Imperial College of Science, Technology and Medicine
University of London
Department of Computing

A Multi-Perspective Framework for Method Integration

Bashar Ahmad Nuseibeh

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering of the University of London, and for the Diploma of Imperial College of Science, Technology and Medicine.

October 1994

To my parents, Ahmad & Norma,

for making this possible ...

Abstract

The development of large and complex systems necessarily involves many people - each with their own perspective on the system, defined by their skills, responsibilities and expertise. This is particularly true for composite systems, that is, systems which deploy multiple component technologies. The intersections between perspectives however, are far from obvious because the knowledge within each is represented in different ways. Furthermore, because development may be carried out concurrently by those involved, different perspectives may be at different stages of elaboration, and may be subject to different development strategies. The problem of how to guide and organise systems development in this setting, we term the "multiple perspectives problem".

This thesis focuses on the specification and design of software-based systems. In particular, it addresses the problem of method integration in this setting, which is viewed as an instance of the multiple perspectives problem.

The thesis elaborates a software development framework based on loosely coupled, locally managed, distributable objects, called "ViewPoints", which encapsulate partial representation, process and specification knowledge about a system and its domain. ViewPoint templates, denoting ViewPoint types, are introduced, and are used as the building blocks of software development methods. Such methods, and the ViewPoints that are created as a result of deploying them, are integrated via pairwise inter-ViewPoint rules, which express the relationships that must hold between ViewPoints in order to achieve consistency.

The thesis proposes a model of ViewPoint interaction in which inter-ViewPoint rules are defined during method design, and invoked and applied during method deployment. Fine-grain process modelling is used as the means of providing local method guidance and of coordinating ViewPoint interaction. In the spirit of multi-perspective development, inconsistencies are tolerated and managed by prescribing rules that specify how to act in the presence of these inconsistencies.

This thesis makes a novel contribution to the organisation and management of multi-perspective software development, and illustrates the role of method engineering, process modelling and consistency management in this setting. The entire ViewPoints framework is supported by a prototype environment, *The Viewer*, implemented to demonstrate the scope and feasibility of the approach.

Preface

Statement of Contribution

Working as part of the Distributed Software Engineering (DSE) Group at Imperial College means working in a team. Consequently, much of the work undertaken in the group is either inspired by, discussed or co-produced with colleagues in the group. This has made the task of distinguishing the author's distinct contribution somewhat more difficult. The work described in this thesis was performed during the author's participation in a number of UK (SERC) and European (Eureka) funded projects. Whereas a large proportion of the work in these projects was collaborative, this thesis describes those aspects to which the author made a distinct contribution. Where necessary, joint work which influenced the direction of the research, is also discussed. Where work was joint or attributable to other researchers, it is appropriately cited between square brackets.

Prior to this thesis, the author produced the following related publications. The ViewPoints framework and preliminary research agenda were described in [Finkelstein et al. 1992a]. The computer-based prototype support environment, *The Viewer*, and associated tools were described in [Nuseibeh & Finkelstein 1992]. A preliminary model of ViewPoint interaction and associated notation for expressing inter-ViewPoint rules were described in [Nuseibeh, Kramer & Finkelstein 1993], and subsequently refined in [Nuseibeh, Kramer & Finkelstein 1994]. The work on inconsistency handling within the ViewPoints framework was described in [Finkelstein et al. 1993] and [Finkelstein et al. 1994], and an exploration of its relationship to process modelling and method guidance appeared in [Nuseibeh, Finkelstein & Kramer 1993] and [Easterbrook et al. 1994]. An account of method engineering and integration in the ViewPoints framework appeared in [Nuseibeh, Finkelstein & Kramer 1994]. Finally, the systems engineering experiences deployed in the examples and case studies described in the thesis first appeared in [Finkelstein et al. 1992d; Finkelstein et al. 1992c], and were later summarised and evaluated in [Finkelstein et al. 1992b]. In all cases, this thesis should be regarded as the definitive account of the work.

Acknowledgements

The only piece of advice from my supervisor, Anthony Finkelstein, that I have chosen to ignore, is his recommendation that I keep the acknowledgements section of this thesis short. I cannot with a clear conscience fail to acknowledge with gratitude the numerous individuals who have collectively shaped this thesis.

This thesis, quite simply, could not have been written without the supervision, guidance and support of Anthony Finkelstein and Jeff Kramer.

As my supervisor, Anthony provided me with the inspiration, encouragement and freedom to pursue my, occasionally bizarre, ideas. As my friend, he has been unfailingly supportive, patient and generous. He has always been ready to listen and advise me on my endless technical and social dilemmas - and for that I am deeply grateful.

For over three years, Jeff consistently provided me with constructive and insightful feedback on my work. He meticulously reviewed endless reports, papers and drafts of this thesis, and effectively treated my bouts of verbal diarrhoea!

Other members of the Distributed Software Engineering Group have also been unfailingly supportive. I would like to thank Keng Ng for his time-saving Mac tips; Steve Crane and Kevin Twidle for guiding me through the Unix maze; and the other members of the group - Naranker Dulay, Sue Eisenbach, Manny Lehman, Jeff Magee and Morris Sloman - who were reassuringly confident that I would write this thesis long before I even settled on a title!

Many of the ideas behind the ViewPoints framework originated during Michael Goedicke's visit to Imperial in 1989. Since then, Michael has consistently contributed to the work, and this thesis has benefited greatly from his feedback.

I would also like to thank Steve Easterbrook for our numerous discussions that clarified the distinctions between inconsistencies and conflicts, and for rigorously testing my method integration notation with examples of his own. I am also indebted to Tony Hunter for guiding me through the literature on logic, including the action-based temporal logic used in chapter 7.

Throughout my stay at Imperial, my fellow students, research assistants and office-mates have provided me with a comfortingly supportive environment. I am particularly grateful to Shi Ching Cheung, Olly Gotel, Stephen Morris and Kostas Karagianidis for their feedback on my work.

Jonathan Moffett deserves special thanks. By the time he left Imperial College in 1992, he had established himself, in the eyes of many Ph.D. students, as a "shadow supervisor"! I could not conceive of submitting my thesis before "Jonathan had had a look at it"! I would like to thank him for meticulously reviewing a complete draft of the beast on short notice.

The Viewer tool, described in chapter 8, was an evolutionary prototype. The feedback from audiences who sat through my early demos was invaluable. The experimental enhancements made to *The Viewer* by past students Alvaro Ballesteros, Fui Kien Lai, Ulf Leonhardt and Thanwadee ("Pomme") Thanitsukkarn were especially helpful.

Anne O'Neill provided me with advice on the Ph.D. procedures at Imperial College, and I am particularly grateful to her for steadying my nerves at times of "crisis". The cheerfulness of Tracy Banton never wavered as she was transformed (through marriage) into Tracy Scott. I thank her for her secretarial support.

I am grateful to my Ph.D. examiners, Pat Hall and Andrew Life, for their valuable suggestions on improving the thesis and developing the research.

My friends at the Imperial College Dance Club made sure I maintained an "upright posture" and high spirits through the long months of thesis-writing, and I am grateful to them for their encouragement and companionship.

Last but not least, the support of my family has sustained me throughout the duration of my Ph.D. Rania, Mustapha, Serena, Bana and Rawan have all, in different ways, contributed to this thesis. My father has never understood why I would want to engage in "smalltalk" for so many years, but encouraged me to do so all the same. My physical well-being was regimentally monitored by my mother who insisted that I simply could not write a Ph.D. thesis on an empty stomach! Thanks Mum and Dad.

Financial Support

The work presented in this thesis was partly funded by:

- *The UK Science and Engineering Research Council (SERC)* as part of its **S**oftware **E**ngineering and **E**ngineering **D**esign (SEED) project.
- *The UK Engineering and Physical Sciences Research Council (EPSRC)* as part of its **V**iewPoint-**O**riented **I**nconsistency Handling using **L**ogic and **A**ctions (VOILA) project.
- *The UK Department of Trade and Industry (DTI) and the Commission of European Communities (CEC)* as part of the Advanced Technology Programme (ATP) of the **E**ureka **S**oftware **F**actory (ESF) project (grant reference number IED4/410/36/002).
- *The European Strategic Programme for Research and Development in Information Technology (ESPRIT)* as part of its Basic Research Action (BRA) Working Group (WG) on **Pro**cess **Mo**delling **Te**chniques: Basic **R**esearch (PROMOTER) project.

The views and opinions expressed in this thesis however, are not necessarily those held by any of the above organisations or individuals, and any errors are, of course, entirely the responsibility of the author.

Table of Contents

Chapter 3 Separating and Integrating Concerns 47

Chapter 4 The ViewPoints Framework 75

Chapter 1 Introduction

With the so-called “software crisis” a painful reality to many firms in the computer industry, there has hardly been a more urgent need to produce high quality software to a deadline. To this end, computer scientists have provided a basket of powerful computer languages and techniques, facilitating the development of modular, maintainable and efficient code. Software engineers on the other hand, have in many ways, attempted to emulate their hardware counterparts, by providing the discipline and formalism so often lacking in large software development projects. This is a thesis in software engineering. With the ever-increasing penetration of complex computer systems into the daily functioning of society, software engineering has emerged as a crucial discipline for guiding the development of such systems.

Complex systems typically deploy multiple technologies and require multiple participants for their development. Thus, while systematic and structured approaches are necessary for the development of such systems, the inevitable proliferation of different views and perspectives on the problem and solution domains also needs to be supported. These views, are invariably expressed and elaborated using different notations and development strategies respectively.

This thesis is about managing complexity, separation of concerns and integrated, systematic software development. It illustrates the complexities of software development in the form of the “multiple perspectives problem”; it proposes a framework within which this problem may be addressed; and attempts to reconcile the desirable separation of concerns provided by the framework with the integration required for systematic software development. This reconciliation is achieved by allowing multiple perspectives during development, while ensuring that the methods by which these perspectives are developed are integrated. In this context then, the central contribution of the thesis is in the area of method integration.

This chapter outlines the background and motivation behind the work, and defines the objectives of the research. A “road map” of the thesis is also sketched.

1.1. Background and Motivation

Large, complex, composite systems [Feather 1987; Doerry et al. 1991] typically deploy multiple technologies and employ multiple, interacting participants for their development. As with any large development exercise, the development of a complex system must be systematic and structured in order to manage this complexity, and in order to facilitate the future maintenance and inevitable evolution of the system [Lehman & Belady 1985].

A software engineering method prescribes development actions and heuristics for deploying one or more notations for specifying a complex system [Tontsch 1990]. A software engineering method however, is more than simply a set of notations and associated procedures for their deployment. Karam and Casselman [Karam & Casselman 1993] for example suggest at least three kinds of properties which may be used to characterise and evaluate a software development method. Technical properties refer to those aspects of a method that relate to its philosophy, life-cycle coverage, structure and features (such as notations, procedures, measures and guidelines). Usage properties refer to those aspects of a method that express its applicability within an organisation and its suitability for a particular purpose. These include the availability of automated tool support. Finally, managerial properties are concerned with a method's support for software development management practices, such as cost estimation, project planning and staffing.

Clearly then, a software development method is a large and complex object, and its suitability for any particular purpose is very much dependent on numerous generic and domain-specific characteristics [Wasserman, Freeman & Porcella 1983]. The study of methods, such as their characterisation or cataloguing, may be termed "methodology engineering", and is distinguished from "method engineering" which is concerned with the design and construction of specific methods [Mullery & Newbury 1991]. Vivarès and Durieux use the term "method modelling" in presenting a framework for modelling software development methods, and develop a formal language for describing such methods [Vivarès & Durieux 1992]. Such formalisation of a software development method may have useful applications such as automated verification, recording of design rationales and reuse.

It is generally not sufficient however, to adopt a software engineering method in order to successfully develop a large and complex system. To use a method effectively it must be supported by computer-based tools, to help automate and support various software development activities prescribed by that method. Computer-Aided Software Engineering (CASE) tools [Gibson 1989] have traditionally provided automated support for methods in the form of notation editors, consistency checkers, code generators and so on [Martin 1988; Forte & Norman 1992]. They have not however, been able to combine this kind of support, with automated support for the software development *process* [Osterweil 1987]. In particular, CASE tools have not, as a rule, been able to provide software developers with "method guidance" [Finkelstein & Kramer 1991; Finkelstein, Kramer & Hales 1992]; that is, guidance as to when and under what circumstances it is appropriate to perform any of a method's steps. Moreover, those tools which *have* had a model of the software development process built into them, have been difficult to adapt to customised or evolving development processes [Lehman 1994] that inevitably exist in many organisations. To make them more “flexible”, many commercial CASE tools have been reduced to being drafting tools, with some syntactic consistency checking built into them [Digital 1991; IDE 1991; Rational 1992].

The complexity of heterogeneous, composite systems - that is, systems deploying multiple technologies and development participants - inevitably requires the decomposition of both problem and solution domains. On the one hand, different development participants may address different aspects of a system under development. Moreover, the views or perspectives held by the different participants on areas of intersection or overlap may also be different. On the other hand, the very existence of different participants frequently implies that they express their areas of concern in different languages and deploy different strategies to do so. This compounds the problem of trying to check the consistency between the different views or perspectives held by the participants.

While an approach that is intolerant of inconsistencies and multiple perspectives may be adopted (and *is* adopted by many organisations that wish to enforce a disciplined development policy), there appears to be mounting evidence that such an approach is not realistic, and that software developers prefer to work with multiple views [Meyers & Reiss 1992; Zahniser 1993] and languages [Petre & Winder 1988] in which inconsistency is tolerated [Balzer 1991]. In fact even in the database world, where traditionally the emphasis has been on constructing universal meta-models or schemas for a database, the need for multiple views and schemas has been acknowledged and addressed [Dayal & Hwang 1984], heterogeneous multidatabases have been explored [Ram 1991; Bright, Hurson & Pakzad 1992], and techniques for schema and view integration have been proposed [Batini, Lenzerini & Navathe 1986; Navathe, Elmasri & Larson 1986; Bouzeghoub & Comyn-Wattiau 1991].

Finally, it is no longer realistic to treat software development as a centralised, sequential set of activities. More often than not, it is a collaborative effort by many development participants, working concurrently in a distributed environment. Thus a framework within which such development takes place must be capable of addressing issues of computer-supported cooperative work (CSCW) [ACM 1992], distributed systems [Mullender 1989] and concurrency [Milner 1989].

1.2. Problem Definition and Objectives

The scenario described above in which multiple participants hold multiple views on the software system they are developing, may be termed “multi-perspective software development”. Software engineering methods designed to support such multi-perspective development must be able to handle the multiplicity of participants, views, development strategies and notations, in such a way that the resultant specification fragments are loosely-coupled yet coherent. This can be achieved by ensuring that the one or more methods used to develop such multi-perspective specifications or systems are integrated. The requirement then, is for effective *method integration* for multi-perspective software development. Moreover, effective method integration must be supported by effective tool integration. One of the objectives of this thesis is to illustrate that the problems of tool integration are, in fact, a subset of method integration problems, and that with the appropriate framework in place, successful method integration facilitates successful tool integration [Kronlöf, Sheehan & Hallmann 1993].

Multi-perspective software development poses particular challenges to the tool developer. On the one hand, individual developers must be free to choose the tools that are most appropriate for their particular purposes, while on the other hand, these tools must be able to communicate and exchange information effectively. This may require some form of standardisation of the exchanged information formats and communication protocols between, at least, any two communicating development participants (or the tools that they deploy). In CASE tools terminology, the consistency relations between partial specifications need to be defined and, when appropriate, these relations need to be checked. The question of how to act in the presence of inconsistency also needs to be addressed. A typical CASE tool in such a setting might flag an error and await some conflict resolution by one or more development participants. This may not be a suitable approach to take if no resolution of the conflicts is immediately apparent.

The *ViewPoints Framework* described in this thesis explicitly addresses the issues raised above. As a generic framework for multi-perspective software development however, it also lays down a detailed agenda for software engineering research. While the framework itself is a novel contribution to software engineering, it specifically identifies the following problems and objectives which are also addressed in this thesis:

1. The need to define or customise software engineering methods for multi-perspective software development. This objective is based on the premise that effective software development requires that development participants adhere to one or more software development methods; and that these methods are constructed or tailored to the particular requirements of the development participants and the organisations they belong to.
2. The need for effective method integration, when more than one method is used to develop a single software system. This in turn requires effective integration of the tools used to support these methods.
3. Method integration within the ViewPoints framework requires:
   i. The ability to express and enact the relationships between multiple ViewPoints. These relationships (which may be used to check consistency between ViewPoints) are effectively the method integration “glue”.
   ii. The ability to describe how to act in the presence of inconsistency. Such inconsistency

handling is an essential step in the *process* of multi-perspective software development, because it is a step towards achieving integration of multiple, inconsistent perspectives.

iii. The ability to model individual development processes deployed by individual development participants. Such "fine-grain" process modelling may then be used to provide effective method guidance.

4. The need to provide computer-based tool support for:

   i. The method engineer, designing and constructing methods;

   ii. The tool developer, developing tools to support the methods designed by the method engineer;

   iii. The method user, deploying methods described by the method engineer, and using tools provided by the tool developer;

   iv. The project manager, monitoring and guiding other method users (noting that the project manager may also be regarded as a method user).

5. The need to demonstrate, via a significant case study, that multi-perspective software development within the ViewPoints framework is feasible and useful.

While this thesis addresses the above objectives within the context of software engineering as a whole, the emphasis, even bias, is on the "earlier" phases of software development such as requirements specification and design. Errors identified during these phases are less expensive to correct in terms of time, effort and money, than those identified during the later stages of development [Boehm 1981]. Moreover, the areas of software specification and design are less mature than, say, implementation and testing, and there is therefore a more urgent need to provide integrated methods to guide software developers during those earlier stages of development.

1.3. Thesis Contribution

This thesis elaborates a framework, supported by tools and populated with techniques and mechanisms, for achieving method integration for multi-perspective software development.

Specifically:

- It describes a framework within which multiple participants, views and methods are supported. This *ViewPoints Framework* [Finkelstein et al. 1992a] applies the "separation of concerns" principle [Ghezzi, Jazayeri & Mandrioli 1991] to reduce many software development complexities by partitioning problem or solution domains into loosely coupled, locally managed, distributable objects called "ViewPoints". Knowledge encapsulated within these ViewPoints is also separated into "slots" which may be used as criteria for selecting or identifying ViewPoints. For example, separation of concerns may be achieved by selecting and developing ViewPoints based on the different representation schemes they employ.

- Within this framework, the notions of method and method integration are defined [Nuseibeh, Finkelstein & Kramer 1994], and the outstanding issues for their achievement are identified and addressed. These include:
  - Expressing the relationships between different ViewPoints [Nuseibeh, Kramer & Finkelstein 1993; Nuseibeh, Kramer & Finkelstein 1994]. These relationships express the *requirements* for method integration. The thesis outlines a simple notation for expressing such relationships between partial specifications residing in different ViewPoints, which is based on pattern-matching between objects, relations, their attributes, types and values.
  - Consistency checking between different ViewPoints [Easterbrook et al. 1994; Nuseibeh 1994]. This is the *implementation* of method integration. Once a relationship between two partial specifications has been defined, then a mechanism and protocol for invoking and applying such a relationship is needed. The thesis outlines a sample communication protocol and associated mechanism that demonstrate consistency checking in the context of distributed development.
  - Inconsistency handling [Finkelstein et al. 1993; Finkelstein et al. 1994]. This is a step that follows consistency checking when multiple perspectives are discovered to be inconsistent. The thesis proposes an *action-based* framework which allows the method engineer to specify how to act in the presence of inconsistency. The role of inconsistency handling for fine-grain process modelling of a ViewPoint-oriented development is also addressed [Nuseibeh, Finkelstein & Kramer 1993].
- Method integration is emphasised as a vehicle for providing integrated tool support and effective development guidance. The thesis demonstrates how inter-ViewPoint rules used for method integration may be used for tool integration. These rules, coupled with the inconsistency handling rules, are also used to provide development method guidance.
- The role of computer-based tool support within the framework is examined [Nuseibeh & Finkelstein 1992]. *The Viewer* environment and associated tools partially demonstrate the feasibility of the ViewPoints approach.
- The applicability of the framework, and associated tools and techniques, is demonstrated via a significant case study and analysis of other systems engineering experiences [Finkelstein et al. 1992b; Finkelstein et al. 1992d; Finkelstein et al. 1992c]. Its feasibility for *systems* as well as software development is also discussed. It is worth noting at this point that the general nature of the ViewPoints framework, and the wide range of issues it addresses, has meant that evaluating it as a whole is not feasible in the context of this thesis. The thesis therefore attempts to demonstrate the applicability of the framework’s constituent contributions using a variety of individual examples. Nevertheless, these examples are drawn from “realistic” software development projects.
- An agenda of outstanding issues for future software engineering research is more clearly defined.

1.4. Thesis Road Map

This thesis is structured along the same lines as the objectives listed in section 1.2.

Chapters 2 and 3 survey related literature which is used to clarify the problem domain and motivate the work.

Chapter 2 discusses the software engineering context of the work described in the thesis, and in particular presents the strengths and weaknesses of current approaches to method-based software development.

Chapter 3 examines in more detail specific approaches to multi-perspective software development, and particularly focuses on those which deploy the notion of views or viewpoints. The chapter also discusses specific frameworks, techniques, protocols and mechanisms for method integration in this setting.

Chapter 4 then presents a detailed description of the ViewPoints framework, within which the problems raised and the objectives identified in chapters 2 and 3 are addressed. In particular, the chapter emphasises the organisational and structuring properties of the framework.

Chapter 5 makes the distinction between method engineering and method use within the framework, and examines the relationship between the two. In particular, it presents an approach to method engineering in which a method is designed and constructed, and then subsequently used in a distributed development setting. While the focus of this chapter is on the activities of method engineering, method use is examined as a vehicle for testing the usability of the methods designed and constructed during the method engineering phase.

Chapters 6 and 7 describe the solutions to three problems arising from the framework, namely, method integration, method guidance and inconsistency handling.

Chapter 6 presents a model for inter-ViewPoint "consistency management" which includes inter-ViewPoint rule expression, invocation and application. The chapter presents a notation for expressing inter-ViewPoint rules, and a protocol and mechanism for rule enaction (invocation and application). Inter-ViewPoint rules are used in this context as a vehicle for method integration within the ViewPoints framework.

Chapter 7 tackles the two issues of process modelling and inconsistency handling. These are addressed under the umbrella of "inconsistency management"; that is, how to develop software systems in general (process modelling), and how act in the presence of inconsistency in particular (inconsistency handling). The use of inconsistency handling rules as part of a ViewPoint's process model that provides method guidance is emphasised.

Chapter 8 then describes *The Viewer,* a computer-based, prototype environment designed and constructed to demonstrate and support the ViewPoints framework. The architecture, implementation and extensions to *The Viewer* are also briefly discussed.

Chapter 9 presents a case study of method engineering and use within the ViewPoints framework. The case study described is partially developed using *The Viewer*. A critical evaluation of the framework and its tool support is then presented.

Finally, **chapter 10** summarises the conclusions and contributions of the thesis and sets an agenda for further work.

Every chapter ends with a chapter summary which also serves to introduce subsequent chapters.

Figure 1.1 below is an overview of the structure and subject matter of the thesis, which may be used to point the reader to particular areas of interest. While the thesis is constructed to be read sequentially from chapter 1 through to 10, chapters 2 and 3 may be skipped if the reader is familiar with related software engineering research, or accepts the brief motivation of the work, presented in chapter 1 and the individual chapter introductions. This is particularly true of chapter 2, which discusses the general software engineering context of the thesis.

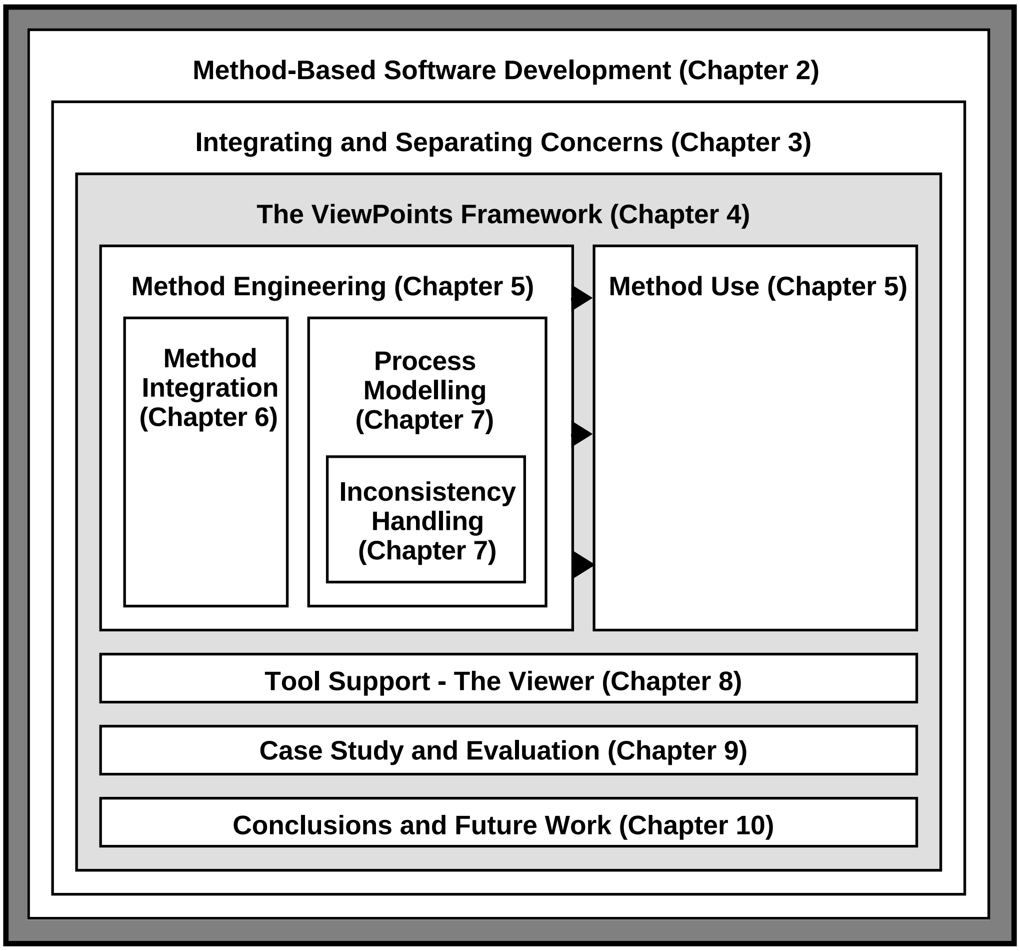

Method-Based Software Development (Chapter 2)
Integrating and Separating Concerns (Chapter 3)
The ViewPoints Framework (Chapter 4)
Method Engineering (Chapter 5)
Method Use (Chapter 5)
Method Integration (Chapter 6)
Process Modelling (Chapter 7)
Inconsistency Handling (Chapter 7)
Tool Support - The Viewer (Chapter 8)
Case Study and Evaluation (Chapter 9)
Conclusions and Future Work (Chapter 10)

Figure 1.1: *Thesis structure and organisation. The subject matter of an "inner" box is within the scope of that of an "outer" box.*

Chapter 2 Method-Based Software Development

This chapter describes the software engineering context of the work presented in this thesis. Alternative software development life cycle models are discussed, and various methods that implement these models are examined. The role of computer-based tool support in this context is also explored. This is followed by an examination of the different kinds of knowledge that are part of, or impact upon, software development. Particular attention is paid to the way this knowledge is traditionally partitioned. The chapter concludes with a summary and a critical evaluation of conventional wisdom in method-based software development.

By necessity, the chapter covers a wide range of software engineering advances and concerns. This is because the ViewPoints framework, described in chapter 4, addresses, or at least touches upon, a number of these issues.

2.1. Methodological Software Engineering

Software engineering is the discipline of software development [Pressman 1992]. This includes development in the early phases of requirements engineering and specification, through to design, implementation and continued maintenance. Software engineering provides and combines *methods* for software development, which in turn provide techniques for specification, implementation, quality assurance, coordination, management and control of development, and effective *automated support* for many of the activities in this process. A model that represents such a development process (its constituent activities and their ordering) is generally termed a software development *life cycle model.*

2.1.1. Life Cycles

It has become almost customary, when introducing the field of software engineering, to describe the stages, steps or phases of the classic software development life cycle model, also known as the "waterfall" model [Royce 1970]. This model identifies the stages of requirements analysis, specification, design, implementation, testing, operation and maintenance, and assumes that they occur in a strict sequential order with well-defined deliverables at the end of every stage. These deliverables include specification documents and implementation code, which are used by managers as milestones for the development projects they are managing.

