and what the meanings of certain keys are. Lastly the unfortunate student as yet has no idea how to leave the editor, nor of what would happen if he tried to.

The remainder of the tutorial document fills in these gaps, resolving the issues of what the `ESC` key is for, and how to leave the editor, in addition to introducing a number of other commands. We could now formulate a viewpoint of the emacs knowledge gained by extending the earlier description, in a piecewise fashion as each new bit of knowledge was gained. In this small case this should not present any problems as the information given does not contradict what was said earlier, and a straightforward refinement defined in terms of "knowledge being enriched" can be observed. This will not always be the case, as we shall see in the next section.

In fact the emacs editor has much more to it than these basic key commands suggest; it is described as an extensible, programmable self-documenting editor with a large user community creating and maintaining code. This results in there being more than

[1]Produced by the Free Software Foundation's GNU (Gnu's Not Unix) project. The acronym Emacs is said to mean anything from "Editing MACros" to "Eats Memory And Crashes Systems".

[2]Assuming that the student we are modelling is a truly model student.

one variety of emacs around, leading inevitably to problems of incompatibility between packages and installations. We return to this topic in section 6.4.

6.2 Manuals and Documentation

A user manual for a program or system will typically take the same path as that outlined in the tutorial: beginning with the broad picture, and getting to the finer details later on. However, in more complex cases there will be more involved than simply fleshing out details; facts glossed over for the sake of simplicity will need to be returned to and perhaps totally re-formulated. This backtracking is illustrated in figure 6-2, and can be explained using the idea of "compromise" identified in Section 4.6. The version numbers in the figure denote the progress being made; there is a refinement (defined here as simply being "better defined") between version 1 and version 2, and between versions 1.1 and 2.1, but a "backward refinement" between versions 2 and 1.1. The relation between version 1 and version 1.1 is not clear; in general they need not be related at all.

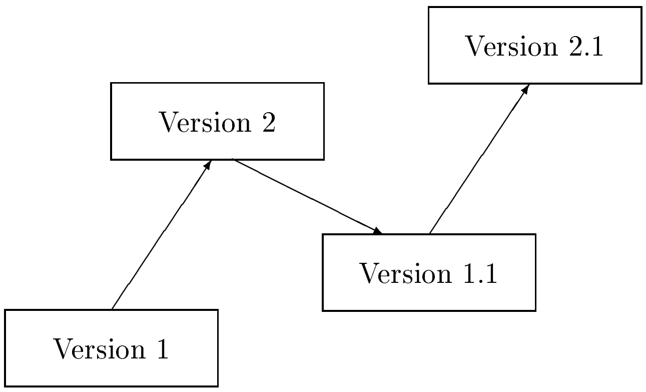

Figure 6-2: Backtracking development

6.2.1 Backward Steps

The picture can easily become more complicated, as the following short "explanation" which might have come from a simplistic manual illustrates.

We have four facts presented to the reader in the following order:

1. All cows are brown.

2. Cows like to eat grass.

3. When cows are lying down in a field, it is about to rain.

4. I lied in viewpoint 1: Some cows are black and white.

After choosing a viewpoint model, such as logical predicates, to represent the given facts, the process of amalgamating the given facts (which we can model as logical conjunction of predicates, perhaps using a language such as Prolog) results in the following accumulation of knowledge (translated back into natural language):

- All cows are brown (viewpoint 1).
- All cows are brown and like to eat grass ($1 \wedge 2$).
- All cows are brown and like to eat grass, and when they're lying down in a field, it is about to rain ($1 \wedge 2 \wedge 3$).
- Cows may be brown or black and white, they like to eat grass, and when they're lying down in a field, it is about to rain ($4 \wedge 2 \wedge 3$).

The refinement relation in the model of logical predicates would be based on implication: viewpoint $3 \Rightarrow$ viewpoint $2 \Rightarrow$ viewpoint 1.

The reader, in recovering from friesian shock, has had update his knowledge. In this case he was safe to amalgamate viewpoint 4 (some cows are black and white) with 2 and 3. However, if this bomb-shell had been dropped at a much later stage in this long road toward becoming an expert in dairy farming, there might be some viewpoint which depended upon the hue of the herd, such as "cows are difficult to spot against a brown background". So the trainee herdsman must sort out those facts which are true of all cows from those which are true only of the brown ones.

6.2.2 Coping with Revised Descriptions

We see that the reader may be left to sort out for himself the inconsistencies a backtracking in an explanation may leave, though a good manual should avoid this happening as much as possible. It is no coincidence that this development bears more than a passing resemblance to merges performed in revision control systems, to which subject we will soon turn our attention.

While the example given above is certainly trivial it is similar in structure to a description which might be given in a user manual, as opposed to a technical reference manual which will be structured in a different way.

A technical manual, documenting how/why a program was written rather than how to use it, is even more likely than a user manual to be difficult to understand, even inaccurate, and as a result, only get referred to in desperation should all else fail. What such a manual should be is well-structured, unambiguous and capable of explaining the design of the code. The paradigm of literate programming, which is gaining in popularity as a means of writing and documenting code, can improve this situation.

6.3 Literate Programming

Literate programming is a means of combining documentation and source code together, typically in a single file. Tools are then used to create program source or readable documentation from this file. The original literate programming tool was developed by Knuth [Knu92] to implement the TEX typesetting software, with the philosophy that "an experienced system programmer ... needs two things simultaneously: a language like TEX for formatting, and a language like `C` for programming ... when both are appropriately combined, we obtain a system that is much more useful than either language separately".

6.3.1 Knuth's WEB

Knuth's system was called `WEB`, underlining the idea that a program is made up of many interconnected pieces[3]. One program, `tangle`, produces C source from a `WEB` document, and another, `weave`, produces TEX source documenting the program. We can think of a `WEB` document as an amalgamation of a program viewpoint and a documentation viewpoint, but the `WEB` system provides much more. Rather than being an over-blown version of "verbose commenting", it enables programs to be *elaborated* in a flexible order; variables do not have to be defined before use, for example, since the `tangle` program will put everything in the right place, and partially defined subroutines and modules can be added to later in the `WEB` document. Thus the order of the document is determined by how the writer thinks the material can best be put across.

A further advantage of literate programming is that the document produced by the `weave` program includes an automatically generated index and table-of-contents, with

[3] Although the name itself is apparently in honour of Knuth's mother-in-law, Wilda Ernestine Bates

each code module cross-referenced. It can also contain anything a TeX document can contain — figures, mathematical formulæand so on.

Figure 6-3 shows an extracted module from an example literate program, in which `<< text >>=` introduces the definition of a program fragment labelled by `text`, `[[ name ]]` identifies `name` as being a variable, and sections are delimited by `@`. The use of these symbols enables the formatting programs to produce indexes of all occurrences of variables, and the section in which a variable is defined, and show the dependencies between sections.

```
@ This program has no input, because we want to keep it simple.
The result of the program will be to produce a list of the first
thousand prime numbers, and this list will appear on the [[output]]
file.

Since there is no input, we declare the value [[m = 1000]] as a
compile-time constant.
The program itself is capable of generating the first [[m]] prime
numbers for any positive [[m]], as long as the computer's finite
limitations are not exceeded.
<<program to print the first thousand prime numbers>>=
program print_primes(output);
  const m = 1000;
        <<other constants of the program>>
  var <<variables of the program>>
    begin <<print the first [[m]] prime numbers>>
    end.
@
```

Figure 6-3: Extract from example literate program written in noweb[Ram94].

One problem which becomes apparent with this style of writing is the need for typographical symbols to denote modules and so on; tool support for creation of literate programs is developing [BG92, BC90, GW91], which will enable the writer to concentrate on documenting the code rather than getting the symbols right.

The use of a single source for code and documentation means that code maintenance should also be easier; the cross-referencing and indexing makes code changes simpler, as the potential side-effects of changes are made explicit.

6.3.2 Literate Programming Applications

Literate programming has itself evolved from its `WEB` origins and many alternative tools have been developed [AO90, Ram94], taking the emphasis away from specific programming languages to allow the best language for the program to be used. Some tools are also able to produce the "woven" human-readable output in the form of hypertext (HTML) for viewing with a World-Wide-Web[4] browser.

6.3.3 Literate Programming and Refinement

It is not only programs that can benefit from the literate approach: written in this way — for example the development of the semantics for the calculator, and in particular the addition of an assignment operation (section 8.5) could be achieved via a `WEB` source document, with additions and alterations to the syntax and semantics presented as modules. Some research [Pep91, Sen92, Mor93, JLM$^+$94] into the marriage of literate programming with formal methods indicates that the paradigm may have much to offer.

A module of a document presented as a literate program is a viewpoint in our terminology. The composition operations provided allow modular structuring, and the `tangle` tool performs the necessary operations to produce an amalgamation — a single, flat file which is not intended to be read by human eyes.

Any viewpoint of the system should be refined by the amalgamation, as compilation of the amalgamation provides a correct implementation of the literate program. Furthermore, it is an intention of the literate programming paradigm to enable an understanding of some part of the actual program — the amalgamation — to be obtained by looking at the relevant part (or parts).

A further use of literate programming tools is to support "site versions" of a source program. Changes to a program can be incorporated into a "change file" which, with a file processing tool like the Unix `diff` command, can be used to automatically update any new version of the source itself. Once again this is simply a case of a viewpoint containing the changes being amalgamated with a viewpoint containing the source: however problems with refinement can arise due to the possibility of overriding parts of the source viewpoint. This is a part of the functionality of revision control systems, as we shall see in the following section.

[4]Different Web!

6.4 Revision Control Systems

A revision control system such as RCS [Tic85] maintains the current version of a program and provides a way to return to previous versions of a program using a series of backward "deltas" or context differences between a version and its predecessor. A new program (or document, or anything contained in a file) is first registered by its writer by checking it in; this results in the file being given a version number, normally `1.1`, and a new *revision group* file called `filename,v` is created, holding the contents of this version. The original file is normally deleted. The checking-in process also involves adding a textual description of the initial version.

Anyone wishing to update the file first checks it out; this results in the contents of the most recent revision in `filename,v` being placed into *filename*, and the status of the file is marked as *locked* to prevent anyone else from trying to update it. If people simply want to examine the file they can check it out without locking it; this simply means they will be unable to check it in again (a master copy of the latest version remains in the group file). An updated version of the file is checked in and a log entry given; the version number will then be updated, normally from `1.1` to `1.2`.

These version numbers are of the form (*release-number.level-number*). A major milestone in the development of the file will result in an increment of the release-number — this is done explicitly by the person doing the checking-in. Otherwise the default action is to increment the level-number.

6.4.1 Ordering between Versions

In a simple, straight-line development there can be said to be an ordering between versions; $\texttt{1.1} \leq \texttt{1.2} \leq \texttt{2.1} \leq \ldots$ The ordering is based simply on "being developed from"; there is no formal requirement that `2.1` be in any way better defined than `1.2`, so we are back to an informal idea of refinement based on "improvement" — `2.1` would be called an improvement of `1.2`, but the nature of the improvement could be anything from a bug-fix to a re-write of a procedure (or more). Correctness cannot be relied upon to be maintained.

Such a development is termed a "slender revision tree"; in reality branches will appear on the tree in situations where versions earlier than the current one need to be worked on; in the above, version `1.2` may be the most recent production version of a program, being used by customers, but version `2.1` is the development version. If a bug is reported in version `1.2`, a fix will be required; this version will have to be checked out and fixed, then checked in on an alternate branch (as calling it `1.3` would upset the ordering — it would not be true to say $\texttt{1.3} \leq \texttt{2.1}$. The branch revision would be numbered `1.2.1.1`. It would then be advisable for the most recent development version to benefit from the bug-fix; the changes in `1.2.1.1` should (if possible) be *merged* with `2.1` to produce `2.1.1.1`. Other situations, such as changes which are made only to a customer site version of a product and must then be made by the customer to the next release when it arrives, will result in similar branching.

6.4.2 Version Ordering and Refinement

The $\leq$ pre-order on version numbers is not a formal refinement ordering, since there is no guarantee of preserving of correctness between the versions themselves. If the product is a text document, for example, the revisions to the text itself are likely to involve addition, replacement and removal of segments of text, and a program being developed is likely to undergo similar alterations. The problem here is that the operation of *overriding* is not monotonic with respect to any meaningful ordering relation.

The ordering in the situation of variant branches becomes more complicated. We do not have, in the above discussion, $\texttt{1.2.1.1} \leq \texttt{2.1}$, for example, but we do have $\texttt{1.2.1.1} \leq \texttt{2.1.1.1}$. Plaice and Wadge[PW93] present a different approach to revision control which stresses the fact that it is the version labels, not the versions themselves, which have an ordering defined on them. Their approach aims to deal more effectively with the presence of variant branches. They define an algebra of versions, with a refinement order defined on them: in this approach numeric version labels are used only for the main branch, with sub-versions or variants labelled alpha-numerically. Refinement is defined as an ordering on version numbers, with an additional axiom for variants: $V \sqsubseteq V\%V'$. Versions can be combined with a *join* operator $+$; the authors present a complete partial order over the version space.

The advantage of this approach is that components which exist in a number of variant forms can be used to build a composite version by selecting the variant most relevant to the required system; the most relevant variant is the closest available in the partial order.

Since the version space is defined with a partial order and composition operations have a precise semantics, it seems that this approach is very much in tune with the refinement ideas presented in this thesis; this is an example of refinement being used to relate together different viewpoints which can be amalgamated in different ways. The join operator represents the amalgamation of two versions but makes no guarantees that the versions *can* be assembled in a meaningful way. In this way the version space idea presents one projection of the ideas in this thesis: that a refinement ordering can be used to relate versions of a system together.

6.4.3 Merging Versions and Amalgamating Viewpoints

We have drawn an analogy with the merging operation in RCS and the amalgamation of viewpoints. How a merger is actually performed is within the power of the human doing the merging; the process is partially automated by a three-way file comparison which compares two revisions with respect to a common ancestor. If these versions are called $r1$, $r2$ and anc respectively, then wherever anc and only one of the revisions agree on a segment of text it is the segment in the differing revision which survives into the merged version. If all three differ an error is flagged and human intervention is necessary.

The merging situation without human intervention can be modelled using compromise (Section 4.6). In this case we have $anc \supseteq r1$ and $anc \supseteq r2$, and the action of the three-way `diff` program is to break the versions into modules containing comparable text segments. Each text segment in the resultant version *merged* is a result of a comparison between each of the three versions, and we will have at least one of $r1 \supseteq merged$, $r2 \supseteq merged$. We cannot guarantee both of these holding, nor $anc \supseteq merged$. However, it should be hoped that in all but the most drastic revisions a compromise should hold between all versions.

6.5 Conflict between Versions

In the case above the compromise relation was applied to the merger case where at least two text segments agreed. It can happen in such configuration management systems that two developers will make changes to the same version of a component, independent of each other. In this case it is quite likely that the ancestor differs from both revisions, and the developers will then wish to merge their revisions. Tools to aid co-operative working (CSCW tools), some of which were discussed in the survey in Chapter 2, will support this process.

The resolution of conflicts will entail a process of amalgamation, guided by negotiation between the developers using a framework such as VOSD, discussed in Chapter 5. The telephone example cited in that chapter is an example of an evolving specification in which viewpoint owners make their own alterations and then have to deal with resulting conflict when the viewpoints are compared. In the worst case the resultant version which resolves the conflict may bear no resemblance to either, so even that weakest of relations, the compromise, will not hold. A more satisfactory resolution would be one that does succeed in reaching a compromise between both developers.

The difference between the two kinds of merger is in the kind of compromise achieved; in the successful 3-way merge the resultant *merged* version is composed of modules (viewpoints), each of which is found in at least one of $r1$ and $r2$. In the conflict-resolved merger each of the modules in the *merged* version is (at best) a *compromise* of the relevant module in at least one of $r1$ and $r2$.

6.6 Summary and Conclusions

This chapter has dealt with the idea that a number of processes through which things are explained can be modelled with viewpoints, amalgamation and refinement. This idea has been applied to an ordered sequence of subjects from tutorial explanations, through user manuals, literate programming, version control systems and on to conflict resolution in a cooperative working framework.

In the case of revision control systems we have seen that existing orderings between versions do not guarantee the correctness of the development, only placing versions in an order according to version number.