While the waterfall model provides a useful framework within which to develop many large software systems, it has a number of short-comings that make it an unrealistic representation of the software development process [Jones 1990; Sommerville 1992; van Vliet 1993]. The model assumes that the different steps prescribed are performed to completion before the next step is started. This is not always the case. For example, a customer's requirements may change during the implementation phase, which will require going back to change the specification and the design. Furthermore, while it is useful from a management point of view, to treat the different phases of the software development life cycle as independent and distinct, this is not usually the case in practice. The different, often concurrent, stages in fact overlap and feed information to each other, and this is a continuous, iterative process. Thus, not only is the waterfall model an inaccurate representation of what actually happens during software development, but it may also cause problems if the deliverables from each phase (e.g., the requirements specification) are frozen prematurely [Gladden 1982; McCracken & Jackson 1982].

Several alternatives to the waterfall model exist. *Rapid prototyping* [Luqi 1989] is one such alternative which may be useful in situations where there is an element of uncertainty. The prototype acts as a model of the system to be built, and may be used by a developer to validate customer requirements. It may also be used to test the viability of a proposed system, and may sometimes be treated as a partial specification of that final system. A prototype is typically modified and adapted many times, and may therefore be regarded as a less thorough, iterative walk-through the steps of the classic waterfall model. Moreover, prototypes are sometimes termed "throwaway", since they are (or should be) discarded after they have demonstrated "proof-of-concept".

Another approach to software development is an *incremental development* approach [Brooks 1987] in which a system is developed piece-meal. The essential parts of the system are initially implemented (these often include user interfaces and high level control structures), after which the user can decide on what he or she regards as particularly important aspects of the system. Having produced such an evaluation at an early stage in the development life cycle, the system is incrementally built, ensuring that each additional set of functions are well tested and integrated into the system. Again, the development process includes several iterations of the classic life cycle stages. One problem with this approach is that incorrect decisions may be made early on during development, which subsequently become more difficult to correct.

The *spiral model* of software development [Boehm 1988] attempts to combine the advantages of the three approaches described above, and proposes risk analysis as a unifying principle. An assessment of management risk items is made at regular stages in the life cycle, and this assessment determines whether or not to move on to the next stage. Such an evaluation of risks is achieved by performing further activities such as simulation, prototyping or detailed analysis. The classic life cycle development steps such as requirements analysis, design and testing are still included in the spiral model, and they are performed iteratively between the regular phases of risk analysis.

The different software development life cycle models described above demonstrate that, while sequential, phased, document-driven approaches appear simpler to manage and understand, in reality software development is a more complex process. During this process, a number of activities are performed iteratively (and possibly concurrently). Moreover, the deliverables produced at the end of each iteration may be partially complete and subject to further modification or refinement.

What life cycle models do not describe are the more detailed procedures or guidelines for performing various constituent steps within each stage, the movement from one stage to another, and the production of deliverables. This is achieved by the use of software development methods.

2.1.2. Methods

The complexity of today's software systems requires the use of well-defined, systematic approaches to their development. Software development methods are engineered to fulfil this role.

A software development method (henceforth called a method) prescribes a set of procedures and guidelines for specifying and/or developing a software system. A method may be concerned with all or part of a software development life cycle. Thus, there are requirements engineering methods such as Software Requirements Engineering Methodology (SREM) [Alford 1977; Alford 1985], design methods such as Structured Design (SD) [Yourdon & Constantine 1979], and software development methods that include both analysis and design steps such as the Structured Systems Analysis and Design Method (SSADM) [Ashworth & Goodland 1990]. The term methodology is sometimes used to describe a collection or system of methods [Ghezzi, Jazayeri & Mandrioli 1991], however, in much of the software engineering literature the terms method and methodology are used interchangeably and are often confused. Strictly speaking a software development methodology is the "study or science of methods".

A method also guides software development by prescribing how to deploy one or more notations. These notations are essential for expressing system requirements, design and implementation. They are also used to describe a wide variety of products and documents that are generated by a software development project. In fact, the notion of a method is inextricably linked to that of notations, which are the primary tool for expressing ideas about, and properties of, a system [Iverson 1980]. Most commercial software development methods utilise at least one notation for describing the products that result from applying the method’s procedures and guidelines. A method that does not have one or more notations in its repertoire is more accurately called a software development life cycle model.

A number of methods have been developed over the last 25 years of software engineering, and these have invariably deployed different notations and procedures for specifying and developing software systems. It is not the objective of this chapter to describe any particular software development method(s) in detail. Rather, the salient features and structure of a number of representative methods is considered. These methods are representative in their varied use of notations, prescriptive procedures and guidelines. These features are referred to in later chapters, when issues of method engineering and integration are addressed.

The *Structured Analysis and Design Technique (SADT)* [Ross 1977; Ross & Schoman 1977; Ross 1985] combines a single, rich, highly graphical language, with a systematic and structured way of specifying systems using this language. The language has over 40 features which necessitates the decomposition of specifications into smaller units that are easier to grasp. Structured analysis (SA) [Ross 1977] - the general philosophy and techniques upon which SADT is based - suggests that “... anything worth saying something about, must be expressed in six or fewer pieces” [Ross 1985]. The application of this suggestion in SADT results in a hierarchical, top-down decomposition of a system and its context. A model in SA is a collection of interconnected diagrams, and multiple models are supported. Every model has an orientation, which includes a context, viewpoint and purpose.

Controlled Requirements Expression (CORE) [Mullery 1979; Mullery 1985] on the other hand uses four different diagramming notations and structured text in seven distinct stages of the method, in order to specify and analyse system requirements. The first stage (problem definition) identifies a customer authority, business objectives, current system problems, new system features, possible future expansion, costs and time scales. These are described textually in a structured form.

The next stage (viewpoint structuring) identifies and organises information processing entities (“viewpoints”) in a project. These viewpoints are arranged in a viewpoint hierarchy using a simple graphical notation. Each viewpoint is then analysed in the next stage (tabular collection) so that the sources of inputs and the destination of outputs to and from each action performed by each viewpoint are identified. This information appears in a series of tabular collection diagrams - one for each identified viewpoint. A data structure diagram for each viewpoint is then constructed in the next stage (data structuring), in which the data (inputs and outputs in the previous stage) is ordered. Data structure diagrams are again simple, graphical, hierarchical representations of this information. The next two stages (single and combined viewpoint modelling) bring together information expressed in the preceding two stages. Timing information and internal data flows are also identified during these two graphical stages. Finally, the constraint analysis stage textually lays down the so-called non-functional requirements of the required system. These include performance and reliability requirements, and environmental, economic and political constraints.

All the graphical stages of CORE have structured textual annotations associated with them, and are developed with the aid of recommended guidelines and heuristics. Although the stages are presented sequentially, requirements analysis using CORE is an iterative, typically concurrent, process.

The *Constructive Design Approach (CDA)* [Kramer, Magee & Finkelstein 1990] for the development of distributed systems is a method structured along procedural rather than notation lines. The method aims to guide the development of a distributed system by separating the system structure, as a set of components and their interconnections, from the functional descriptions of individual components. This, its designers argue, facilitates the description, construction and evolution of distributed systems.

The method prescribes five recursive, informal steps which take the system developer from a high level architectural design, to a running distributed system. The method is less dependent on a particular notation than most methods, in that initial design may be laid down very loosely in terms of “components” and their main data flows. Control (synchronisation) between components is introduced during an interface specification step, and this is followed by a component elaboration step in which component types are identified and the functional behaviour of “primitive” components is described in the language of choice. The system configuration is then constructed by instantiation and interconnection of components to form distributable logical nodes. Again, such a configuration may be described in a configuration language of the designers’ choice. Finally, modification and evolution of the system is performed by the replacement or addition of nodes.

Jackson System Design (JSD) [Jackson 1983; Cameron 1989] combines distinct descriptive notations with prescriptive procedures on how to use them. The method combines an object-oriented design approach with functional decomposition, and attempts to address most aspects of the software development life cycle - from analysis to implementation. JSD comprises three distinct stages: a modelling stage, a network stage and an implementation stage. During the modelling stage, graphical (tree-like) process structure diagrams are constructed, in which the real world is analysed and represented in terms of actions (or events) that may affect entities. During the network stage, a (graphical) system specification network is constructed in which an entire system is described as a network of interconnected communicating processes. Finally, in the implementation stage, the network of processes is transformed into a sequential implementation in a programming language of the developers’ choice.

Table 2.1 lists the four methods described above, and tabulates their use of notations and procedures, and the extent to which guidelines or heuristics are provided. The table is not meant as a competitive comparison of the methods, but as an illustration of the different emphasis that these different methods place on the three classification criteria tabulated. These criteria ultimately influence the structure of methods, the way in which they are deployed, and the applications for which they are used. They are adapted from the generic model for representing design methods described in [Potts 1989]. Comparative reviews of methods are abundant in the literature and include [Maddison 1983; Olle, Sol & Tully 1983; Olle et al. 1991] and [De Champeaux & Faure 1992] to name a few.

Method	Notations	Procedures	Guidelines
SADT	Actigrams	Few fixed procedures ("a discipline of thought" [Ross 1985]).	Many heuristics.
CORE	1. Structured text 2. Viewpoint structuring 3. Tabular collection 4. Data structuring 5. Single viewpoint modelling 6. Combined viewpoint modelling 7. Structured text	Documented step-by-step procedures for constructing specifications using notations provided.	Many heuristics, guidelines and examples, but there are ambiguities in the notations and the procedures for deploying them.
CDA	1. Configuration structure 2. Configuration language 3. Programming language(s)	1. Structure & component identification 2. Interface specification 3. Component elaboration 4. Construction 5. Modification and evolution	Few guidelines and heuristics, due to relative immaturity of the method.
JSD	1. Process structure diagrams 2. System specification network 3. Programming language	1. Entity-Action identification 2. Network modelling 3. Transformation	Few heuristics, but many examples in the literature and industry.

Table 2.1: *The variation in emphasis on notations, procedures and guidelines in four different software development methods.*

Many other methods have been developed over the years, each having its strengths and weaknesses for the development of systems installed in particular application domains [Wasserman, Freeman & Porcella 1983]. More recently there has also been a proliferation of object-oriented analysis and design methods [Arnold et al. 1991; De Champeaux & Faure 1992] which deploy a number of different notations for describing systems in terms of objects encapsulating state and behaviour. While the advantages accrued from building modular systems based on an object-oriented model are many (e.g., easier maintenance and more reuse), the object-oriented development process (which includes development procedures, heuristics and guidelines) is less well understood than for many traditional methods. This is partly because there is much less experience in the use of object-oriented methods, and partly because the object-oriented model of software development is more iterative, less ordered and more heavily oriented towards reuse [Henderson-Sellers & Edwards 1990; Korson & McGregor 1990].

The successful deployment of a method greatly depends on its ability to *guide* the software developer in developing a system. A “good” method should therefore be prescriptive enough to be able to recommend what development activity to do next, but flexible enough to allow a developer to occasionally diverge from recommended method procedures and guidelines. Ideally, a method should also be able to help developers recover from mistakes made during development. One weakness of current software development methods is that in many instances, the granularity of the activities supported by the method is too coarse to provide effective or helpful guidance. Method guidance in such cases is at an inappropriately high level. Thus for example, a method might recommend that “some editing should be performed” (it may even invoke an appropriate editor), but it does not give any more detail on exactly what to edit.

A method’s success is also dependent on its ability to fit in with an organisation’s management practices. It should integrate well into an organisation, and support managerial aspects of development such as team communication, cost estimation, project planning and staffing [Karam & Casselman 1993]. Moreover, a method should be tailorable, both in terms of the notations that it deploys and the procedures it prescribes. In fact, it has recently become popular for methods to be “engineered” [Kumar & Welke 1992], that is, systematically custom built for particular organisations and projects, or for particular application domains. Such *method engineering* [MEMM 1993] also encompasses the integration of multiple methods or method fragments in order to build new methods that bring together the desired or desirable features of several, tried and tested, methods. It is particularly critical that the method integration in this setting is effective, since the new, integrated methods should offer more than the sum of their parts [Kronlöf 1993a].

2.1.3. Tools

Using a method to develop a large, complex system can be a time-consuming and tedious process. Computer-based tool support is therefore essential for methods to be effective for large-scale software development. This section discusses the role of automated tool support and the kinds of tools that are available for supporting method-based software development. Three categories of support tools are addressed: Computer-Aided Software Engineering (CASE) tools, Integrated Project Support Environments (IPSEs), and method engineering (meta-CASE) tools. The role of database technology and a Database Management System (DBMS) are also examined in this context.

2.1.3.1. Computer-Aided Software Engineering (CASE) Tools

The term *computer-aided software engineering* is commonly used to describe a wide range of computer-based software development tools. These include simple, standalone tools, such as editors and debuggers, or more complex, integrated “environments” that combine several tools and an infrastructure (e.g., operating system services and a DBMS) to support them [Pressman 1992].

The last decade or so has seen CASE tools [Gane 1990] steadily increase in popularity [Byte 1989]. A large number of such tools have been constructed to support a wide range of software development activities, and substantial experience has been gained in their use [McClure 1989]. The software engineering literature has consistently reported on advances in CASE tools technology [Software 1988; CACM 1992; Software 1992]. The premature, or some would argue misconceived [Lehman 1987], expectation of fully automating the software development process however, has not been met, and a number of research issues remain [Balzer 1985; Voelcker 1988].

A simple CASE tool typically supports, or partially automates, a software development activity such as specification using a particular notation. A distinction has been made (e.g., [Voelcker 1988]) between *upper CASE* tools which support "front-end" development activities such as planning, requirements engineering and design, and *lower CASE* tools which provide support for "back-end" development activities such as implementation and programming support. Lower CASE tools, such as interpreters and static analysers, have been around for many years, and are consequently better understood in general and more widely used. Upper CASE tools on the other hand, support activities such as requirements engineering which by their nature deal with artefacts that may be volatile and vague. Frequently, front-end specification and design activities deploy highly graphical notations that are typically less formal and less amenable to automated support. Therefore, upper CASE tools have traditionally provided basic (diagram) editors and consistency checkers for specification syntax checking. Often, they have been restrictive in the notations they support, difficult - if not impossible - to customise, and irritating in their continuous flagging of transient inconsistencies. Thus, while CASE tools have made great strides in supporting software development, they still leave much to be desired so far as their usability, flexibility and coverage of development activities are concerned [Acly 1988; Martin 1988].

One additional issue that has become increasingly critical for the successful adoption of CASE tools technology in software development organisations, has been that of *tool integration.* The notion of having an integrated development environment populated with CASE tools, or alternatively, a number of standalone CASE tools working cooperatively, is seen as a fundamental, desirable objective. This is because, as in other engineering disciplines, software engineers need to have a variety of tools at their disposal, should be able to choose the configuration of tools that meet their requirements, and should be able to use them individually or together, as and when their requirements demand. Many of the complexities of large-scale software development arise from the inability of tools or information systems to "talk" to each other (e.g., exchange and/or share data) [Oddy 1990], and integrated CASE (I-CASE) [Software 1992] is a key technology for reducing these complexities. Different approaches to tool integration [Wybolt 1991] are discussed in more detail in chapter 3 (section 3.4).

2.1.3.2. Integrated Project Support Environments (IPSEs)

An *Integrated Project Support Environment* provides a collection of analysis, programming and management tools that cover an entire software development life cycle. These tools support one more or software development methods that are integrated together by the IPSE. The measure of tool integration is "the extent to which tools agree" [Thomas & Nejmeh 1992] and this is a property of the relationships defined between the tools. An IPSE provides a framework and mechanisms for defining and implementing these integration relationships. Tool integration however, is a necessary but insufficient component technology of an IPSE. An IPSE is an infrastructure for providing common services such as a generic user-interface, a common database or object management system, and a common tool set for providing common functionality between the integrated tools. These services are important not only for the provision of tool integration, but also for interface, process, team and management integration [Brown & McDermid 1992]. A distinction is made [Brown 1989; Göhner 1991] between a *homogeneous (closed) IPSE,* in which many CASE tools are integrated relatively easily by a single organisation building the IPSE, and a *heterogeneous (open) IPSE,* in which tools from different vendors make integration more difficult.

An IPSE is sometimes called a *method-based environment* as opposed to a *programming support environment* (the scope of the latter being more limited in terms of life cycle coverage) [van Vliet 1993]. This is because an IPSE can support one or more software development methods, rather than the activity of programming alone. In this context, the IPSE is based on a specific model of all, or part, of the software development life cycle. Examples of IPSEs striving to achieve this ideal are SIGMA [Akima & Ooi 1989], ISTAR [Dowson 1987], PACT [Thomas 1989b], ECLIPSE [Bott 1989] and ARCADIA [Taylor et al. 1989].

The UNIX environment [Kernighan & Pike 1984] is one of the best known open programming support environments in wide use. Although it began as an operating system in the late 1960s, it has grown to contain many of the integration mechanisms and tools that justify it also being classified as an IPSE. UNIX provides a primitive text-based line interface, but supports a number of program development tools including a variety of editors (e.g., vi, emacs and awk), configuration management and version control tools (e.g., SCCS and RCS), and many more. Used in conjunction with the X Window System [Mikes 1990], many of the weaknesses in the original UNIX user interface have now been alleviated. UNIX remains however a difficult environment to master by the novice, and requires a separate project management layer if it is to be used effectively for large-scale software development.

A number of different kinds of software development environments exist, which provide a subset of the functionality or life cycle coverage of a full scale IPSE. These include *language-oriented environments* [Sommerville 1992; van Vliet 1993] which are programming support environments built around a specific programming language, and which take advantage of the structuring mechanisms of that language. Examples of such environments include the Smalltalk-80 [Goldberg 1984] and Interlisp [Teitelman & Masinter 1981] environments.

A *public tool interface* (PTI) is a set of interface primitives, which may be used by tool and environment builders, that define basic integration facilities such as object and process management and tool communications [Sommerville 1992]. One prominent PTI upon which many IPSEs have been built is the Portable Common Tool Environment (PCTE) [Boudier et al. 1988; Thomas 1989a]. The motivation behind PCTE is to provide a standard layer upon which software tool vendors can build and market their tools. PCTE provides services in much the same style as UNIX tool services, with a User-Interface Management System (UIMS) for building and manipulating user interfaces, and an Object Management System (OMS) for data management (whose data model is based on an entity-relationship model [Chen 1976]).

Another PTI is the Common APSE Interface Set (CAIS) [Oberndorf 1988], a standard tool interface for the Ada Programming Support Environment (APSE) [Buxton 1980]. CAIS offers broadly the same kind of services as PCTE. In fact, the similarities between CAIS-A [Munck, Oberndorf & Ploedereder 1989] and PCTE+ [Boudier et al. 1988] (the subsequent revisions of the original CAIS and PCTE, respectively) are such, that the future evolution of these interfaces will be coordinated towards producing a new standard called PCIS [Sommerville 1992]. In general however, PCTE has been the more popular of the two PTIs, and a number of environments have been built using it, including the ECLIPSE and PACT environments mentioned above.

Finally, a class of tools known as *environment generators* are also an area of increasing research. These generate language or application-specific environments, typically from some kind of formal description. Examples of these include Gandalf [Habermann & Notkin 1986], in which programming environments (e.g., for Modula-2 and Ada) are generated from grammar definitions; and, the structured transformational approach to generating application-specific environments described in [Garlan, Cai & Nord 1992].

2.1.3.3. Meta-CASE Technology

Meta-CASE tools are software tools that support the design and generation of CASE tools. The environment generators described above are one particular kind of such tools. They take a formal description of a language or application as input, and generate one or more tools to support that language or application. Meta-CASE tools for some applications (also called application generators [Cleaveland 1988]) have been available for a number of years [Horowitz, Kemper & Narasimhan 1985]. These include compiler-compilers such as Yacc [Johnson 1975], code generators [CASE 1987] such as Tags [Lewis 1990], and third and fourth generation language systems [Misra & Jalics 1988] such as COBOL and Oracle respectively. Meta-CASE tools that generate CASE tools to support software development methods however, have only recently started to emerge and are sometimes called *method meta-CASE tools* [Alderson 1991].

Figure 2.1 is a schematic overview of a generic method meta-CASE environment. The figure assumes that a "method definition" is sufficient to implement the tool supporting that method. In practice, some tool-specific information (such as a user-interface definition) is also needed before the meta-CASE tool generator can produce a CASE tool that supports the defined method.

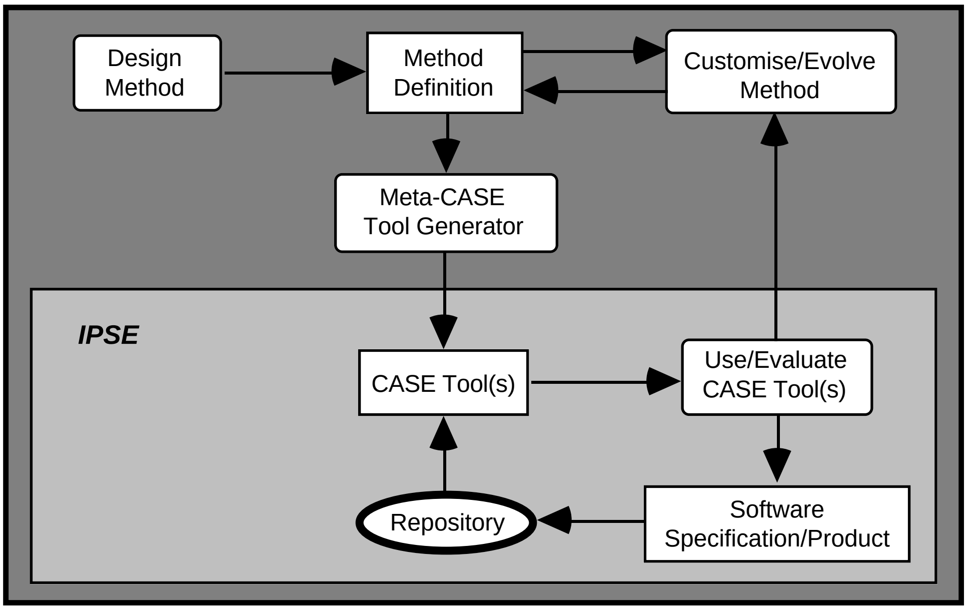


Figure 2.1: *A generic, method-based meta-CASE environment. Rectangles with rounded corners represent activities or processes whose inputs and outputs are represented by the arrows. Normal rectangles represent artefacts which are outputs produced by, or inputs to, activities or processes. The oval with the thick border represents a repository and its data/object management system.*

Method meta-CASE tools that efficiently generate usable CASE tool sets rely on a number key factors and technologies. The method for which tool support is required must be well understood and documented, so that a formal, or at least precise, description of that method [Sommerville, Welland & Beer 1987] can be produced and used as input to the meta-CASE tool. A good meta-CASE tool also facilitates the analysis, design and description of a method whose full and formal definition is not readily available. In this setting, the meta-CASE tool is a support tool for the method designer or engineer, and is therefore sometimes called a *Computer-Aided Method Engineering (CAME)* [Harmsen & Brinkkemper 1993] tool.

Meta-CASE tools must also be capable of documenting the definitions of, typically graphical, notations and their relationships, and therefore rely on the successful incorporation of generic (diagram) editing and rule checking systems and techniques [Welland, Beer & Sommerville 1990]. Care has to be taken not to allow developers to use these facilities to diverge wildly from their organisations' method standards. Thus, it is preferable that a meta-CASE tool is only used by method engineers to define their organisation's method and tool standards, and that individual software developers are not allowed to change or deviate from these. In fact, because of these and other fears, the market for meta-CASE tools has not been large and has consisted mainly of CASE tool builders rather than end users [Svoboda 1993].

An attractive, general property of meta-CASE tools is their ability to specify and generate themselves; that is, like a compiler-compiler, a meta-CASE tool should also be able to bootstrap itself. One example of a commercial meta-CASE tool that has been tested in this way, is the IPSYS Tool Builder's Kit (TBK) [Alderson 1991]. TBK is an integrated collection of generic tools and function libraries that may be used to specify software development methods, and to generate quality CASE tool sets to support them. It allows a tool developer to define the syntax and semantics of the various notations deployed by a method, provides a database interface based on a functional model of data which may be used to specify a method's (and its products') data model, and includes a UIMS based on OSF/Motif [OSF 1990] for defining the presentation of user input to and output from the generated tools.

Another commercial meta-CASE tool is the Virtual Software Factory (VSF) [Pocock 1991b; Pocock 1991a]. VSF provides a unified formal definition of the method to be supported and a kernel environment for the execution of that definition. A method definition in VSF consists of two parts: the construction of a semantic model of the method in a special-purpose language called Cantor; and, a syntactic definition of textual and graphical "views" in a Viewpoint Definition Language (separate sub-languages for defining textual and graphical views are provided). The VSF execution kernel, called the Analyst Workbench (AWB), then animates the formal method definition to produce a CASE tool set to support that method.

A number of other meta-CASE systems are also commercially available. These include Excelerator with Customizer [Rossi 1993; Svoboda 1993], MetaEdit [Smolander 1993], Paradigm Plus [Backhurst 1993] and ObjectMaker (Tool Development Kit) [ObjectMaker 1991].

Finally, Metaview is a multi-faceted research project, one of whose aims is the construction of a "metasystem" for describing and automatically generating custom software development environments [Sorenson, Tremblay & McAllister 1988; Sorenson & Tremblay 1993]. The Metaview architecture consists of a meta level, an environment level and a user level. At the meta level, a meta model is defined which is sufficiently expressive to support the description of a large class of development techniques (called "environments" in [Sorenson, Tremblay & McAllister 1988]). At the environment level, the environment definition produced at the meta level is "processed" to generate a particular tool configuration. The tools generated at the environment level are then ready to be used by developers (users) at the user level of the architecture.

To summarise, meta-CASE tools have a number of uses in the context of method engineering and CASE tools development. A meta-CASE tool automates the development of CASE tools to support one or more methods. It facilitates the process of extending and customising existing CASE tools. It is an aid to the design and construction of new methods, and facilitates prototyping of CASE tools for these methods, which in turn may clarify customer (developer) requirements. Finally, it is a vehicle for method integration in which a common repository and shared data model for different methods are used to provide an integration thread.

2.1.3.4. Software Engineering Repositories

A *software engineering repository*, together with the operations required to manage it, is central to CASE, IPSEs and meta-CASE technology. A repository is fundamentally a database that holds the artefacts produced by a software development process. These artefacts include specifications, implementations and other project documentation. Traditional information systems database models and database management systems however, are inadequate for managing the variety and complexity of these software development artefacts [Welke 1992]. Such models and systems are oriented towards supporting large quantities of information that is usually structured as simple short records. *Software engineering databases* [Brown 1989] on the other hand, are designed to support large quantities of information with varying structure and granularity. Transactions on such databases are typically *long transactions* (minutes, hours, or even days) as opposed to the *short transactions* (fractions of a second) on traditional databases. Therefore traditional database locking techniques and mechanisms are not feasible for software engineering databases [Sommerville 1992; Hurson, Pakzad & Cheng 1993]. In the following, we use the terms “repository” and “database” interchangeably, although there are subtle differences [Pressman 1992]. The term database is generally associated with information (processing) systems, whereas repositories tend to support software engineering environments.

Information systems databases are defined in terms of a *data model* of the information they hold. Variations of the Entity-Relationship (ER) model proposed by Chen [Chen 1976] have often been used to represent such data models. A software engineering repository is defined in terms of a *meta-model*, which determines how information is stored in that repository, and how it may be accessed, viewed and maintained [Welke 1989]. Repositories use a generalisation of the traditional DBMS or file system called an *Object Management System (OMS).* This provides support for the storage and manipulation of objects and relations between objects. These objects may vary in size and complexity, and the OMS is designed to support such variation. Such an OMS is a central part of PCTE for example, and is one of the key features that distinguish PCTE from an environment such as UNIX.

In the context of I-CASE, it is useful to examine a software engineering repository in terms of the kinds of artefacts it can store and the specific services it provides [Pressman 1992]. The artefacts that a repository can hold are numerous and include all problem and system knowledge and documentation. Services range from traditional DBMS services such as non-redundant data storage, high-level access, transaction control, security, and ad hoc data queries and reports, to more specialised CASE services such as storage of sophisticated data structures and process/project management. Used in this way, a software engineering repository can ensure data integrity, facilitate information sharing, and provide tool integration, method enforcement and document standardisation.

Software engineering databases or repositories have been used in a number of software development environments and applications. ConceptBase [Jarke & Peters 1993] is a software engineering and information system repository based on a deductive and object-oriented language called Telos [Mylopoulos et al. 1990]. It has been used as a meta system for method modelling [Jarke & Peters 1993], as a process-oriented integration model for software development environments [Jarke 1992], and as knowledge base for a variety of applications such as hypertext co-authoring [Eherer & Jarke 1991] and teamwork support [Hahn, Jarke & Rose 1991].

In the Odin project [Clemm & Osterweil 1990], the use of an OMS for tool integration is examined. The system proposed is based on the idea of integrating tools around a "centralized *[sic]* store of persistent software objects" (i.e., a repository). This repository contains two kinds of structures: a Derivation Graph, which specifies type and tool interconnections, and a Derivative Forest, which specifies how instances of the types (objects) are related to each other.

The role of a software engineering repository in software development has triggered a flurry of activity in the area of standards. The PCTE public tool interface is one such standard developed to ensure that environments and tools built on top of it integrate and interact together by way of the common repository and OMS. Other standards have been developed for representing software development information in a universal format in order to facilitate tool integration and data exchange. Two such standards on trial are the CASE Data-Interchange Format (CDIF) [CDIF 1993] and the IEEE 1175 standard [IEEE 1992]. CDIF focuses on standardising the data formats used to exchange information between CASE tools. This includes standardising both the syntax and semantics of information exchanges. The IEEE 1175 standard on the other hand, provides a reference model for CASE tool interconnections. The model attempts to define methods or mechanisms for implementing information exchanges, processes for transferring information, and the kinds of information to be transferred.

Another kind of software engineering repository is a *knowledge base* [Symonds 1988], a term commonly used in the field of Artificial Intelligence (AI) and which is finding increasing applications in software engineering [Puncello et al. 1988]. Knowledge-engineering tools, such as expert system shells, are used to build and maintain such a knowledge base which typically contains a representation of facts, a set of production rules for drawing conclusions from the facts, and reasoning strategies that specify how problems are solved by the application of production rules. In [Balzer, Cheatham & Green 1983], an alternative, futuristic, knowledge-based approach to software development was proposed. The approach suggested the use of an “automated assistant” that guides, supports and automates software development by interacting with, and maintaining software development information stored in, a knowledge base. Thus, the knowledge base may, for example, contain general-purpose or domain-specific implementation knowledge such as algorithms and data structures. Such a knowledge base is becoming a de facto requirement for modern software development environments that are designed to support reuse [Barstow 1985; Prieto-Díaz 1991].

In whatever form it takes, a software engineering repository is indispensable for supporting software development. The need for a *centralised* repository however is less clear. Centralised databases are almost universally used in IPSEs for their perceived benefits in providing a powerful model for tool integration, security and information integrity. Centralised repositories for large-scale software development however, are based on complex meta-models and schemata. In a rapidly evolving environment in which new methods and tools are being added and evolved, these centralised structures quickly become difficult to understand, maintain and extend, gradually becoming a bottleneck. Providing for physical distribution of a database, while maintaining logical centralisation, has some benefits as does the use of emerging object-oriented database technology [Hurson, Pakzad & Cheng 1993]. Sommerville [Sommerville 1993] suggests that software development environments should be modelled as cooperating communities of agents rather than centralised or specialised “bureaucracies”. Moreover, the trend in database research also favours increased decentralisation. Consequently, *multidatabase systems* have become an area of steadily increasing interest [Bright, Hurson & Pakzad 1992].

A multidatabase is a set of autonomous databases, managed together without a global schema [Litwin, Mark & Roussopoulos 1990]. A multidatabase management system, also called a multidatabase or federated system, is characterised by a multidatabase language. A heterogeneous multidatabase system supports multiple database systems with different database models, languages and services [Ahmed et al. 1991]. A special class of research multidatabases, known as interoperable systems [Bright, Hurson & Pakzad 1992], offer the most loosely-coupled of all types of multidatabase systems. Interoperable systems are not globally database-oriented, which allows local systems to include a variety of heterogeneous systems, such as expert or knowledge-based systems. Communication is through standard protocols between nodes.

A multidatabase environment ensures local autonomy of the constituent databases. However, the other major issue in multidatabase design [Litwin, Mark & Roussopoulos 1990], data redundancy, is more problematic. Data redundancy in multidatabases causes well-known update and consistency problems. Recently however, there appears to be a trend towards relaxing consistency constraints for software development databases in which transient inconsistencies are more likely to exist. This is particularly useful during the early stages of the development such as during the requirements elicitation stage when multiple conflicting requirements are likely to exist.

Fortunately, this has been recognised in the database world, and some remedies, such as allowing the evaluation of time constraints in additional to the usual data-value constraints, have been proposed [Rusinkiewicz, Sheth & Karabatis 1991]. A strong case for tolerating and supporting inconsistency in database and configuration management is made in [Gabbay & Hunter 1991] and [Schwanke & Kaiser 1988] respectively.

In summary, decentralised repositories appear to be more realistic software development support infrastructures. They are consistent with the requirement for decentralised software development tools, but of course there is no escaping the need for stronger integration models to “glue” these decentralised repositories together again - when the need arises. The problems of enforcing inter-database consistency however are not as critical for software development as they are for traditional data processing applications, since the software development process deals with inconsistent data all the time. Thus, tolerating inconsistency [Balzer 1991] can be a feasible approach for the effective support of such a process.

2.1.4. Evaluation

Section 2.1 has discussed current approaches to method-based software development. The discussion has also raised a number of issues that highlight the limitations and desirable objectives of these approaches. What follows is a summary and brief evaluation of these issues, limitations and objectives.

- Different software development life cycle models have been proposed, many of which enforce the ordering of development stages or activities, and which require the production of deliverables at the end of various stages. Others, while relaxing ordering constraints and requirements for intermediate deliverables, result in development projects that are more difficult to manage.
- Software development methods are elaborated instances of life cycle models. Many methods have been developed, each with features and limitations that make them more or less desirable for the development of particular applications. The objective of creating the ideal or standard method is not a realistic one. Rather, methods should be engineered and/or integrated to produce “custom methods” appropriate for the particular organisations, people and projects in hand.
- CASE tools support all or part of a software development method. Often, in an attempt to make these tools more generic and reusable, they are constructed without faithfully supporting their associated method. IPSEs attempt to provide tool integration facilities, and some IPSEs allow developers to customise the process by which these tools are invoked and used. The granularity of such processes in general however is too coarse to provide useful method guidance (it is at the tool invocation level rather than at the level of the objects manipulated by those tools).

- Meta-CASE tools technology fits in between CASE tools and IPSEs, but meta-CASE tools are often standalone. They require precise method definitions as input, which makes them attractive tools for specifying development processes and for standardising the use of notations in organisations. Conversely, they may also be used to customise processes and notations to suit particular needs. Such method customisation manifests itself in the tools that these meta-CASE tools generate.
- Software engineering repositories are often the central part of IPSEs' infrastructures and are used to facilitate tool integration. However, they are often independent, centralised, monolithic structures, with schemas that are difficult to understand, extend and maintain. Moreover, they do not fit the model of development that includes multiple participants, using multiple methods and tools, in a distributed environment. There is therefore a need for decentralised repositories such as multidatabases. This would also prevent repositories from becoming the bottleneck of software development activities.

2.2. Software Engineering Knowledge

Software engineering is a knowledge-intensive activity. Successful software development requires that development participants understand the problem domain in which they are working, and the solution domain in which their system will be installed. In other words, they must possess *domain knowledge.* The participants must also be able to perform the various actions necessary to build the required system. In this context, they must possess software development *process knowledge.*

In order to communicate and document knowledge about a problem, its solution, and the process of development, development participants must express this knowledge in some form of language, notation or representation scheme. In this context, they must possess *representation knowledge* with which to produce the desired descriptions. Finally, the manifestation of any development process in a particular domain, is a product. In software terms, this is a series of specifications and associated implementation - described in terms of one or more representations. In this context, this *specification or product knowledge* is the software development deliverable.

Figure 2.2 summarises the four different kinds of knowledge associated with software development. The following four sub-sections discuss each of these in more detail, and present related research contributions and open issues. Section 2.2.5 then critically examines the problems associated with the current global view of software engineering knowledge, and examines the lines along which the different kinds of knowledge have traditionally been partitioned.

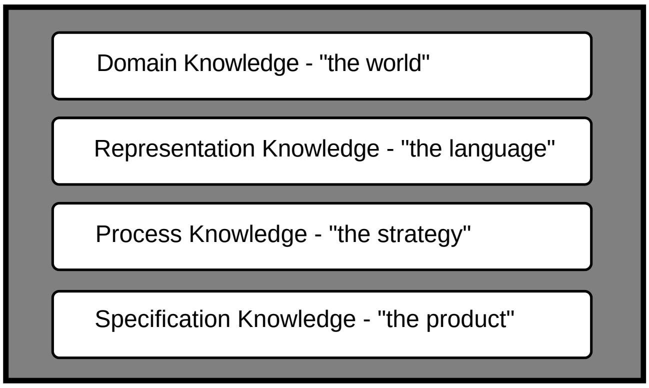

Domain Knowledge - "the world"
Representation Knowledge - "the language"
Process Knowledge - "the strategy"
Specification Knowledge - "the product"

Figure 2.2: *Software engineering knowledge.*

2.2.1. Domain Knowledge

Knowledge about a software system's application domain is critical to the successful development of that system. On the one hand, analysts eliciting system requirements must *understand* the problem domain for which a software solution is required. On the other hand, system developers should be able to *reuse* domain-specific abstractions and components when designing and implementing that system.

Domain knowledge is difficult to elicit and represent, since a large part of it is usually not well understood or represented by the "domain expert" (the customer) who possesses it. Furthermore, design and implementation decisions and rationales are also difficult to elicit and represent. *Domain analysis* is "the process by which information used in developing software systems is identified, captured, and organized ..." [Prieto-Díaz 1990].

The difficulty in capturing and representing domain knowledge is partly due to the fact that much of this knowledge is in the form of encoded expertise, intuition and rules-of-thumb, that domain experts find very difficult to formalise, or at least express. Many AI researchers have grappled with these problems for some time. The role of the "knowledge engineer" has long been recognised in the AI community as an essential participant in the development of knowledge-based or expert systems. Even more demanding is the question of how to go about capturing the "creative" activities of development, or representing the mental models that experts hold about particular domains [De Bono 1978].

The above difficulties notwithstanding, there is still much domain analysis and knowledge acquisition which can, and has, been applied successfully to the understanding and representation of domain knowledge. Barstow [Barstow 1985] for example, reports on work that explores the representation and implementation of domain knowledge about programming, for use by an automatic programming system in the application domain of oil well logging.

Maiden and Sutcliffe [Maiden & Sutcliffe 1992] on the other hand, explore the use of more general abstractions which can be reused and tailored for particular domains. The proposed process through which this may be achieved is that of *analogy:* developers choose domain abstractions that are analogous to the application domain they are working with, which reduces the time and effort required to capture domain knowledge that is common or shared between application domains. This work is demonstrated further in [Maiden & Sutcliffe 1993], where the reuse of domain abstractions is empirically illustrated. However, Kramer warns that abstractions which are too abstract may be too general to be useful [Kramer 1993].

Barstow also explores the need to specify a system *and* its domain [Barstow 1993]. He concludes that domain modelling is not only important for providing a better understanding of the phenomena one wants to support with a software system, but also because domain models (i) provide a natural means for organising a system and specifying its components, and (ii) make customisation and evolution of systems easier, since these activities can be stated in terms of the domain models. Jackson and Zave confirm this view by suggesting that "specifications should describe domains explicitly" [Jackson & Zave 1993], and that such domain descriptions may or may not be independent of the system that also needs to be specified.

A number of projects have utilised domain knowledge, analysis and modelling in their approaches. The Programmer's Apprentice project [Rich & Waters 1990] uses the notion of "cliches" to refer to standard methods for dealing with particular tasks in a particular domains. The Draco project [Neighbours 1984] examines the construction of software from reusable, domain-specific components. These components are described in a so-called "domain language" in which analysis information about a problem domain is represented in terms of objects and operations. The Requirements Modelling Language (RML) [Borgida, Greenspan & Mylopoulos 1985] is a language for representing domain knowledge about some portion of the world. Once constructed, this "requirements model" ideally becomes the only body of knowledge about the world being modelled. Well defined portions of this model can then be realised as a software system.

Finally, a complementary and increasingly popular approach to knowledge acquisition, that originated from anthropology, is *ethnography* [Hammersley & Atkinson 1983]. In a software engineering context, this is the process of observing customers in their natural or daily work environment, and subsequently analysing these observations in order to gain a better understanding of the structure, organisation and practices of those customers [Sommerville et al. 1993]. Thus, ethnographers attempt to understand a problem domain with a view to eliciting customer requirements. Knowledge acquired in this way may be passed on to system developers who can then design and implement a software system that operates in the observed domain. The approach is particularly useful when the application domain is large and complex, and customers are unsure where their problems lie, and what scope there is for software automation and support. It is also useful for the aforementioned reason that domain experts often find it difficult to articulate their expertise.

One software engineering example that integrates ethnography into the requirements engineering process is reported by [Sommerville et al. 1992]. The study describes how ethnographic analysis is used to prototype an air traffic control system which is then used as input to the requirements engineering process for the full-scale system.

2.2.2. Representation Knowledge

Software development, including specification, design and implementation, may be regarded as a process of *description* [Jackson 1991; Jackson 1993]. To produce descriptions, software developers require languages or notations[1] with which to represent the elements of these descriptions. Different notations have different strengths and weaknesses which affect their expressiveness and determine how appropriate they are for particular application domains. There is no doubt however that notations are indispensable tools for communicating and documenting facts and ideas [Iverson 1980]; that is, for representing knowledge[2].

“In computer science, a good solution often depends on a good representation.” [Woods 1983]. The choice of representation depends on the kind of knowledge that needs to be represented. Natural language for example, is a multi-purpose representation that is suitable for producing a wide range of descriptions. The complexity of natural language however, makes it very difficult to be analysed or interpreted by machines, which is the ultimate goal of a software specification or implementation. Moreover, it is possible (and is often the case) that natural language is used to produce vague or ambiguous descriptions. Again, this is not suitable if a computer is to execute such descriptions. Therefore, researchers have developed numerous representations that vary in their complexity, formality, structure, and ultimately, their *expressive power* for describing particular kinds of knowledge.

The expressiveness of a notation is determined by the domain in which it is used and the constructs it provides. When specifying systems, analysts and designers often find it easier to produce graphical descriptions of facts and ideas. In fact, Larkin and Simon [Larkin & Simon

[1] Although the terms “notation” and “language” are commonly used interchangeably, we make a subtle distinction. A notation is a set of constructs that may be used in a variety of ways to represent knowledge. A language on the other hand, is a set of constructs and the rules for using them together. A language therefore has a grammar, whereas a notation may have many different grammars associated with it.

[2] There is again a distinction between knowledge representation and representation knowledge. Knowledge representation is concerned with exploring ways of structuring and representing (domain) knowledge (e.g., using objects, logic, data structures, etc.). Representation knowledge on the other hand refers to knowledge about a particular notation or representation scheme - which will be used to express or articulate knowledge.

1987] compare “diagrammatic” and “sentential” representations, and conclude that the proper use of graphical representations (diagrams) can significantly improve problem solving and expression. Moreover, customers who initially provide imprecise requirements in natural language, also find that graphical descriptions are easier to understand. Implementors on the other hand, who may prefer to use graphical notations, are more likely to produce software in text-based notations or languages because the latter are more easily interpreted or executed by computers.

AI researchers have also explored a variety of ways in which to represent knowledge [Computer 1983]. One well-known approach has been the decomposition of knowledge into modules called *frames* [Minsky 1975], which may then be used to make various kinds of inferences. Frames are also analogous to the now popular *objects,* and may be structured to exhibit properties such as “instantiation” and “inheritance”. Frames also contain “frame variables” or “slots” in which knowledge is held.

Whatever representation is ultimately chosen to represent knowledge, the next step is to provide a precise or formal definition of that representation. This is important to remove ambiguities from the representation, and to facilitate the provision of tool support in the form of syntax-directed editors, interpreters, compilers and so on. To define representations, grammars or “meta-level” descriptions are often used. One common approach for defining text-based languages, is to describe their respective grammars in *Backus-Naur Form (BNF)* [Aho & Ullman 1977]. This entails providing a description of the “terminals” (constants or primitive constructs) and “non-terminals (variables) of the language, and specifying allowable or alternative concatenations of those constructs. Another approach is to use an *entity-relationship* description [Chen 1976] of the notation, replacing the constructs of the notation by “entities” and their connectors by relationships. The latter approach is more common for defining graphical notations.

Defining the syntax and semantics of representations is also important for facilitating translations between representations. This is becoming an increasingly important area of research, where many have acknowledged the need to use multiple languages or notations for specifying, or even implementing large and complex systems [Wile 1986; Petre & Winder 1988; Zahniser 1993]. In fact, much of the work in the so-called area of *multi-paradigm software development* [Zave 1989; Meyers & Reiss 1991], is greatly dependent on a deep understanding of the different languages or notations used in the different specification or programming paradigms. A critical review of multi-paradigm development techniques is deferred to chapter 3.

Finally, it is worth noting that software processes may also be treated as descriptions. Whether these descriptions are in the form of programs [Osterweil 1987] or models [Lehman 1987] is the subject of much debate. In general however, knowledge about the software development process can, and usually is, represented separately from other domain knowledge.

2.2.3. Process Knowledge

A process is “a series of actions which produce a change or development” [Dictionary 1987][3]. A *software process* prescribes the activities by which software is produced. These include software development actions and their ordering, timing, synchronisation and coordination. *Software process technology* [Finkelstein, Kramer & Nuseibeh 1994] addresses a number of additional software process issues that impact upon software development. These include process programming [Osterweil 1987], modelling [Curtis, Kellner & Over 1992], evolution [Lehman 1994] improvement [Humphrey, Snyder & Willis 1991] and management [Humphrey 1989].

The term *software development process knowledge* is used to refer to all of the above software process activities. A common and effective way to understand, represent and manipulate process knowledge is through the use of software process modelling. *Software process modelling* is the construction of abstract descriptions of the activities by which software is developed. In the area of automated software development, the focus is on models that are enactable; that is, executable, interpretable, or amenable to automated reasoning. Modelling the software development process is a means of understanding the ways in which complex software systems are designed, constructed, maintained and improved [Finkelstein, Kramer & Hales 1992]. A software process model may also be used to provide method guidance; that is, as a vehicle for answering questions such as “what should I do next?”, or more importantly “how do I get out of this mess I’m now in?” In general, process models have not been successful in answering the second kind of question.

Software process modelling is a complex activity that can span the entire software development life cycle, from requirements analysis and specification to system implementation, evolution and maintenance. From an organisational point of view, it is desirable to model the overall development process including the coordination and interaction of a large number of development participants. From the individual developer’s point of view, while coordination and interaction are still important, the focus is on modelling development activities that fall within the domain of concern or responsibility of that individual developer.

This last issue of process model *granularity* has been a stumbling block for many attempts at software process modelling. In general, process models have addressed processes that are very coarse-grain [Curtis, Kellner & Over 1992]. *Coarse-grain* processes are “unaware” of the representation schemes they manipulate and therefore treat them as opaque or “vanilla” objects with no internal structure or semantics [Nuseibeh, Finkelstein & Kramer 1993]. This is in contrast with *fine-grain* process models which represent activities at the level of individual developers or

[3] A process is sometimes also called a “method”, but this is not the terminology deployed in this thesis. A method in the context of this thesis also includes representations together with the process by which descriptions in those representations are produced.

tools, and the representations that these developers and tools use. Fine-grain process models are therefore more effective vehicles for providing method guidance to developers, as opposed to coarse-grain models which may be too abstract or generic to be useful [Curtis, Kellner & Over 1992]. Perry's work on modelling humans [Perry 1992] and policies [Perry 1990; Perry 1991] in the software development process is an example of experiments in fine-grain process modelling.

Finally, a large proportion of the work on coordination and cooperation in Computer-Supported Cooperative Work (CSCW) also falls under the umbrella of software process modelling and support. For example, Godart [Godart 1993] describes COO, a transaction model to support cooperation and coordination between multiple software developers. The model uses goal-oriented process modelling coupled with a concurrency control protocol, to provide consistent cooperation and coordination between developers who share intermediate results.

Support for collaborative software development is also provided by CSCW environments such as ConversationBuilder [Kaplan et al. 1992]. ConversationBuilder provides different protocols for shared activities such as editing and bug tracking, and demonstrates how very different kinds of process models can be supported by the same environment. This is an attractive feature for software development involving multiple development participants, since different participants are likely to follow their different processes, and communicate in different ways depending on the individual goal they have.

Software development process knowledge is the key to consistently producing high quality software. How such knowledge is represented and used is still an area of ongoing research [SEJ 1991]. Moreover, the question as to what represents an effective software development process is as yet unanswered. Researchers at the Software Engineering Institute (SEI) at Carnegie-Mellon University (CMU) however, have proposed a Capability Maturity Model (CMM) [SEI 1991b; SEI 1991a] of software processes that may be helpful in evaluating such processes. This model identifies five levels of "process maturity" in organisations:

- level 1 in which the process is ad hoc and chaotic,
- level 2 in which the process is repeatable,
- level 3 in which the process is defined,
- level 4 in which the process is managed, and
- level 5 in which the process is optimised.

Moving up these levels indicates a greater level of process maturity, which in turn suggests the production of higher quality software products. While the CMM may not be the most comprehensive model of all software processes [Finkelstein 1992], it is nevertheless a step forward in process assessment and improvement, and thus product improvement.

2.2.4. Specification (Product) Knowledge

Software *specification knowledge* refers to the body of knowledge collected about the software development artefacts that are generated by a software development process. These artefacts include specifications, designs, implementation code and documentation. The term “specification” is therefore used generically to refer to all software development “products”, from initial requirements to final deliverables, and including all intermediate descriptions or documents in between. Since software itself is a “non-tangible” entity, specification or product knowledge is in the form of descriptions articulated in one or more representations. Thus, any particular software development deliverable (whether intermediate or final), is an instance of representation knowledge; and the process of producing this deliverable is an instance of process knowledge. Moreover, the product itself embeds knowledge about the application domain in which it resides.

Domain, representation and process knowledge have already been addressed in the context of software product development. Therefore, this section addresses specification knowledge about a product *after* it has been produced in some form or another. The kinds of specification or product knowledge that are examined include *analysis and testing* of a product, its development *rationale,* and other *metrics*. With the exception of development rationale, this thesis does not address any of the above activities. Therefore, this section only provides a brief overview of one or two of these issues for completeness.

Once a partial specification or product has been developed and expressed in a particular representation, a number of operations may be performed on it, and various kinds of information may be extracted from it. For example, a concurrent system specification may be analysed for liveness and deadlock, or verified for consistency and completeness. Moreover, an implemented system may be tested for bugs and usability, and its performance measured. The information collected from such analysis, verification, testing and measurement, represents different kinds of specification or product knowledge. This information is then fed back into the development process to correct, improve and generally maintain the software product. Software development processes which do not take into account specification knowledge feedback [Lehman 1994], almost certainly, generate final products that fail to meet their specifications.

It is frequently necessary to for a developer to explain the *reasons* for including components in a specification, and the *reasoning* behind the development decisions that resulted in the developed product. Recording a development or design rationale during the software development process is one way to capture some of this information [Potts & Bruns 1988]. Design rationale information may therefore include both process and product knowledge. This is because both the artefacts and the process of deliberation (rationale) by which they where produced needs to be documented and explained.

Other related questions include: how does one go about capturing a design rationale and what representation does one use to document it? The first question remains problematic because of the difficulties of embedding the recording of a design rationale in the software development process, and ensuring that such a rationale is indeed recorded by development participants. More answers to the second question exist. Lee and Lai [Lee & Lai 1991] propose a framework for evaluating the expressive adequacy of rationale representations, based on progressively differentiating the elements of representations. One such representation proposed by Lee and Lai is their Decision Representation Language (DRL) for recording elements of a development decision making process, such as alternatives, their evaluations, the arguments behind these evaluations and the criteria used to make these evaluations. Another well known representation is gIBIS [Conklin & Begeman 1988], which uses the representation elements “issue”, “position” and “argument” to record rationale arguments. Unlike DRL however, gIBIS has no facility to represent goals which are useful for representing deliberations and multiple viewpoints.

Finally, an important attribute of specification knowledge is consistency. The final product delivered by a software development process must be internally consistent and free from any errors or contradictions. Such inconsistencies inevitably arise in large projects and therefore analysis and testing techniques need to be deployed in order to remove, or at least identify, these inconsistencies. In software specification for example, a common problem is to ensure the consistent use of terminology and its associated semantics. One way to explore semantic inconsistencies in a large problem domain is through the use of ontology [Wand & Weber 1990]. This field of study addresses the *meaning* of terms and the use of *meaning preserving* transformations between descriptions such as specifications and programs.

2.2.5. Partitioning Software Engineering Knowledge

The partitioning of software engineering knowledge into domain, representation, process and specification knowledge undoubtedly has advantages. For example, one would clearly want to separate knowledge about a domain from knowledge about how to represent that domain knowledge. Moreover, a software developer with knowledge and experience in using a certain development process, cannot be expected to combine this with equal knowledge about, and experience of, every application domain he or she addresses. Therefore, it is clearly important that the different kinds of software engineering knowledge identified are somehow separated.

Unfortunately, partitioning software engineering knowledge in this way is not sufficient. With increasingly complex, *heterogeneous* software systems being built, each of the kinds of software engineering knowledge identified is also composed of many parts. For example, a software system may be used simultaneously in different domains. It may also be described (specified, implemented and documented) in different representations; and it may have been developed by many developers enacting different processes. Finally, the end product may have several different components.

Clearly then, the different kinds of software engineering knowledge may themselves partitioned as, for example, shown in figure 2.3. And here lies the problem. There is a mismatch between the partitions of different kinds of knowledge. For example, while one developer may be partitioning a software development problem along the lines of the different representation schemes that will be used to express that problem, another may be slicing up that same problem in terms of the deliverables (e.g., specifications and products) that he or she is required to produce. This may be workable for small projects, but it makes the task of identifying and checking the relationships and inter-dependencies between the partitions of different kinds of knowledge much more difficult for large, complex and distributed development projects.

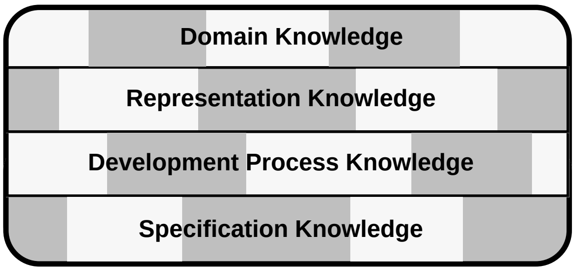


Figure 2.3: *A global view of software engineering knowledge decomposition.*

What is needed then, is a better way of partitioning software engineering knowledge, that maintains some of the advantages of the traditional separation of domain, representation, process and product knowledge, while at the same time maintaining the links between them.

Chapter 3 discusses the "separation of concerns" principle, which will be used to explore alternative ways of partitioning software engineering knowledge to the one suggested by figure 2.3 above. Of course, the issue here is not only how do we cut up software engineering knowledge, but also how do we do this in such a way as to allow some form of integration at a later stage.

2.3. Chapter Summary and Evaluation

This chapter has examined the software engineering context of the work described in this thesis. By necessity, it covers a diverse range of software engineering advances and concerns. This is because the ViewPoints framework described in chapter 4 addresses, or at least touches upon, a number of these issues. Many of the topics covered in this chapter will be revisited at some stage later in the thesis to demonstrate that the proposed framework is consistent with currently successful software engineering practices, and to evaluate the author's contribution to the current state-of-the-art.

This chapter has critically examined the software development process as exemplified by a number of life cycle models described in the literature. These included the waterfall, incremental, prototyping and spiral models of development. The notion of a method as a set of procedures, guidelines and notations was discussed, with reference to a number of prevalent methods such as CORE, JSD, SADT, the CDA and others. CASE tool support for such methods was also discussed, and the role of IPSEs in this setting was examined. In particular, the role played by databases and repositories for method and tool integration was explored.

This tour of the literature has highlighted:

1. *Weaknesses of, and alternatives to, traditional software development life cycles.* In particular, it identified a need for life cycle models that facilitate distributed, concurrent and iterative development, with both intermediate and final deliverables.
2. *Requirements for effective software engineering methods.* Specifically, it identified the need to customise and/or integrate different methods, in order to produce “tailored” methods that satisfy the particular needs of different organisations and/or projects.
3. *Difficulties in providing method-based tool support and integration,* for the effective support of different software development processes. In particular, it identified the need to move away from support environments which rely on centralised (database) architectures for integration.
4. *Different kinds of software engineering knowledge.* In particular, software engineering research has identified addressed domain, representation, process and specification knowledge. Each however has been partitioned independently of the others, making integrated support for these different kinds of knowledge much more difficult.