We have seen that straightforward refinement, and even the weaker co-refinement, cannot help in all situations where we can intuitively see an "enhancement" taking place; backtracking developments such as Figure 6-2 involve a form of "backward refinement" or compromise, which cannot preserve correctness. In the chapters which follow we treat two examples of explanations in more detail than the overview presented in this chapter, using viewpoints and refinement with some success to explain the explanations which are taking place.

Chapter 7

Incremental Development of an Algebraic Specification

In this chapter we present an example of an explanation in the form of an *incremental specification* taken from a journal paper, and apply our refinement ideas to model the relations between successive steps.

7.1 Introduction

Incremental specification is a common paradigm for presenting and developing a system; presenting information a piece at a time improves clarity and enables developer and reader to continue at a convenient pace without getting prematurely bogged down in details.

The example used for this chapter comes from [BCG+89], an incremental specification of part of the functionality of the Apple Macintosh Toolbox Event Manager. We begin by summarising the example as presented in [BCG+89], and then consider how a refinement relation should be defined for the style of specification used in the example. We then return to each stage in the development of the example to model the development between each incremental step as a refinement, in order to construct a correctness proof for the development.

7.2 Development of Toolbox Event Manager Specification

The event manager is provided for the benefit of applications developers, enabling them to select the next input event (key press, mouse click etc) from a queue of pending events and take the action appropriate to that event. It is the selection of the next element of the queue that is the concern of this example.

We begin the development with an algebraic specification for a simple, unbounded queue, by specifying a function *NextAndRest* which, given a queue, will return a tuple whose first element is the next element in the queue and whose second element is the remainder of the queue. The queue itself is represented as:

$$queue = empty \mid add(queue, element)$$

and our initial version of *NextAndRest* is as follows:

$$\boxed{\begin{aligned} & NextAndRest_1(add(q,e)) = \\ & \quad \textbf{if } isEmpty(q) \\ & \quad \textbf{then } \langle e, q \rangle \\ & \quad \textbf{else } \langle n, add(r,e) \rangle \textbf{ where } \langle n, r \rangle = NextAndRest_1(q). \end{aligned}} \tag{7.1}$$

Thus our initial requirement for the next element in the queue is that it should be the first (the one that has been there longest). The next element in a queue represented as $(add(add(add(empty, a), b), c), d)$ is thus a. Note that we have swept under the carpet such issues as the definitions of $empty, add, element$ as well as the fact that *NextAndRest* is only defined for non-empty queues; such are the liberties of partial specifications.

With equation 7.1 as our initial specification, we can begin the incremental process of fleshing out the specification. Motivation for each elaboration is a crucial part of the development, so we include direct textual quotes from [BCG+89] at each stage.

> ...the application programmer can pre-empt the queue order by selecting specific event types ...thus ignoring prior queued events of non-selected types. We therefore modify *NextAndRest* to take an extra parameter "*m*" to be thought of as a "mask", and we assume ...a function "*wanted*" which tests whether a given element is to be included according to a given mask.

The resulting new version of *NextAndRest* is first written as

$$
\begin{aligned}
&\mathit{NextAndRest}_2(add(q,e),m) = \\
&\quad \textbf{if } wanted(e,m) \textbf{ and } NoneWanted(q,m) \\
&\quad \textbf{then } \langle e,q\rangle \\
&\quad \textbf{else } \langle n, add(r,e)\rangle \textbf{ where } \langle n,r\rangle = \mathit{NextAndRest}_2(q,m) \\
&NoneWanted(empty,m) = \textbf{ true} \\
&NoneWanted(add(q,e),m) = \textbf{ not } wanted(e,m) \textbf{ and } NoneWanted(q,m)
\end{aligned}
\tag{7.2}
$$

Comparing the structure of this version with the previous one leads to a useful generalisation, via a new function *None*:

$$
\begin{aligned}
None(empty,t) &= \textbf{ true} \\
None(add(q,e),t) &= \textbf{ not } t(e) \textbf{ and } None(q,t)
\end{aligned}
\tag{7.3}
$$

New versions of both the previous specifications of *NextAndRest* can now be presented;

$$
\begin{aligned}
&\mathit{NextAndRest}_1(add(q,e)) = \\
&\quad \textbf{if } None(q, \lambda\, e'.\ \textbf{true}\ ) \\
&\quad \textbf{then } \langle e,q\rangle \\
&\quad \textbf{else } \langle n, add(r,e)\rangle \textbf{ where } \langle n,r\rangle = \mathit{NextAndRest}_1(q).
\end{aligned}
\tag{7.4}
$$

$$
\begin{aligned}
&\mathit{NextAndRest}_2(add(q,e),m) = \\
&\quad \textbf{if } wanted(e,m) \textbf{ and } None(q, \lambda\, e'.wanted(e'.m)) \\
&\quad \textbf{then } \langle e,q\rangle \\
&\quad \textbf{else } \langle n, add(r,e)\rangle \textbf{ where } \langle n,r\rangle = \mathit{NextAndRest}_2(q,m)
\end{aligned}
\tag{7.5}
$$

The full development is presented in tables 7.1 and 7.2, together with the motivating text from the original paper.

Our aim in the remainder of the chapter is to express each of these versions of the specification as the result of an amalgamation of a previous stage with an “incremental viewpoint”, one which encapsulates the information being added. Note that we do not insist on performing the amalgamation with the most recent stage; as the following development will illustrate, it can be more natural to combine increments with earlier, more abstract versions. We begin by considering the style of specification used in the example, and identifying a suitable refinement relation.

Text	Viewpoint
Initial Viewpoint	$\mathit{NextAndRest}_1(\mathit{add}(q,e)) =$ $\quad\textbf{if } \mathit{isEmpty}(q)$ $\quad\textbf{then } \langle e, q\rangle$ $\quad\textbf{else } \langle n, \mathit{add}(r,e)\rangle$ $\quad\textbf{where } \langle n, r\rangle = \mathit{NextAndRest}_1(q).$
The initial viewpoint is re-written to introduce a function *None*, generalising the test to be applied for selection of the next element.	$\mathit{NextAndRest}_1(\mathit{add}(q,e)) =$ $\quad\textbf{if } \mathit{None}(q, \lambda\, e'.\ \textbf{true})$ $\quad\textbf{then } \langle e, q\rangle$ $\quad\textbf{else } \langle n, \mathit{add}(r,e)\rangle$ $\quad\quad\textbf{where } \langle n, r\rangle = \mathit{NextAndRest}_1(q)$ $\mathit{None}(\mathit{empty}, t) = \textbf{ true}$ $\mathit{None}(\mathit{add}(q,e), t) =$ $\qquad\qquad\textbf{not } t(e) \textbf{ and } \mathit{None}(q,t)$
"...the application programmer can pre-empt the queue order by selecting specific event types ...We therefore modify *NextAndRest* to take an extra parameter "m" to be thought of as a "mask", and we assume ...a function "wanted" which tests whether a given element is to be included according to a given mask"	$\mathit{NextAndRest}_2(\mathit{add}(q,e), m) =$ $\quad\textbf{if } \mathit{wanted}(e,m)$ $\quad\quad\textbf{and } \mathit{None}(q, \lambda\, e'.\mathit{wanted}(e'.m))$ $\quad\textbf{then } \langle e, q\rangle$ $\quad\textbf{else } \langle n, \mathit{add}(r,e)\rangle$ $\quad\quad\textbf{where } \langle n, r\rangle = \mathit{NextAndRest}_2(q,m)$
"...The candidate queue elements for selection are now restricted ...further to those of maximal priority in the queue ...The criterion for selecting e from $\mathit{add}(q,e)$ is that there be no wanted elements in q which are not of lower priority than e"	$\mathit{NextAndRest}_3(\mathit{add}(q,e), m) =$ $\quad\textbf{if } \mathit{wanted}(e,m)$ $\quad\quad\textbf{and } \mathit{None}(q, \lambda\, e'.\mathit{wanted}(e'.m)$ $\qquad\quad\textbf{and not } \mathit{higher}(e,e'))$ $\quad\textbf{then } \langle e, q\rangle$ $\quad\textbf{else } \langle n, \mathit{add}(r,e)\rangle$ $\quad\quad\textbf{where } \langle n, r\rangle = \mathit{NextAndRest}_3(q,m)$

Table 7.1: Viewpoint presentation, extracted from [BCG+89]. Continued on page 84

Text	Viewpoint
Some events need to be treated as if they form part of a stack rather than a queue. The **not** *higher* function is generalised to a *preEmpts* function which acts on queue-type or stack-type events. “...We assume a predicate “sType” which indicates that an event is of “stack” (i.e. last-in-first-out) type ...We denote its complement by “qType” ...Equal-priority events are either all of queue type or all of stack type ...If $sType(e)$, the criterion is that there be no wanted elements of *higher* priority than e”.	$NextAndRest_4(add(q,e),m) =$ $\quad \textbf{if } wanted(e,m)$ $\qquad \textbf{and } None(q, \lambda\, e'.wanted(e'.m)$ $\qquad\qquad \textbf{and } preEmpts(e',e))$ $\quad \textbf{then } \langle e,q\rangle$ $\quad \textbf{else } \langle n, add(r,e)\rangle$ $\qquad \textbf{where } \langle n,r\rangle = NextAndRest_4(q,m)$ $preEmpts(e',e) =$ $\quad qType(e) \textbf{ and not } higher(e,e')$ $\quad \mid sType(e) \textbf{ and } higher(e',e)$
Some events are of *rType*, which means that related events must be discarded. “...adding a new activate event causes ones already in the Event Pool to be discarded, while the presence of a deactivate event in the Event Pool means that for the moment any incoming ones are discarded ...[we] remove the discarded (de)activate events at the time that a (de)activate event is selected from the Event Pool”.	$NextAndRest_5(q,m) =$ $\quad \langle n, \textbf{if } rType(n) \textbf{ then}$ $\qquad RemoveAll(q,n) \textbf{ else } r\rangle$ $\qquad \textbf{where } \langle n,r\rangle = NextAndRest_4(q,m)$ $RemoveAll(empty,e) = empty$ $RemoveAll(add(q,e'),e) =$ $\quad \textbf{if } related(e,e') \textbf{ then } r \textbf{ else } add(r,e')$ $\quad \textbf{where } r = RemoveAll(q,e)$

Table 7.2: Viewpoint presentation, continued from page 83.

7.3 Refinement of Algebraic Specifications

Formal specification techniques are generally classed as model-based or algebraic. Z and VDM are model-based, in the sense that a system is modelled in terms of understood mathematical concepts such as sets, sequences and relations, by defining states and operations in terms of how the state is affected. Algebraic (or property-based) specifications specify an object or type in terms of relations between operations on that type. Co-refinement [Ain95] has been developed in terms of Z and the refinement calculus. Algebraic specification provides us with an opportunity to consider refinement in different surroundings. A full algebraic specification in a language such as CLEAR [BG86] or Larch [GHW85] will include a *sort* (type) definition, declaration of imported specifications, signatures of operations, and equational axioms defining the operations themselves. By contrast, the style of algebraic specification we are using is concerned only with the axioms; sorts and signatures are omitted, but assumed to be defined. Basic operations are also assumed to be defined and imported from a library of such operations. What we are left with, as will become clear in the following section, is a collection of equational axioms acting as a function definition; a specification written in this style is thus a partial specification, and this fits our idea of viewpoints.

7.3.1 Refinement Relation for Algebraic Specification

Any refinement relation must be correctness-preserving, as we have seen in Chapter 3. We will define correctness of an implementation of an algebraic specification as follows: if the implementation terminates at least for those initial conditions for which the algebraic specification is defined, and establishes final conditions satisfied by the specification, it is correct with respect to the specification. This definition means that the implementation can be more deterministic than the specification, which as we have noted might not be acceptable in some circumstances - but it allows an implementation of *NextAndRest* to be defined for the case where the queue is empty, which our specifications do not.

A refinement relation for this style of "functional" algebraic specification can be expressed as a relation between functions. We use the conservative extension introduced in Section 3.5.3:

Definition 7.1 (Refinement) *For functions f and g, $f \sqsubseteq g$ iff*

$$\operatorname{dom} f \subseteq \operatorname{dom} g$$

and

$$\forall x \in \operatorname{dom} f : g(x) = f(x)$$

So g is applicable wherever f is, and g will be at least as correct as f; thus a function defined on positive integers is refined by one defined over the natural numbers, as long as it is indistinguishable from the first when limited to positive integer input.

This relation is a pre-order and preserves correctness, so it has the necessary properties to allow stepwise development as outlined in Chapter 3.

7.4 Extension and Enrichment

There is a clear association between the refinement theory we have been discussing and the theory of abstract data types [Ehr82]. In this theory a specification is a triple $D = \langle S, \Omega, E \rangle$ where S is a set of sorts (types), Ω a set of operation signatures over S and E a S-sorted set of Ω equational axioms.

Classically an *extension* describes a function f between specifications D_0 and D_1 such that D_0 is a sub-specification of D_1; this means that $S_0 \subseteq S_1$, $\Omega_0 \subseteq \Omega_1$, and $E_0 \subseteq E_1$. A *conservative* extension describes the case where f preserves correctness.

An *enrichment* classically describes an extension where there is no addition to the types of the specification, only to the operation signatures and equational axioms.

However in the literature (for example, [ST95]) "enrichment" is applied as a general term for the addition of sorts, operations and axioms to a specification. This is the sense in which we use it, in for example Section 3.6. Successive enrichments are in effect what is taking place in this example; however since our viewpoints identify only equational axioms, rather than sorts and operations, the refinement relation is axiomatic.

7.4.1 Composition Operations

Operations for composing specifications are for our purposes limited to function composition ($f \circ g$), application ($f(g)$) and λ abstraction.

$f \circ g$ denotes a function whose domain is the domain of g and whose range is the range of f; composition is only possible where $\operatorname{dom} f = \operatorname{ran} g$.

$(f \circ g)(x) = f(g(x))$.

$f(g)$ denotes the application of higher-order function f to function g. Thus the domain of f is a function space (f is also called a *functional*):

$\operatorname{dom} f = (\alpha \to \beta)$

For example, if *double* : $x \mapsto 2x$ is a function defined on integers and *map* is a function with signature $(\alpha \to \beta) \to (\alpha \ list \to \beta \ list)$, then $map(double)$ will be a function which takes a list of integers and returns the list with each element doubled.

We need to be convinced that the composition operations discussed above preserve monotonicity, as discussed in Section 3.4. Observe that for refinement with function composition, for functions f, g, h:

$$
\begin{array}{lcl}
f \sqsubseteq g \Rightarrow & & \\
 & \operatorname{dom} f \subseteq \operatorname{dom} g & \text{(refinement definition)} \\
\Rightarrow & \operatorname{dom}(h \circ f) \subseteq \operatorname{dom}(h \circ g) & (\operatorname{dom}(h \circ f) = \operatorname{dom} f \text{ for any } h, f) \\
 & \forall\, x \in \operatorname{dom} f : g(x) = f(x) & \text{(by definition)} \\
\Rightarrow & \forall\, f(x) \in \operatorname{dom} h : h(g(x)) = h(f(x)) & (h \text{ is a function}) \\
\Rightarrow & h \circ f \sqsubseteq h \circ g & \text{(refinement definition)}
\end{array}
$$

A similar derivation supports the monotonicity of function application;

$$f \sqsubseteq g \Rightarrow h(f) \sqsubseteq h(g) \tag{7.6}$$

7.4.2 Data Refinement

Data refinement is discussed in general terms in Section 3.5.5. By analogy with the notion of an abstraction function between abstract and concrete states, we define data refinement for algebraic specification by defining a mapping function between new and old domains and ranges; thus for a data refinement from a function defined on integers to one defined on reals, an abstraction function would be one casting reals back to integers. We can adapt the refinement definition, Definition 7.1 to give the following for data refinement:

Definition 7.2 (Data refinement) *For functions f and g, $f \preceq_{abs} g$ iff*

$$\mathrm{dom}(f \circ abs) \subseteq \mathrm{dom}\, g$$

and

$$\forall\, x \in \mathrm{dom}\, g : f(abs(x)) = abs(g(x))$$

We use $f \preceq_{abs} g$ to denote data refinement through the abstraction function abs. In fact the abstraction function has two distinct parts: one for the domain and one for the range (codomain). The domain abstraction recovers the domain of the initial function, and the range abstraction the range of the initial function. In the above abs has been used for both, as will be the case when the domain and the range are equal.