This chapter has also served as motivation for the next part of the literature survey presented in chapter 3. Chapter 3 examines in more detail particular attempts at separation of concerns to reduce software development complexity, with particular emphasis on approaches that support multiple perspectives during development. The next chapter also picks up on the open issues in method engineering and integration, which need to be addressed if the desirable separation of concerns is to be reconciled with the desired integration.

Chapter 3 Separating and Integrating Concerns

This chapter motivates the work described in this thesis by discussing related research in the areas of multi-perspective software development and method integration. The chapter has two main themes. The first is “separation of concerns” for reducing software development complexity. In particular, the use of multiple *viewpoints or perspectives* as vehicles for such separation is examined.

The second theme is “integration” in the context of method-based, multi-perspective software development. In particular, the focus is on *method integration,* which is identified as a key enabling technology for other kinds of integration such as tool integration. Existing approaches, architectures, techniques and mechanisms for method integration are critically examined.

The chapter concludes with a summary and an evaluation of existing work, reiterating outstanding research issues that will be addressed in the remainder of the thesis.

3.1. Separation of Concerns

Herbert Simon defines a complex system as one “made up of a large number of parts that interact in a non-simple way. In such systems the whole is more than the sum of the parts ...” [Simon 1981]. He further proposes that complex systems exhibit a hierarchical structure, and are therefore better understood if decomposed into component parts. Such decomposition (originally proposed by Simon in 1969) is the basis for the separation of concerns principle.

Separation of concerns is an activity by which a complex system is decomposed, according to various criteria, into simpler units. The objective of such a decomposition is to be able to address only those concerns or criteria that are of interest, ignoring others that are unrelated. For example, one may separate the analysis of a motor vehicle’s functionality from aesthetic considerations, or separate the development of the user interface features of a software package from the development of its computational functions.

There are many possible criteria for separating concerns during the development of software systems [Ghezzi, Jazayeri & Mandrioli 1991; Alford 1994]. One common criterion is time.

Temporal separation of concerns decomposes a development process into time slots during which different activities are performed. Temporal separation may also be useful for ordering of activities. For example, requirements analysis is commonly separated from, and performed before, system implementation. The waterfall life cycle model described in chapter 2 (section 2.1.1), is based on this kind of temporal separation.

Another criterion for separation is based on *spatial* considerations. For example, software development may be decomposed into activities performed at physically different locations.

Separation in terms of *qualities* is also common. For example, one may wish to deal separately with the efficiency and correctness of a given program [Ghezzi, Jazayeri & Mandrioli 1991].

The complexity of a software development project may also be reduced by separating the different kinds of software engineering knowledge that are involved[4]. For example, software developers frequently produce data and process models of a complex system and address these separately. This kind of *view* separation is often used when different *parts or aspects* of a complex system need to be addressed separately. The π-paradigm described in [Goedicke 1990], applies such separation of concerns to divide a complex software description into smaller (more understandable) partial descriptions (called views). These are then superimposed to form the complete description. Separation of concerns in software development based on views or viewpoints is examined in more detail in section 3.3.

Yet another common criterion for separation of concerns in computer science and software engineering is that of a *role or agent.* The notion of an agent has been particularly addressed in the areas of AI, Distributed Artificial Intelligence (DAI) [Weihmayer & Brandau 1990], and increasingly in CSCW. For example, a complex development project may be decomposed into smaller and simpler sub-projects according to the roles, responsibilities or expertise of the agents assigned to develop these sub-projects. Combining or coordinating sub-projects in such scenarios however is a non-trivial task. For example, in DAI, a common objective is cooperative problem-solving by multiple agents [Shoham 1990; Jennings & Mamdani 1992; Werner 1992; McCabe 1993]. These agents have individual or common goals, and interact according to one or more well-defined protocols in order to achieve these goals. These protocols have been the subject of significant research and will be discussed in more detail in section 3.4 since they are also vehicles

[4] An increasingly common criterion for separation of concerns in computer science and software engineering is an object [Wegner 1987]. Objects encapsulate two kinds of knowledge in a single entity: data (domain knowledge) and the operations (process knowledge) that operate on that data. Object-oriented analysis (OOA) and design (OOD) methods separate concerns (in problem and solution domains respectively) based on such objects (e.g., [Coad & Yourdon 1991] and [Booch 1991] respectively).

for integration.

Finally the notion of a software development *paradigm* is becoming an increasingly feasible criterion for separation. This is particularly so with multi-paradigm software development systems becoming more common [Zave 1989; Meyers & Reiss 1991; Spinellis 1994]. A paradigm refers to a development approach characterised by its view and representation of a system and the world in which that system operates. Examples of software specification and programming paradigms include the functional, logic, object-oriented and imperative paradigms. Increasingly complex software systems may require the development of their different aspects or components using different paradigms. In such cases, the notion of a paradigm is a useful coarse grain criterion for separating concerns and decomposing these systems and their development processes.

In conclusion, separation of concerns is an increasingly important tool for handling the complexity of many modern software systems. It relies on decomposing a problem or system into its constituent components that are, as far as possible, unrelated, or at least loosely coupled. These, hopefully simpler, components may then be addressed (developed, analysed, decomposed, etc...) independently, possibly by different development participants, at different times, in different locations and so on. It can also be argued that, in many instances, the “real world” is naturally composed of many different activities or components that have separate concerns, and that actively applying the separation of concerns principle is actually a process of modelling or representing the “real world”.

3.2. Integration

Separating concerns is an important step towards reducing the complexity of software systems, making them easier to develop, understand and maintain. However, “... having divided to conquer, we must now reunite to rule.” [Jackson 1990]. In other words, having decomposed a system into different activities or components, it is then often necessary to achieve some level of integration between these activities or components.

Integration is the process of making something whole [Dictionary 1987]. In software engineering, the scope for integration is wide, and like separation of concerns, may be based on a number of different criteria. For example, one may wish to integrate or combine the activities of different development participants, so that they are synchronised and coordinated. Alternatively, one may wish to integrate the separate sub-products of different development activities, in order to produce a single deliverable (such as a software system).

3.2.1. Scope

A number of different kinds of integration are particularly relevant to the work presented in this thesis. These include method, tool, view and system integration. All these fall under the general umbrella of software engineering knowledge integration, and are loosely related as shown schematically in figure 3.1 below.

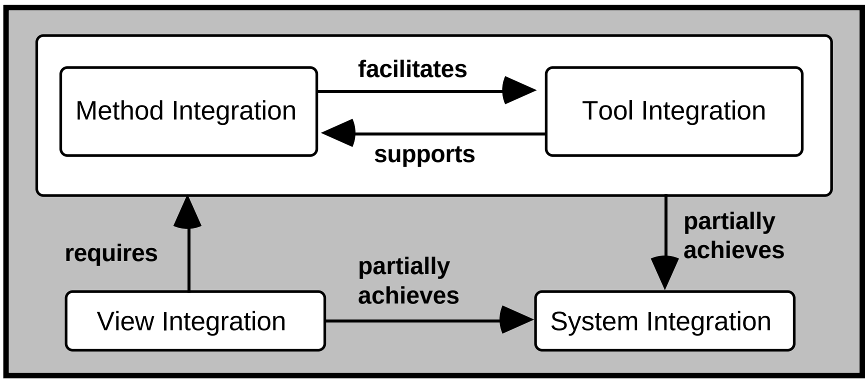


Figure 3.1: *A perspective on software engineering knowledge integration.*

Software engineering knowledge integration refers to the process of combining different, or the same, kinds of software engineering knowledge (examined in section 2.2), in meaningful and productive ways. For example, one may wish to achieve process knowledge integration [Mi & Scacchi 1992], where a number of different process descriptions are combined, either to produce a single process description, or to ensure the smooth communication and coordination between different processes. Such process or control integration is only one of the facets of tool integration; others include data and presentation integration.

Tool integration in the context of computer-aided software engineering refers to the combination of two or more (especially multi-vendor) software development tools, in such a way that these tools can then communicate and exchange information in order to achieve some software development goal. A tool integration framework is the infrastructure containing the mechanisms for achieving integration [Wybolt 1991]. Tool integration frameworks provide high level architectures which position and relate the various tool integration components, which normally include at least an object management system, an integration agent (such as a tool communication protocol), and the tools themselves.

Tool integration is usually the prime objective of an Integrated Project Support Environment (IPSE) [Wasserman 1990]. Unfortunately, this has detracted somewhat from the more fundamental objective of an IPSE - that of providing integrated tool support for *method-based* software development. Thus, the focus has often been on frameworks, infrastructures, protocols, mechanisms and standards for tool integration, with only a secondary requirement of supporting software development methods. This is in contrast with a more desirable approach of developing frameworks and techniques for method integration, which can then be used as the basis for tool support and integration [Kronlöf, Sheehan & Hallmann 1993].

Method integration [Kronlöf 1993a] focuses on the study and synthesis of multiple methods which possess individual and collective features that make them an attractive combination for developing particular (software) systems. Tool integration in this context is only an issue after questions such as "what methods should be integrated?" and "how will these methods be used together" are answered. Method integration issues will be discussed in more detail in section 3.4, it is however worth reiterating a point that has often been neglected: method integration *facilitates*, and should be a pre-requisite to, tool integration (as illustrated in figure 3.1).

The objective of both method and tool integration is the production of an integrated software system. Therefore, *system integration* in this context refers to the development *product,* rather than the development process itself. Since a system consists of a number components connected together in some way, system integration is the process of establishing these connections. System integration is desirable in software development for many reasons. These include customer requirements for a single integrated deliverable, and because integrated systems have "emergent properties" [Checkland 1981] that are not necessarily possessed by their component parts.

Finally, for complex systems which have been developed from multiple perspectives or views, some form of *view integration* is often necessary. In multi-perspective software development, views may be used to represent development steps, notations, products or aspects of a system and therefore view integration is necessary for achieving system integration, which in turn can only be achieved by successful method and tool integration. The notions of "view" and "view integration" have different meanings in the literature depending on their context and are discussed in more detail in section 3.3.

3.2.2. Agenda

Thus far, this chapter has highlighted the desirability of separating concerns for reducing software development complexity, and coupled this with the need for integration of concerns to various degrees, at various stages of development. The choice of concerns to use as the basis for separation are many, and the scope for integration is wide. This thesis addresses the development of complex systems from multiple perspectives, and therefore particular requirements for separation and integration of concerns are more relevant than others.

On the one hand, separation of concerns based on the views, viewpoints or perspectives that developers hold on a problem or solution domain is desirable; while, on the other hand, integrating the methods by which these view, viewpoints or perspectives are developed is necessary in order to achieve integration of the development process and its products.

Thus, there is a need for *method integration for multi-perspective software development.* The next two sections review related research contributions in this area, and highlight outstanding problems that need to be addressed.

3.3. Multiple Perspectives

This section discusses the role and use of multiple "views", "viewpoints" and "perspectives" in software development. These terms are often used interchangeably in the literature, so unless otherwise stated, no explicit distinction is made between them in this chapter.

Two uses of the term "view" are particularly relevant in the context of this work. A view may represent an opinion, thought or perception of an item or area of concern. It should be expressible in some language or notation. Alternatively, a view may represent any one of the aspects, components or parts of something [Dictionary 1987].

The former use of the term allows partial or total overlap between views, while the latter implies disjoint areas of concern. Overlapping views may give rise to inconsistencies and conflicts, while disjoint views may need to be combined or linked into an integrated whole.

The following four sub-sections examine the use of multiple views in software engineering. The proliferation of multiple viewpoints in requirements engineering is examined in section 3.3.1. This is followed in section 3.3.2 by an exploration of multi-view software specification and programming, in which multi-paradigm development is presented as a special kind of multi-view development. Section 3.3.3 discusses the notions of inconsistency and conflict in multi-view development environments, and examines some approaches (such as negotiation and argumentation) that address these notions. Finally, the role and use of views in the database world are briefly examined in section 3.3.4.

3.3.1. Viewpoints in Requirements Engineering

The requirements of large, complex systems are often conflicting and contradictory, and almost always difficult to understand, elicit and specify accurately and completely. Customers may be unsure of their exact system requirements, and analysts may have a different understanding or interpretation of these requirements and the problem domain in which a required system will operate. Approaches which support multiple viewpoints in requirements engineering are therefore particularly useful, even necessary. A representative sample of such approaches is listed in table 3.1, and described in more detail below.

Approach	Definition of viewpoint
SA [Ross 1977]	An expression of interest
CORE [Mullery 1979]	An information processing entity
Viewpoint resolution [Leite & Freeman 1991]	A standing or mental position held by an individual examining or observing a universe of discourse
VOA [Kotonya & Sommerville 1992]	A service recipient
Goal-directed approaches [Dardenne, Fickas & van Lamsweerde 1993] [Robinson 1990]	A domain model with associated objectives ("a belief")

***Table 3.1:** Viewpoints in requirements engineering.*

One of the earliest uses of viewpoints in requirements engineering was in Structured Analysis (SA) [Ross 1977; Ross & Schoman 1977]. In SA (also described in chapter 2, section 2.1.2), a viewpoint expresses an interest in some aspect of a system. A viewpoint is also associated with a context and purpose, which together describe the orientation of an SA model. SADT [Ross 1985], the proprietary method based on SA, treats a viewpoint as a conceptual notion more than a formal vehicle for capturing and expressing system requirements. However, the SADT method has a rich notation and many heuristics for constructing multiple models of a system, which in turn support multiple perspectives in the requirements specification process.

The Controlled Requirements Expression (CORE) method [Mullery 1979; Mullery 1985] on the other hand, identifies viewpoints explicitly early on in the requirements elicitation and specification process, based on a systematic problem definition in which customer authorities are also identified. Viewpoints in CORE represent information processing entities, which are systematically analysed as the method unfolds. Viewpoints in CORE however are orthogonal, that is, they are non-overlapping. This is unrealistic and unsatisfactory if multiple viewpoints or perspectives on a real system are to be supported.

Leite and Freeman make a distinction between viewpoints, perspectives and views [Leite 1989; Leite & Freeman 1991]. A viewpoint is defined as a standing or mental position held by an individual examining or observing a universe of discourse. A viewpoint is identified by the individual holding it (such as that individual's name and role in the universe of discourse). A perspective on the other hand, is a collection of facts modelled according to a particular modelling scheme and a particular viewpoint. The representation of these facts in a particular representation scheme or notation therefore defines a perspective. A view in this context is an integrated collection of perspectives constructed systematically by a "view-construction process".

Based on the above definitions, Leite and Freeman propose a partial requirements elicitation technique for early validation of requirements. This so-called "viewpoint resolution" technique identifies discrepancies between two different viewpoints, classifies and evaluates these discrepancies, and integrates the alternative solutions into a single representation in a proposed language called VWP1. While viewpoint resolution is a useful requirements validation technique, it is very specific in its scope, requiring viewpoints to be represented in a single language (VWP1) in which all the analysis is then performed. Moreover the *early* validation of requirements somewhat diminishes the usefulness of having multiple viewpoints during requirements elicitation, where inconsistencies and conflicts are normally inevitable. Finally, viewpoint resolution does not address the outstanding and difficult problem of viewpoint communication which is necessary for viewpoint integration or reconciliation.

Kotonya & Sommerville propose a viewpoint-oriented analysis (VOA) approach to requirements elicitation, analysis and definition [Kotonya & Sommerville 1992]. This approach uses the notion of viewpoints to represent entities external to the system being analysed, which can exist without the presence of the system. Viewpoints in the VOA approach are service recipients, which may provide the system with data or control information that is necessary for service provision. While the services received by viewpoints represent functional requirements, these services may also be associated with constraints which represent non-functional requirements. VOA thus provides a framework for integrating functional and non-functional requirements, facilitating the early identification and resolution of inconsistencies when they arise.

The notion of a viewpoint in the VOA approach however is not entirely intuitive. A VOA viewpoint is not in any sense a “view” on the system since it is external to that system and interacts with it. Therefore, while the use of viewpoints in this context may be useful in trying to understand the behaviour of a system, it does not provide a framework or mechanism for representing multiple views of a system, or for tolerating the inevitable inconsistencies that exist during the requirements elicitation process.

Other approaches to requirements elicitation are less explicit in their use of the term viewpoint, but nevertheless make reference to it. For example, in [Dardenne, Fickas & van Lamsweerde 1993], a goal-directed approach to requirements elicitation is proposed in which system requirements are constructed by identifying goals, which may then be associated with agents, their actions, roles, responsibilities and so on. Goals identified in this manner may be used to provide the “roots” at which conflicts can be resolved and multiple viewpoints reconciled. In [Robinson 1989], Robinson proposes an approach to reconciling (“integrating”) multiple specifications (“viewpoints”) using (domain) goals. This is taken further in [Robinson 1990], in which negotiation is proposed as a vehicle for such reconciliation, and in [Robinson & Fickas 1994], in which collaborative requirements engineering is supported by a planner, domain abstractions and automated decision science techniques.

3.3.2. Multi-Paradigm and Multi-View Software Development

The use of multiple viewpoints in software development is not restricted to requirements engineering. It is often desirable to specify and implement different aspects or parts of a system, using different languages to do so. This section examines approaches that deploy multiple languages ("views") for specifying or implementing software systems. Based on the contention that multi-paradigm development approaches are a special kind of multi-view approach, approaches to multi-paradigm software development are also discussed. In fact, Meyers defines a multiparadigm environment to be a multi-view development environment in which views "... do not interact during editing. In such an environment, different parts of a single system may be developed using different languages or language paradigms, and the system as a whole is simply the union of its constituent parts." [Meyers 1993]. Table 3.2 lists the approaches that will be discussed.

Technique/Tool [Researchers]	Scope
PRISMA [Niskier, Maibaum & Schwabe 1989a]	Multi-View Specification
[Barroca & McDermid 1993]	Multi-View Specification
Conjunction as Composition [Zave & Jackson 1993]	Multi-Paradigm Specification
Viewpoint specifications [Ainsworth et al. 1994]	Multi-View Specification
Overlays [Rich 1981]	Multi-View Programming
PECAN [Reiss 1985]	Multi-View Programming
GARDEN [Reiss 1987]	Multi-Paradigm Programming
FIELD [Reiss 1990]	Multi-View Programming
Compositional approach [Zave 1989]	Multi-Paradigm Programming
Call gates [Spinellis 1994]	Multi-Paradigm Programming
Semantic Program Graphs [Meyers 1993]	Multi-View Specification/Programming

Table 3.2: *Multi-view and multi-paradigm development approaches.*

3.3.2.1. Specification

Software specification is a general term used to refer to both requirements and design description activities. Requirements specification approaches that are explicitly viewpoint-oriented were discussed in the previous section (3.3.1). This section examines a number of specification approaches that may be used for both requirements and design specification, and which are also viewpoint-oriented.

PRISMA is a pluralistic knowledge-based system for specifying software systems [Niskier, Maibaum & Schwabe 1989b; Niskier, Maibaum & Schwabe 1989a]. It is particularly suited to software requirements specification since it allows the simultaneous use of multiple viewpoints or formalisms, called views, to describe system requirements. PRISMA provides support for three kinds of views: Entity Relationship Models, Data Flow Diagrams and Petri Nets. These views are captured using Prolog as a meta-language. Thus, while the integration of views is not explicitly addressed, they are kept mutually consistent via view translation/consistency checks. This makes integration into a single formal specification much simpler, particularly if they are expressed in the same/unifying meta-language (Prolog).

The process of translating different views into a single meta-language for consistency checking however, has all the disadvantages of traditional centralised (database) architectures (described in chapter 2, section 2.1.3.4). In particular, meta-languages are difficult to design and extend, and there is a large overhead in translating all views into one language, especially in cases of dynamically changing views that evolve during development.

Barroca and McDermid describe another view-oriented approach to the specification of real-time systems [Barroca & McDermid 1993]. The approach accepts the premise that no single notation can represent all the requirements or characteristics of a real-time system, and proposes the combination of different notations to allow the representation of different views of, mainly reactive, systems. The proposed approach focuses on views of behaviour, functionality and data flow, and suggests that modecharts, timed statecharts and real-time logic are particularly important specification notations for such systems. The importance of understanding and specifying the relationships and communication protocols between these different views is also highlighted, as well as the need for a unifying (integration) framework.

Zave and Jackson propose the construction of system specifications by composing many partial specifications, each written in a specialised language that is best suited for describing its intended area of concern [Zave & Jackson 1993]. They further propose the composition of partial specifications as a conjunction of their assertions in a form of classical logic. A set of partial specifications is then consistent if and only if the conjunction of their assertions is satisfiable. The approach is demonstrated using partial specifications written in Z and a variety of state-based notations.

Zave and Jackson’s approach is powerful because it relies on a simple operator (conjunction) to compose partial specifications, but it assumes that classical logic may be used as an underlying universal formalism into which all the different languages may be translated. While this has the advantage of simplifying the analysis of specifications, it carries with it the overhead and drawbacks of using a common or universal formalism. Moreover, while the approach identifies consistency checking as problematic, the handling of inconsistent specifications is not addressed.

While Zave and Jackson do not use the term “view” explicitly in their approach, a partial specification written in a formal language may nevertheless be regarded as a view on the system being specified. Ainsworth et al. make this more explicit in their approach to making large specifications easier to understand [Ainsworth et al. 1994]. Their approach uses the notion of a “viewpoint” to represent a partial specification written in a formal language (all the example presented are in Z). Viewpoints represent self-contained partial specifications of a system, which may be amalgamated to give a description of the complete system. Amalgamation is a composite process in which the data and operations of individual viewpoints are considered separately. It is broken down into three steps:

- preparation - in which elements in the viewpoints to be amalgamated are identified (this may involve some simple pre-processing),
- co-refinement - in which syntactic (data co-refinement) and semantic (operation co-refinement) clashes are resolved, and,
- coalescence - in which the resultant non-interfering viewpoints are combined together into a single specification.

This approach may be useful for reducing the complexity of formal specifications written in a single language, but it is unclear how it may be applied to large-scale examples. It is also unsuitable for specifications written in multiple languages. Moreover, the amalgamation process, while described generally, appears difficult to generalise to specification languages other than Z. Finally, the notion of a viewpoints as “points of view” on a system, which are potentially conflicting or inconsistent, is not tolerated.

3.3.2.2. Programming

Programming is a composite activity that includes program editing, compiling, debugging and so on. This section describes a number of approaches and environments that support these different activities by providing multiple views within which to program. It also presents approaches to multi-paradigm programming, in which the usual assumption of a single implementation language no longer holds.

One of the earliest approaches that explicitly recognises the existence of, and need to represent, multiple, overlapping views in program development was proposed by Rich in 1981 [Rich 1981]. The approach proposes the modelling of programs and data structures using multiple “points of view”, and suggests the use of a formalism called “overlays” to support such an approach. An overlay is a triple made up of two “plans” (each plan representing a view) and a set of “correspondences” (logical equalities) between these two plans. The correspondences between plans represent the relationships between views and are not limited to programs and data structures.

While the approach may be feasible for small, simple programs and data structures, it requires considerable additional effort for larger scale program development. A programmer must systematically describe all plans in terms of “roles” (such as operations and tests) and constraints (such as data and control flows). The correspondences between elements of the plans then also have to be laboriously specified. Nevertheless, overlays are useful for viewing data structures for example, where it may be helpful to regard a single structure as an instance of several abstractions.

A significant body of work on multi-view software development environments has also been generated by Steven Reiss at Brown University. Three development environments are of particular interest: PECAN [Reiss 1985], GARDEN [Reiss 1987] and FIELD [Reiss 1990].

PECAN is a family of program development systems that support multiple views of a single developer’s program. These views may be different representations of a program or its corresponding semantics. Like Rich’s views, PECAN allows multiple views of shared data structures. PECAN views however may be inspected (viewed) concurrently and are automatically updated. All the views are internally represented as a single abstract syntax tree which is never seen by the software developer. What the developer does see are different views that may be edited interactively. Supported views include a syntax-directed editor view, a graphical flow chart view, a symbol table view, a data type view, execution views and others. PECAN however is a single user environment that supports multiple views during program construction, and in which all programs are ultimately represented as abstract syntax trees. Scott Meyers (a former student of Reiss) recognised that abstract syntax trees are insufficient to represent the wide range of views that software developers may require. He therefore proposed another canonical representation, based on a graph structure called “semantic program graphs”, which provides more expressive capabilities for a wide range of semantics employed in software development. These include programming languages such Ada and Pascal, and specification notations such as Petri Nets [Meyers 1993].

Reiss on the other hand identified a need to create new views dynamically during development. This he addressed in the GARDEN environment and family of tools. GARDEN supports the creation of multi-paradigm visual languages by using an object-oriented model, with a common base language, to define the type structure, syntax and semantics of different languages. Within this framework programs are represented as objects, and the interaction between programs is modelled using conventional object-oriented message passing protocols. GARDEN reduces the conceptual bottleneck of the common underlying representation of PECAN, and replaces it with a common base language that is visible to the user/programmer.

Following on from GARDEN, FIELD is a tool integration environment in which tools interact via message passing, coordinated by a message server. Views in FIELD represent the different products produced by the different tools. The basic idea in FIELD is that existing (Unix) tools may communicate if they are modified to send and receive messages. A message server regulates these messages, ensuring that each tool receives only those messages that are of interest to it. This approach circumvents the bottlenecks imposed by having the common representation and base language of the PECAN and GARDEN environments respectively, but it does create a new communication bottleneck in the shape of the message server. The FIELD environment is nevertheless feasible for multi-view software development, and its approach has been commercially exploited in environments such as Hewlett-Packard's SoftBench [Cagan 1990].

In [Zave 1989], Zave also suggests that a compositional approach to multi-paradigm programming is more fruitful than approaches that rely on a single (merged) multi-paradigm language [Hailpern 1986]. Zave proposes that a multi-paradigm program is written as a collection of single-paradigm programs which are then composed, that is, allowed to execute concurrently and cooperatively to form a whole functioning system. Zave tests this kind of composition on a number of paradigms including CSP [Hoare 1985], Prolog, Awk [Aho, Kernighan & Weinberger 1988] and Paisley [Zave & Schell 1986]. The procedure or function call however remains the most widely used conceptual composition vehicle, although recent alternatives have been suggested, such as the conjunction operator described in the previous sub-section [Zave & Jackson 1993].

Spinellis proposes an approach to multi-paradigm programming in which paradigms are treated as object classes, and multi-paradigm programs as instances of these classes [Spinellis 1994]. This allows for abstracting common characteristics across paradigms, and for designing multi-paradigm environment generators and implementing multi-paradigm programming support environments. Inter-operation between paradigms is accommodated by an abstraction called an (import or export) "call gate". The role of a call gate is to implement data and control transfer functions, and to document the transfer conventions used and expected. Unfortunately, while this approach is conceptually sound (and is amply supported by fully implemented demonstrators), it is heavily implementation-oriented, and fundamentally, interoperability is provided by the underlying implementation language (C in this case). Like the GARDEN environment however, the approach however does provide a unifying (object-oriented) framework for developing multi-paradigm environments and applications, even though the scope for supporting "upstream" software development activities such as specification and design appears to be limited.

3.3.3. Representing and Managing Multiple Views

The difficulties in integrating multiple views for multi-view software development may be alleviated by appropriately representing and managing these views. For example, Shilling and Sweeney extend the object-oriented paradigm in order to support both users and developers of multi-view environments [Shilling & Sweeney 1989]. They propose that views may be implemented by (1) defining multiple interfaces in object classes, (2) controlling the visibility of instance variables, and (3) allowing multiple copies of an instance variable to occur within an object instance.

Meyers identified five approaches to integrating multi-view software development systems, which, to varying degrees, derive their integration capabilities from the way in which information is represented [Meyers 1991]. The five approaches are:

- Shared file system: in which tools (views) communicate by reading and writing to shared files. The different tools have their own internal representation schemes, and the file system is effectively the integration mechanism.
- Selective broadcasting: in which tools communicate via message passing, usually moderated by a message server.
- Simple databases: in which integration is achieved through the use of a common database. This is similar to the shared file system approach, except that databases provide a more formal and structured integration agent than the flat file.
- View-oriented databases: in which the simple databases approach is extended to allow different views on the shared database to have different underlying data structures which may be changed independently. Since the views are on the shared database, consistency is automatically maintained by the database system.
- Canonical representations: in which a single representation is shared by all tools. Meyers favours this approach and has suggested the use of a single canonical representation into which all views may be translated and then related [Meyers & Reiss 1991].