7.4.3 Co-Refinement

We can expect to come across situations, as discussed in Chapter 3, in which straightforward data-refinement is too strong to describe what is going on. We can adapt the co-refinement definition from Section 4.5 for our specification style as:

Definition 7.3 (Co-refinement)

$$\begin{aligned} & f \ \underline{\sqsubseteq\!\!\!\!\circ}\ _{l,r}\ g \ \textit{iff} \\ & \qquad \mathrm{dom}(f \circ l) \subseteq \mathrm{dom}\, g \\ & \wedge \quad \forall\, x \in \mathrm{dom}\, g : r \Rightarrow f(l(x)) = l(g(x)) \end{aligned}$$

Co-refinement is defined in terms of data refinement; an alternative notation for the above (c.f. Section 7.4.2) would be

$$r \Rightarrow f \preceq_l g$$

l is a function from $(\mathrm{dom}\, g)$ to $(\mathrm{dom} f)$ (and from $(\mathrm{ran}\, g)$ to $(\mathrm{ran} f)$ - we again make the simplifying assumption that domain and range are equal). l is the *link* for this co-refinement (see Section 4.5.1), and is a surjective function. r is the restriction, which we represent as a predicate specifying the conditions under which a refinement relation holds.

In the case where domain and range are different the definition can be simply extended to give a domain link and a range link. In the remainder of the example we will only be talking about links on the domains of the functions, so the implicit range link is the identity mapping.

7.4.4 Augmented Co-Refinement

We use a similar definition of augmented co-refinement to that developed in Chapter 3, to aid in the composition of specifications.

For "augmented" functions $(f, \mathcal{L}_f, \mathcal{R}_f)$ and $(g, \mathcal{L}_g, \mathcal{R}_g)$ (see definition 4.6), augmented co-refinement from f to g holds with link $\mathcal{L}_{fg}$ and restriction $\mathcal{R}_{fg}$ iff:

$$
\begin{array}{rc}
 & f \preceq_{(\mathcal{L}_{fg} \wedge \mathcal{R}_{fg})} g \\
\wedge & \mathcal{L}_g = \mathcal{L}_{fg} \circ \mathcal{L}_f \\
\wedge & \mathcal{R}_g = \mathcal{R}_f \wedge \mathcal{R}_{fg}
\end{array}
$$

7.5 Viewpoint Development

Each stage in the development can be seen in terms of viewpoint amalgamation. If we consider the first version of *NextAndRest* (equation 7.1) as our initial viewpoint, each of the following stages can then be represented as the result of an amalgamation of a previous stage with an "incremental viewpoint"; one which encapsulates the information to be added. This corresponds with the way in which a system might be informally described in terms of "what the hearer already knows" plus "additional information".

As each viewpoint will be some development of a previous viewpoint, there will be some relation between it and the original viewpoint. We will investigate this relation – our method is to encapsulate each change in such a way as to indicate what information is being added, and identify how this change is to be combined with previous versions to produce a new version. In Section 7.6 we look again at the viewpoints produced at each stage in the development and consider the relation between them.

We first look at the initial adjustment to the specification.

7.5.1 $NextAndRest_1$

The change from equation 7.1 to equation 7.4 is fairly minor, involving the re-writing of *isEmpty*(q) as $None(q, \lambda\, e.\, \mathbf{true})$. This is not much of an increment in that no information is added; we are simply presenting an alternative version of the specification. However, we do need to justify the replacement. We do this by forming an abstraction of the function $NextAndRest_1$, which can then be instantiated to form a new function equivalent to the old one. We present this method in detail at this stage, as it will be used in later stages.

The function $NextAndRest_1$ can be generalised as follows;

$$
\begin{aligned}
& AbsTest(add(q,e))\ \ test = \\
& \quad \textbf{if}\ test(q,e) \\
& \quad \textbf{then}\ \ \langle e, q\rangle \\
& \quad \textbf{else}\ \ \langle n, add(r,e)\rangle\ \textbf{where}\ \langle n, r\rangle = AbsTest\ \ q\ \ test
\end{aligned}
\tag{7.7}
$$

The effect of the abstraction is to define a function which will select, as the next element, the one that is nearest the end of the queue which satisfies the function *test*. $NextAndRest_1(q)$ is then expressible simply as an instantiation, $NextAndRest_1(q) = AbsTest\ \ q\ \ \lambda(q,e).isEmpty(q)$ — in this case, the test ignores the current element and succeeds only if the rest of the queue is empty.

Since we have (from the definition of *None*, equation 7.3):

$$
isEmpty = \lambda\, q.None(q, \lambda\, e.\ \textbf{true}\ ) \tag{7.8}
$$

we can conclude from the monotonicity of function application (equation 7.6)

$$
\begin{aligned}
AbsTest\ \ q\ \ \lambda(q,e).isEmpty(q) &= AbsTest\ \ q\ \ \lambda(q,e).(\lambda\, q.None(q,\lambda\, e'.\ \textbf{true}\ ))(q) \\
&= AbsTest\ \ q\ \ \lambda(q,e).None(q,\lambda\, e'.\ \textbf{true}\ ) \qquad (7.9)
\end{aligned}
$$

We have renamed the variable e in $\lambda\, e.\ \textbf{true}$ to e', to show that this is different from the other e in the equation.

Starting at the other end, the new version of our function (equation 7.4) can be parameterised to a form equal to equation 7.9, and we can conclude that the versions are equivalent (and hence trivially refine each other). Equation 7.8 is kept as a record of the development.

In the next, and remaining, stages of the development we will begin with a table summarising the activity at this stage, in terms of the current viewpoint, incremental viewpoint and amalgamated "product" viewpoint.

7.5.2 $NextAndRest_2$: Mask

Current Viewpoint	$NextAndRest_1$
Incremental Viewpoint	$Mask$
Product Viewpoint	$NextAndRest_2 = NextAndRest_1 \circ Mask$

The information to be added here is the mask, used to select only those events which are wanted. The description given in [BCG+89] motivates us to think of the addition of the mask by specifying the new viewpoint in terms of the old one; if all the events fit the mask, we would have a viewpoint equivalent to the previous version. Thus if we have a function which acts as a filter, throwing out anything not fitting the mask, we can then plug the filtered version of the queue into the original function.

We make use a higher-order function *filter*, defined as follows;

$$filter\ fn\ empty = empty \tag{7.10}$$

$$\begin{aligned} & filter\ fn\ (add(q, e)) = \\ & \quad \textbf{if}\ (fn\ \ e)\ \textbf{then}\ add((filter\ fn\ \ q), e) \\ & \quad \textbf{else}\ (filter\ fn\ \ q) \end{aligned} \tag{7.11}$$

filter applies the function *fn* to each element of the queue; if the function returns nil the element is removed from the queue.

With this we can build

$$\begin{aligned} & Mask(q, m) = \\ & \quad filter\ \ \lambda\, e.wanted(e, m)\ \ q \end{aligned} \tag{7.12}$$

With this function we can now specify the new version of *NextAndRest* as follows;

$$\boxed{NextAndRest_2(q, m) = NextAndRest_1(Mask(q, m)) \qquad (7.13)}$$

What we have done here is to represent the new version as an amalgamation of the initial version ($NextAndRest_1$) and the increment ($Mask$).

This way of specifying the new version has the advantage over the old version of encapsulating the added information succinctly, in a manner closer to the way the increment was textually described. We can also clearly see what relation this version has to the previous one; it will behave in the same way as $\mathit{NextAndRest}_1$ if all events fit the mask, i.e. $\forall\, e.\mathit{wanted}(e, m)$.

Note that the two versions ($\mathit{NextAndRest}_1$ and $\mathit{NextAndRest}_2$) are not equivalent. This is because the functions are defined on different domains, due to the introduction of the mask. What we have is a (restricted) co-refinement (discussed in Section 7.4.3). The restriction is $(\forall\, e.\mathit{wanted}(e, m))$, as identified above.

We can also identify a link in this co-refinement. It must map tuples of *queue* and *mask* back to queue: the function $\lambda(q, m).q$ will achieve this.

Using QUEUE and MASK to denote the domains of variables q and m respectively, we have

$$\begin{aligned} \operatorname{dom} \mathit{NextAndRest}_1 \circ \lambda(q, m).q &= (\mathit{QUEUE} \times \mathit{MASK}) \\ &= \operatorname{dom} \mathit{NextAndRest}_2 \end{aligned}$$

which satisfies the first part of our co-refinement definition (Definition 7.3), and

$$\begin{aligned} &(\forall\, e.\mathit{wanted}(e, m) \Rightarrow \\ &\quad \mathit{NextAndRest}_1(\lambda(q, m).q(q, m)) = (\mathit{NextAndRest}_2(q, m))) \\ &\quad \forall (q, m) \in \operatorname{dom} \mathit{NextAndRest}_2 \end{aligned}$$

which satisfies the second part, so we have

$\mathit{NextAndRest}_1 \mathrel{\underline{\circledcirc}}_{(\lambda(q,m).q, \forall\, e.\mathit{wanted}(e,m))} \mathit{NextAndRest}_2$.

7.5.3 $\mathit{NextAndRest}_3$: Priorities

Current Viewpoint	$\mathit{NextAndRest}_2 = \mathit{NextAndRest}_1 \circ \mathit{Mask}$
Incremental Viewpoint	$\mathit{Priority}$
Product Viewpoint	$\mathit{NextAndRest}_3 = \mathit{Priority} \circ \mathit{Mask}$

The remaining versions of *NextAndRest* take advantage of the common format introduced in the initial development. We again use the abstraction function from Section 7.5.1:

$$
\begin{aligned}
&\mathit{AbsTest}(\mathit{add}(q,e))\ \mathit{test} = \\
&\qquad \textbf{if}\ \mathit{test}(q,e) \\
&\qquad \textbf{then}\ \ \langle e,q\rangle \\
&\qquad \textbf{else}\ \ \langle n,\mathit{add}(r,e)\rangle\ \textbf{where}\ \langle n,r\rangle = \mathit{AbsTest}\ q\ \mathit{test}
\end{aligned} \tag{7.14}
$$

Supplying new values for *test* will produce new versions as required; the way we progress is different in emphasis to that presented in the original development, and owes something to the approach of [PW93], referred to in Chapter 6[1]. We create "Priority" and "Stack" versions, and "Mask" is a variant or sub-version which can be combined with the main versions as we choose. We present the increment for a priority queue as

$$
\begin{aligned}
&\mathit{Priority}(q) = \\
&\qquad \mathit{AbsTest}\ q\ \ \lambda(q,e).\mathit{None}(q,\lambda\, e'.\ \textbf{not}\ \mathit{higher}(e,e'))
\end{aligned} \tag{7.15}
$$

We can now combine this increment with the mask increment to give $\mathit{NextAndRest}_3$ as described:

$$
\boxed{\mathit{NextAndRest}_3(q,m) = \mathit{Priority}(\mathit{Mask}(q,m))} \tag{7.16}
$$

Just as we had a co-refinement between $\mathit{NextAndRest}_1$ and $\mathit{NextAndRest}_2$, we have a co-refinement between *Priority* and $\mathit{NextAndRest}_3$ with link $\lambda(q,m).q$ and restriction $\forall\, e.\mathit{wanted}(e,m)$. The proof is the same:

$$
\begin{aligned}
\operatorname{dom} \mathit{Priority} \circ \lambda(q,m).q &= (\mathit{QUEUE} \times \mathit{MASK}) \\
&= \operatorname{dom} \mathit{NextAndRest}_3
\end{aligned}
$$

and

$$
\begin{aligned}
&(\forall\, e.\mathit{wanted}(e,m) \Rightarrow \\
&\qquad \mathit{Priority}(\lambda(q,m).q(q,m)) = (\mathit{NextAndRest}_3(q,m))) \\
&\qquad \forall (q,m) \in \operatorname{dom} \mathit{NextAndRest}_3
\end{aligned}
$$

satisfying the co-refinement definition, and so

[1] In their notation we might use *NextAndRest1%Mask%Priority* to denote the Priority variant of the Mask variant of version 1 of NextAndRest

$$Priority \sqsubseteq_{(\lambda(q,m).q,\forall\, e.wanted(e,m))} NextAndRest_3.$$

We are more interested in the relation between $NextAndRest_2$ and $NextAndRest_3$. They are defined as

$$\begin{array}{rcl} NextAndRest_2 & = & NextAndRest_1 \circ Mask \\ NextAndRest_3 & = & Priority \circ Mask \end{array}$$

so we can break the problem down by examining the relation between $NextAndRest_1$ and $Priority$.

Now, since

$$\begin{array}{rcl} NextAndRest_1 & = & AbsTestq\, \lambda(q,e).None(q, \lambda\, e'.\ \mathbf{true}\ ) \\ Priority & = & AbsTestq\, \lambda(q,e).None(q, \lambda\, e'.\neg\ higher(e,e')) \end{array}$$

the two functions will be equivalent when the next element is guaranteed to have the highest priority, i.e.

$$\forall(q,e)(None(q, \lambda\, e'.\ \mathbf{true}\ ) = None(q, \lambda\, e'.\neg\ higher(e,e')))$$

One way to achieve this is to place a restriction on the refinement so that all elements are of equal priority:

$$\begin{array}{l} \forall(e,e').\neg\ higher(e,e') \Rightarrow \\ \qquad (\forall(q,e).None(q, \lambda\, e'.\ \mathbf{true}\ ) = None(q, \lambda\, e'.\neg\ higher(e,e'))) \end{array}$$

This restriction is arguably a little strong — all we required was that the "Next" element always be of highest priority. Another way to achieve this would be to sort the queue by priority before taking the next element, but this would be computationally expensive and would result in the "Rest" of the queue returned by the function also being sorted — imposing a strong restriction on the correctness of the function.

With the "equal priority" restriction we now have a co-refinement,

$$NextAndRest_1 \trianglerighteq_{(id, \forall (e,e').\neg\, higher(e,e'))} Priority$$

as we can see by examining the domains:

$$\begin{aligned} \operatorname{dom} NextAndRest_1 \circ id &= QUEUE \\ &= \operatorname{dom} Priority \end{aligned}$$

This trivially satisfies the first co-refinement rule and

$$\begin{aligned} (\forall (e,e').\neg\, higher(e,e') \Rightarrow NextAndRest_1(q) &= Priority(q)) \\ &\qquad \forall\, q \in \operatorname{dom} Priority \end{aligned}$$

satisfies the second.

We are permitted to compose functions and maintain co-refinement, as long as we keep the restriction: thus

$$\begin{aligned} NextAndRest_1 &\trianglerighteq_{(id, \forall (e,e').\neg\, higher(e,e'))} && Priority \\ \Rightarrow (NextAndRest_1 \circ Mask) &\trianglerighteq_{(id, \forall (e,e').\neg\, higher(e,e'))} && (Priority \circ Mask) \\ \Rightarrow NextAndRest_2 &\trianglerighteq_{(id, \forall (e,e').\neg\, higher(e,e'))} && NextAndRest_3 \end{aligned}$$

We consider other relations between $NextAndRest_1$, $NextAndRest_2$ and $NextAndRest_3$ in Section 7.6.

7.5.4 $NextAndRest_4$: Queue/Stack

Current Viewpoint	$NextAndRest_3 = Priority \circ Mask$
Incremental Viewpoint	$QueueStack$
Product Viewpoint	$NextAndRest_4 = QueueStack \circ Mask$

Whereas auxiliary functions like *wanted* and *higher* were left unspecified in previous versions, here we give a definition for the function *preEmpts*, which uses some (undefined) predicates *sType* and *qType*. This we add to our viewpoint:

$$\begin{aligned}
&\mathit{preEmpts}(e', e) = \\
&\quad \mathit{qType}(e) \textbf{ and } \textbf{not } \mathit{higher}(e, e') \mid \mathit{sType}(e) \textbf{ and } \mathit{higher}(e', e)
\end{aligned}$$

The "Queue/Stack" increment can then be presented as:

$$\begin{aligned}
&\mathit{QueueStack}(q) = \\
&\quad \mathit{AbsTest}\ q\ \ \lambda(q, e).\mathit{None}(q, \lambda\, e'.\mathit{preEmpts}(e', e))
\end{aligned} \tag{7.17}$$

Note that this increment replaces the previous one (*Priority*); it uses *PreEmpts* where *Priority* used **not** *higher*. We can see, however, that where all elements satisfy *qType*, the two will be equivalent.