In almost all the implementations of the above approaches, an overriding requirement is the need to maintain consistency between the various views or the tools that support the development of these views. Various frameworks, models, techniques and tools for managing consistency have been proposed. For example, Hagensen and Kristensen classify and formally define various techniques for *consistency handling* in software development [Hagensen & Kristensen 1992]. Notions of inconsistency and conflict however are usually treated as undesirable, and integration frameworks and representations are designed to disallow such inconsistencies and conflicts, or at least identify and eradicate them when they occur. This however is not universally accepted, and table 3.3 summarises approaches that advocate *inconsistency handling.*

Approach	Mechanism	Scope
Living with inconsistency [Schwanke & Kaiser 1988]	Smarter recompilation	Programming (Configuration Management)
Tolerating inconsistency [Balzer 1991]	Pollution markers	Programming
Making inconsistency respectable [Gabbay & Hunter 1991]	Meta-level temporal rules	(Logic) Databases
Lazy inconsistency [Narayanaswamy & Goldman 1992]	Announce proposed changes	Cooperative software development

Table 3.3: *Inconsistency handling in software development.*

Schwanke and Kaiser suggest that during large systems development, programmers often circumvent strict consistency enforcement mechanisms in order to get their jobs done [Schwanke & Kaiser 1988]. In the configuration management project CONMAN, an approach to "living with inconsistency" during development is proposed. The CONMAN programming environment helps programmers handle inconsistency by:

- identifying and tracking six different kinds of inconsistencies (without requiring them to be removed),
- reducing the cost of restoring type safety after a change (using a technique called "smarter recompilation"), and,
- protecting programmers from inconsistent code (by supplying debugging and testing tools with inconsistency information).

Balzer proposes the notion of "tolerating inconsistency" [Balzer 1991] by relaxing consistency constraints during development. The approach suggests that inconsistent data be marked by guards ("pollution markers") that have two uses: (1) to identify the inconsistent data to code segments or human agents that may then help resolve the inconsistency, and (2) to screen the inconsistent data from other segments that are sensitive to the inconsistencies. The approach does not however provide any mechanism for specifying actions that need to be performed in order to handle these inconsistencies.

Gabbay and Hunter suggest "making inconsistency respectable" [Gabbay & Hunter 1991] by proposing that inconsistencies be viewed as signals to take external actions (such as "asking the user" or "invoking a truth maintenance system"), or as signals for taking internal actions that activate or deactivate other rules. Again, the suggestion is that "resolving" inconsistency is not necessarily done by eradicating it, but by supplying rules that specify how to act in the presence of such inconsistency. Gabbay and Hunter further propose the use of temporal logic to specify these meta-level rules.

"Lazy" consistency is proposed as the basis for cooperative software development [Narayanaswamy & Goldman 1992]. This approach favours software development architectures where impending or proposed changes, as well as changes that have already occurred, are announced. This allows the consistency requirements of a system to be "lazily" maintained as it evolves. The approach is a compromise between the optimistic view in which inconsistencies are assumed to occur infrequently and can thus be handled individually when they arise, and a pessimistic approach in which inconsistencies are prevented from ever occurring. A compromise approach is particularly realistic in a distributed development setting where conflicts or "collisions" of changes made by different developers often occur. Lazy consistency maintenance supports activities such as negotiation and other organisational protocols that support the resolution of conflicts and collisions.

Of course, some inconsistencies and conflicts cannot be tolerated indefinitely. Therefore, techniques for conflict resolution between conflicting views are often necessary. Easterbrook proposes a model of conflict resolution with computer-supported negotiation [Easterbrook 1991]. The model is particularly suitable for requirements elicitation from multiple perspectives by multiple development participants, in which these perspectives may be conflicting or contradictory. Conflicts are broken into components, and the issues underlying the disagreements are identified. Possible resolutions are then generated and evaluated according to the criteria associated with the issues. Future viewpoints or perspectives may be generated by this process.

Negotiation is also the technique adopted by Finkelstein and Fuks, who propose a formal dialogic framework for collaborative development and conflict resolution between interacting viewpoints [Finkelstein & Fuks 1989]. The framework has wider applicability in area of software specification, where specifications are constructed from multiple viewpoints and in which these viewpoints “negotiate, establish responsibilities and cooperatively construct a specification.”. In its proposed form however, it is only suitable for dialogue between two parties, and requires extension to handle multi-party dialogue.

Conflict resolution may also be regarded as part of the process of achieving integration between multiple perspectives [Robinson 1989]. Robinson defines perspectives as alternative and conflicting beliefs, and not projected views of a consistent belief set. Perspectives are created by associating a domain model with objectives. Once a perspective is associated with a specification, it may then be integrated with specifications of other perspectives. Again, negotiation is used as a vehicle for conflict resolution [Robinson 1990].

Finally, the notions of conflict resolution and negotiation are both part of a process of argumentative development. “Argumentation is the activity of reasoning about a design” [Fischer, McCall & Morch 1989]. It includes activities such as discussion and negotiation, which may be used to improve the quality of a product and to resolve conflicts that may arise during a development process. VIEWPOINTS is an issue-based hypertext system containing useful information for the design of kitchens [Fischer, McCall & Morch 1989]. It is an example of a graphical tool providing kitchen designers with arguments for and against answers to specific design problems. It further illustrates the power of using hypertext to navigate through non-linear information structures.

3.3.4. Views on Databases

Finally, it is worth making a few comments about the use of multiple views in the database literature. Broadly speaking, two approaches exist. In the first approach a view on a database is used to represent a projection on that database, in which partial database information is extracted and presented in that view. The modification of one view modifies the underlying database and is usually associated with a “trigger” operation for modifying (updating) any other related or overlapping views on that same database. View integration in this context, is a measure of data integrity in the database. Well integrated views means that changing one view on the database ensures the correct and immediate change of all other related views [Dayal & Hwang 1984; Batini, Lenzerini & Navathe 1986; Navathe, Elmasri & Larson 1986; Bouzeghoub & Comyn-Wattiau 1991].

The second approach is the use of views in the increasingly popular world of multidatabases. As discussed in chapter 2 (section 2.1.3.4), a multidatabase is a set of autonomous databases, managed together without a global schema. In this setting each of the constituent databases contains partial information and may therefore be regarded as a view. View integration in this context means integrating the different constituent databases via a number of techniques. These include the use of a multidatabase management system, a multidatabase language, or via direct one-to-one database system interoperability [Litwin, Mark & Roussopoulos 1990; Ram 1991; Bright, Hurson & Pakzad 1992].

Multidatabases appear to be more attractive than traditional (centralised) databases, since they are better models of software engineering repositories, being more amenable to distribution and offering customised representation and optimised retrieval operations for individual developers. Moreover, multidatabases offer more scope for tolerating inconsistencies [Rusinkiewicz, Sheth & Karabatis 1991] if required, since the view integration mechanisms are separate from other database operations.

3.4. Method Integration

A software development method is a set of procedures and heuristics for deploying one or more notations for describing a software system. The descriptions produced include the specification, implementation and associated documentation of that system. As discussed in chapter 2 (section 2.1.1), different methods are suitable for different stages of development and for different application domains. “Constructing a new method for a given purpose from two or more existing methods is called *method integration* ... The aim of method integration is to combine the strengths and reduce the weaknesses of the selected methods when applied to the engineering process in question.” [Kronlöf, Sheehan & Hallmann 1993].

This section discusses different approaches to method integration. Research work in this area is limited, but many of the techniques and tools discussed provide the building blocks for such integration. In particular, related work on tool integration is examined.

3.4.1. Philosophy

Chapter 2 illustrated the benefits of method-based software development in terms of improved developer productivity and software quality. In a single software development project, different developers may use different methods for different development stages or different problem domains. Method integration is a means of providing more customisation and greater flexibility in development, which in turn, also improves productivity and software quality. Since a large software development project is likely to employ many, physically distributed, development participants with conflicting, complementary or different views, a method integration framework must be able to handle both the decentralisation[5] of, and the inconsistencies between, development participants and the views that they hold.

Approaches to method integration include: (i) integration by *common features* such as a common underlying model, (ii) *restricting* the investigation to compatible subsets of methods such as language subsets, (iii) *extending* one "main method" with ideas or features taken from other "supplementary methods", or (iv) *strengthening* a main method by replacing or overriding some of its ideas with ideas taken from supplementary methods [Kronlöf & Ryan 1993]. In practice however, the integration of methods is *ad hoc* and is a side-effect of tool integration. This is unfortunate, since tool integration is the *implementation* of the *intentions* of method integration [Wybolt 1991]. As a result, much of the research work on integration addresses frameworks for achieving tool integration and providing mechanisms for interoperability and sharing of information between tools. While such frameworks and mechanisms are relevant in the context of method integration, they are fundamentally implementation-oriented and often do not address the more conceptual and semantic issues of method integration[6].

The remainder of this section (3.4) examines current approaches to integration with particular emphasis on those aspects relevant to method integration. Critical evaluations of these approaches are also made, and in particular, their suitability for multi-perspective software development is examined. Section 3.4.2 discusses general integration frameworks and architectures, and is followed in section 3.4.3 by an examination of the integration mechanisms and protocols that may be used within these frameworks and architectures.

[5] We adopt the following convention: a decentralised system is also distributed, whereas a distributed system may be only partly (e.g., logically) decentralised. For example, a distributed system may have decentralised control (e.g., autonomous workstations) but centralised data (e.g., a shared database).

[6] The emphasis on, and motivation behind, tool integration however is understandable. There already is a huge investment in current CASE technology (e.g., tools, training, customer base, etc...), and therefore integration between different (especially multi-vendor) tools is essential.

3.4.2. Frameworks and Architectures

An integration *framework* defines the setting or context within which tools, techniques and software products are related and may be examined. It often also outlines the philosophy of the integration approach deployed. An integration *architecture* on the other hand, describes in more detail the building blocks of the framework, with some indication of how these building blocks interact[7].

In the Eureka Software Factory (ESF) architecture for example [Fernström, Närfelt & Ohlsson 1992], a layered view of communication flows is proposed as part of a "software factory" conceptual reference model (CoRe) [Gera et al. 1992]. Communication in ESF is performed across a single "software bus", although the building blocks of the architecture (people, tools and platforms) perceive direct one-to-one communication (figure 3.2). While the ESF architecture is a step towards decentralisation, its software bus is a communication bottleneck that is normally associated with integration architectures that have strictly centralised control. Furthermore, while the ESF architecture describes interactions among people, platforms and tools, it does not specifically address *method* interactions.

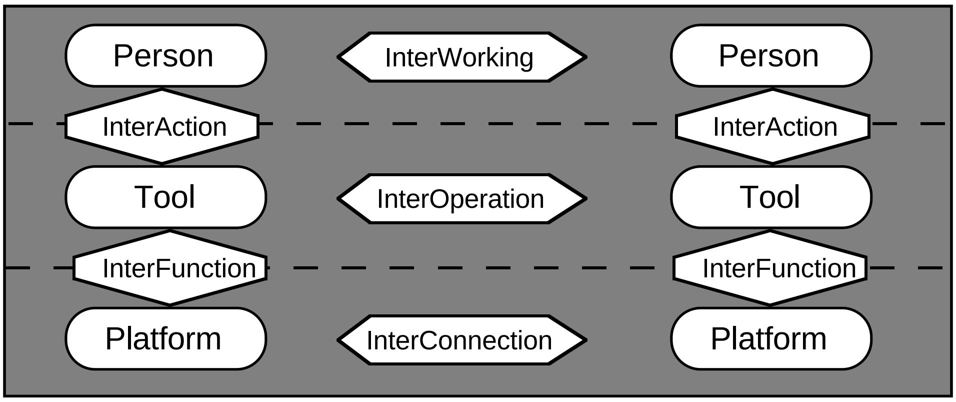


***Figure 3.2:** Communication across the ESF "software bus".*

In general, centralised integration architectures (figure 3.3a) rely on a common ("centralised") entity (e.g., a repository or server) to facilitate integration. Decentralised integration architectures (figure 3.3b) on the other hand, rely on (often ad hoc) one-to-one integration of the constituent components. A whole range of architectures between these two extremes exist. For example, some physically centralised databases are logically distributed (that is, they may 'appear' as different databases to different parts of an organisation), and some distributed systems have centralised control (e.g., in a client-server system, distributed clients rely on a centralised server for control).

[7] In this context, the terms "framework" and "architecture" are often used interchangeably. The term "software architecture" however, is also used to refer to a high level design description of a software system expressed in an appropriate notation [Garlan & Shaw 1993].

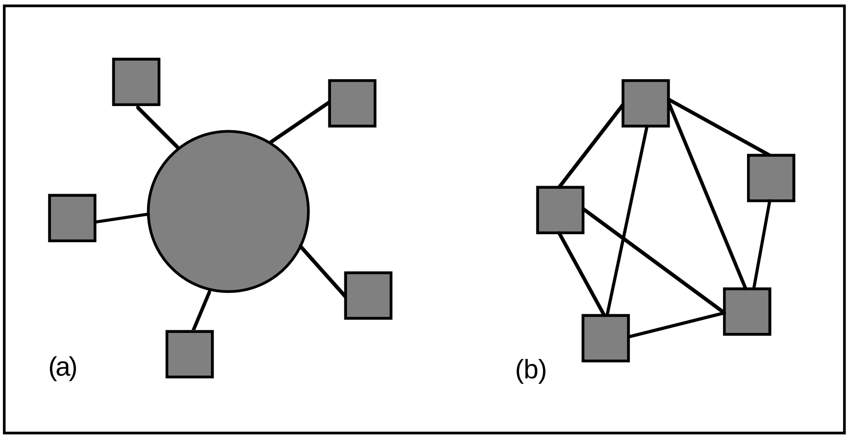


Figure 3.3: *Centralised versus decentralised integration architectures.*

Integration architectures have been specifically addressed by researchers on tool integration. “A current problem in tool integration is the general lack of agreement of tool developers over the appropriate integration mechanisms, levels of integration, and standards for presentation, data and control integration” [Wasserman 1990]. Presentation integration addresses the need to have a coherent and consistent visual view of information generated by different tools. This normally involves some standardisation of the tools’ user interfaces, for example by running these tools under the same windowing system.

Data integration normally refers to the process of building a consistent body of information produced by different tools. This allows the syntax and semantics of data produced by one tool to be understood and used by another. Control integration on the other hand, is the process of coordinating and synchronising the operations and functions of different tools so that they work cooperatively together. This may include the invocation of one tool from another. Control integration is sometimes also called process integration, but strictly speaking it is only a subset of it. Process integration tends to encompass organisation-wide processes which are not necessarily tool-oriented.

While data, control and presentation integration are different facets of (tool) integration, they need to be addressed collectively in order to achieve the eventual common goal of constructing a software system. An integration framework or architecture serves this purpose.

In section 3.3.3, five alternative approaches to integrating multi-view software systems using a shared file system, selective broadcasting, simple databases, view-oriented databases and canonical representations were discussed [Meyers 1991]. All these approaches are, to varying degrees, instances of the centralised integration architecture shown in figure 3.3a, in that they rely on a shared entity (database, file, message server or representation) to provide the integration.

The shared repository however, remains the primary integration agent of most integration architectures [Yin 1994] such as PCTE (described in section 2.1.3.2). This has led to a number standard reference models (such as CDIF and IEEE 1175) that define the nature of information exchange between tools (section 2.1.3.4). Data integration is achieved within such architectures via the use of a common schema or meta-model of that data. The Software through Pictures (StP) development environment integrates multiple representation schemes (“views”) using this approach [IDE 1991].

A shared message server, such as that deployed in FIELD and the HP SoftBench environments, is an alternative integration architecture which provides better control integration. The message server is used to channel messages to and from different tools, and thus performs selective broadcast which reduces, but does not eradicate, the centralisation of the system.

Support for decentralisation in such integration architectures is important because of the increasingly distributed nature of software development. Unfortunately, commonly used repositories are not appropriate for supporting distributed development. Shared repositories are based on single (monolithic) meta-models or schemas that are difficult to distribute, understand, maintain and extend. They can therefore evolve into bottlenecks rather than into integration agents of software development environments.

The ESPRIT project Atmosphere [Obbink 1991] addresses method integration in some detail. While accepting the different facets of tool integration identified above, two facets of method integration are also identified. These are: *intra-process integration*, which addresses the combination of different methods, and *inter-process integration*, which addresses the combination of configuration management with system integration.

The focus of Atmosphere then is method composition within and between various development processes. This is explored via a number of different case studies that attempt to perform *pairwise* integration of different methods [Kronlöf 1993a]. The result of such a method integration exercise has highlighted number of problematic issues, useful guidelines and lessons learned. These include the need to resolve *terminology* clashes between methods, identify the different *hierarchies* that different methods deploy, identify and resolve *inconsistencies* in method and tool products, and provide or use *transformation* tools for traceability and consistency checking [Kronlöf 1993b]. Other individual attempts at pairwise integration of methods, particularly formal and structured methods [Semmens, France & Docker 1992; Polack, Whiston & Mander 1993], also confirm these findings.

While Atmosphere and other method integration projects raise a number of important issues surrounding method integration, no general framework within which such integration issues may be addressed has emerged. Kronlöf and Ryan however, do suggest a systematic approach to method integration, comprising of the following steps which need to be performed in order to produce a “new” integrated method [Kronlöf & Ryan 1993]:

1. Define the target process of the integrated method, including the definition of development steps (development procedures) and the ordering of these steps.
2. Select the methods to be integrated, ideally because they support the target process, but frequently for political, historical or pragmatic reasons (such as the availability of tool support or local expertise).
3. Define the underlying model of the integrated method in order to identify the classes of objects represented, manipulated and analysed by this method. An underlying model of a method provides a representation of the product(s) generated by that method.
4. Define the language mappings between (subsets of) the languages of the "original" methods. Since an underlying model of the integrated method has already been defined (in step 1), the potentially large number of pairwise mappings ($(n^2 + n)/2$ for n languages) is reduced to n mappings (that is, one mapping from each language into the underlying model). Of course, this assumes that pairwise mappings are required between *all* the original languages, which is not always the case.
5. Provide guidance to the user on how to use the new integrated method. Ideally, much of the guidance information should be derived from the definition of the development process (that is, from a process model of that process), but case studies, examples and heuristics are also important sources of such method guidance. Integrating the guidance provided by many methods however, remains problematic.

While the above steps describe an approach to method integration, what is also needed is a method integration *framework*, to facilitate the combination of *any* two or more methods in such a way that above issues (plus inconsistency handling and tool support) may be addressed in a systematic way. Kronlöf and Ryan suggest the use of a "meta-method" for method integration, though the construction of such a method may be very difficult (and almost amounts to a quest for a "universal" method). The need for such an organisational framework has also been echoed by [Norman & Chen 1992], who propose a framework for integrated CASE [Chen & Norman 1992]. The framework however, like many others, primarily addresses tool integration rather than method integration.

3.4.3. Protocols and Mechanisms

Whereas frameworks and architectures provide the context within which integration may be achieved, protocols and mechanisms are the vehicles for achieving such integration. A *protocol* specifies the rules, conventions and activities by which entities interact, while a *mechanism* implements such a protocol. Since integration is a process of combining together two or more entities (e.g., tools, methods, systems, etc...), communication protocols and mechanisms are needed to perform this combination.

Communication protocols between interacting entities differ in their objectives. This imposes different requirements on the granularity and synchronisation of the information exchanged. In DAI for example [Weihmayer & Brandau 1990], a common objective is cooperative problem-solving by multiple agents [Shoham 1990; Jennings & Mamdani 1992; Werner 1992; McCabe 1993]. These agents have individual or common goals, and interact according to one or more well-defined protocols in order to achieve these goals.

A well known protocol deployed in many multi-agent systems is negotiation using Contract Nets [Smith & Davis 1981]. In this protocol an explicit agreement (contract) is established - after a bidding process - between a manager node that generates a task, and another contractor node willing to execute that task. This contract is then used as a basis for a private two-way communication (transfer of information) until the task is executed to completion by the contractor[8].

White and Purtilo define an interaction protocol in the context of distributed systems as “a set of rules and conventions that a module must follow to communicate or synchronise with another module” [White & Purtilo 1992]. The remote procedure call (RPC) [Birrel & Nelson 1984] is one such protocol used in many distributed systems deploying a client-server architecture [Evans 1992]. This protocol relies on node-to-node message-passing in which a client sends the name of a procedure and the procedure’s parameters to a server and then waits for a response. The server on the other side, waits for a request to arrive, processes it locally when it does arrive, and then returns the result to the client. The client in this protocol is aware of the server and delegates the execution of particular tasks to that server. The RPC protocol may be used in the context of other cooperative protocols which have more general cooperative goals.

In general, client-server architectures classify communication protocols under two broad headings: data shipping (e.g., caching) and function shipping (e.g., remote service, RPC) [Levy & Silberschatz 1990; Satyanarayanan 1991]. In a function shipping protocol, the client sends data and the operation code to a server, then possibly waits for a result. Implementations of this kind of protocol are widespread, but may suffer from communication load overhead and potentially high network traffic. In a data shipping protocol on the other hand, the client asks/fetches information from a server and performs operations on that information locally. Implementations of this protocol may be inefficient if there is a high access/operation invocation frequency, which may be needed if there is a high rate of change of the fetched server information.

Collaborative work architectures also deploy a variety of communication protocols. The PACT architecture for example offers the infrastructure for clusters of agents (programs that encapsulate

[8] We adopt the terminology used by [Werner 1992] in which cooperation/collaboration requires communication. Communication may result in conflict which may be resolved by negotiation.

engineering tools) to interact through facilitators that "translate tool-specific knowledge into and out of a standard knowledge-interchange language." [Cutkosky et al. 1993]. This is a so-called "federated architecture". An alternative to this approach is demonstrated in the MIT Dice architecture in which a network of agents communicate through a shared workspace called a "blackboard" [Sriram & Logcher 1993]. An agent in Dice is a combination of a user and a computer.

Finally, multi-paradigm programming (discussed in section 3.3.2.2) may also need communication protocols for integration. Multi-paradigm systems require the coordination, cooperation and synchronisation of a number of program fragments that together provide the overall desired functionality of the system. In contrast with multi-agent systems in which a specific objective is goal-directed cooperative problem-solving, multi-paradigm programming systems bear a closer resemblance to multi-view systems [Meyers 1991] in which multiple program fragments are treated as views on a problem or solution domain. In this setting then, communication protocols support activities such as *interoperability* [Wileden et al. 1990; Wileden et al. 1992], *transformation* [Wile 1992], *merging* [Feather 1989a; Feather 1989b; Lippe & van Oostrom 1992] and *composition* [Zave & Jackson 1993] of views. Table 3.4 lists various techniques for integrating multiple partial specifications. These techniques may be used as the building blocks of method integration and are not necessarily mutually exclusive.

Technique	Example mechanisms deployed [Researchers]
Composition	1. Conjunction [Zave & Jackson 1993] 2. Amalgamation [Ainsworth et al. 1994]
Interoperability	1. Specification level [Wileden et al. 1992] 2. Programming level [Hayes & Schlichting 1987] 3. User level (negotiation); e.g.: • contract nets [Smith & Davis 1981], • using domain goals [Robinson 1990]
Merging	1. Operation based [Lippe & van Oostrom 1992] 2. Evolutionary transformations [Feather 1989a; Feather 1989b]
Transformation	1. Syntax integration [Wile 1992] 2. Meta-system approach [Boloix, Sorenson & Tremblay 1992]

Table 3.4: *Some techniques for (method) integration.*

To reiterate briefly, formal composition is the process of combining together formal partial specifications which describe different, possibly overlapping views, of a system. As discussed in section 3.3.3.1, Zave and Jackson propose the translation of partial specifications into first order predicate logic, and then use the conjunction operator to compose these specifications. Ainsworth et al. propose the use of a composite process called amalgamation for systematically analysing and combining partial specifications written in Z.

Interoperability is a means for entities to communicate or work together. In the context of multi-perspective software development these entities may be views, agents, specifications, programs and so on. Hayes and Schlichting for example, propose an approach to constructing distributed mixed language programs by adding a generic RPC facility to each constituent language [Hayes & Schlichting 1987]. The Mixed Language Programming (MLP) system they propose facilitates program interoperability by using a distinct process for each program component and a data description language, called the Universal Type System (UTS) language, to specify system interfaces and type data. Wileden et al. address the type model aspect of interoperability, and propose a model for supporting so-called Specification Level Interoperability (SLI) [Wileden et al. 1990; Wileden et al. 1992]. SLI defines type compatibility in terms of the specification (properties) of objects, rather than the representation (structure) of those objects. This allows interoperating programs to communicate directly in terms of higher-level, more abstract types, and hides low-level details of these programs' representations from each other.

Negotiation is also a form of interoperability between communicating parties (normally human agents). For example, the contract nets protocol described in this section uses negotiation to solve problems and resolve conflicts, and Robinson uses it as a vehicle for integrating perspectives in requirements specification [Robinson 1990].

Transformation is a fundamental technique that is deployed to some degree in almost all integration approaches. For example, in both the composition and interoperability approaches described above, some transformation of information takes place, such as translations into predicate logic or a universal type system, respectively. Wile addresses the issue of integrating specifications or programs written in different languages, by proposing the use of transformations (mappings) between the grammars of those languages [Wile 1992]. This is facilitated by the use of parameterised grammars described in an extension of BNF. The technique is supported by an automated system called Popart, which among other things generates languages from their grammar definitions and facilitates the translation through direct transformation between these languages. The Metaview system described in chapter 2 (section 2.1.3.3) also uses a transformational approach based on the development of a set of transformation rules expressed in a transformational language [Boloix, Sorenson & Tremblay 1992]. Transformations between "environments" are made by using a common formal model (entity-aggregate-relationship-attribute - EARA) for representing source and target specification environments.

In the context of multi-perspective software development, merging is the process of combining the results of parallel development activities. This is a particularly difficult to achieve if the activities are interfering (e.g., during distributed, cooperative development) or the results of these activities are overlapping (e.g., version control). Many merging techniques also make use of different kinds of transformations.

In operation-based merging [Lippe & van Oostrom 1992], development operations are recorded in the form of "transformations", which themselves are composed of object management system operations called "primitive transformations". Conflicts between primitive transformations can then be resolved by a user through: (i) imposing an ordering on primitive transformations, (ii) deleting a primitive transformation, and (iii) editing a transformation if the previous options have not been sufficient.

Feather proposes the use of "evolutionary" transformations [Feather 1989b] in the process of constructing specifications by combining parallel elaborations [Feather 1989a]. Evolutionary transformations are incremental elaborations (refinements and adaptations) of different specifications, which may then be replayed in serial order to merge these specifications. The parallel development of specifications achieves separation of concerns in a distributed development setting, while the merging integrates these specifications - detecting and highlighting interference (such as conflicts and inconsistencies) in the process.

3.4.4. Evaluation

Clearly, method integration approaches need to address a large number of diverse issues. Techniques for combining partial specifications are needed, and in many cases this requires more than simply composing these specifications using some "special" operators. Exchange of information, transformations and conflict resolution may also be necessary. If development is taking place in a concurrent and distributed setting, then cooperative and collaborative communication protocols and mechanisms are needed to support this kind of development.

Brinkkemper introduces the notion of "modelling transparency" in order to understand and use the relationships between different partial specifications [Brinkkemper 1993]. Four degrees of modelling transparency between CASE tools are identified, ranging from stand-alone tools with non-accessible repositories which do not support modelling transparency, to hypertext-like tools in which the relationships between any two tools may be established, thus supporting a high degree of modelling transparency.

Consistency checks or actions between partial specifications may be modelled as relations between these specifications [Castelfranchi, Miceli & Cesta 1992]. Therefore, it is important to understand and explicitly express relationships between partial specifications in order to perform the various integration techniques described in the section above.

Other considerations that must be taken into account, especially in a distributed environment, include the ability to identify what partial specifications are related and where they reside in a distributed system or network. Gasser lists three alternative definitions of agent *identity* in multi-agent systems [Gasser 1992]:

1. The identity of an agent may be defined by specifying a *unique identifier* to it. This definition is popular in distributed systems where an identifier may be a machine name or network address for example.
2. An agent may be given an identity by some *specification of its behaviour*. In other words, by stating the set of actions that the agent can perform.
3. The identity of an agent can also be defined by *the relations that other agents have towards it.* That is, the identity of an agent may depend upon how other agents react to its behaviour.

The diversity of the issues involved, clearly motivates the requirement for an organisational and integration framework within which these issues may be addressed. This framework should allow distributed development by multiple development participants, and should facilitate the integration of the various products generated by these participants. The framework should also facilitate the construction and deployment of tool support, which in turn necessitates the provision of integrated method engineering capabilities upon which to base such support. Clearly, the framework should facilitate method construction and integration, in order to support multi-perspective software development.