The advertised version of $\mathit{NextAndRest}_4$ is now described as:

$$\boxed{\mathit{NextAndRest}_4(q, m) = \mathit{QueueStack}(\mathit{Mask}(q, m))} \tag{7.18}$$

We again approach the relation between $\mathit{NextAndRest}_3$ and $\mathit{NextAndRest}_4$ by looking at their constituent functions:

$$\begin{aligned}
\mathit{NextAndRest}_3 &= \mathit{Priority} \circ \mathit{Mask} \\
\mathit{NextAndRest}_4 &= \mathit{QueueStack} \circ \mathit{Mask}
\end{aligned}$$

$$\begin{aligned}
\mathit{Priority} &= \mathit{AbsTestq}\, \lambda(q, e).\mathit{None}(q, \lambda\, e'.\neg\ \mathit{higher}(e, e')) \\
\mathit{QueueStack} &= \mathit{AbsTest}\ q\ \ \lambda(q, e).\mathit{None}(q, \lambda\, e'.\mathit{preEmpts}(e', e))
\end{aligned}$$

Priority and QueueStack will be equivalent whenever *preEmpts* is equivalent to $\neg$ *higher*:

$$\forall (q, e)(\mathit{None}(q, \lambda\, e'.\neg\ \mathit{higher}(e, e')) = \mathit{None}(q, \lambda\, e'.\mathit{preEmpts}(e', e)))$$

A sufficient restriction (again, arguably too strong) for this to be achieved is $\forall\, e.\mathit{qType}(e)$. We then have

$$Priority \sqsubseteq_{(id\ ,\ \forall\, e.qType(e))} QueueStack$$

and by composition

$$\begin{aligned} (Priority \circ Mask) &\sqsubseteq_{(id\ ,\ \forall\, e.qType(e))} (QueueStack \circ Mask) \\ \Rightarrow NextAndRest_3 &\sqsubseteq_{(id, \forall\, e.qType(e))} NextAndRest_4 \end{aligned}$$

Other alternatives can also be built, such as a maskless priority queue which we discuss in Section 7.6.

7.5.5 $NextAndRest_5$: Remove

Current Viewpoint	$NextAndRest_4 = QueueStack \circ Mask$
Incremental Viewpoint	$Remove$
Product Viewpoint	$NextAndRest_5 = Remove \circ QueueStack \circ Mask$

The idea behind this final stage is that, as well as being of $qType$ or $sType$, events may be additionally of $rType$; if such an event is selected, all "related" events must be discarded from the queue. While $NextAndRest_5$ is presented by simply wrapping an auxiliary function around $NextAndRest_4$, the increment itself is not dependent upon any particular version of $NextAndRest$ — it might make most sense to describe it this way, but there is no reason why we should not build it on top of any other version. If we create a simpler "front end" function $Remove$:

$$Remove(q, e) = \langle e, \textbf{if } rType(e) \textbf{ then } RemoveAll(q, e) \textbf{ else } q\rangle \tag{7.19}$$

for $RemoveAll$ as defined above (Table 7.2), then the advertised new version can be specified as:

$$\boxed{NextAndRest_5(q, m) = Remove(NextAndRest_4(q, m))} \tag{7.20}$$

or

$$NextAndRest_5(q, m) = Remove(QueueStack(Mask(q, m))) \tag{7.21}$$

We can see the function $Remove$ as the compliment of our earlier $Mask$.

For co-refinement between $NextAndRest_4$ and $NextAndRest_5$, we observe that they are equivalent when *Remove* is the identity function. This will be achieved if

$$(\forall\, e . \neg\ rType(e))$$

which gives us a sufficient (strong) restriction:

$$\begin{aligned} &\forall\, e\ \neg\ rType(e) \Rightarrow \\ &\quad (Remove \circ QueueStack \circ Mask = QueueStack \circ Mask) \\ &\quad \Rightarrow NextAndRest_4 \mathrel{\underline{\sqsubseteq}}_{(id\ ,\ \forall\, e.\neg\ rType(e))} NextAndRest_5 \end{aligned}$$

We now go on to consider what other relations hold between the identified viewpoints.

7.6 Relationships between Viewpoints

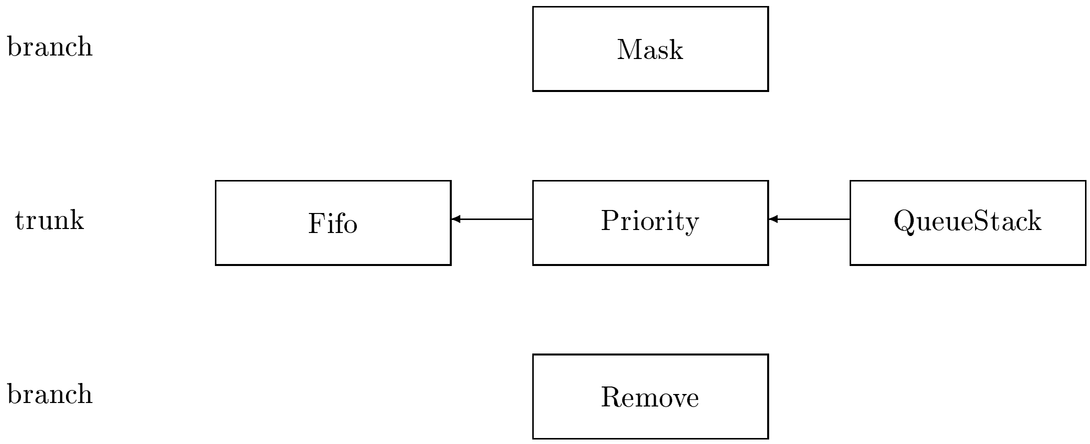

Figure 7-1: Viewpoint version dependencies. Priority is built on Fifo, Queuestack is built on Priority.

As we have noted, the advantage of organising the increments in this way is that we have isolated those parts which can be presented in any order and those which depend on previous stages. Figure 7-1 illustrates this. We can create whatever version of *NextAndRest* we wish to, making a "pick & mix" selection of one item from the central "trunk", and optionally one or both of the branches. Thus we can have simply the vanilla *Fifo* version (the simple First-In-First-Out queue, corresponding to $NextAndRest_1$), a *Mask 'n' Fifo* version corresponding to $NextAndRest_2$, a *Priority 'n' Mask* version corresponding to $NextAndRest_3$, etc. We can also build alternative versions such as *Fifo 'n' Remove* (a version of $NextAndRest_1$ which removes any events related to the selected one if applicable) or *Priority*, which acts simply as the presented increment above ($NextAndRest_3$ without the mask).

More formally, these different versions can be combined using function composition for our amalgamation operator.

Using this notation, we can build twelve combinations as follows:

1. $Fifo = \lambda\, q.\ AbsTest\ q\ \ \lambda(q,e).None(q, \lambda\, e'.\ \mathbf{true}\ )$ $\quad NextAndRest_1$
2. $Priority = \lambda\, q.\ AbsTest\ q\ \ \lambda(q,e).None(q, \lambda\, e'.\ \mathbf{not}\ higher(e,e'))$
3. $QueueStack = \lambda\, q.\ AbsTest\ q\ \ \lambda(q,e).None(q, \lambda\, e'.\ preEmpts(e',e))$
4. $Fifo \circ Mask = \lambda(q,m).\ Fifo(Mask(q,m))$ $\quad NextAndRest_2$
5. $Priority \circ Mask = \lambda(q,m).\ Priority(Mask(q,m))$ $\quad NextAndRest_3$
6. $QueueStack \circ Mask = \lambda(q,m).\ QueueStack(Mask(q,m))$ $\quad NextAndRest_4$
7. $Remove \circ Fifo = \lambda\, q.\ Remove(Fifo(q))$
8. $Remove \circ Priority = \lambda\, q.\ Remove(Priority(q))$
9. $Remove \circ QueueStack = \lambda\, q.\ Remove(QueueStack(q))$
10. $Remove \circ Fifo \circ Mask = \lambda(q,m).\ Remove(Fifo(Mask(q,m)))$
11. $Remove \circ Priority \circ Mask = \lambda\, q.\ Remove(Priority(Mask(q,m)))$
12. $Remove \circ QueueStack \circ Mask = \lambda\, q.\ Remove(QueueStack(Mask(q,m)))$ $\quad NextAndRest_5$

A number of co-refinements are observable here: to begin with, we have shown in the derivations (Section 7.5) that there is a co-refinement between each of the versions on the main branch, i.e.

$$Fifo \sqsubseteq_{(id\ ,\ \forall(e,e').\neg\ higher(e,e'))} Priority \sqsubseteq_{(id\ ,\ \forall\, e.qType(e))} QueueStack$$

and we have used function composition to show

$$\begin{aligned}
\mathit{Fifo} \circ \mathit{Mask} \quad & \mathrel{\underline{\circledcirc}}_{(id\ ,\ \forall (e,e').\neg\, higher(e,e'))} \mathit{Priority} \circ \mathit{Mask} \\
& \mathrel{\underline{\circledcirc}}_{(id\ ,\ \forall\, e.qType(e))} \mathit{QueueStack} \circ \mathit{Mask}
\end{aligned}$$

or, to give them their original names,

$$\begin{aligned}
\mathit{NextAndRest}_2 \quad & \mathrel{\underline{\circledcirc}}_{(id\ ,\ \forall (e,e').\neg\, higher(e,e'))} \mathit{NextAndRest}_3 \\
& \mathrel{\underline{\circledcirc}}_{(id\ ,\ \forall\, e.qType(e))} \mathit{NextAndRest}_4
\end{aligned}$$

and we also have, from Sections 7.5.2 and 7.5.5:

$$\begin{aligned}
\mathit{NextAndRest}_1 \quad & \mathrel{\underline{\circledcirc}}_{(\lambda(q,m).q\ ,\ \forall\, e.wanted(e,m))} \mathit{NextAndRest}_2 \\
\mathit{NextAndRest}_4 \quad & \mathrel{\underline{\circledcirc}}_{(id\ ,\ \forall\, e.\neg\, rType(e))} \mathit{NextAndRest}_5
\end{aligned}$$

Thus each of the functions presented in the original development is a co-refinement of its predecessor (and of all its predecessors), i.e. (omitting links and restrictions for the moment)

$$\mathit{NextAndRest}_1 \mathrel{\underline{\circledcirc}} \mathit{NextAndRest}_2 \mathrel{\underline{\circledcirc}} \mathit{NextAndRest}_3 \mathrel{\underline{\circledcirc}} \mathit{NextAndRest}_4 \mathrel{\underline{\circledcirc}} \mathit{NextAndRest}_5$$

Figure 7-2 illustrates the relations between each of the viewpoints presented. The bold numerals **1**. . . **5** indicate $\mathit{NextAndRest}_{(1\ldots5)}$ respectively.

Using the augmented co-refinement relation (Section 7.4.4) to collect links and restrictions as we go, we have the following:

$$\begin{aligned}
\mathit{NextAndRest}_1 \quad & \mathrel{\underline{\tilde{\circledcirc}}}_{(\lambda(q,m).q\ ,\ \forall\, e.wanted(e,m))}\ \mathit{NextAndRest}_2 \\
& \mathrel{\underline{\tilde{\circledcirc}}}_{(\lambda(q,m).q\ ,\ \forall\, e.wanted(e,m)\ \wedge\ \forall\, e,e'.\neg\, higher(e,e'))}\ \mathit{NextAndRest}_3 \\
& \mathrel{\underline{\tilde{\circledcirc}}}_{(\lambda(q,m).q\ ,\ \forall\, e.wanted(e,m)\ \wedge\ \forall\, e,e'.\neg\, higher(e,e')\ \wedge\ \forall\, e.qType(e))}\ \mathit{NextAndRest}_4 \\
& \mathrel{\underline{\tilde{\circledcirc}}}_{(\lambda(q,m).q\ ,\ \forall\, e.wanted(e,m)\ \wedge\ \forall\, e,e'.\neg\, higher(e,e')\ \wedge\ \forall\, e.(qType(e)\ \wedge\ \neg\, rType(e)))} \\
& \mathit{NextAndRest}_5
\end{aligned}$$

We can then conclude from the transitivity given by augmented co-refinement

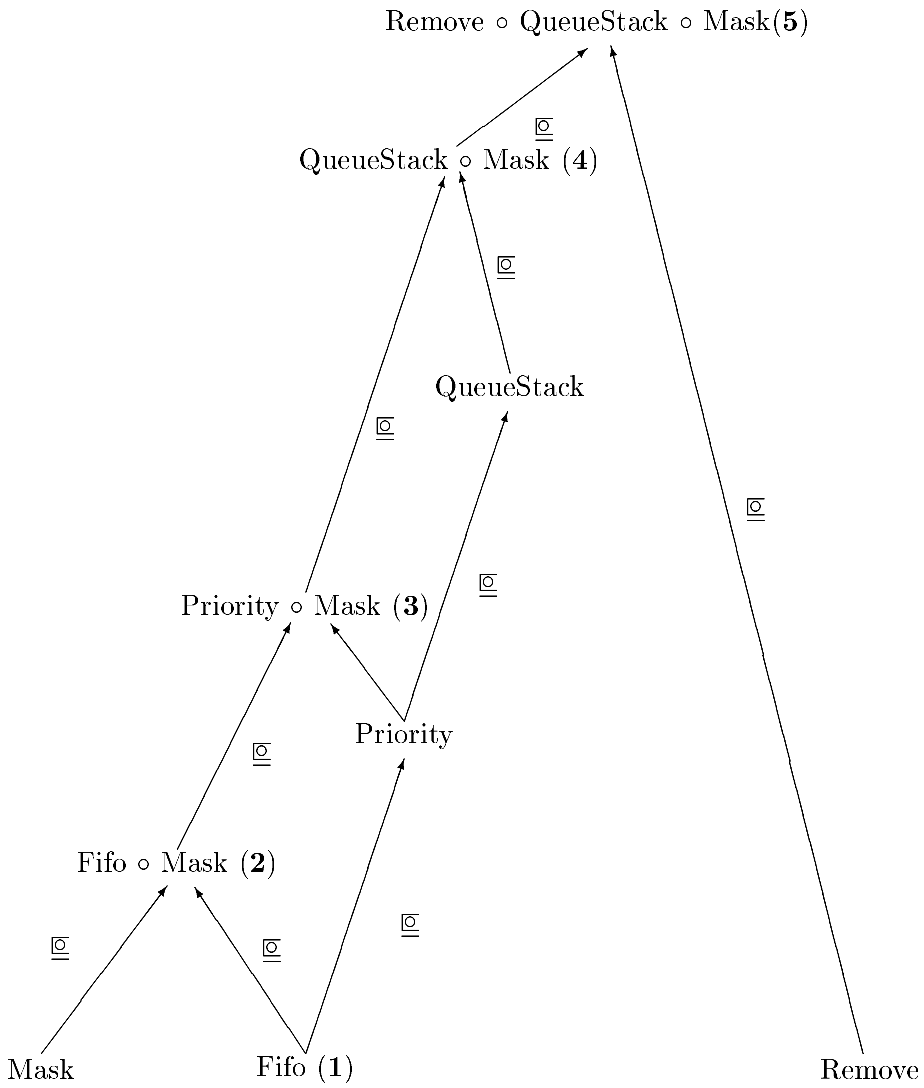

Figure 7-2: Viewpoint relations

$$NextAndRest_1 \quad \tilde{\underline{\underline{\odot}}}_{(\lambda(q,m).q\ ,\ \forall\, e.wanted(e,m)\ \wedge\ \forall\, e,e'.\neg\, higher(e,e')\ \wedge\ \forall\, e.(qType(e)\ \wedge\ \neg\, rType(e)))}\ NextAndRest_5$$

So the final version is a co-refinement, with a strong restriction, of the initial one. We have shown that, under certain defined circumstances, $NextAndRest_5$ will behave as $NextAndRest1$.

7.6.1 Alternative Viewpoint Presentation

The presentation of the increments given here is not, of course, unique. One alternative which keeps the same number of increments but presents them differently is to make each viewpoint a "filter", like the *Mask* increment above. The initial viewpoint, which we have called *Fifo*, remains unchanged as does the *Mask* increment. The *Priority* increment can be presented as a sorting function, $PrioritySort(q)$, which takes a queue and sorts it so that the highest priority event is at the head of the queue, and the order of equal priority events is preserved.