Finally, the ideal integration framework must also avoid the traditional bottlenecks that arise from the use of centralised architectures that rely on universal and monolithic representation schemes for integration. Thus the framework in this setting must be distributable, support multiple methods and views, and allow incremental integration [Lindstrom 1993] and reuse.

3.5. Chapter Summary: Research Requirements and Objectives

This chapter has presented two main bodies of work. On the one hand, the desirable “separation of concerns” principle was examined, and various “concerns” were identified as useful for reducing the complexity of software systems. The notions of “view, “viewpoint” and ”perspective” were examined as vehicles for separation, and their varied deployment in software engineering applications was discussed.

On the other hand, the “integration” of separated concerns was also examined. Frameworks and architectures for integration were discussed, and in particular the techniques, protocols and mechanisms that constitute the building blocks of such integration frameworks and architectures were explored.

To summarise, this chapter has specifically motivated the need:

1. *To separate concerns during software development* in order to reduce the complexity of the software development process and the product(s) generated by such a process. This includes allowing multiple, possibly inconsistent, views on a software system, in order to support multiple development participants working in a distributed setting. Thus, the chapter

motivated the need to support multi-perspective software development.

2. *To provide an organisational framework for integrating of methods, tools and views,* that is generic and distributable. Such a framework allows the desirable separation of concerns, and may be populated with the techniques necessary to integrate such concerns. The framework thus supports method engineering and integration, facilitates the provision of integrated tool support, and allows multiple development participants to hold multiple views.
3. *To provide techniques for identifying, expressing and checking relationships between concerns,* in order to use these techniques for method, and consequently tool and view, integration. These techniques include combining partial specifications (“views”), checking consistency and handling inconsistencies between them, and allowing them to interoperate collaboratively. In all these techniques, various kinds of transformations must also be supported.

The objective is therefore to make use of individual integration techniques within a framework for method integration, in order to provide for the combination of different methods that are most suitable for particular organisations or applications.

While the literature explored in this chapter offers advances in many areas of multi-perspective software development and integration, these are not collectively addressed within a uniform framework. In particular, method integration is addressed, if at all, in an ad hoc manner. No systematic approach to utilising the available techniques is adopted. Moreover, there is no explicit support for multiple views in this context.

Like chapter 2, this chapter has covered a diverse range of software engineering advances and concerns. This is because the ViewPoints framework, described in the next chapter, addresses a number of these issues and concerns. Again, each of the topics covered in this chapter will be revisited later in the thesis to evaluate the author’s contribution to the current state-of-the-art.

Chapter 4 The ViewPoints Framework

This chapter describes the ViewPoints framework for multi-perspective software development. It defines the notion of a "ViewPoint", the various kinds of knowledge it encapsulates, and the way this knowledge is separated and structured into "slots". The notion of a "ViewPoint Template" is also introduced and defined, and presented as a form of reusable, primitive method. All concepts are illustrated by example fragments. The chapter goes on to evaluate the framework, contrasting the strengths of a multi-perspective development approach with the difficulties in achieving integration in such a setting. In particular, this chapter outlines issues relating to tool support, process modelling, inter-ViewPoint consistency checking and inconsistency handling. These issues are addressed in more detail in subsequent chapters.

The work described in this chapter addresses many of the objectives originally outlined in [Finkelstein et al. 1989], and builds upon the framework described in [Finkelstein, Goedicke & Kramer 1990]. The ViewPoints framework as described in this chapter however, is the author's own original contribution.

4.1. Motivation and Objectives

This section reiterates the motivation for developing the ViewPoints framework by describing the so-called "multiple perspectives problem". This is then used to outline the objectives of the work, namely, to support multi-perspective software development in a distributed setting.

4.1.1. Motivation: The Multiple Perspectives Problem

The development of most large and complex systems involves many people, each with their own perspective on the system defined by their skills, responsibilities, knowledge and expertise. This is particularly true for composite systems which deploy a number of different technologies. For example, a lift system is typically composed of separate software, electrical and mechanical components.

Inevitably, the different perspectives of those involved in the development process of such systems intersect and overlap, giving rise to a requirement for consistency checking and coordination. The intersections, however, are far from obvious because knowledge within each perspective is represented in different ways. Moreover, because development may be carried out concurrently by those involved, different perspectives may be at different stages of elaboration and may be subject to different development strategies.

The problem of how to organise and guide development in this setting - many actors, sundry representation schemes, diverse domain knowledge, different development strategies - we term “the multiple perspectives problem”. Although this is a general problem of large systems development, we will focus only on (multi-perspective) *software* development - a subset of *systems* development.

4.1.2. Objective: Multi-Perspective Software Development

The multiple perspectives problem serves to clarify our research requirements and agenda. Thus, there is a need to:

- Structure, organise and manage the different perspectives that exist during software development.
- Check consistency and handle inconsistencies between the different perspectives, by transferring and transforming information between them.
- Support concurrent, cooperative development in a distributed setting.

Addressing each of these needs requires contributions in different areas of software engineering. What is primarily needed to begin with however, is an organisational framework within which these issues may be addressed. In this chapter we propose the ViewPoints framework, whose building blocks are *ViewPoints*[9], which provide the infrastructure and organisation necessary to address the above issues. We call multi-perspective software development within the ViewPoints framework *ViewPoint-Oriented Software Engineering (VOSE).*

4.2. ViewPoints

The term “view” or “viewpoint” is used in different ways in software engineering. In the database world, a view is often regarded as a projection or partial representation of a global knowledge source (which is treated as “the truth” against which all views have to check consistency). Overlaps (which potentially lead to inconsistencies) are maintained centrally by the global knowledge source. Views in this context are therefore “read-only” descriptions.

[9] We use the term “ViewPoint” with a distinctive uppercase “V” and “P”. This should not be confused with the ViewPoints system described by Fischer and his colleagues which is a hypertext-based design rationale tool [Fischer, McCall & Morch 1989] (it is also described briefly in chapter 3, section 3.3.3).

Increasingly however, software engineers have regarded views as loosely coupled knowledge sources which can be modified independently. A common, constructed database or representation is responsible for identifying overlaps and resolving inconsistencies between these views. Views in this setting are more akin to tools that have independent identities, behaviour and state (*c.f.* objects in object-oriented technology). As discussed in chapter 3, this is useful because it allows development participants to combine the strengths of many languages and to explore alternative solutions to problems. Many of the multi-view and multi-paradigm systems and techniques (also described in chapter 3) support this notion of a view.

To treat views as active entities (that is, as tools with some functionality in addition to their state) requires that these views interact and exchange information. Imposing a common, centralised representation or knowledge source to facilitate this interaction and exchange however, gradually becomes a bottleneck. Common representations are difficult to construct, maintain and extend, and are less amenable to both physical and logical distribution. In order to support a model of views that allows multiple partially, totally, or non-overlapping views (i.e., one that allows multi-perspective software development), views must be allowed to develop independently if necessary, and interact directly when the need arises. To achieve this, a view must combine both the notion of a "perspective" (opinion, belief, point of view), with that of a "viewer" (development participant, agent, actor, knowledge source) holding that view. We call such a view a "ViewPoint", and it is characterised by its extension of the usual notion of a view as a "partial specification" through the addition of a "partial process" that is used to construct that partial specification.

4.2.1. Definition

We define ViewPoints to be loosely coupled, locally managed, distributable objects. Each ViewPoint encapsulates partial knowledge about a system and its domain - *expressed in a suitable representation scheme* - *together with partial knowledge about the overall process of development.* A ViewPoint therefore contains three kinds of software engineering knowledge:

- *Representation knowledge* about the notation, language or representation scheme deployed by that ViewPoint;
- *Specification knowledge* about the particular aspect of the system or problem that the ViewPoint is concerned with; and,
- Software development *process knowledge* about how to deploy a particular representation to produce a specification.

A ViewPoint's "owner" is responsible for the development of that ViewPoint, and provides *domain knowledge* (including experience or expertise) to that ViewPoint. ViewPoint owners are described in section 4.2.3.

4.2.2. Slots

The different kinds of knowledge that a ViewPoint encapsulates are structured and separated into five "slots"[10]: *(1) representation style (2) work plan (3) domain (4) specification and (5) work record* (Fig. 4.1). These slots are used to separate concerns within a ViewPoint in terms of the different kinds of knowledge they contain, are useful in that they give ViewPoints a uniform structure, and serve to identify and separate different ViewPoints.

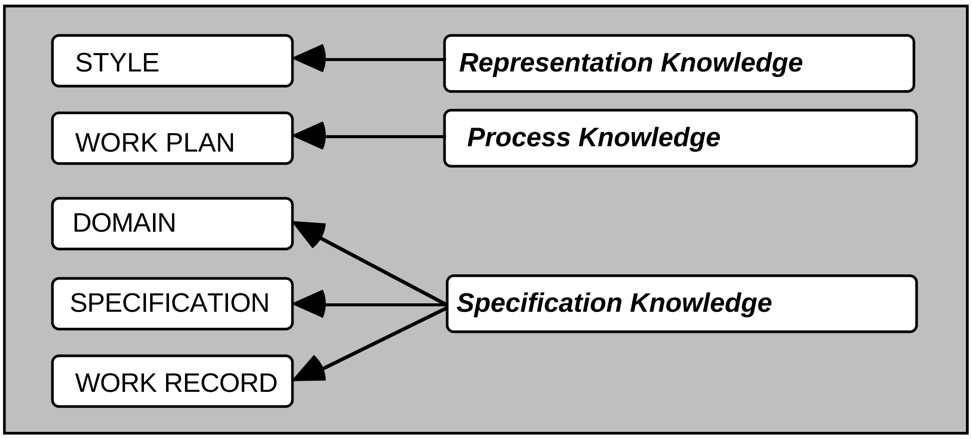


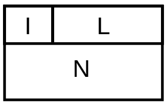
Figure 4.1: *The five slots of a ViewPoint.*

4.2.2.1. Representation Style

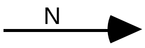
The representation style slot (hereafter referred to simply as the style slot) contains a definition of the representation scheme deployed by a ViewPoint. It therefore contains a meta-level description of the notation or language used by a ViewPoint to describe the area of concern of that ViewPoint.

Text-based languages are often defined by a BNF grammar description, while for graphical notations entity-relationships may be more suitable. In this thesis, *we use a notation based on objects and relations that have typed attributes and values to describe a ViewPoint's representation style*. An example of an object is a "Process" in a data flow diagram which in our representation may be defined as follows:

[10] The term "slot" is borrowed from frames in AI (chapter 2, section 2.2.2) where it is used as a knowledge or data container - also called a "frame variable" (c.f. instance variables in object-oriented terminology).

OBJECT: Process		
ATTRIBUTES	**TYPES**	**VALUES**
Name (N)	String	
Identifier (I)	Integer	
Location (L)	String	
Icon	Bitmap	

A relation on the other hand binds two or more objects (which may be expressed as parameters of that relation). For example, a "Transition" in a state transition diagram is a relation between two "State" objects, and may be defined as follows:

RELATION: Transition (State, State)		
ATTRIBUTES	**TYPES**	**VALUES**
Name (N)	String	
Icon	Bitmap	

While such a meta-level representation is more suited to graphical languages, in principle it can also be used for text-based languages. One approach would be to identify the terminals in the BNF definition and use these as the objects in our representation.

The style slot then, provides the basic definition of the syntax of notation used by a ViewPoint. How this notation is deployed is described in the work plan slot.

4.2.2.2. Work Plan

The work plan slot defines the actions, and the recommended ordering of those actions, necessary to deploy the representation scheme defined in the style slot. The work plan thus contains a series of actions and a process model that describes the circumstances under which any of the actions may be performed. We identify five distinct kinds of actions:

- *Assembly actions.* These are the basic actions necessary to assemble (construct) a ViewPoint specification in the defined representation scheme. They may be regarded as a list of basic "editing" actions that one would expect a CASE tool supporting such a ViewPoint to provide. Assembly actions include adding and removing objects, and linking and unlinking objects - by, respectively, adding and removing any associated relations between these objects. For example, the most basic list of assembly actions for developing a state transition diagram would be:

```
add(State)

remove(State)

link(Transition (State, State))

unlink(Transition (State, State))
```

- *In-ViewPoint check actions.* These are the actions that check that ViewPoint specifications are locally consistent, according to some in-ViewPoint rules. In-ViewPoint rules essentially define the semantics of a representation style by defining the constraints that should hold in order for a ViewPoint specification to be "well-formed". Different method designers may choose to impose different in-ViewPoint rules on ViewPoint specifications. In a functional hierarchy for example, an in-ViewPoint rule may prescribe that there should be more than one child for any decomposed parent function, while other method designers may allow the decomposition of a parent function into a single child (Fig. 4.2). An in-ViewPoint check then tests whether or not this rule holds; that is, it checks whether or not any decomposed function in the hierarchy has more than one child. Other in-ViewPoint check actions may include checking for duplicate function names, that there is one root for each functional hierarchy, and so on.

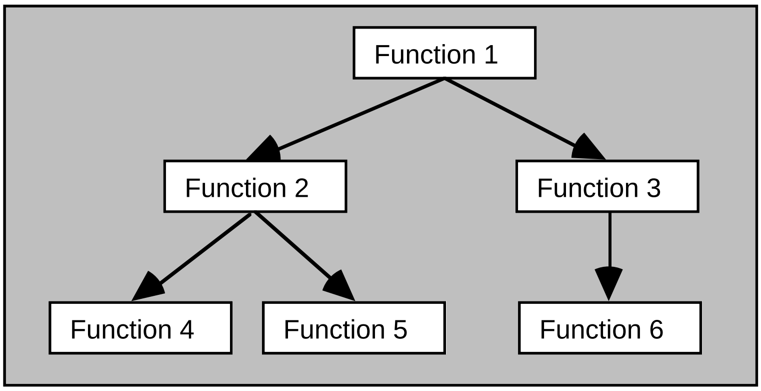


Figure 4.2: *A functional decomposition hierarchy. An in-ViewPoint check action will flag an inconsistency in this diagram only if there is an in-ViewPoint rule laid down by the method designer specifying that a single decomposition of a function is illegal. Such a rule would detect that "Function 3" is decomposed into "Function 6" only.*

- *Inter-ViewPoint check actions.* These are the actions that check the consistency between ViewPoints, according to some inter-ViewPoint rules. Inter-ViewPoint rules define the relationships between the representation styles of different ViewPoints, and therefore typically describe the areas of overlap between ViewPoints. By checking inter-ViewPoint rules in a well defined order, they may also be used as a vehicle for synchronising or coordinating the development activities of two different ViewPoints. This order is prescribed by local ViewPoint process models. As we will see in chapter 5, inter-ViewPoint checks are fundamental to method integration within the ViewPoints framework. Inter-ViewPoint rules are the integration "glue", and the check actions are the vehicle for applying these rules.

In the process of expressing inter-ViewPoint relations, we identified four different kinds listed and explained in Table 4.1 below.

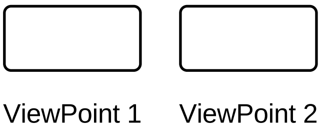
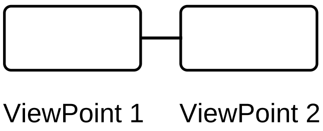
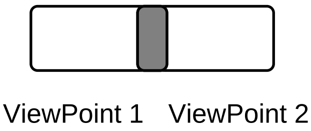
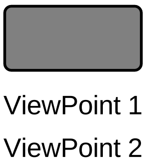
	The two ViewPoints are independent, non-overlapping and unrelated (except in that the method from which they are created requires both ViewPoints to exist - but the ViewPoints themselves, and their contents, are unrelated). For example, a method may require the development of a ViewPoint describing the functional hierarchy of a software system and another ViewPoint documenting the financial resources available to the project.
	The two ViewPoints are non-overlapping, but there is some existential relationship in which the existence of one depends in some way on the existence of the other. For example, the Z method [Spivey 1989] requires that for each Z schema ViewPoint, there is an associated textual description ViewPoint.
	The two ViewPoints are partially overlapping, with a partial specification in one related to a partial specification in the other. This is the most common type of relationship. For example, a source agent in a CORE[11] tabular collection diagram must be a named agent in the agent hierarchy. Relationships between partially overlapping ViewPoints describe syntactic and semantic correspondences between notations and domains of concern respectively.
	The two ViewPoints are totally overlapping - they describe the same domain using the same representation scheme. We may (1) require that any conflicts, discrepancies or inconsistencies be eventually resolved so that the two ViewPoints are made to say the same thing, or (2) accept that the two ViewPoints represent two different "views" of the same domain (e.g., different solutions to the same problem) that require evaluation and a choice to be made between them.

Table 4.1: *Inter-ViewPoint relations. Shaded areas represent overlaps between ViewPoints.*

- *ViewPoint trigger actions.* These are actions that create new ViewPoints. In a sense they are meta-level actions because their application yields new ViewPoints. To make the ViewPoints framework fully distributable, these actions are part of individual ViewPoint work plans, and are not necessarily performed centrally. Thus, ViewPoints themselves may create new ViewPoints "on-the-fly" from within their own individual work plans. Creating a ViewPoint from a ViewPoint "template" is discussed in more detail in section 4.3.

[11] CORE uses the term "viewpoint" to refer to an information processing entity. We use the term "agent" in its place to avoid the clash in nomenclature with our "ViewPoints".

- *Inconsistency handling actions.* These are actions that may be performed in the presence of inconsistency, and are typically a subset of the other work plan actions listed above. They may however include further actions that are not explicitly part of a ViewPoint work plan, such as the user actions needed to resolve conflict, consult experts, or engage in further fact finding and research. Often, they are actions that invoke specialised tools to handle particular problems. Inconsistency handling actions may be used to handle both in- and inter-ViewPoint inconsistencies, although the problems associated with *inter*-ViewPoint handling are much more difficult. Inconsistency handling is examined in more detail in chapter 7, where logic-based inconsistency handling rules are used as part of a ViewPoint's process model to specify how to act in the presence of inconsistency. These rules are of the general form:

INCONSISTENCY implies ACTION

All of the above work plan actions are actions that *may* be performed by a ViewPoint developer. A ViewPoint process model defines *when* it is appropriate or recommended that any of these actions be performed. In other words, the process model provides guidance on the development strategy of the ViewPoint specification. In demonstrating the ViewPoints framework, we have adopted the following notation to describe a ViewPoint's local development strategy:

{preconditions} ⇒ [agent, action] {postconditions}

This reads:

if the listed *preconditions* hold, then,

if the *agent* performs the *action*, then,

the list of *postconditions* will hold

Again, the way in which a ViewPoint process model is used will be discussed in more detail in chapter 7. Suffice to say at this point that we have found that modelling the process in this "fine-grain" way facilitates the provision of useful development guidance and help. In this context, granularity is at the level of individual work plan actions, rather than the traditionally coarse-grain actions such as tool invocations.

Moreover, modelling individual ViewPoint development processes locally is a step towards a fully distributable ViewPoints framework, in which the "overall" development process is also distributed. In fact, we believe that the notion of an overall development process or process model is not useful in this context. What we have in reality, and need to support, are different views on an overall process. These include organisational, managerial, local development views and so on.

4.2.2.3. Domain

The domain slot provides a means of classifying a ViewPoint. It contains a label that identifies the area of concern of that ViewPoint. The choice of this label is based on any permanent identifier of the ViewPoint (that is, permanent within the life time of that ViewPoint). Thus, a ViewPoint domain typically identifies part of a system under development. It may also identify the owner (developer) of that ViewPoint - but a ViewPoint owner may change several times during the life time of a ViewPoint, and is therefore less suitable for identifying a ViewPoint.

ViewPoint domains may be ad hoc labels that are partially, totally or non-overlapping. Any overlaps and other relationships between such domains are established during the ViewPoint-oriented development process.

4.2.2.4. Specification

The specification slot contains a description of the identified domain, in the notation defined in the style slot. It is the product of applying work plan actions to produce a partial specification of a system - partial because a ViewPoint is normally a partial description of an overall system.

The specification slot is a view in the traditional sense, in that it contains a description of part of a problem or its solution from a particular perspective. A specification is the product of software development, and is often one of the deliverables of a ViewPoint-oriented development process.

4.2.2.5. Work Record

The basic work record slot contains a list of the work plan actions that were performed to produce a ViewPoint specification. Thus, the work record contains a development *history* of a ViewPoint specification, and is therefore one way of defining the development *state* of a ViewPoint.

A work record may be annotated to provide a development *rationale*. Each action may be annotated separately (to explain why that action was performed for example), or the work record may be annotated as a whole (to explain the development strategy of a ViewPoint, say). These annotations may take the form of simple or structured text, or may be expressed in more sophisticated rationale languages (such as the “issues” and “arguments” of gIBIS discussed in section 2.2.4).

A work record is therefore an elaborated instantiation of the work plan, and has many uses other than providing a development history and rationale. Portions of the work record may be *replayed* to explore alternative development strategies or to produce different versions of a specification. Information in the work record may also be used during consistency checking and inconsistency handling, since it is a temporally ordered list of actions that may be analysed and reasoned about in this context.

Other ViewPoints, such as those of development managers, may also use work record information to monitor the development progress of individual developers by “inspecting” their work records, and ensuring that sufficient documentation and justification for development actions has been provided.

Finally, information useful for producing project reports is also held in the work record, which may be compiled in various ways to generate structured project reports or documents (as specified by contractors or standards organisations for example).

4.2.3. Owners

A ViewPoint owner is an agent (person or tool) responsible for the development of a ViewPoint. In other words, a ViewPoint owner is responsible for enacting a ViewPoint work plan to produce a ViewPoint specification for a particular domain. Typically, a ViewPoint owner is a human user or group of users, although a computer system (e.g., a knowledge-based or expert system) may also play this role.

An agent may simultaneously be the owner of several ViewPoints, but each ViewPoint has one and only one owner (even though this owner may be a group of people). A ViewPoint’s owner may change over time.

A ViewPoint owner may “filter” out the relevant domain knowledge necessary to include in a ViewPoint, and therefore also serves as a kind of “interface” between a ViewPoint and the external world in which that ViewPoint resides.

We will not examine the role of a ViewPoint owner any further in this thesis. However, do we distinguish between two kinds of ViewPoint ownership described in outline below. The first is from a management perspective in which ownership of a ViewPoint is equivalent to having “responsibility for” developing that ViewPoint. This also includes responsibility for the contents of the ViewPoint. The second kind is more from a developer or user perspective. Here ownership of a ViewPoint means “authority to” access or modify that ViewPoint. A ViewPoint owner may “delegate” authority of a ViewPoint to a new owner (which is often the case in large development projects). For example, a ViewPoint owner, while maintaining responsibility over one ViewPoint, may transfer control (delegate authority) to another owner to develop that ViewPoint. What results is a ViewPoint ownership hierarchy, with associated responsibilities and authorities (which may reflect actual organisational relations and structures). Such a structure may also be analysed and perused separately - for requirements traceability purposes for example [Gotel & Finkelstein 1994a; Gotel & Finkelstein 1994b].

4.2.4. Specifications

A *system specification* in the ViewPoints framework is a configuration (structured collection) of ViewPoints (Fig. 4.3). This is consistent with actual specification practices where both the development process and the final deliverables are a series of documents - as opposed to a single monolithic specification. Thus, the ViewPoints framework recognises this heterogeneity of development products, and provides a means of structuring and organising them (it is still possible, of course, to construct a single ViewPoint representing the entire system specification, but we believe that such a "universal" ViewPoint is difficult to construct, understand and maintain, and is therefore undesirable).

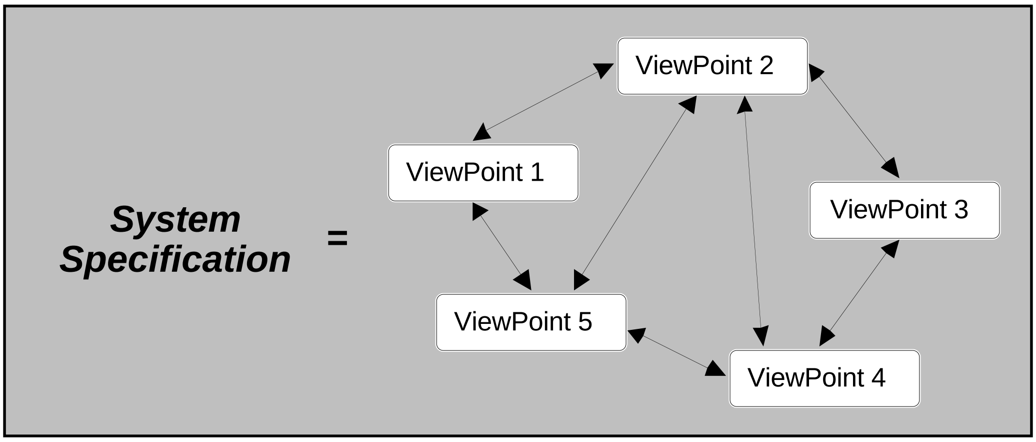


Figure 4.3: *A system specification is a configuration (structured collection) of ViewPoints. Arrows represent inter-ViewPoint relationships.*

4.2.5. Interaction Infrastructure

The five ViewPoint slots encapsulate representation, process and specification knowledge, while the ViewPoint owner adds domain knowledge. While this covers the different kinds of software engineering knowledge necessary to develop a system, the ViewPoints framework requires additional support for interaction between ViewPoints. Support for such interaction is provided by a ViewPoint *interface* which holds, transforms and analyses information transferred between ViewPoints. A ViewPoint interface specifies what ViewPoint knowledge is visible to other ViewPoints - hiding any private or temporary knowledge in the process. The major role of a ViewPoint interface is to support inter-ViewPoint communication, which in turn is the vehicle for method integration. ViewPoint interfaces and inter-ViewPoint communication will be discussed in more detail in chapter 6 (section 6.4).

4.3. ViewPoint Templates

Many ViewPoints, while containing descriptions of different domains, may use the same representation scheme and development strategy to produce these descriptions. Moreover, a single organisation may deploy a limited number of development techniques or methods in their development process which they repeatedly use to specify and build systems. The notion of a ViewPoint "type" or "class" is therefore useful in this context. We call such a ViewPoint type a "ViewPoint template".

4.3.1. Definition

A ViewPoint template (hereafter referred to as a "template") is a ViewPoint in which only the style and work plan slots have been elaborated (Fig. 4.4). In other words, it only contains the definition of the notation and development process deployed by a ViewPoint. A single template is therefore a description of a single development technique.

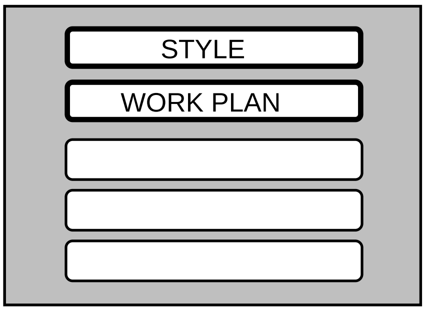


Figure 4.4: *A ViewPoint template is a ViewPoint with only the style and work plan slots elaborated.*

A ViewPoint is created by *instantiating* a ViewPoint template. A single template may be instantiated many times to yield many ViewPoints (Fig. 4.5). Moreover, one template may be an elaboration, restriction or modification of another; that is, one template may inherit the style and/or work plan of another. A template is therefore a reusable description, facilitating reuse by both instantiation and/or inheritance. We currently support instantiation of templates, but not inheritance between them. Conceptually however, both forms of reuse are available.

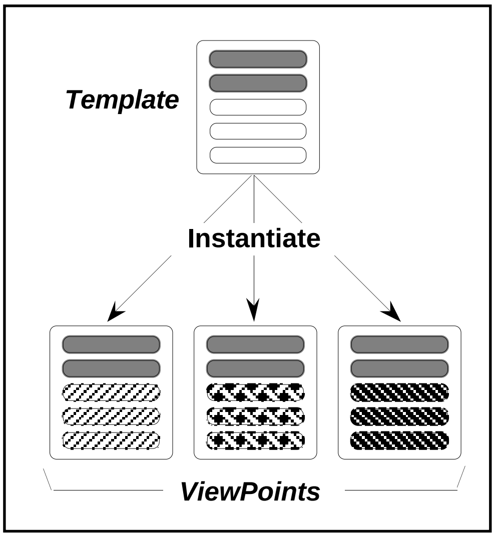


Figure 4.5: *Instantiating a ViewPoint template. A ViewPoint template may be instantiated many times to yield many different ViewPoints.*

4.3.2. Methods

A software engineering method in the context of the ViewPoints framework is a configuration (structured collection) of ViewPoint templates (Fig. 4.6). In other words, a method is a structured collection of development techniques. Each technique, denoted by a template, may be regarded as "primitive", loosely coupled, locally managed method from which other composite methods may be constructed (integrated). Alternatively, each technique may describe a particular stage or step in a development method, in which case tighter integration between the templates may be required.