$NextAndRest_3$ is then presented as $NextAndRest_1 \circ PrioritySort \circ Mask$.

The *QueueStack* increment is harder to achieve. A function is needed which (regardless of whether the queue has already been sorted) reverses the order in the queue of all *sType* events. Thus the effect of this filter is to render the queue as entirely *qType*. Calling this function $Stack2Queue$, we can present $NextAndRest_4$ as

$$NextAndRest_1 \circ Stack2Queue \circ PrioritySort \circ Mask$$

The final *Remove* increment is already presented as a filter, so we present $NextAndRest_5$ as simply

$$Remove \circ NextAndRest_1 \circ Stack2Queue \circ PrioritySort \circ Mask$$

This presentation of the viewpoints is not necessarily preferable; while it would appear more straightforward to present everything in terms of filters and function composition rather than application and instantiation, the tortuous nature of the *QueueStack* increment does not lend itself to comprehension. An advantage of this approach is that it is easier to see which of the steps can be combined arbitrarily and in different orders. The

Mask, if used, must be the first function to be used because it transforms the domain. *Remove* can only be applied at the end, as it acts on the selected element. The two filters *PrioritySort* and *Stack2Queue*, however, can be exchanged without affecting the result, they can or removed independently. This differs from the previous presentation in which *QueueStack* superseded *Priority* due to the use of the *higher* function.

7.7 Summary and Conclusions

We have succeeded in showing that a refinement relationship holds between each of the steps in the presentation, thus providing a formal basis behind this method of incremental specification, and demonstrated how new information in a system can be presented as an independent unit.

We are still dealing with a slightly idealised example; the development is in reality post-hoc, in that it does not reflect the inevitable back-tracking that would occur if the development were begun from scratch (a situation identified in Section 4.6.1 as consisting of related compromises). However, the example is representative as an *explanation* of the functionality of the event queue.

Implementing the resulting viewpoints could lead to some inefficiencies, especially in the last section where a *PrioritySort* function has to sort the queue each time *NextAndRest* is called. This is an illustration of a point made in Section 2.1, that what makes a good modularisation (or explanation, in this case) from the point of view of the specifier may not be so good from the point of view of an implementer. In this case the implementer might choose to implement the *add* function so that it always maintains the queue in a sorted state.

The presentation of these independent units is comparable, especially in the additional use of "filters" in the previous section, to the use of pipes and filters in the UNIX operating system [Wal93]. The units as presented can be thought of as re-usable building blocks.

In the next chapter we consider the application of viewpoints and refinement to an explanation of the semantics of programming languages.

Chapter 8

Denotational Semantics

We have already noted the necessity of expressing the semantics of a language to enable us to reason about any viewpoint expressed in that language. Dijkstra's guarded command language (Section 4.2) has its semantics expressed in terms of predicate transformers, and we saw in Section 4.3 how it has been extended to make the process of program development more straightforward.

In this chapter we will look at how the language in which the semantics of programming languages are expressed can similarly be used as a basis for refinement of viewpoints. The viewpoints we are dealing with are specifications of a programming language, so we are looking at things one level up from previous examples (and considering the semantics of semantics).

8.1 Introduction

While it is not normal to speak of refinements between semantic descriptions, a common approach in texts such as [Gor79, Sch86] is to begin with a very simple language and add features to it. We will introduce some concepts in denotational semantics and provide a simple example borrowed from [Sch86], to illustrate the idea of adding features to a description of a programming language, which we can model as amalgamating a viewpoint with another viewpoint representing the changes. We then use refinement to to investigate the relationship between the resulting viewpoint and its components.

8.1.1 Semantics of Programming Languages

The description of a programming language's syntax, typically in Backus-Naur form (BNF), specifies the set of all legal statements in a language and hence is an important piece of knowledge to implement a compiler for that knowledge; a parser for the language can be built from the BNF definition. This says nothing about the *meaning* of the statements, however.

A possible approach to the definition of a programming language's semantics is to provide a compiler for that language. This corresponds to the *operational* approach, exemplified by the Vienna Definition Language [Weg72]. A second approach is *axiomatic* semantics, in which logical axioms and inference rules related to the language's constructs are given, and a formal proof can be built to show a program has a certain property.

Denotational semantics provides a bridge between these two approaches, as it provides a means of defining semantics in a non-operational way by modelling a program's meaning as a mathematical function. The main feature of denotational semantics is a *valuation function*. Valuation functions are high-level and abstract, providing a modular structure is especially of interest to us in the present work.

The approaches are complementary: each provides a basis for reasoning, and each can be said to have its own place in the development of a computer language. Axiomatic semantics is the most abstract and can be used to give initial specifications of a language. A denotational definition may follow from that, which can be used in a proof that it satisfies the axioms given in the more abstract semantics. The denotational definition can then be implemented using an operational definition.

Denotational semantics has a wide body of literature and research associated with it [Sch86, Ten76] and its modular features suggest it will be amenable to consideration from our "viewpoints" approach.

8.2 Basic Form of a Denotational Description

A denotational description provides a function corresponding to a program, to be applied to its initial state. This function is defined over the sub-expressions of the program. Figure 8-1 provides an example denotational description.

There are three parts to the description:-

Abstract Syntax:

P ∈ Program
S ∈ Expr-Seq
E ∈ Expr
N ∈ Numeral

P ::= ON S
S ::= E TOTAL S | E TOTAL OFF
E ::= E_1 + E_2 | E_1 * E_2 | IF E_1,E_2,E_3 | LASTANS | (E) | N

Semantic Algebras:

Truth Values
Domain $t \in Tr = \mathbf{B}$
Operations
$true, false : Tr$
$not : Tr \to Tr$
$or : Tr \times Tr \to Tr$
$(\_ \to \_ [] \_) : Tr \times D \times D \to D$

Natural Numbers
Domain $n \in Nat = \mathtt{N}$
Operations
$zero, one, \ldots : Nat$
$plus, times : Nat \times Nat \to Nat$
$equals : Nat \times Nat \to Tr$

Valuation Functions:

$\mathbf{P}$:Program$\to Nat^*$
$\mathbf{P}[\![\mathtt{ON\ S}]\!] = \mathbf{S}[\![\mathtt{S}]\!](zero)$

$\mathbf{S}$:Expr-Seq$\to Nat \to Nat^*$
$\mathbf{S}[\![\mathtt{E\ TOTAL\ S}]\!] = \lambda n.\ let\ \ n' = \mathbf{E}[\![\mathtt{E}]\!](n) in\ \ n'\ \ cons\ \ \mathbf{S}[\![\mathtt{S}]\!](n')$
$\mathbf{S}[\![\mathtt{E\ TOTAL\ OFF}]\!] = \lambda n.\ \mathbf{E}[\![\mathtt{E}]\!](n)\ \ cons\ \ nil$

$\mathbf{E}$:Expr$\to Nat \to Nat$
$\mathbf{E}[\![\mathtt{E}_1 + \mathtt{E}_2]\!] = \lambda n.\ \mathbf{E}[\![\mathtt{E}_1]\!](n)\ \ plus\ \ \mathbf{E}[\![\mathtt{E}_2]\!](n)$
$\mathbf{E}[\![\mathtt{E}_1 * \mathtt{E}_2]\!] = \lambda n.\ \mathbf{E}[\![\mathtt{E}_1]\!](n)\ \ times\ \ \mathbf{E}[\![\mathtt{E}_2]\!](n)$
$\mathbf{E}[\![\mathtt{IF\ E}_1,\mathtt{E}_2,\mathtt{E}_3]\!] = \lambda n.\ \mathbf{E}[\![\mathtt{E}_1]\!](n)\ \ equals\ \ zero \to \mathbf{E}[\![\mathtt{E}_2]\!](n)\ []\ \mathbf{E}[\![\mathtt{E}_3]\!](n)$
$\mathbf{E}[\![\mathtt{LASTANS}]\!] = \lambda n.\ n$
$\mathbf{E}[\![\mathtt{(E)}]\!] = \lambda n.\ \mathbf{E}[\![\mathtt{E}]\!](n)$
$\mathbf{E}[\![\mathtt{N}]\!] = \lambda n.\ \mathbf{N}[\![\mathtt{N}]\!]$

$\mathbf{N}$: Numeral $\to Nat$ (definitions omitted)

Figure 8-1: Calculator semantics, from [Sch86]

1. Abstract Syntax — defined in terms of syntax domains and BNF rules. Rather than use terminal symbols as a *concrete* syntax BNF description would, each rule is expressed in terms of the tokens (or words) which are members of a syntax domain. For example in Figure 8-1, `P` $\in$ `Program` indicates that `P` is an arbitrary non-terminal in the syntax domain `Program`.

2. Semantic Algebras — the domains and operations that are used to describe the semantics of the language. In Figure 8-1, domain *Nat* has nullary operations *zero*, *one*, ..., and binary operations *plus*, *times*, *equals*. The domain of truth values *Tr* includes an infix *choice* operator $(\_ \rightarrow \_ \,[]\, \_) : Tr \times D \times D \rightarrow D$, which provides an if–then–else construct.

3. Valuation Functions — the main part of the denotational description, which defines a function for each legal sub-expression of the program. In figure 8-1, the valuation function $\mathbf{P}[\![\texttt{ON S}]\!]$ has signature `Program`$\rightarrow$ *Nat* and denotes the meaning of a program as the meaning of a further valuation function S on the text of the program `S`. The brackets $[\![\ldots]\!]$ serve to separate syntax tokens from function definition.

Modelling a program's meaning as a function makes it possible to compare two statements in a language and show that they are equivalent if they have the same denotation.

8.2.1 Building Semantic Domains

The domains used in the semantic algebra can be based on *primitive* domains such as **B**,**N**, or *compound* domains which are formed by sums or products of domains, functions on domains, or lifting domains.

An example in Figure 8-1 is Nat^* which is a *sequence* or *list space* on *Nat* with the following operations for building lists:

$nil : Nat^*$
$cons : Nat \times Nat^* \rightarrow Nat^*$
$hd : Nat^* \rightarrow Nat$
$tl : Nat^* \rightarrow Nat$
$null : Nat^* \rightarrow Tr$

and the following *simplification properties*

$hd(a\ \ cons\ \ l) = a$
$tl(a\ \ cons\ \ l) = l$

$null(nil) = true$
$null(a\ \ cons\ \ l) = false$

Simplification properties are used in proofs about properties of a semantic description.

The product space $(A \times B)$ has operation builders:

$\forall\, a \in A, b \in B : (a, b) \in A \times B$
$fst : A \times B \rightarrow A$
$snd : A \times B \rightarrow B$
and simplification properties:

$fst(a, b) = a$
$snd(a, b) = b$

Function domains have the following operation builders:

- Lambda abstraction: $(\lambda\, x.e) \in A \rightarrow B \bullet \forall\, a \in A, e[x \backslash a]$ is a unique value in B.
- Function application: for $g : A \rightarrow B$ and $a \in A$, $g(a) \in B$

Lifted domains are used for functions which may not return a value for all elements in their domain: for example the *div* function is not defined for $(n\ \ div\ 0)$. The symbol $\perp$, pronounced "bottom", is used for an undefined value. A lifted domain $A_\perp$ is formed by union: $A_\perp = A \cup \{\perp\}$.

The abbreviation *let* is used for a general form of lambda abstraction: $(let\ x = e_1\ in\ e_2)$ abbreviates $\lambda\, x.e_2)e_1$, if the domain of e_1 is not lifted.

8.3 Semantic Description as Specification

A semantic description is a formal specification, and any programming language which implements that description must satisfy the specification. There are similarities with the style of specification used in an algebraic specification language such as CLEAR [BG86], with *sorts* defined by the semantic algebras and *equational axioms* by the valuation functions. In common with Z and VDM we have a "definition" part and a "predicate" part.

However, the use of axioms is more a feature of axiomatic semantics; what we are missing here is that denotational semantics express the meaning of a program as a *function*, and a semantic description is itself a function on a program. The signature of a semantic description S is $S : \mathit{Program} \to (A \to B)$, where A and B are the domain and range of the function which models the meaning of the program.

A denotational semantic viewpoint can be defined as any semantic description, with the possibility of some gaps and inconsistencies in the description. A viewpoint that will be used as a specification of a programming language, however, needs to be consistent and complete.

Before we go any further we need to identify a suitable refinement relation for semantic viewpoints. We begin with a definition of *correctness* for semantic descriptions and programming languages.

8.3.1 Correctness

Definition 8.1 (Correct implementation) *A programming language is a* correct *implementation of a semantic description iff all programs in that language satisfy the valuation function in the semantic description.*

In particular this definition means that a programming language which contains statements to which no valuation function applies will not be a correct implementation of the semantic description.

We can now say that for a programming language L, another language L' preserves the correctness of L if it also satisfies L's semantic description. However, this suggests that if L' has been modified to allow a wider range of program statements, correctness will not be preserved: L's semantic description will not include the extra program statements.

Following this line of reasoning, if a semantic description S is amalgamated with another to form a new description S', any programming language which implements S' should also be a possible implementation of S. The problem is that our definition of correctness is too strong to allow any modifications to the semantics other than those which maintain equivalence. Any two languages which are both correct with respect to a semantic description will have the same meaning, since it is the semantic description which defines their meaning. If they have the same meaning, the languages are equivalent.

However, we know how to deal with problems of this sort — we simply weaken our definition of correctness.

Definition 8.2 (Consistent implementation) *A programming language L is a* consistent *implementation of a semantic description* S *iff, for all programs* P *in the language* L *(*$P \in L$*), either* $S(P)$ *is undefined or* P *satisfies* S*.*

8.3.2 Refinement Relation for Semantic Descriptions

With this weakened definition of "correctness", we can require that, for a refinement relation to hold between a semantic description S and an extended description S', any implementation of S' must be a consistent implementation of S.

Recalling that a semantic description is an expression of a program's meaning as a function, we can define a refinement relation as follows:

Definition 8.3 (Refinement) *For semantic descriptions* S *and* S'*, and respective implementations* L *and* L'*,* $S \sqsubseteq S'$ *iff for all programs* $P \in L$*,*

$P \in L'$ *and*
$S(P) \equiv S'(P)$

Thus refinement between semantic descriptions is defined as equivalence between meanings of a program in each of the semantic descriptions. The meaning of a program is a function too, from input to output or from initial state to final state, so $S(P) \equiv S'(P)$ exactly when their graphs are equal, that is $S(P)(x) = S'(P)(x)$ for all x.

This refinement relation is reflexive (if $S = S'$ and $L = L'$, refinement will hold) and transitive: for S, S', S'' and L, L', L'' such that $S \sqsubseteq S'$ and $S' \sqsubseteq S''$, we have for all programs P,

$$(P \in L \Rightarrow P \in L') \wedge (P \in L' \Rightarrow P \in L'')$$
$$\leftrightarrow (P \in L \Rightarrow P \in L'')$$

and

$$(S(P) \equiv S'(P)) \wedge (S'(P) \equiv S''(P))$$
$$\leftrightarrow S(P) \equiv S''(P)$$

The relation preserves our weaker definition of correctness, *consistent implementation*: if L is a consistent implementation of S, and we derive (S', L') such that $S \sqsubseteq S'$ and L' is a consistent implementation of S', then L' is a consistent implementation of S.

The proof is as follows: if we take any program P in the language L', it may or may not be a valid program in the language L. If it is, then $P \in L$ and by the refinement relation $S'(P) \equiv S(P)$, so consistency is satisfied. Otherwise, $S(P)$ will not be defined so consistency is satisfied.

8.3.3 Composition Operators

To develop a denotational description of a programming language we need to define operations to build specifications, in addition to the domain construction operators introduced in Section 8.2.1. These can then be used in an amalgamation of two viewpoints representing denotational descriptions, and as long as we use operators that are monotonic with respect to the refinement ordering, we will be able to show that the amalgamation refines its constituent viewpoints.

Using the notation $syn(S)$,$alg(S)$ and $val(S)$ to refer to the abstract syntax, semantic algebra and valuation function parts of semantic viewpoint S, we can identify operations to perform on a semantic viewpoint:

- Adding to the abstract syntax

  If we wish to extend the range of expressions in our programming language, we start by making an addition to the abstract syntax. If we simply do this, however, we have an inconsistent viewpoint as the valuation function is not defined over the new expression. Thus after adding to the syntax, if we wish to regain consistency (which we are only obliged to do if we wish to implement the viewpoint), the valuation function must also be extended.