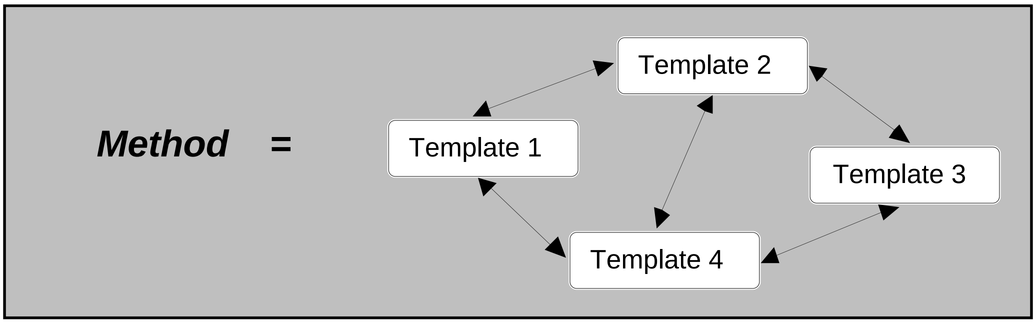


Figure 4.6: *A software engineering method in the ViewPoints framework is a configuration (structured collection) of ViewPoint templates.*

4.4. The Framework

The ViewPoints framework raises a number of software engineering issues. While providing the infrastructure for supporting multiple methods and views in a distributed, collaborative software development setting, it also identifies and clarifies many outstanding research problems, and sets out a detailed agenda for further work.

4.4.1. Architecture[12]

The architectural objective of the ViewPoints framework is to move away from centralised towards decentralised architectures (Fig. 4.7). Thus, ViewPoints are “distributable”, although this does not exclude having centralised ViewPoints containing global data or control information. There are several levels of decentralisation however, with varying levels of difficulty to achieve them. For example, decentralising a specification by breaking it into smaller, more manageable modules has become commonplace and fits exactly into a ViewPoint-oriented development model. Using physically or logically decentralised databases to store parts of the specification on the other hand is more difficult. Moreover, if the underlying schemas of the specifications in the different databases are also different, then managing the specification development process is, with current technology, problematic.

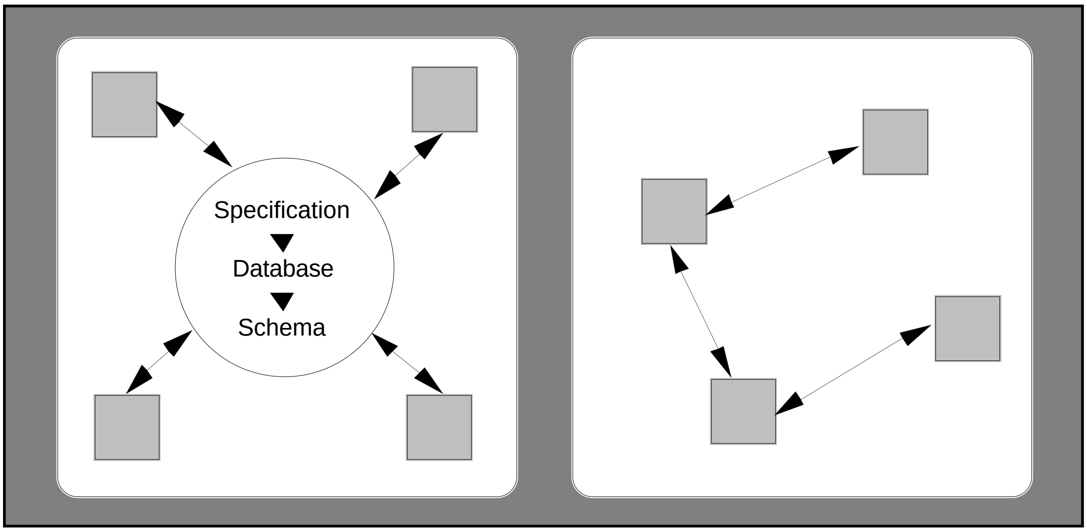


Figure 4.7: *Centralised versus decentralised architectures. The ViewPoints framework is inherently decentralised.*

Our current approach is to have a totally decentralised architecture for the ViewPoints framework, and to work on the outstanding issues of integration and management, rather than adopt a centralised architecture which we then attempt to decentralise. We also favour the use of pairwise checks of relationships between ViewPoints as the integration vehicle.

Moreover, we argue that the decentralised architecture of the ViewPoints framework is a better reflection of “real” development by multiple development participants who hold multiple perspectives. Development participants are naturally distributed, and while they may share knowledge, or even hold a common view of the world, this knowledge is distributed among them

[12] We use the term “architecture” to describe the structuring elements and communication principles of a framework. Thus, we can talk of centralised architectures, distributed architectures, pipeline architectures and so on.

and is not held centrally. In fact typically, much redundant or duplicate information is held by different development participants, and therefore a single, centrally shared body of information is an inaccurate representation of "reality".

4.4.2. Tool Support

The scope for computer-based tool support of VOSE centres around two broad activities: "method engineering" and "method use".

To support method engineering, tools are needed to describe and assemble templates, and to construct CASE tools to support ViewPoint specification development based on these templates. Template description requires tools such as text and graphical editors to define template styles, and process modelling tools to define template work plans. Template libraries are also useful for reusing pre-defined templates. CASE tools construction requires either programming or meta-CASE tool kits to build or generate, respectively, tools to support various templates.

To support method use, that is, ViewPoint-oriented software development, requires a distributed system infrastructure to support distributed ViewPoints and their interaction, and a flexible environment within which to develop and manage ViewPoints. The latter includes environments that facilitate the use of the tools constructed or generated by the method engineering process.

The Viewer prototype environment, described in chapter 8, illustrates the scope for tool support within the ViewPoints framework, and explores various areas amenable to automated support. The assumption that we make in *The Viewer* is that ViewPoint template support tools are built from scratch, as opposed to using pre-existing tools such as those available on UNIX. This is more of a pragmatic working assumption, but in practice the transition from a "traditional" to a ViewPoint-oriented development environment will have to be facilitated by translators to and from various existing tools' representation schemes.

The ViewPoints framework is also a vehicle for tool integration. This is achieved by treating tool integration as a special case of method integration. Method integration, as we shall discuss in chapter 6, is the process of integrating ViewPoint templates by means of inter-ViewPoint checks managed by a process model. If ViewPoint template definitions are made precise and complete enough to generate CASE tools from these definitions, then these same checks and associated process model that provided method integration, can also be used for tools integration (Fig. 4.8).

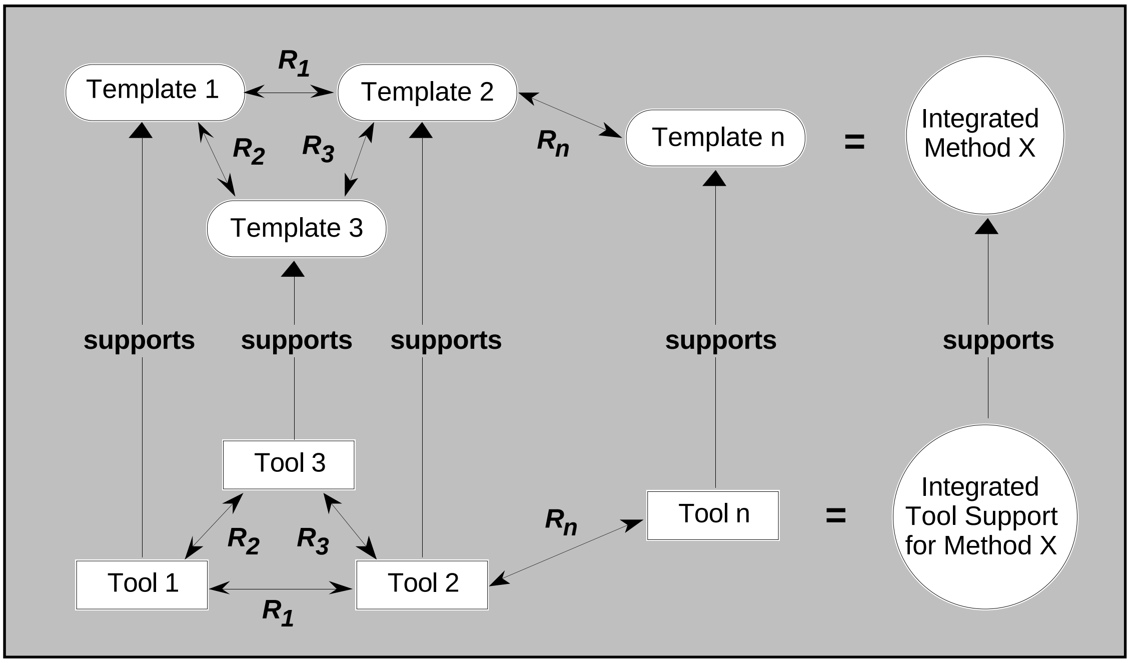


Figure 4.8: *Tool integration is a special case of method integration.*

4.4.3. Analogous Approaches

The structure and elements of the ViewPoints framework have analogies in a number of related computer science approaches. These have influenced the design of the framework, which has been able to draw upon the advances made by these approaches.

4.4.3.1. The Object Model

The ViewPoints framework is an *object-based* framework [Wegner 1987]. ViewPoints are analogous to “objects” in the traditional object model. They have a “state” (with the specification and work record slots as “instance variables”) and “behaviour” (with “methods” defined in the work plan and, possibly, style slots). They have a unique identity, marked by the their domain and owner. They are also instantiated from “classes” (templates), but there is currently no explicit support of inheritance - which makes the framework “object-based” as opposed to “object-oriented”. Inheritance is not part of the current framework for pragmatic reasons: there is a tension between inheritance and encapsulation - which is needed for effective distribution. Thus, while conceptually inheritance between templates is desirable, current technology makes it difficult to implement efficiently in a distributed environment. Table 4.2 summarises these and other analogies between object-orientation and VOSE.

Object-Orientation	VOSE
Object (instance)	ViewPoint
Class	Template
Encapsulation of state and behaviour	Encapsulation of state in specification and work record
Object identity	ViewPoint domain and owner
Information hiding: objects can only be changed by object operations	Information hiding: ViewPoint specifications can only be changed by work plan actions
Inheritance	Not available (omitted to maximise encapsulation of templates and distribution of ViewPoints)
Polymorphism: the binding of a message to different methods, depending on the class of the receiving object	Polymorphism "simulated" (because we have neither inheritance nor dynamic binding): identical work plan actions may be used to build specifications based on different styles, depending on the template instantiated
Message passing	Message passing - available but not in the strict object-oriented sense; rather, in the distributed systems context: via inter-ViewPoint communication protocol

Table 4.2: *VOSE and Object-Orientation.*

4.4.3.2. Databases

A ViewPoint is also analogous to an autonomous database and its database management system (DBMS). Alternatively, a collection of ViewPoints can be regarded as a multidatabase system. The definition of a ViewPoint style is analogous to the definition of a database data model or schema. Actual database data is stored in the specification, domain and work record slots. The work plan has multiple analogous roles in the database world. It may be regarded as the application that manipulates and accesses the data or the DBMS that manages the database. It also describes inter-database dependencies in the form of inter-ViewPoint checks and ViewPoint trigger actions. Table 4.3 summarises the analogies between databases and the ViewPoints framework.

Database in a Multidatabase System	ViewPoint in VOSE
Database data model and schema	Style
Data	Specification, domain and work record
Database Management System (DBMS)	Work plan
Application	Work plan process model
Inter-database dependencies	Work plan: inter-ViewPoint and ViewPoint trigger actions

Table 4.3: *VOSE and databases.*

4.4.3.3. Artificial Intelligence

To a lesser extent, the ViewPoints framework has analogies in the fields of Artificial Intelligence (AI) and Distributed Artificial Intelligence (DAI). ViewPoints for example, are analogous to frames in AI. Their specification and work record slots are analogous to “frame variables”, which can only be modified by operations defined in their respective work plan slot. In DAI on the other hand, cooperative problem-solving by multiple, distributed agents is analogous to ViewPoint-oriented software development. These agents have individual or common goals, and interact according to one or more well defined protocols in order to achieve these goals - as do ViewPoints. While the ViewPoints framework and DAI frameworks are different in many respects, the interaction protocols between agents in DAI are particularly appropriate in a distributed VOSE setting. In fact, the contract nets protocol for negotiation between agent described in chapter 3 (section 3.4.3) for example, is directly applicable in the ViewPoints context with little modification.

4.4.4. Integration Issues

The primary objective of this thesis is to achieve method integration in the context of multi-perspective software development, exemplified by VOSE. The ViewPoints framework facilitates multi-perspective software development by deploying multiple ViewPoints to represent multiple perspectives and the development participants that hold these perspectives. The architecture of the framework is inherently decentralised, allowing distributed, concurrent development. ViewPoint templates facilitate the definition of the development techniques from which ViewPoints are created and developed. A single ViewPoint template represents a primitive method, while a collection of templates constitutes a more complex, composite method.

The framework thus achieves separation of concerns *across* ViewPoints in terms of views and development participants, and *within* ViewPoints in terms of the different kinds of software engineering knowledge that each ViewPoint encapsulates. Separation of concerns is also achieved at the method engineering level in terms of the different templates that constitute methods, and at the method use level in terms of the different ViewPoints that constitute systems specifications.

The *integration* of these concerns at different stages however is not as straightforward, and requires contributions to different aspects of the framework. This in turn has an impact on the nature of the ViewPoint-oriented software development process, and the way in which activities are performed within the framework. This section discusses the areas of contribution of this thesis that impact upon method integration.

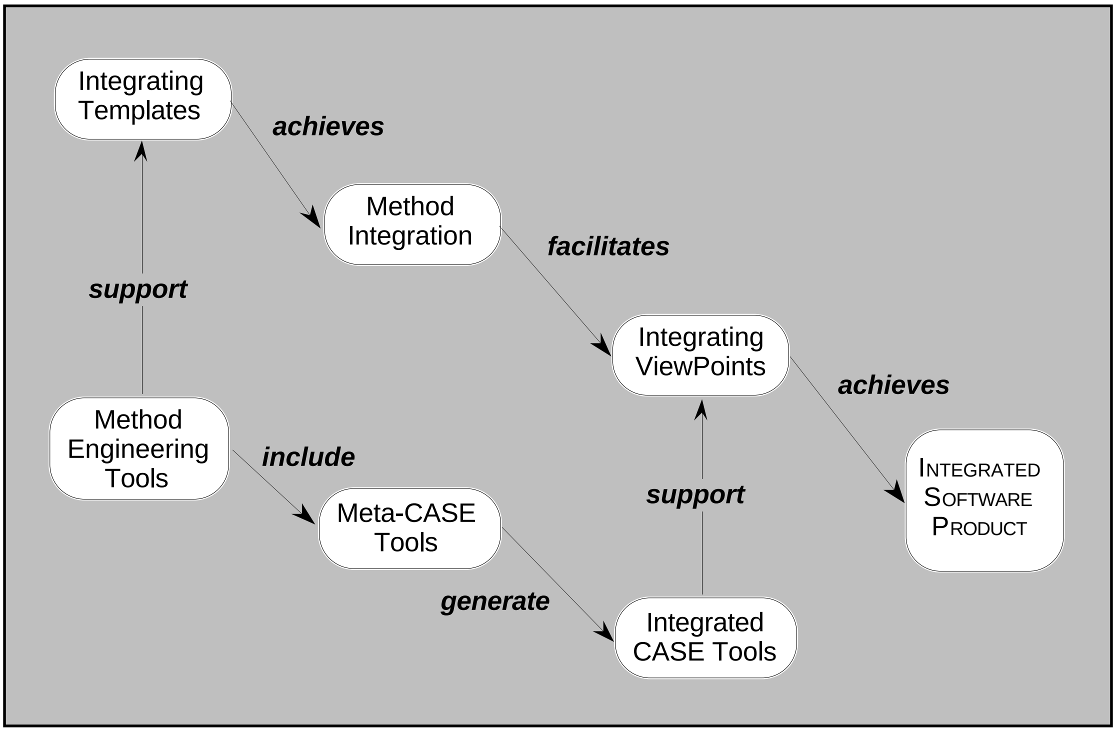


Figure 4.9: *Perspectives on integration in the ViewPoints framework.*

Fig. 4.9 shows the scope for integration within the ViewPoints framework. It also illustrates the need to explore other activities within the framework (including different facets of integration) in order to achieve the desired method integration. During method engineering, integration is at the template level, that is, ensuring that the relationships between templates are identified and expressed. During method use (ViewPoint development), integration is between ViewPoints, that is, ensuring that inter-ViewPoint rules are invoked at the appropriate times and applied correctly. The objective here is to produce an integrated software system (specification and/or implementation).

4.4.4.1. Method Engineering and Integration

Method engineering within the ViewPoints framework is simplified by the fact that a single software engineering method may be designed and constructed in a modular fashion by both *defining* individual templates and *reusing* pre-defined templates from a reuse library of such templates. Tool support for this process is also simplified in that method engineering tools in this context are either structured editors or meta-CASE tools for generating CASE tools to support, mostly simple, templates. The next chapter, addresses the issues of method engineering in more detail, and examines the impact that this process has on method use.

Method integration in the ViewPoints framework is ultimately concerned with inter-ViewPoint relationships - how they are expressed, when they are invoked and how they are applied. These relationships first manifest themselves as intra- and inter-template relationships, which we collectively call inter-ViewPoint relationships. The requirements of method integration are laid down during method engineering when templates are defined, and are “measured” by the extent to which the relationships between ViewPoints instantiated from these templates are found to hold. Chapter 6 examines method integration in the ViewPoints framework in more detail, but it is worth noting at this stage that integration in this context does not mean tight coupling between templates or ViewPoints. Rather, it is simply the satisfaction of rules that the method designer *chooses* to define between templates, and consequently between ViewPoints instantiated from these templates.

4.4.4.2. Process Modelling and Method Guidance

A key feature of ViewPoints is that they are *locally managed.* In other words, each ViewPoint has a local work plan of actions for constructing its own ViewPoint specification. This specification construction process is driven or guided by a local process model that also forms part of the ViewPoint’s work plan. Chapter 7 addresses process modelling in the ViewPoints framework, and particularly focuses on the use of “fine-grain process modelling” for providing effective development guidance for individual ViewPoint specification developers.

Fine-grain ViewPoint process models are also used to guide the invocation of inter-ViewPoint rules and thus synchronise the development of different ViewPoint specifications. A ViewPoint’s process model therefore specifies when it is appropriate to check consistency between ViewPoints and how to act in the presence of inconsistency. These steps are necessary in the process of producing an integrated software system.

4.4.4.3. Consistency Checking and Inconsistency Handling

Consistency checking between ViewPoints is a vehicle for integrating these ViewPoints. It is the activity in which two or more ViewPoints compare knowledge and ascertain whether or not the relationships that supposedly hold between them do in fact hold. When these relationships hold, they are the integration “link” between partially, totally or non-overlapping ViewPoints. In chapter 6, the process of checking two ViewPoint specifications according to the defined method integration rules, is examined, and the interaction protocol that is used to perform such checking is described. This includes the transfer of information from one ViewPoint to another, as part of either the consistency checking or more general software development process.

Of course, consistency checks between ViewPoints often expose inconsistencies. However, in the process of achieving integration during software development, we advocate tolerating transient inconsistencies and supporting more *inconsistency handling.* Thus, we are able to tolerate partial inconsistency with respect to the rules defined. In fact, in general we can only speak of “partial consistency” in software development, although it may be possible to define “total consistency” as the satisfaction of *all* the defined consistency rules. Inconsistency handling in the ViewPoints framework is discussed in more detail in chapter 7.

4.4.4.4. Concurrency and Distribution

Support for concurrency and distribution is not explicitly addressed in this thesis, although many of the design decisions of the framework are made with these two issues in mind. ViewPoints are largely autonomous entities which may be developed concurrently and interact intermittently to check consistency and exchange information. Such interaction is managed by a protocol that specifies *what* information needs to be exchanged and *when* it is allowable to do so. While the protocols we explore in this thesis (chapter 6) attempt to take concurrency issues into account, no new concurrency control mechanisms are proposed. Concurrency control is beyond the scope of this thesis, but a mechanism such as the one proposed by Barghouti [Barghouti 1992] may be appropriate in the ViewPoints setting.

The ViewPoints framework is also inherently, and deliberately, distributable. Its architecture is based on a decentralised model of interacting ViewPoints. These interactions are driven by local (decentralised) process models according to inter-ViewPoints rules that are also distributed among the different ViewPoints. There are no “special” ViewPoints that manage overall development or individual ViewPoint interactions. ViewPoints are autonomous, encapsulating as much knowledge as is necessary to proceed in their local specification development and interactions with other ViewPoints. This, of course, does not discount the possibility of having “management” ViewPoints, “organisational” ViewPoints and so on, but these have the same status as other ViewPoints, and may also be “developed” in a distributed environment like other ViewPoints.

4.5. Chapter Summary

In this chapter, we presented an outline of the ViewPoints framework for distributed, multi-perspective software development. The structuring elements of the framework, ViewPoints and ViewPoint templates, were described, and the scope for automated tool support of ViewPoint-Oriented Software Engineering (VOSE) was presented. The influences of other computing disciplines on the design and implementation of the framework were briefly examined, and outstanding research issues of integration within the framework were set out.

In the following chapters, we will address, in varying degrees of detail, these open research issues. The objective of our examination will be to support integrated multi-perspective software development within the ViewPoints framework, by ensuring that methods constructed within this framework are also integrated.

An examination of method engineering, and its effects on method use, within the framework is particularly appropriate, and is the subject matter of the next chapter.

Chapter 5 Method Engineering and Method Use

This chapter describes the activity of method engineering within the ViewPoints framework. It discusses how a method is designed and constructed by method engineers, and illustrates how such an engineering process affects the way in which a method is used. The role of meta-CASE technology in this setting is discussed, particularly in supporting reuse for the method engineer and providing effective tool support for the method user. Method integration is identified as a key outstanding problem of method engineering and is addressed in the next chapter.

5.1. Software Development Participants

Multi-perspective software development engages multiple participants who invariably hold different views on a problem or solution domain. These participants include *clients* who have different, conflicting and complementary requirements of the software system they wish to acquire; and *developers* who must elicit, specify, analyse and validate these requirements, and then design and build a software system that satisfies the requirements. We use ViewPoints, as described in chapter 4, to represent the different perspectives of both clients and developers.

Another participant in the software development process is the *method engineer*. In general, a method engineer is responsible for designing and constructing a development method that, in turn, provides developers with systematic procedures and guidelines for deploying one or more notations to describe problem or solution domains. Increasingly, the role of the method engineer has become more application domain specific, with “customised” or “situation-specific” methods assembled for particular organisations or projects [Harmsen & Brinkkemper 1993]. This is facilitated by computer-aided method engineering (CAME) tools, such the meta-CASE tools discussed in chapter 2 (section 2.1.3.3).

In theory, the activities of method engineers precede those of software developers. In practice however, the roles and activities of method engineers, software developers and clients are inextricably linked and often overlap (figure 5.1). Both clients and software developers will make demands on the method engineer: the client may require that the software developer use a particular method, and the system requirements may necessitate the customisation of that method for a particular project. The method engineer, while ultimately interacting mostly with a software developer by delivering a method for him to use, will do this based on feedback from both developer and client. His role may also extend throughout the development life-cycle (that is, even after delivering the method to the software developer), since the method deployed may be modified or enhanced as the development proceeds (e.g., because of inevitable process evolution [Lehman 1994]).

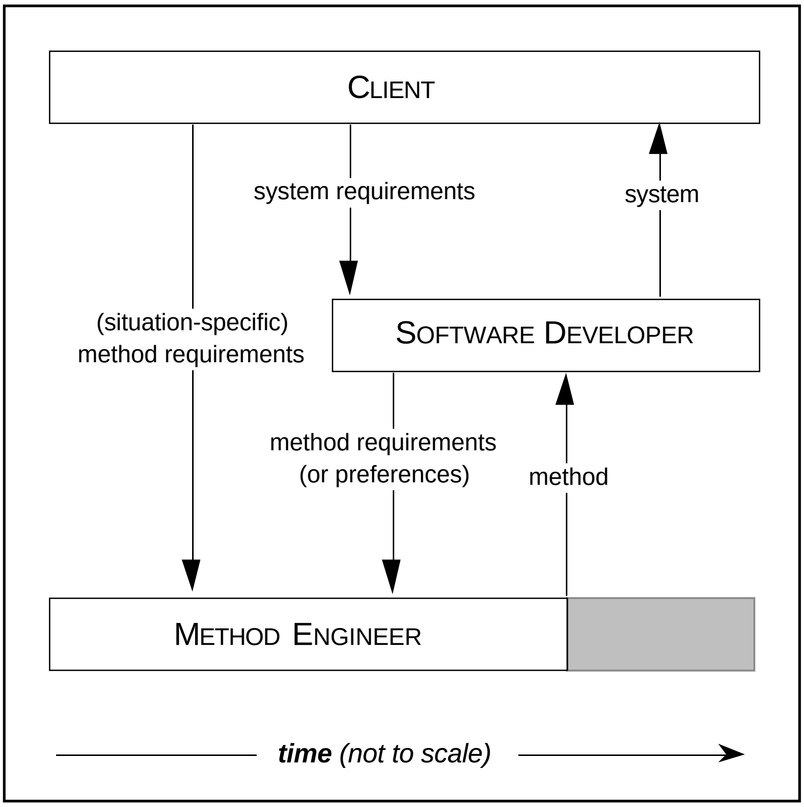


Figure 5.1: *Software development participants. The shaded portion of the method engineer's activities indicates that his involvement in the development may continue even after delivering the software development method to the software developer.*

We now focus our attention on the role and activities of the method engineer in the context of the ViewPoints framework. Naturally, the kinds of method that such a method engineer produces impact upon the way these methods are used - by other development participants in general, and by software developers in particular.

5.2. Method Engineering for ViewPoint-Oriented Method Use

In the ViewPoints framework, a software engineering method is a configuration[13] (structured collection) of ViewPoint templates. These templates are therefore the building blocks of methods, from which system specifications (ViewPoint configurations) are created. The structure of a method in this context is determined by intra- and inter-template relationships. Intra-template relationships are relationships between ViewPoints instantiated from the *same* template, whereas inter-template relationships are between ViewPoints instantiated from *different* templates.

Figure 5.2 captures a number of activities that occur during both method engineering and method use. The particular method shown in the figure is composed of six templates, labelled T1 to T6. The arrows between the templates represent inter-template relationships. Thus, T1 is related to both T2 and T3, while T2 and T3 are related to T4 and T5, respectively. Both T4 and T5 are related to T6. The directions of the arrows also indicate the "preferred" or recommended order of high level development activities; e.g., the order in which the various templates should be instantiated. In this context, the role of the method engineer is to identify, describe and relate the different templates that constitute the required method.

The thick grey arrows in figure 5.2 represent instantiation actions on templates (the "ViewPoint trigger actions", described in chapter 4, section 4.2.2.2). The result of an instantiation action on the template Tx is a ViewPoint VP_{Tx}. Note that a single template such as T4 may be instantiated many times to yield as many ViewPoints (so for example, T4 is instantiated twice in the figure to yield ViewPoints $VP1_{T4}$ and $VP2_{T4}$).

Finally, the various ViewPoints that are created as a result of this templates instantiation process, are themselves related by instantiations of the relationships defined between the templates they were created from. Thus, the inter-template relationship, $\Re$, between templates T2 and T4, becomes an inter-ViewPoint relationship between all ViewPoints instantiated from T2 and T4, respectively. In other words, VP_{T2} is related to both $VP1_{T4}$ and $VP2_{T4}$ by instantiations of the relationship, $\Re$). The instantiation of templates and their relationships is an activity that takes place during so-called "ViewPoint-oriented development", that is, during method use, and is discussed at length in chapter 6.

[13] The term "configuration" is borrowed from "configuration programming" [Kramer 1991], which advocates the use of separate structural descriptions of distributed systems in terms of components and their communication interconnections. In fact, the idea of applying configuration programming principles to method engineering in the ViewPoints framework were first explored in [Kramer & Finkelstein 1991].