- Extending the valuation function

  This can be done without also making the syntax change, but again an inconsistent viewpoint will result. We will only have a refinement if both operations are performed. For the moment we will insist that the new valuation function introduce no new semantic domain, i.e. operations used in the function are already defined in the semantic algebra. For new syntactic expression $token\texttt{::=}e$ and new valuation $f[\![e]\!]$

  $S \sqsubseteq S'$, where $S' = S[syn(S)\backslash syn(S) \cup \{token\texttt{::=}e\}, val(S)\backslash val(S) \cup \{f[\![e]\!]\}]$, provided $f[\![e]\!]$ and $token\texttt{::=}e$ are not already defined in S, and all semantic domains used in the definition of f are in S. If f is already defined in S, the signature of f must be identical.

  This is indeed a refinement, since any program for which $S(P)$ is defined will have the same meaning in S'. The operation of extending the syntax and valuation function is monotonic, since the new function will not interfere with any function already in the viewpoint.

With these basic operators we can go through a simple example, using the earlier introduced calculator example.

8.4 Example: Adding a Button to a Calculator

This example comes from [Sch86], which presents the semantics of a simple calculator. Figure 8-1 is a summary of the semantics; we are dealing with a calculator which, in addition to being able to cope with sums and products, has an elementary decision operator (`IF`).

The valuation functions can be expanded on by the following commentary. A calculator program consists of a press of the ON button followed by a sequence of expressions. After the final expression has been terminated with a press of the TOTAL button, the OFF button terminates the program. The calculator also has a last-answer memory feature, hence the form of the signature for $\mathbf{S} : \mathtt{Expr\text{-}Seq} \rightarrow \mathtt{Nat} \rightarrow \mathtt{Nat}^*$. An expression sequence is evaluated as a function from the current "LASTANS" (initially `zero`) to a sequence of numbers.

A sample program would thus be "`ON 3 + 4 TOTAL IF LASTANS 5,6 TOTAL LASTANS + (2 * 7) OFF`". This is evaluated as follows:

$\mathbf{P}[\![\mathtt{ON\ 3\ +\ 4\ TOTAL\ IF\ LASTANS\ 5,6\ TOTAL\ LASTANS\ +\ (2\ *\ 7)\ OFF}]\!]$
$\hat{=}\ \mathbf{S}[\![\mathtt{3\ +\ 4\ TOTAL\ IF\ LASTANS\ 5,6\ TOTAL\ LASTANS\ +\ (2\ *\ 7)\ OFF}]\!](\mathit{zero})$
$\hat{=}\ \mathit{let}\ n' = \mathbf{E}[\![\mathtt{3\ +\ 4}]\!](\mathit{zero})\ \mathit{in}$
$\qquad n'\ \mathit{cons}\ \mathbf{S}[\![\mathtt{IF\ LASTANS\ 5,6\ TOTAL\ LASTANS\ +\ (2\ *\ 7)\ OFF}]\!](n')$

The **E** expression evaluates to:

$\mathbf{E}[\![\mathtt{3\ +\ 4}]\!](\mathit{zero})$
$\hat{=}\ \mathbf{E}[\![\mathtt{3}]\!](\mathit{zero})\ \mathit{plus}\ \mathbf{E}[\![\mathtt{4}]\!](\mathit{zero})$
$\hat{=}\ \mathbf{N}[\![\mathtt{3}]\!]\ \mathit{plus}\ \mathbf{N}[\![\mathtt{4}]\!]$
$\hat{=}\ \mathit{three\ plus\ four}\ \hat{=}\ \mathit{seven}$

so the **S** expression becomes:

$\mathbf{S}[\![\mathtt{IF\ LASTANS\ 5,6\ TOTAL\ LASTANS\ +\ (2\ *\ 7)\ OFF}]\!](\mathit{seven})$

And so on, leading to an output sequence of $(\mathit{seven}, \mathit{six}, \mathit{twenty})$.

8.4.1 Incremental Viewpoint

An exercise in [Sch86] deals with adding a new button to the calculator which should test for equality, to be used in conjunction with the **IF** button; thus we would be able to say

```
IF (5+4)=(3*3),2,3
```

which would give 2. This suggests that the test for equality should return zero for true, non-zero for false, and the new valuation function would look like this:

$\mathbf{E}[\![\mathtt{E}_1 = \mathtt{E}_2]\!](n) = \mathbf{E}[\![\mathtt{E}_1]\!](n)\ \mathit{equals}\ \mathbf{E}[\![\mathtt{E}_2]\!](n) \rightarrow \mathit{zero}\ []\ \mathit{one}$

A new part also needs to be added to the abstract syntax, for `E ::=` $\mathtt{E}_1$ `=` $\mathtt{E}_2$.

We now come to the question of how a new version of the semantic description given in figure 8-1 is formed with this new feature. We consider our initial viewpoint to be this semantic description.

We can express the additional information as an incremental viewpoint, as follows:

Abstract Syntax: $\mathtt{E} \in \mathtt{Expr}$ $\mathtt{E} ::= \mathtt{E}_1 = \mathtt{E}_2$
Semantic Algebras: **Truth Values** ... **Natural Numbers** ...
Valuation Functions: $\mathbf{E} : \mathtt{Expr} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat}$ $\mathbf{E}[\![\mathtt{E}_1 = \mathtt{E}_2]\!](n) = \mathbf{E}[\![\mathtt{E}_1]\!](n)\ \mathit{equals}\ \mathbf{E}[\![\mathtt{E}_2]\!](n) \rightarrow \mathit{zero}\ []\ \mathit{one}$

What we have is a valuation function only defined for expressions of the form $\mathtt{E}_1 = \mathtt{E}_2$. This viewpoint is clearly quite useless on its own, being recursively defined with no base case. It is fact arguable whether we need to include the algebras for truth values and natural numbers, but we would argue that it is, since in amalgamating the viewpoint with the original we need to know whether any new semantic domains are to be added.

8.4.2 Amalgamation

Having identified the new information we now need to amalgamate it with the initial viewpoint. We divide this into coalescence planning and coalescence stages (Section 3.2).

In coalescence planning we identify what changes need to be made to achieve the amalgamation. To add the new piece of syntax to the viewpoint we need first check that it is unique, i.e. that $\mathtt{E} ::= \mathtt{E}_1 = \mathtt{E}_2$ is not already in the viewpoint. We operate under the assumption that items with the same name refer to the same item, i.e. that $\mathtt{E}$ in the incremental viewpoint is the same as $\mathtt{E}$ in the original.

The result of this check is that the new syntax is not already defined, but that the syntax domain $\mathtt{E}$ is already defined. The resulting action to perform in coalescence will therefore be to augment the syntax domain $\mathtt{E}$ with the new form. This check also shows that a valuation function needs to be defined for the new syntax.

The next stage is to check the semantic algebra. The algebraic domains in the incremental viewpoint are a subset of those in the original one, and have the same operations and signatures.

Finally the valuation functions. The new function $\mathbf{E}$ is already defined in the original viewpoint, but not on the expression $\mathtt{E}_1 = \mathtt{E}_2$. So it can be added, after checking that $\mathbf{E}$'s signature is the same as in the original viewpoint. This check also discharges the condition on the addition of abstract syntax, as the new valuation function defines the semantics of the new piece of syntax.

The result of the planning stage is that the following action should take place (where *Calc*, *Inc* and *Calc_New* refer to the initial, incremental and amalgamated viewpoints):

> **Action:** *Calc_New* is to be formed from
>
> $$\begin{aligned} &\mathit{Calc}[\mathit{syn}(\mathit{Calc})\backslash \mathit{syn}(\mathit{Calc}) \cup \mathtt{E} ::= \mathtt{E}_1 = \mathtt{E}_2, \\ &\quad \mathit{val}(\mathit{Calc})\backslash \mathit{val}(\mathit{Calc}) \cup \\ &\qquad \mathtt{E}[\![\mathtt{E}_1 = \mathtt{E}_2]\!](n) = \mathbf{E}[\![\mathtt{E}_1]\!](n)\ \mathit{equals}\ \mathbf{E}[\![\mathtt{E}_2]\!](n) \rightarrow \mathit{zero}\ [\,]\ \mathit{one}] \end{aligned}$$

The coalescence stage is then simply the putting into effect of the above action. The checks and justifications which came out of the planning stage will form a part of the amalgamation trail.

8.4.3 Refinement

Our amalgamated product, *Calc_New*, is a refinement of *Calc*, because the expression above defining the new viewpoint uses only the operations we have defined in Section 8.3.3, which satisfy our refinement definition.

The feature added could not be said to be very complicated, however. We next take into account a more complex feature to be added to the calculator, involving a change to the semantic algebras of the description.

8.5 Example: Adding Assignment

We now want to be able to deal with a further new feature, which is a more involved alteration: adding an assignment operation. The physical calculator will need a number of extra buttons for identifiers, to be called `A`,`B`,`C`,`D`, and a `:=` button (for "becomes", or "takes the value"). New expressions would be of the form:

```
A := 5 + 6
B := IF (A=4),3,2
D := (A + 4) * (A + 5)
```

Such an increment needs a number of changes:

- New syntax domain for identifiers, with BNF rule. A decision needs to be made as to whether `I := E` is a normal expression or not. Expressions of the form `(A := 5) + A` would be legal in some languages — for example in C, the statement `b = (a=5) + a;` assigns the value `5` to `a` and `10` to `b`. For the calculator this would over complicate matters, so we will identify `I := E` as a special kind of expression by altering the syntax for expression sequences S to include `I := E TOTAL S`.
- New semantic domain, *Store*, a function from identifiers to natural numbers, with operations to create an empty store, read and update the store
- New valuation functions for identifiers and assignments. An identifier is simply an expression, whose meaning is the natural number in the store referenced by the identifier. The meaning of an assignment is to update the store for a given identifier.
- New signatures for expression sequences S and expressions E, which will need be altered to take into account the store, rather than just the value held in the simple

LASTANS store. It is natural to explain the store as an extension of the single-cell memory we already have. The valuation functions for expression sequences and expressions will change from

S : Expr-Seq $\rightarrow$ Nat $\rightarrow$ Nat*
E : Expr $\rightarrow$ Nat $\rightarrow$ Nat

to

S : Expr-Seq $\rightarrow$ (Nat $\times$ Store) $\rightarrow$ Nat*
E : Expr $\rightarrow$ (Nat $\times$ Store) $\rightarrow$ Nat

since both the last answer and the current store are to be maintained.

These changes will allow a program sequence such as the following:

Input	Display
10 + 5 TOTAL	15
A := LASTANS TOTAL	15
LASTANS + 2 TOTAL	17
B := A * 10 TOTAL	150

8.5.1 Further Composition Operators

In order to make these changes we will need more composition operators than those already given in Section 8.3.3. The first applies to the semantic algebra:

- A new domain can be added to the algebra by stating its signature and operations; as long as the names of the domain and its operations are not already used in the viewpoint, this operation will not affect the rest of the viewpoint (compare the addition of local variables in the refinement calculus). So for any semantic viewpoint S and new semantic domain D,

  $S \sqsubseteq S[alg(S) \backslash alg(S) \cup D]$ provided D is free in S.

  This operation has the necessary property of monotonicity, because $alg(S) \subseteq (alg(S) \cup D)$.

- Changing valuation functions. We have already seen that valuation functions can be extended. We can also *transform* them, as in Section 3.5.5, by changing their range. This involves the use of an abstraction function from the old range to the new one: i.e. we replace a valuation function $\mathbf{F} : D \rightarrow X \rightarrow Y$ by $\mathbf{F'} : D \rightarrow X' \rightarrow Y'$ with an abstraction function $a : (X' \rightarrow Y') \rightarrow (X \rightarrow Y)$, where for arbitrary piece of abstract syntax `d`,

$\mathbf{F}[\![\mathtt{d}]\!] = a(\mathbf{F'}[\![\mathtt{d}]\!])$

If a suitable abstraction a can be found, the valuation function F can be exchanged for F'. However, F is used in function composition by other valuation functions, and may use other valuation functions in its definition: these may also need to be changed if F is replaced by F'. We have to make some limitations on the form of F', as follows:

We define the *top-level* syntactic domain of a denotational description as the (unique) domain which does not feature as an non-terminal (i.e. on the right-hand-side) in the BNF of any other syntactic domain. Thus in the calculator example `P` is the top-level syntactic domain. A top-level valuation function is the valuation function for the top-level domain.

We require for our refinement definition that for semantic descriptions S and S', any program in the implementation of S is also a valid program, with the same meaning, in the implementation of S'. The meaning of a program P in S is defined as $S(P)$, which is the return value of the top-level valuation function of S when applied to P. Thus our definition is equivalent to $\mathbf{T}_S(P) = \mathbf{T}_{S'}(P)$ for all programs P in the implementation of S.

The general form of $\mathbf{T}_S$ is $\lambda\, d.\Phi(\mathbf{F}_i(d_i))$, where the $\mathbf{F}_i$ are valuation functions called on sub-expressions d_i of the syntactic expression d, and Φ is the function applied to the results of these valuations. If any of the F_i are changed we require that it is possible to change Φ to keep $\mathbf{T}_S$ constant. Thus:

$\exists\, \Phi'.\ \Phi(\mathbf{F}_i(d_i)) = \Phi'(\mathbf{F'}_i(d_i))$

The definition of $\mathbf{T}_S$ can then be changed to $\mathbf{T'}_S = \Phi'(\mathbf{F'}_i(d_i))$, and since $T_S = T'_S$, the new semantic description refines the old one.

The effect of this restriction on F' and the creation of Φ is to reconstruct the monotonicity that is otherwise lost by the transformation.

8.5.2 Incremental Viewpoint

Figure 8-2 shows an incremental viewpoint representing the changes we need to make to the abstract syntax, semantic algebras and valuation functions.

8.5.3 Amalgamation

The coalescence planning phase will identify the following changes to be made before *Calc_New* can be amalgamated with the incremental viewpoint *Assign*:

```
Abstract Syntax:

S ∈ Expr-Seq
E ∈ Expr
I ∈ Identifier

S ::= I := E TOTAL S | I := E TOTAL OFF
E ::= I
```

`Semantic Algebras:`

`Natural Numbers`

`...`

`Identifiers`

`Domain` $i \in \mathit{Id} = \mathit{Identifier}$

`Store`

`Domain` $s \in \mathit{Store} = \mathit{Id} \to \mathit{Nat}$

`Operations`

$\mathit{init\_store} : \mathit{Store}$
$\mathit{init\_store} = \lambda\, i.\ \ \mathit{zero}$
$\mathit{read\_store} : \mathit{Id} \to \mathit{Store} \to \mathit{Nat}$
$\mathit{read\_store} = \lambda\, i.\, \lambda\, s.\ \ s(i)$
$\mathit{write\_store} : \mathit{Id} \to \mathit{Nat} \to \mathit{Store} \to \mathit{Store}$
$\mathit{write\_store} = \lambda\, i.\, \lambda\, n.\, \lambda\, s.(\lambda\, j.\ \ j = i \to n\ []\ s)$

`Valuation Functions`

$\mathbf{S}$:`Expr-Seq`$\to (\mathit{Nat} \times \mathit{Store}) \to \mathit{Nat}^*$
$\mathbf{S}[\![\texttt{I := E TOTAL S}]\!] = \lambda(n, s).$
$\quad \mathit{let}(n', s') = (\mathtt{E}[\![\mathtt{E}]\!](n, s), \mathit{write\_store}[\![\mathtt{I}]\!]\ \ \mathbf{E}[\![\mathtt{E}]\!](n, s)\ \ s)$
$\qquad \mathit{in}\ \ n'\ \ \mathit{cons}\ \ \mathbf{S}[\![\mathtt{S}]\!](n', s')$
$\mathbf{S}[\![\texttt{I := E TOTAL OFF}]\!] = \lambda(n, s).$
$\quad \mathit{let}(n', s') = (\mathtt{E}[\![\mathtt{E}]\!](n, s), \mathit{write\_store}[\![\mathtt{I}]\!]\ \ \mathbf{E}[\![\mathtt{E}]\!](n, s)\ \ s)$
$\qquad \mathit{in}\ \ n'\ \ \mathit{cons}\ \ \mathit{nil}$

$\mathbf{E}$:`Expr`$\to (\mathit{Nat} \times \mathit{Store}) \to \mathit{Nat}$
$\mathbf{E}[\![\mathtt{I}]\!] = \lambda(n, s).\mathit{read\_store}[\![\mathtt{I}]\!]s$

Figure 8-2: *Assign*: Incremental viewpoint for the addition of assignment

- Abstract Syntax: `I` $\in$ `Identifier` is a new domain. `S` $\in$ `Expr-Seq` and `E` $\in$ `Expr` are defined in *Calc_New*, but *Assign* gives them new BNF forms. There is no conflict here so the syntax of the amalgamation will be formed by $syn(Calc\_New) \cup syn(Assign)$. New valuation functions need to be given for the new syntax.

- Semantic Algebras: The domain *Nat* is identical in both viewpoints. *Id* is new in *Assign*. The semantics can be formed by $alg(Calc\_New) \cup alg(Assign)$.

- Valuation Functions: Functions S and E have different signatures in *Assign* from those in *Calc_New*. Comparing $val(Calc\_New)$ with $val(Assign)$ shows that further changes are necessary: each of the equations for S and E in $val(Calc\_New)$ must be altered to have the same signature. We have defined those expressions that actually *use* the store; others will simply pass it along with the last answer memory.

  So the coalescence action is twofold: first the functions from the old viewpoint must be updated have the same signature as the new one, and then the amalgamated viewpoint is formed by collecting these functions.

All but the last of the coalescence activities are straightforward. They use operations we have introduced in Sections 8.3.3 and 8.5.1. The valuation function changes need the transformation operation we defined in Section 8.5.1.

We are changing the functions for **S** and **E**, giving them new signatures:

S :Expr-Seq$\rightarrow (Nat \times Store) \rightarrow Nat^*$

and

E :Expr$\rightarrow (Nat \times Store) \rightarrow Nat$

The abstraction function a_S is defined for **S** as:

$a_S : ((Nat \times Store) \rightarrow Nat^*) \rightarrow (Nat \rightarrow Nat^*)$
$a_S = \lambda f.\ \lambda n.\ f(n, init\_store)$

and similarly a_E:

$a_E : ((Nat \times Store) \rightarrow Nat) \rightarrow (Nat \rightarrow Nat)$
$a_E = \lambda f.\ \lambda n.\ f(n, init\_store)$

The old versions of **E** and **S** are equivalent to a_E(**E'**) and a_S(**S'**), for new versions **E'** and **S'**. We now need to consider the effect on the other valuation functions in the viewpoint.

The only other valuation function that features as part of the definition of **E** is **N**. This definition will not need to be changed as the *Nat* argument was not passed to **N** in the original viewpoint.

This leaves only the top-level valuation function **P**, which must define an equivalent meaning for any program, between old and new versions. The old definition of **P** is:

$\mathbf{P}$:Program$\rightarrow$ Nat^*
  $\mathbf{P}[\![\text{ON S}]\!] = \mathbf{S}[\![\text{S}]\!](zero)$

Due to the signature change in **S'**, the right hand side of this definition must now provide a $(Nat \times Store)$ argument. The function we choose for Φ is, by no coincidence, similar to our abstraction function: it maps the `(zero)` to the corresponding value in the new domain, `(zero,` *init_store*`)`. Our new **P** is therefore

$\mathbf{P}'$:Program$\rightarrow$ Nat^*
  $\mathbf{P}'[\![\text{ON S}]\!] = \mathbf{S}'[\![\text{S}]\!](zero)$

The coalescence step then consists simply of assembling the new functions, and removing the ' decorations.

8.6 Further Changes: Continuations

We have already mentioned (Section 8.1.1) that alternative versions of semantics exist (axiomatic, operational). Within the denotational approach there are also differences; up to now we have been using *direct* semantics. Such semantics emphasise the structure of a language; an equation like

$$\mathbf{E}[\![\text{E}_1 + \text{E}_2]\!](\text{n}) = \mathbf{E}[\![\text{E}_1]\!](\text{n}) \text{ plus } \mathbf{E}[\![\text{E}_2]\!](\text{n})$$

makes no attempt to prescribe the intended order of evaluation. In some applications however the concept of concept of *control* is necessary to express order of evaluation; it can be modelled by a semantic construct called a *continuation*.

This allows modelling of control flow as presented in "goto" jumps, returns from procedures, errors or interrupt handling. *Continuation* semantics incorporates this idea, representing the "rest of the program" as a continuation (a function from machine-state to machine-state, taking the current state to the state at the termination of the program); this provides a way to override the "normal continuation" for jumps, error handling and so on.

Standard denotational semantics texts [Gor79, Sch86] deal with the complexity of the continuations issue by first explaining direct semantics and then motivating the addition of continuations, in a manner similar to the discussion in the previous paragraph. This is clearly an incremental change such as we have been discussing.

The general process followed in texts is to present the semantics of some command in direct semantics and introduce a *command continuation*, using it to replace the direct semantics version with one in continuation style; in the case of [Sch86] this is via a direct semantics which involves a command stack, which is replaced by the continuation function. Similarly expression semantics are altered to include the order of evaluation.

The question of the relationship between direct and continuation semantics is quite complex — proving that two definitions describe the same program is made difficult by the different structures of the definitions. A better way, coinciding with our approach to explanations in this thesis, would be to derive a continuation semantics definition from a direct one.

8.6.1 Constructing Continuation Semantics

As an example, suppose we have a valuation function

$$\mathbf{C}_D : \mathit{Command} \rightarrow \mathit{Store}_\perp \rightarrow \mathit{Store}_\perp$$

in direct semantics. Adding continuations to this valuation function we would expect to result in

$$\mathbf{C}_C : \mathit{Command} \rightarrow \mathit{Cmdcont} \rightarrow \mathit{Cmdcont}$$

where the command continuation is $\mathit{Cmdcont} = \mathit{Store}_\perp \rightarrow \mathit{Store}_\perp$.

Sethi and Tang [ST80] proposed a method for constructing continuation semantics from a direct semantics. Their method is to define, for each construct $[\![\mathrm{C}]\!]$, the continuation semantics from the direct semantics. An overview of the method follows.

If F and F' are domains of function values in direct and continuation semantics respectively, it is possible to define transformations for each term T in F to derive a corresponding T' in F'. The derivation is done in small stages by using an auxiliary operator δ as a syntactic place-holder, starting with δT, which represents the (as yet undefined) continuation semantics version of T in F', and 'pushing' δ onto smaller sub-expressions of T using transformation rules.

To show that T and T' represent related values, all intermediate expressions in the derivation have meanings in either the direct or continuation domain. Representing a term as a tree, any sub-term below δ represents values from the direct domain and the rest represent values from the continuation domain. Thus δ is used as a bridge between the direct and continuation worlds to enable the derivation to be made a step at a time. We can make a comparison between this and the progression from specification to program; the refinement calculus uses the specification statement as a bridge for a similar reason.

A predicate is defined to test the equivalence of values from each of the domains. For direct value $v \in V$ and continuation value $w \in K \to A$, the values satisfy the equivalence predicate if either both are $\perp$, both $\top$ or, for v' equivalent to v, $w = \lambda\, k.k(v')$ for expression continuation k. This predicate is written $\mathbf{pk}(v, w)$.

In each step of the transformation from a $v \in V$ to a $w \in K \to A$ from $v \in V$, the equivalence predicate is maintained. The method is an incremental development, of the same nature as we have been modelling in this chapter. We can use the stipulations we make about refinement to verify that the method is acceptable for generating a correct continuation term T' from a direct term T.

- **Correctness**. We define correctness as follows: for two terms T, T', T' is correct with respect to T if the equivalence predicate defined above is satisfied ($\mathbf{pk}(T, T')$).

- **Monotonicity**. The δ function has the following property:

  For term T consisting of subterms T_1, T_2, we write their composition as $T = T_1 \oplus T_2$ for monotonic operation $\oplus$. Then $\delta\, T = \delta(T_1 \oplus T_2) = \delta(T_1) \oplus \delta(T_2)$.

  This property is necessary for the "pushing" method of term derivation, and also gives us the property of preservation of monotonicity we need.

- **Refinement relation**. We can define a refinement relation as follows:

  For some terms v in the direct domain and w in the continuation domain such that $\mathbf{pk}(v, w)$, if w' is obtained from w by transformation operations and $\mathbf{pk}(v, w')$, then w is refined by w'.

  This refinement relation maintains correctness and is reflexive (w is obtained from w by the identity transformation) and transitive (since each transformation maintains the predicate).

So the process of transformations outlined above is valid as a refinement method, as it has the necessary properties we have been discussing.

8.7 Summary and Conclusions

In this Chapter we have been able to use the refinement methods we have developed in Chapters 3 and 4 to examine developments in denotational semantics. We have used refinement to reason about the relations between versions of a simple calculator, and shown that the same ideas can be used to examine the foundations of a published approach to developing a continuation semantics definition from a direct one.

Chapters 7 and 8 have dealt in detail with the theme of "explanation as incremental development". The chapters have used different, but related, formalisms. The use of retrieve relations between stages can be compared to the category-theoretical approach of [FM95] in which functors are used to map between objects and morphisms; this could be termed as a generalisation of our approach as we have considered only relations between steps expressed in a single formalism; category theory can be used to relate different formalisms through functors, thus addressing the issue of heterogeneity in specification development. These functors are a more theoretical way of describing relations as we have considered them in this work.

Chapter 9

Summary and Conclusions

The aim of this thesis has been to demonstrate that, in a number of fields in software engineering and more generally in the way people explain things, viewpoints and refinement are a natural and useful way to describe what is going on. We have done this by introducing a number of case studies where explanations are given, and showing in each case that the information given can be encapsulated as a viewpoint, and related to explanations of other parts of a system, sometimes as part of an amalgamation. In this concluding chapter we present a brief summary of where we've been and then draw some conclusions about our approach and areas of further research that may result.

9.1 Summary

9.1.1 Viewpoints can be used to model a number of processes in software engineering

We set out a number of pages ago by investigating the use of viewpoints in software engineering (Chapter 2). This brought to light a wide range of viewpoint techniques, not all of which explicitly used the term "viewpoint" to describe what they were doing. They were included in the survey because they nevertheless used partial descriptions to model, or maintain the development of, a system under consideration. There were different agendas in the use of viewpoints, though most were interested in them as an aid to the elicitation and validation of requirements ([LF91, NMS89, RW91, KS92]). Not all allowed overlap of viewpoints (multiple perspectives) or the use of different notations for different parts of the problem (multiple paradigms). A major conclusion from the survey was that formalising the relations between viewpoints and dealing with inconsistencies were key issues.

9.1.2 Any investigation of refinement and viewpoints needs a formal basis

For this reason we went on (Chapter 3) to consider the notion of a refinement relation between viewpoints as an aid to modelling the relationships between viewpoints. The foundations of refinement theory, and the necessary properties of a refinement relation and specification composition operator, were presented. Conclusions were that before identifying a refinement relation, a definition of correctness of a final product with respect to an initial specification was necessary. Refinement can then be defined in such a way as to ensure correctness is preserved.

Amalgamation of viewpoints can only be reasoned about when those viewpoints are expressed in a language with a defined semantics, in which it is possible to say whether two viewpoints are equivalent.

In Chapter 4 we took an example of a refinement approach, Morgan's refinement calculus, which is built upon the semantic characterisation of a simple programming language (Dijkstra's wp semantics). This was used to illustrate the ideas from Chapter 3.

We also considered situations where a refinement definition might be too strong, and the use of weaker relations such as co-refinement, where under certain conditions B could refine A, on to "compromise" where B behaved mostly as A, provided a customer gave way on some details. This last kind of "improvement", characterised also as "backward refinement", is useful for modelling such occasions as backtracking or amalgamating conflicting viewpoints, but preservation of correctness cannot be relied upon.

9.1.3 Other work on viewpoints can be assessed with this formal basis

Returning to an example from the survey chapter, we considered in more detail the distributed framework of VOSD (Chapter 5), and considered how we might model the relationships between those ViewPoints as refinements, and how the VOSD approach to inconsistency can be modelled using our co-refinement and compromise ideas.

9.1.4 Explanations can be explained with these ideas

Chapter 6 considered, at an introductory level, a number of ideas based on the theme of "explaining how people explain things"; from user manuals through literate programming to version control systems, we demonstrated that viewpoints, amalgamation and refinement can be successfully used to model, explain and reason about what is going on.

9.1.5 Viewpoints can model, and provide guidance for, example developments

The examples of "explanatory development" which followed considered in more detail the ideas outlined in previous chapters, to put some practical flesh on the theoretical bones. In each case a clear notion of refinement and a composition operation, both conforming to the properties set out in Chapter 3, were necessary to be presented before going on with the development.

- Chapter 7 provided a detailed investigation of amalgamations and used the ideas of refinement developed to verify that the new version correctly represented all the information collected.
- Chapter 8 dealt with the development of semantic descriptions by first identifying a viewpoint describing the initial information, encapsulating the new information as a new viewpoint, and forming an amalgamation of the two to give the full state of information at the new stage.

These chapters demonstrated that, for specification development of this kind, viewpoints and refinement could be used to good effect to describe what goes on. They also showed that, as long as certain mathematical properties were held by the development method (monotonic operators, defined notion of correctness), viewpoints and refinement provide a systematic approach to proving that derived or amalgamated viewpoints have the properties we require of them.

9.2 Conclusions

Our aim as stated in the Introduction was twofold: to demonstrate the use of viewpoints in everyday explanations (*Viewpoints in Practice*); and to show how the formal theory of refinement, co-refinement and amalgamation could be used to model and validate these explanations (*Explanations explained*). We first needed to define what we meant by a viewpoint, which we have tried to do in general terms so as to encompass, as far as possible, the differing perspectives many people have on what constitutes a viewpoint.

9.2.1 Viewpoints in Practice

The conclusion of Chapters 2, 5 and 6 is that the concept of viewpoints, in some form, are used in many situations. They are utilised in models and frameworks to aid the requirements, specification and/or coding process, in validation and elicitation; but they can also be used to describe other, more disparate fields.

9.2.2 Explanations Explained

The use of amalgamation, and the exploration of the relationships between viewpoints with refinement, has been dealt with at some length in the remaining chapters. We have concentrated on using a refinement relation appropriate to the viewpoints being considered; thus for the algebraic specification style used for the Toolbox Event Manager, a relation based on conservative extensions was appropriate. For denotational semantics a relation based on the equivalence of languages implementing a semantic description was used. The relations were shown to be pre-orders, preserving correctness and monotonicity, and so were adequate for our purposes.

By contrast a relation based on compromise, which is a pre-order but does not preserve correctness, is not sufficient as a refinement relation. This does not prevent us reasoning about compromise developments such as backtracking and removal of inconsistencies — but it shows us that we must not rely on the correctness or composability of a "compromised" viewpoint.

9.2.3 Backward Refinement and Modularity

This issue of compromise or "backward refinement" is likely to be worthy of further study. The research into viewpoints initiated at Bath is part of a long-term interest in modularity, exemplified in key papers by Parnas [Par72b, Par72a]. Chapter 2 raised the subject of elaboration of such modules [Fea89a, Fea89b], from which much of the present work stems. The need to backtrack in such a development, inevitable in real-life developments, justifies further investigation of ways to express formally the relationship between steps in the development.

9.2.4 Tools

There are a number of CASE tools available for specification development, in addition to those cited in Chapter 2, which provide an integrated environment for developing and maintaining multiple versions of systems. Statemate [HLN+90], a tool used to produce and maintain different graphical representations of a system (statecharts, data-flow diagrams and activity charts), is an example of this. Tool support for viewpoints is provided by the experimental systems of VOSD and VOA [KS92], which as we have seen (Sections 2.3.6 and 2.3.8) provide an environment for the development of viewpoints. It would be useful to provide an addition to the set of tools which used a theorem prover (such as HOL [Gor87] or its commercial incarnation, ICL Proofpower [Jon92]) for investigation and verification of relations between developed viewpoints.

9.3 Future research

This thesis has identified some issues which could be investigated further.

- More varieties of refinement exist than we have considered here: for example the failures-divergences model of CSP [Hoa85], in which refinement holds between two specifications if the failures and divergences (situations leading to non-termination) have been reduced. The application of ideas such as co-refinement and compromise to such notions of refinement could be investigated in a similar way to the approach used here.
- We have noted that a development in which a new version is more deterministic than the old one might, in some circumstances (such as safety or security critical applications) be regarded as an incorrect development. In this thesis we used an explicit definition of correctness in which removing non-determinism was permitted, but further examples in which this would not be the case could be interesting: refinement would be limited to an equivalence relation.
- There are other fields in which an ordering relation is induced as part of a development. For example in object-oriented modelling, an object decomposition produces a model of an entity in terms of generalised classes, such as a hierarchy that divides *Living Things* into *Mammals* and *Non-mammals* and goes on down to humans. At each level of the hierarchy there is a relation, usually called "is-a" – a human is a mammal and a mammal is a living thing, so a human is a living thing. The is-a relation is a pre-order, but object modelling provides other kinds of relation which are unlikely to be pre-orders, such as "part-of" and general associations. There are roles for viewpoints in this field: formalisation of relations aids in reasoning about inheritance, for example. Although approaches such as VOSD use an object-based model for their viewpoints, the presence of

an ordering being classes (and between classes and their instances) suggests that refinement could be exploited further.

- Chapter 6 introduced a number of subjects which encroached on other territories, such as Artificial Intelligence, knowledge representations and natural language understanding. These were only dealt with on a superficial level: there is a great deal of scope for identifying how these fields approach the issues of conflict and inconsistency in accumulated knowledge, and how this knowledge is represented, in order to use viewpoints and refinement to model the process.

We conclude that explanations have, to some extent, been explained as set out in the aims of this thesis, and the problem areas of backtracking and compromise have been identified as worthy of further research. Interest in viewpoints is increasing at the present time in a manner that threatens to render the survey in Chapter 2 obsolete: it is hoped that the material presented here will contribute to the further development of these viewpoint methodologies.

To begin with it had been too stark, too crazy, too much what the man in the street would have said "Well, I could have told you that" about. Then some phrases like 'Interactive Subjectivity Framework' were invented, and everyone was able to relax and get on with it.

Douglas Adams, Life the Universe and Everything

Bibliography

[ACGW93] M Ainsworth, AH Cruickshank, LJ Groves, and PJL Wallis. Formal specification via viewpoints. In J Hosking, editor, *Proc. 13th New Zealand Computer Society Conference*, pages 218–237. New Zealand Computer Society, 18–20 August 1993.

[AF89] JS Anderson and S Fickas. A proposed perspective shift: viewing specification design as a planning problem. In *Proc. 5th Int. Workshop on Software Specification and Design*, pages 177–184, Los Alamitos, CA, 1989. IEEE Comp. Soc. Press.

[Ain95] M Ainsworth. *Viewpoints and Refinement: A formal basis for viewpoint amalgamation using refinement techniques.* PhD thesis, University of Bath, 1995.

[ALN+91] J-R Abrial, MKO Lee, DS Neilson, PN Scharbach, and IH Sorenson. The B-Method. In *VDM '91: Formal Software Development Methods. LNCS 552*, pages 398–405. Springer-Verlag, 1991.

[AO90] Adrian Avenarius and Siegfried Oppermann. FWEB: A literate programming system for Fortran 8X. *ACM SIGPLAN Notices*, 25(1):52–58, 1990.

[ARW96] M Ainsworth, S Riddle, and PJL Wallis. Formal validation of viewpoint specifications. *Software Engineering Journal*, pages 58–66, January 1996.

[AW94] M Ainsworth and PJL Wallis. Co-refinement. In D Till, editor, *Proc. 6th BCS-FACS Refinement Workshop*, pages 151–166, City University, London, 5th–7th January 1994. Springer-Verlag.

[Bac78] R-J Back. On the correctness of refinement steps in program development. Technical Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[Bac88] R-J Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[BC90] Marcus E. Brown and Bart Childs. An interactive environment for literate programming. *Journal of Structured Programming*, 11(1):11–25, 1990.

[BCG$^+$89] CT Burton, SJ Cook, S Gikas, JR Rowson, and ST Sommerville. Specifying the Apple Macintosh^{TM} Toolbox Event Manager. *Formal Aspects of Computing*, 1:147 – 171, 1989.

[BG86] RM Burstall and JA Goguen. An informal introduction to specifications using CLEAR. In N Gehani and AD McGettrick, editors, *Software Specification Techniques*, pages 363–389. Addison-Wesley, 1986.

[BG92] Judy M. Bishop and Kevin M. Gregson. Literate programming and the LIPED environment. *Journal of Structured Programming*, 13(1):23–34, 1992.

[BH95] P Besnard and A Hunter. Quasi-classical logic: Non-trivializable classical reasoning from inconsistent information. Technical report, Department of Computing, Imperial College, London, UK, 1995.

[BLN86] C Batini, M Lenzerini, and AB Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[Daw91] J Dawes. *The VDM-SL reference guide.* Pitman Publishing, 1991.

[Dij72] EW Dijkstra. Notes on structured programming. In OJ Dahl, EW Dijkstra, and CAR Hoare, editors, *Structured programming*, pages 1–72. Academic Press, 1972.

[Dij76] EW Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Eas92] S Easterbrook. Domain modelling with hierarchies of alternative viewpoints. Cognitive Science Research Paper CSRP 252, University of Sussex, 1992.

[Ehr82] H D Ehrich. On the theory of specification, implementation and parameterisation of abstract data types. *Journal of the ACM*, 29(1):206–227, January 1982.

[EN94] S Easterbrook and B Nuseibeh. Managing inconsistencies in an evolving specification. Technical report, Imperial College Dept of Computing, August 1994. Submitted for publication.

[EN95] S Easterbrook and B Nuseibeh. Managing inconsistencies in an evolving specification. In *Proc. 2nd International Symposium on Requirements Engineering (RE '95)*, pages 48–55, York, UK, March 1995. IEEE CS Press.

[Fea89a] MS Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198–208, 1989.

[Fea89b] MS Feather. Detecting interference when merging specification evolutions. In *Proc. 5th Int. Workshop on Software Specification and Design*, pages 169–176, Los Alamitos, CA, 1989. IEEE Comp. Soc. Press.

[FF89] A Finkelstein and H Fuks. Multi-party specification. In *Proc. 5th Int. Workshop on Software Specification and Design*, pages 185–195, 1989.

[FGH+93] A Finkelstein, D Gabbay, A Hunter, J Kramer, and B Nuseibeh. Inconsistency handling in multi-perspective specifications. In *Proc. 4th European Software Engineering Conference*, Garmisch, Germany, 13–17th September 1993. Springer-Verlag. Also Technical report no. Doc 93/2.

[Fin88] A Finkelstein. Re-use of formatted requirements specifications. *Software Engineering Journal*, 3(5):186–198, 1988.

[FJ90] JS Fitzgerald and CB Jones. Modularizing the formal of a database system. In D Bjorner, CAR Hoare, and H Langmaack, editors, *VDM '90: VDM and Z – formal methods in software development. LNCS 428*. Springer-Verlag, 1990. Also University of Machester Technical Report UMCS-90-1-1.

[FKN+92] A Finkelstein, J Kramer, B Nuseibeh, L Finkelstein, and M Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.

[FM95] JL Fiadeiro and T Maibaum. Interconnecting formalisms: Supporting modularity, reuse and incrementality, 1995.

[FN88] S Fickas and P Nagarajan. Critiquing software specifications. *IEEE Software*, pages 37–47, November 1988.

[GHW85] JV Guttag, JJ Horning, and JM Wing. The Larch family of specification languages. *IEEE Software*, 2(4), 1985.

[Gol83] NM Goldman. Three dimensions of design development. In *Proc. 3rd Nat. Conf. Artificial Intelligence*, pages 130–133, Washington D.C., Aug 1983.

[Gor79] Michael J C Gordon. *The denotational description of programming languages*. Springer-Verlag, 1979.

[Gor87] Michael J C Gordon. HOL: A proof generating system for higher-order logic. In G Birtwistle and PA Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. 1987.

[Gri81] D Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[Gro92] The RAISE Language Group. *The RAISE specification language*. BCS Practitioner Series. Prentice-Hall International, 1992.

[GW91] E. M. Gurari and J. Wu. A WYSIWYG literate programming system (preliminary report). In *1991 ACM Computer Science Conference: March 5–7, 1991, San Antonio Convention Center, San Antonio, Texas: Proceedings: Preparing for the 21st Century*, pages 94–104. ACM, 1991.

[HJN93] IJ Hayes, CB Jones, and JE Nicholls. Understanding the differences between VDM and Z. Technical Report UMCS-93-8-1, Department of Computer Science, University of Manchester, United Kingdom, 1993.

[HLN+90] D Harel, H Lachover, A Naamad, A Pnueli, M Politi, R Sherman, A Shtull-trauring, and M Trakhtenbrot. Statemate — a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[HN95] A Hunter and B Nuseibeh. Managing inconsistent specifications: Reasoning, analysis and action. Technical Report 95/15, Department of Computing, Imperial College, London SW7 2BZ, UK, October 1995.

[Hoa72] CAR Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hoa85] CAR Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[JFH92] WL Johnson, M Feather, and DR Harris. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869, 1992.

[JLM+94] DT Jordan, CJ Locke, JA McDermid, CE Parker, BAP Sharp, and I Toyn. Literate formal development of Ada from Z for safety critical applications. In V Maggioli, editor, *SAFECOMP '94. 13th International Conference on Computer Safety, Reliability and Security*, pages 1–10, Anaheim, California, USA, October 1994. European Workshop on Industrial Computer Systems Technical Committee 7.

[Jon90] CB Jones. *Systematic Software Development using VDM.* International Series in Computer Science. Prentice-Hall, second edition, 1990.

[Jon92] RB Jones. Methods and tools for the verification of critical properties. In R Shaw, editor, *Proc. 5th BCS-FACS refinement workshop*. Springer-Verlag, 1992.

[JZ93] M Jackson and P Zave. Domain descriptions. In *Proc. IEEE International Symposium on Requirements Engineering*, pages 56 – 64. IEEE Computer Society Press, 1993.

[Knu92] DE Knuth. *Literate Programming.* Stanford University, Center for the Study of Language and Information, 1992.

[KS92] G Kotonya and I Sommerville. Viewpoints for requirements definition. *Software Engineering Journal*, 7(6):375–387, 1992.

[KS94] G Kotonya and I Sommerville. Integrating safety analysis and requirements engineering. Technical Report SE/3/1994, Lancaster University Computing Department, LANCASTER, LA1 4YR, UK, 1994.

[LF91] JCSP Leite and PA Freeman. Requirements validation through viewpoint resolution. *IEEE Transactions on Software Engineering*, 17(12):1253–1269, December 1991.

[LFKN95] U Leonhardt, A Finkelstein, J Kramer, and B Nuseibeh. Decentralised process enactment in a multi-perspective development environment. In *Proc. 17th International Conference on Software Engineering (ICSE-17)*, pages 255–264, Seattle, USA, 1995. IEEE CS Press.

[LLC91] SE Lander, VR Lesser, and ME Connell. Knowledge-based conflict resolution for cooperation among expert agents. In *Computer-Aided Cooperative Product Development. LNCS 492*, pages 253–268. Springer-Verlag, 1991.

[Mai91] N Maiden. Analogy as a paradigm for specification reuse. *Software Engineering Journal*, 6(1):3–16, 1991.

[Mor87] JM Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[Mor88a] CC Morgan. Auxiliary variables in data refinement. *Information Processing Letters*, 1988. Reprinted in [MRG88].

[Mor88b] CC Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1):17–27, 1988. Reprinted in [MRG88].

[Mor88c] CC Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988. Reprinted in [MRG88].

[Mor93] CC Morgan. The refinement calculus, and literate development. In B Moller, H Partsch, and S Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 161–182. Springer Verlag, 1993.

[Mor94] CC Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice-Hall, second edition, 1994.

[MRG88] CC Morgan, K Robinson, and P Gardiner. On the refinement calculus. Technical Monograph PRG-70, Oxford University Computing Laboratory Programming Research Group, 1988.

[Mul79] GP Mullery. CORE — a method for controlled requirements specification. In *Proc. 4th Int Conference on Software Engineering*, pages 126–135, 1979.

[Mul84] GP Mullery. Acquisition — environment. In M Paul and HJ Siegert, editors, *Distributed Systems — Methods and Tools for Specification. LNCS 190*, chapter 3, pages 45–130. Springer-Verlag, 1984.

[NFK94] B Nuseibeh, A Finkelstein, and J Kramer. Method engineering for multi-perspective software development. Information and Software Technology Journal, 1994.

[NKF94] B Nuseibeh, J Kramer, and A Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994.

[NMS89] C Niskier, T Maibaum, and D Schwabe. A look through PRISMA: Towards pluralistic knowledge-based environments for software specification acquisition. In *Proc. 5th Int. Workshop on Software Specification and Design*, pages 128–136, Los Alamitos, CA, 1989. IEEE Computer Soc. Press.

[Par72a] DL Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[Par72b] DL Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330 – 336, 1972.

[Pep91] P. Pepper. Literate program derivation: a case study. In M. Broy and M. Wirsing, editors, *Methods of programming. Selected papers on the CIP-Project*, pages 101–124. 1991.

[PST91] B Potter, J Sinclair, and D Till. *An Introduction to Formal Specification and Z*. International Series in Computer Science. Prentice-Hall International (UK) Ltd, United Kingdom, 1991.

[PW93] J Plaice and WW Wadge. A new approach to version control. *IEEE Transactions on Software Engineering*, 19(3):268–276, 1993.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[RBP+91] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorenson. *Object-Oriented Modelling and Design*. Prentice-Hall International Inc., New Jersey, 1991.

[Rob89] WN Robinson. Integrating multiple specifications using domain goals. In *Proc. 5th Int. Workshop on Software Specification and Design*, pages 219–225, 1989.

[RW91] HB Reubenstein and RC Waters. The Requirements Apprentice: automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, 1991.

[Sch86] DA Schmidt. *Denotational semantics: a methodology for language development*. Allyn and Bacon, 1986.

[SEJ96] Software engineering journal special issue: Viewpoints for software engineering, January 1996.

[Sen92] C. T. Sennett. Demonstrating the compliance of Ada programs with Z specifications. In *Proc. 5th BCS-FACS Refinement Workshop*, pages 70–87, 1992.

[Som96] I Sommerville. *Software Engineering.* Addison-Wesley, fifth edition, 1996.

[Spi89] JM Spivey. *The Z Notation - A Reference Manual.* Prentice-Hall, 1989.

[SS89] JJ Shilling and PF Sweeney. Three steps to views — extending the object-oriented paradigm. *SIGPLAN Notices*, 24(10):353–361, 1989.

[ST80] R Sethi and A Tang. Constructing call-by-value continuation semantics. *Journal of the ACM*, 27(3):580–597, July 1980.

[ST95] D Sannella and A Tarlecki. Model-theoretic foundations for program development: basic concepts and motivation. Submitted for journal publication, March 1995.

[Sto92] DA Stokes. Towards a formal specification of revisable CORE: allowing for change. *Software Engineering Journal*, 7(6):393–408, 1992.

[Ten76] RD Tennant. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, 1976.

[Tic85] WF Tichy. RCS — a system for version control. *Software–Practice and Experience*, 15(7):637–654, July 1985.

[Wal92] PJL Wallis. A new approach to modular formal description. Technical Report 92-57, University of Bath, 1992.

[Wal93] PJL Wallis. Looking at building blocks: An experiment in component description. In *Second International Workshop on Software Reusability, Lucca, 1993: Position Paper Collection*, 1993.

[War93] N Ward. Adding specification constructors to the refinement calculus. In JCP Woodcock and PG Larsen, editors, *FME'93: Industrial-Strength Formal Methods. LNCS 670*, pages 652–670, Odense, Denmark, 1993. Formal Methods Europe, Springer-Verlag.

[Weg72] P Wegner. The Vienna Definition Language. *Computing Surveys*, 4:5–63, 1972.

[Win88] J Wing. A study of 12 specifications of the library problem. *IEEE Software*, 5:66–76, 1988.

[Wir71] N Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221 – 227, 1971.

[Woo89] JCP Woodcock. Structuring specifications in Z. *Software Engineering Journal*, 4(1):51–66, 1989.

[ZJ93] P Zave and M Jackson. Conjunction as composition. *Transactions on Software Engineering and Methodology*, 2(4):379–411, 1993.