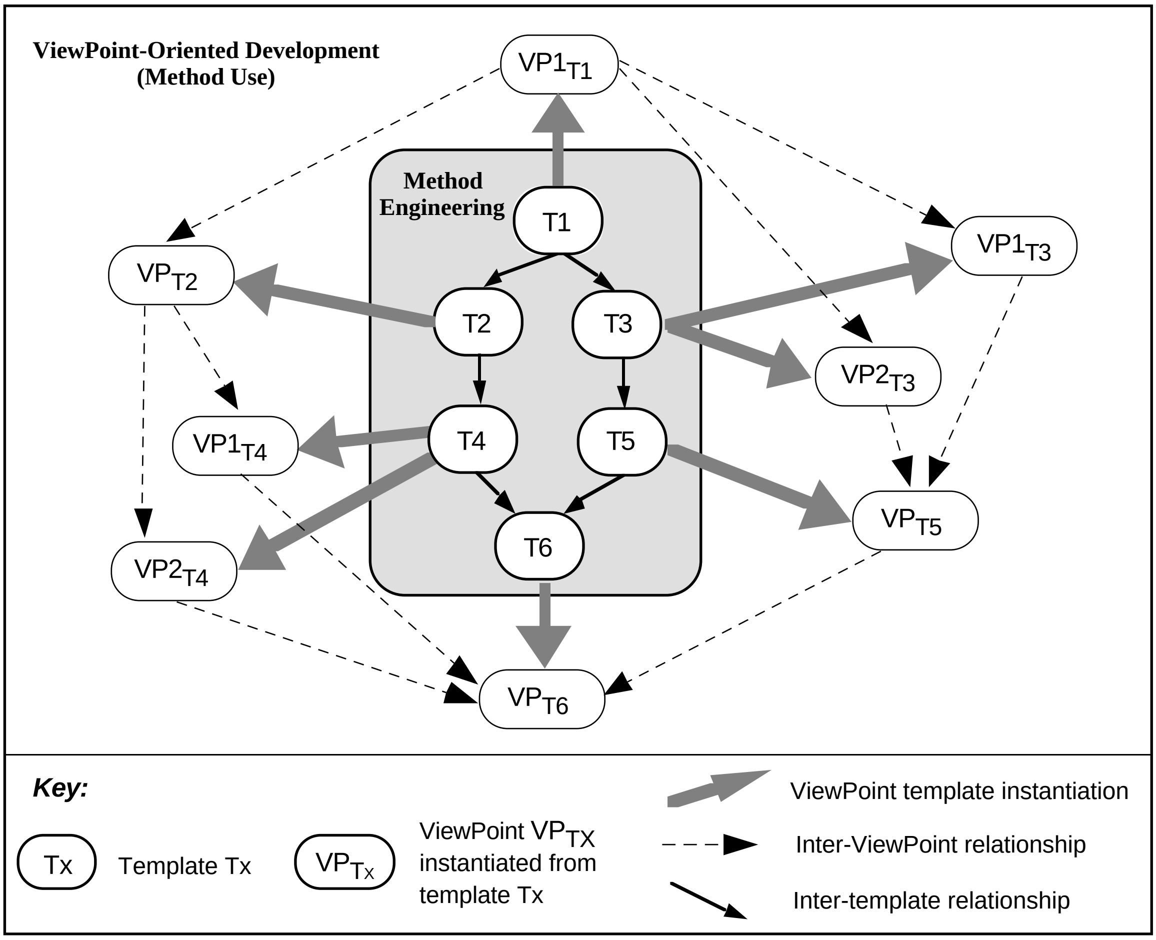

ViewPoint-Oriented Development
(Method Use)
Method Engineering
T1
T2
T3
T4
T5
T6
VP1$_{T1}$
VP$_{T2}$
VP1$_{T3}$
VP2$_{T3}$
VP1$_{T4}$
VP2$_{T4}$
VP$_{T5}$
VP$_{T6}$
Key:
Tx
Template Tx
VP$_{Tx}$
ViewPoint VP$_{TX}$ instantiated from template Tx
ViewPoint template instantiation
Inter-ViewPoint relationship
Inter-template relationship

Figure 5.2: *Method engineering for ViewPoint-oriented method use. Method engineering includes constructing the ViewPoint templates that make up a method, and defining the inter-template relations between them. ViewPoint-oriented development (method use) includes instantiating ViewPoint templates to create new ViewPoints and instantiating inter-template relations to establish inter-ViewPoint relations.*

Figure 5.2 also illustrates some of the complexities of software engineering methods and the different ways by which they may be deployed. In general, methods do not prescribe a sequential series of procedures, but are rather composed of several loosely coupled stages each containing one or more development techniques or sub-techniques (recall that we use the term “development technique” to mean a representation scheme and an associated procedure for deploying it). Thus by treating ViewPoint templates as components in a configuration, a system specification is then developed by creating instances of those templates, which are created or “spawned” dynamically as the development proceeds.

What figure 5.2 does *not* show is the iterative process of refinement by which individual ViewPoints are (partially) developed, and then modified and evolved as a result of traversing the inter-ViewPoint relationship arrows (which represent inter-ViewPoint consistency checks, transformations and transfers). In order to understand and follow such a ViewPoint-oriented development process, the method engineering process must also be understood. Thus, a method in the ViewPoints framework must be designed and constructed in such a way as to allow

ViewPoints to be created when needed, partially developed if necessary, and iteratively checked and refined as the development process proceeds.

5.3. Method Design

In this section, we address method design in the context of a ViewPoint-oriented method engineering process. We outline the steps by which methods are constructed, provide guidelines about how they are - or should be - designed, and comment upon the way in which the design of such methods affects the way in which they are used. The need for method integration in this context is also motivated.

5.3.1. A ViewPoint-Oriented Method Engineering Life Cycle

The objective of method engineering in the ViewPoints framework is to produce a configuration of ViewPoint templates. In other words, the role of the method engineer is to identify, design and construct the templates that make up a method. Typically, method engineers either "invent" new methods from scratch, or assemble (reuse and adapt) methods from a reuse library of method fragments (templates). The way in which methods are "cut up" into their constituent templates is a central method design activity, that relies on experience and the application of some simple heuristics. Figure 5.3 outlines the ViewPoint-oriented method engineering process, described in more detail below.

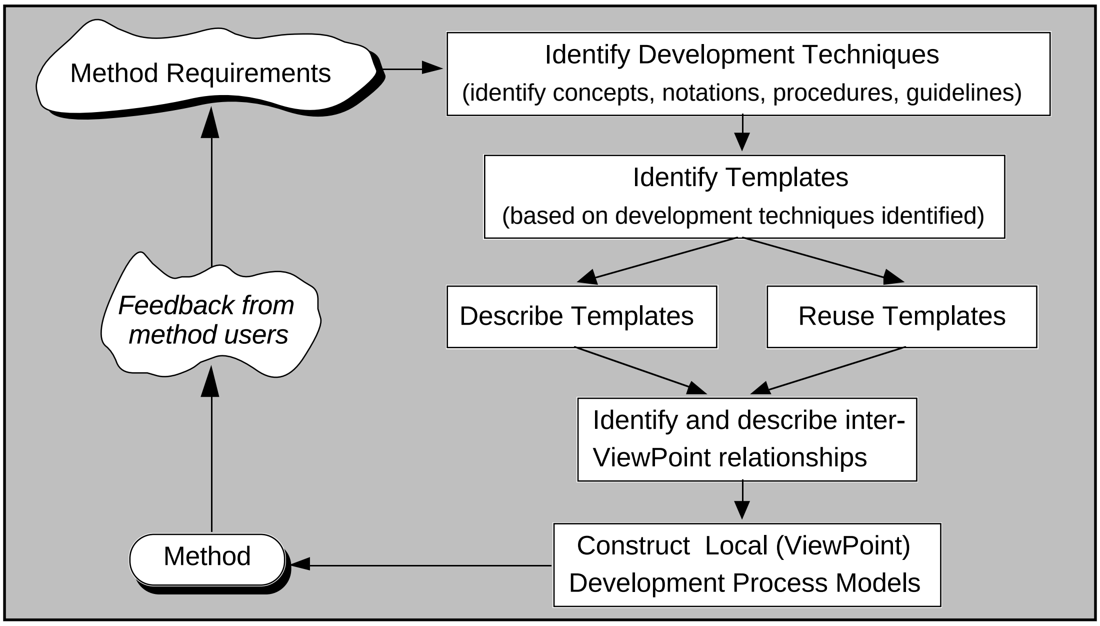

Method Requirements
Identify Development Techniques
(identify concepts, notations, procedures, guidelines)
Identify Templates
(based on development techniques identified)
Feedback from method users
Describe Templates
Reuse Templates
Identify and describe inter-ViewPoint relationships
Method
Construct Local (ViewPoint) Development Process Models

***Figure 5.3:** Method design and construction in the ViewPoints framework.*

- From somewhat uncertain requirements of a software engineering method, a method engineer identifies the development techniques that the target method will deploy. This includes identifying the different notations and development strategies that will be part of that method. For example, Rumbaugh and his colleagues (the method engineers of the Object Modelling

Technique, OMT [Rumbaugh et al. 1991]), identified three kinds of modelling techniques required for the production of an object-oriented design, namely, object, dynamic and functional modelling. For this purpose, they then adapted three well known development or diagramming techniques (entity-relationship, state-transition and data flow) and used them (respectively) to represent the three models of OMT.

- The next step is then to identify the ViewPoint templates that need to be constructed in order to describe the techniques identified above. A good heuristic in such situations is to identify as many simple development techniques as possible (e.g., those that deploy simple notations) and then choose a single template to describe each technique. Of course a method engineer may not have the luxury of choosing what development techniques (and hence templates) to use; rather, he may be restricted in terms of the kinds of techniques or notations required by his customers[14].
- Once the required templates have been identified, they must then be defined more precisely. In other words, the style and work plan slots for each template must be elaborated. Alternatively, templates already described in other methods may be adapted to satisfy the requirements of the new method. Such adaptation usually involves changes in local template style and template work plan actions, particularly the inter-ViewPoint check actions described in chapter 4 (section 4.2.2.2).
- Inter-ViewPoint check actions apply inter-ViewPoint rules. These rules are method-specific attributes of templates. They describe intra- and inter-template relationships, and are the only non-local properties of templates that method engineers need to be concerned with at this stage. If a template has been reused from a different context (method), then these rules also need to be modified or completely re-written. If the template has been defined from scratch then these rules need to be written from scratch. Inter-ViewPoint rules play a number of important roles in method engineering and method use, and will be discussed in much more detail in chapter 6. Primarily however, they relate the different templates of methods together, and are therefore a vehicle for method integration. Method integration facilitates multi-perspective software development of integrated software systems.
- The final method engineering step is to define a local development (process) model for each individual template. Each process model defines when various work plan actions should take place; e.g., when inter-ViewPoint rules should be invoked and checked. Therefore, ViewPoints' local process models also serve to synchronise communication between ViewPoints by coordinating invocation and application of inter-ViewPoint check actions. The coordination of ViewPoint process models in this setting may also be called process integration.

[14] "Customers" of method engineers are software developers who deploy (use) the method(s) produced by method engineers.

The result of the method engineering process described above is a *method* that satisfies the initial requirements, and may be delivered to, and used by, software developers to develop their respective software systems. Of course, in the real world, a method is used by software developers whose experience is then fed back to method engineers who adapt, improve and maintain that method over time. While the dynamic evolution of methods (during method use) is not currently supported by our framework (once these methods' constituent templates have been instantiated), it is clearly an issue that *does* need to be addressed, then carefully supported and controlled.

5.3.2. ViewPoint Template Design

Software development methods are complex artefacts. They typically deploy several representation schemes and often have to cooperate with each other. ViewPoint templates help method engineers reduce the complexity of software development methods by basing their design on the separation of concerns principle. What may appear as a large and complex method is decomposed into its constituent development techniques, which in turn are described as ViewPoint templates. Within each template, the notation and local development process are also separated.

Nevertheless, the design of complex methods still requires skill and insight. With or without the ViewPoints framework in place, "bad" methods may still be designed. A method engineer may choose to define complex methods by combining totally inappropriate development techniques. Thus, while the ViewPoints framework can facilitate the development of partial specifications in different languages, if the relationships between these languages are not clear to the method engineer, then this will be reflected in the use of that method and the software system produced.

For example, a method designer wanting to describe a method that combines Z and CSP notations in a single integrated method, will still have to identify the relationships between these two formal "methods". This may not be easy with or without the ViewPoints framework (although it has been done successfully in the past [Benjamin 1989]). However, what the ViewPoints framework *does* provide is a means of reducing the complexity of methods in general. So the formal method Z for example, may be simplified by describing it using two separate but related templates: one for elaborating schemas and another for textual descriptions. A relationship between these two templates would be that "for every Z schema, there is (or should be) an associated textual description".

Choosing the appropriate ViewPoint templates that constitute a method is a important design activity. Once these templates have been identified however, they also need to be internally designed and elaborated. The representation style of a template is normally easier to define than a template's work plan. Representation schemes may be defined using standard syntax grammar definitions such as BNF grammars, entity-relationship definitions or graph grammars [Rekers 1993]. Work plans on the other hand describe development strategies or processes, and modelling such development activities is still an area of (growing) research [Finkelstein, Kramer & Nuseibeh 1994]. Our approach in the ViewPoints framework has been a pragmatic one. We attempt to keep templates simple by basing them on development techniques that deploy simple notations and development strategies. This simplifies the design of template work plans, because their constituent development process models are also simplified.

In designing and defining ViewPoint template work plans, it is useful to make the distinction between *development actions* and *development rules.* Development actions describe the procedures that may be performed by ViewPoint developers to construct ViewPoint specifications, while development rules constrain development by specifying when, and under what circumstances, various actions should be performed. Development rules describe, for example, local (in-) and global (inter-) ViewPoint relationships that can be used, respectively, to check the consistency of ViewPoint specifications locally and in the context of systems development as a whole. Thus, development rules are part of a ViewPoint work plan's process model, specifying, for example, how to act in the presence of inconsistency.

Again, in designing ViewPoint template work plans, it is useful to distinguish between two kinds of development actions: *canonical development actions* and *analysing actions.* Canonical development actions are of the form:

$$\boxed{d_c: s \rightarrow s'}$$

in which the action, d_c, modifies a specification, s, in a some way to produce a new specification, s'. Canonical development actions can be derived from the syntactic structure of a representation scheme. Thus, we can use a syntax-oriented editor, for example, to implement such actions. Of course, arbitrary modifying actions (such as the assembly actions of the work plan described in chapter 4) are also possible.

Analysing actions, d_a, are of the form:

$$\boxed{d_a: s \rightarrow ag}$$

in which the action, d_c, on the specification, s, results in an agenda, ag, of actions that need to be performed. Formally, an agenda can be represented as a set of pairs (s, d) built from partial specifications and proposed actions. Thus, in this context, the pair (s, d) means that the development action, d, should at some time in the future be applied to partial specification, s. The inconsistency handling actions, described briefly in chapter 4 and in more detail in chapter 7, are examples of such analysing actions. In fact, both in- and inter-ViewPoint check actions are also examples of analysing actions; e.g., when a check action is performed and no inconsistency is detected, then the agenda is updated with a list of other checks that need to be performed.

To summarise, once method designers have chosen the ViewPoint templates that constitute the methods they are designing, they must then decide what development rules they wish to apply on the chosen template representation styles. To apply these rules, they must also specify what actions developers (method users) are allowed to perform, and under what circumstances it is advisable that they do so. The above distinctions are consistent with our earlier organisation of ViewPoint template work plans (in terms of assembly actions, check actions and so on). They do however, provide method engineers with additional insight into the design of methods in our framework .

5.4. Method Construction Support

This section discusses activities that support the process of method construction - before, during and after method design. In particular, it examines the scope for reuse in the ViewPoints framework, the way in which this may be achieved by the method engineer, and the limitations on its successful deployment in this setting.

The scope for tool support of method engineering in general is also addressed, again highlighting the limitations of automated support for method design and construction. In particular, the relationship between method and tool construction is explored, emphasising the role of meta-CASE technology in this context.

Note that method engineering in the ViewPoints framework can be distributed and collaborative. Different templates can be defined by different method engineers with different expertise. However, modelling distributed and collaborative method engineering is beyond the scope of this thesis, but clearly, the modular structure of the framework does facilitate this kind of activity.

5.4.1. Reuse

The ViewPoints framework provides method engineers with an infrastructure for reuse. The most obvious reusable component in the framework is the ViewPoint template, which may be reused in at least three different ways (shown schematically in figure 5.4):

- The first way is to treat templates as standalone *components* belonging to a “reuse library” (of reusable templates). They can be individually “plugged” into the appropriate method, typically after some adaptation. Reusing templates in this way can save method engineers significant time and effort during method design and construction, by reducing the necessity to define templates from scratch. However, the problems of classifying, maintaining and accessing large libraries of reusable components remain - but are beyond the scope of this thesis. Moreover, method engineers need to invest significant planning and design effort in order to build and populate such reusable template libraries.
- The second way is to reuse ViewPoint templates through *instantiation*. This is currently the most common kind of reuse in the ViewPoints framework, and straddles both method

engineering and method use. Templates are analogous to “types” or “classes” containing generic information which is passed on to instances of these templates. This allows multiple ViewPoints to share the same notations and/or development strategies - if they are instantiated from the same template. Reuse by instantiation however, usually means that “type level” (template level) information is duplicated in each instance (ViewPoint).

- The third way, which we currently do not support, is to reuse template knowledge through *inheritance.* Many templates may share the same representation and/or process knowledge. To avoid duplicating such knowledge in each new template that is constructed, a “super-template/sub-template” structure may be developed (*c.f.* super-classes/sub-classes in object-oriented inheritance structures). For example, one could define an “abstract template” (*c.f.* abstract class) that describes a generic data flow diagramming technique, with two further sub-templates for respectively representing two notational variations of this technique [DeMarco 1978; Gane & Sarson 1979][15]. A formal definition of inheritance between ViewPoint templates has been explored in [Mödl 1991], but we do not support it in the ViewPoints framework described in this thesis[16].

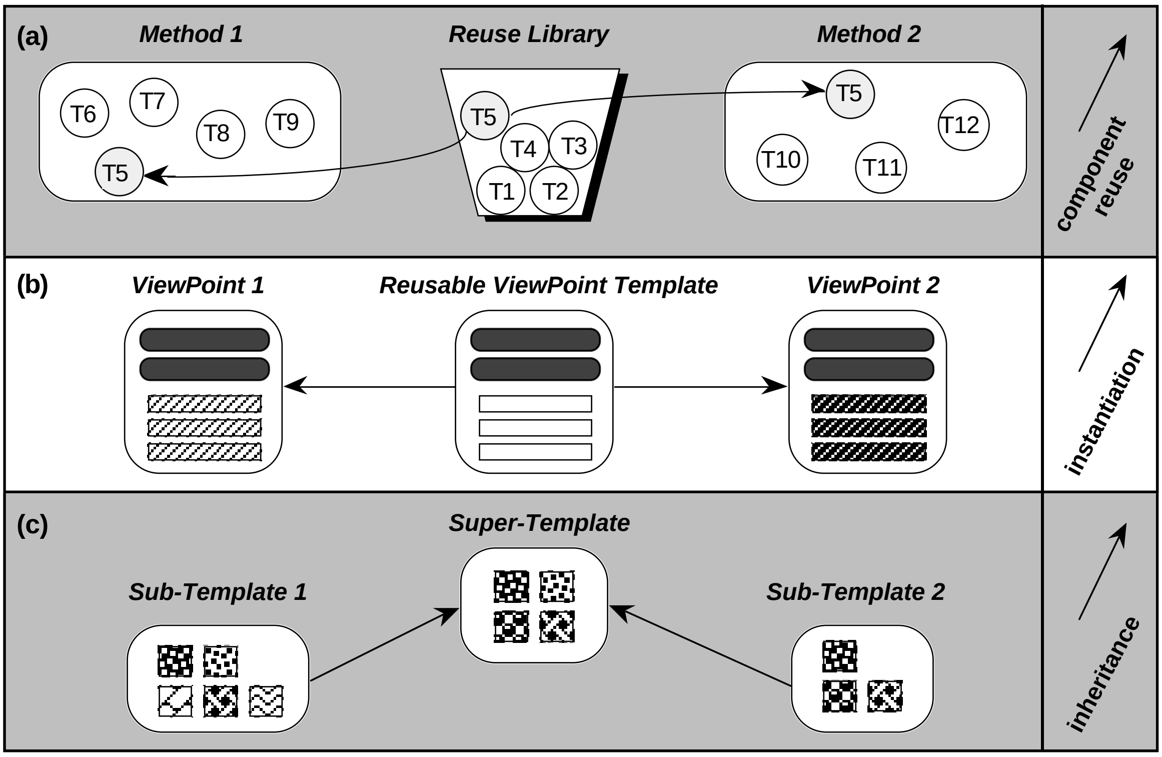


Figure 5.4: *ViewPoint templates as vehicles for reuse. (a) Templates as reusable components; (b) Multiple ViewPoints instantiated from the same (reusable) template; and (c) Templates sharing (reusing) knowledge through inheritance.*

[15] The only differences between the two techniques are stylistic; e.g., one uses rectangles with rounded corners rather than circles to represent processes.

[16] See section 4.4.3.1 in chapter 4 for an explanation of why inheritance has been excluded from the current ViewPoints framework.

Reuse in the ViewPoints framework is an aid to method construction. Significant time and effort can be saved by reusing existing templates and adapting them to new method requirements. Templates are highly reusable components - except that is, for that part of the their work plans which specifies inter-ViewPoint relationships. Strictly speaking, inter-ViewPoint check actions should be defined separately from individual templates because they are not part of any single development technique per se, but rather describe relationships that method engineers wish to express between these techniques. They are also more method-dependent, and therefore less reusable, than the remaining elements of ViewPoint templates. As a pragmatic compromise, while we still include inter-ViewPoint check actions (and their associated rules) in template definitions (in order to maximise the distributability of ViewPoints instantiated from these templates), we nevertheless maintain them separately within their respective templates so that they can be accessed and modified more easily when the templates are reused.

The incorporation of reuse into the method design process however, is not as clear. Designing reusable components (i.e., designing templates *for* reuse) and designing methods *with* reuse are still areas of active research [Hall 1992b; Hall 1992a]. What the ViewPoints framework does, is provide an infrastructure for facilitating these activities.

Finally, it is worth observing that, from a method user’s point of view, the ViewPoints framework also facilitates reuse. ViewPoints themselves are reusable (domain-specific) components, that can be classified, abstracted and combined. However, reuse of domain abstractions [Maiden 1992] is not a method engineering activity, and is also beyond the scope of this thesis.

5.4.2. Automated Tools

Computer-aided method engineering (CAME) supports the design and development of methods and their associated (CASE) tools. In the ViewPoints framework, CAME includes *tools* that support template definition, reuse and maintenance, and *environments* that facilitate CASE tools development (the latter also include meta-CASE tools for generating CASE tools to support methods and their constituent templates). Actual support for the ViewPoints framework is provided by *The Viewer* prototype environment, and is described in more detail in chapter 8. In this section, we briefly outline the “ideal” requirements for CAME tools support in this context.

A ViewPoint-oriented method engineering support environment should, at the very least, provide tools for basic editing of ViewPoint templates. Structured editors for template styles, and tools to facilitate process modelling in template work plans, are particularly useful. A templates library or database management system is also needed for effective deployment of reuse in this context.

In order to integrate methods effectively, inter-ViewPoint rules between templates need to be identified and expressed. Automated tools that facilitate the construction and syntax checking of these rules are therefore desirable. Moreover, in some methods, an inter-ViewPoint rule defined in one template may have an “inverse” in another template to which it relates. Tool support in this context may be helpful in automatically generating the inverse of a rule in the corresponding or related template.

A ViewPoint template defines one of a method’s constituent development techniques. It can therefore serve as a specification for a CASE tool developed to support that technique. Using “traditional” meta-CASE technology, a template definition may be used to generate a CASE tool to support ViewPoint development based on that template. We have found that effective tool integration can also be achieved in this setting, if the methods (templates), on which the tools are based, are also integrated. In fact, the same inter-ViewPoint rules that are used to integrate methods can be used to integrate tools. Again, chapter 8 addresses tool integration in this context in much more detail.

While method engineering and method use are usually two completely separate software engineering activities, it is desirable that the relationship or link between method engineering and method use is clear in any environment that supports the ViewPoints framework.

The Viewer environment attempts to make contributions towards satisfying the above requirements.

5.5. Method Use

Finally, we briefly examine the impact of our method engineering process on method use.

In general, method use is the deployment of a method to specify and develop software system. In the ViewPoints framework, this is characterised by the creation and development of many ViewPoints, each of which addresses a particular aspect or part of the system being developed. In this way, ViewPoint-oriented software development supports the proliferation of multiple views during requirements, multiple solutions during design, and multi-paradigm programming during implementation. One could even treat the different pieces of system documentation as different ViewPoints (e.g., programmer manuals, user manuals, multi-lingual manuals, etc.).

The separation of concerns provided by ViewPoints facilitates distributed, collaborative software development by multiple development participants. Each ViewPoint can be used to represent one or more stakeholders in the development process, in addition to the particular views of the world or system that these stakeholders are concerned with. Each individual ViewPoint’s process model guides the local development of its ViewPoint’s specification, and coordinates periods of interaction with other ViewPoints. While there needs to be some starting point to any software development project (that triggers the creation of, at least, the first ViewPoint), development from then on can proceed in a fully distributed fashion, with individual ViewPoints “spawning” (instantiating templates) to create other ViewPoints on-the-fly (this is the role of the "ViewPoint trigger actions" described in chapter 4 - section 4.2.2.2).

Similarly, the "end" of a software development project is also subjective. Since ViewPoint-oriented software engineering (VOSE) results in a number of partial specifications (ViewPoints), one satisfactory endpoint to such a development process may be the existence of a particular configuration of ViewPoints that have been checked to some satisfactory level of consistency. One can then select that particular collection of ViewPoints and group them together as the system's "requirements", while another collection can represent the system's "design", and so on. This reflects that actual nature of software development, in which development is incremental and in which even the "final" delivered system evolves over time.

5.6. Chapter Summary

This chapter has outlined a ViewPoint-oriented approach to method engineering.

Within the context of the ViewPoints framework, methods are composed of ViewPoint templates related by inter-ViewPoint rules. The role of the method designer is to select the appropriate ViewPoint templates that constitute a method, and then describe their individual representation styles and work plans (including any inter-ViewPoint relationships and local development strategies). This method design process may also involve reusing (and typically adapting) existing ViewPoint templates.

The method construction process can also be supported by a variety of automated tools. Some of these tools facilitate the management and support of the method engineering life cycle in this context, while others facilitate the development of CASE tools to support methods in general and methods' constituent development techniques (templates) in particular. Meta-CASE tools fall into the latter category, and provide one route from method engineering to computer-aided method use.

This chapter has also emphasised the need to combine (integrate) different methods - or in ViewPoints terminology, to combine (integrate) templates. The role of inter-ViewPoint rules as a vehicle for such integration was proposed and briefly discussed. The next chapter addresses such method integration in more detail.

Chapter 6 Method Integration

This chapter proposes a model of inter-ViewPoint consistency management, in which "inter-ViewPoint communication" is used as a vehicle for implementing method integration. While chapter 5 identified method integration as primarily a method engineering activity, its effectiveness can only be demonstrated by the actual development of an integrated software system. Thus, the model described in this chapter straddles the activities of both method engineering and method use, in order to illustrate how an integrated method can be used to develop an integrated multi-perspective specification. While the emphasis of the chapter is on expressing the inter-ViewPoint relationships that are used for method integration, the actual checking of these relationships is also addressed, and a sample protocol for inter-ViewPoint consistency checking is presented. Since consistency checking may also identify inconsistencies, the model presented incorporates an inconsistency handling dimension, which is described in more detail in the next chapter.

We begin with an initial working definition of method integration in the context of the ViewPoints framework, which is elaborated further at the end of the chapter in light of the material presented. We then discuss the role of ViewPoint interaction and coordination as vehicles for managing consistency, handling inconsistencies, and developing integrated multi-perspective specifications. Our terminology in this context is also defined. The discussion is used as the basis for developing a model of ViewPoints integration, whose implementation[17] achieves the method integration we require.

6.1. Method Integration in the ViewPoints Framework: Terminology

This section discusses the terminology we use in our approach to method integration in the ViewPoints framework. In particular, we attempt to clarify our use of the terms method integration, consistency and inconsistency management, and ViewPoint interaction,

[17] We are not necessarily referring to a *software* "implementation" of our model (although *The Viewer* prototype described in chapter 8 *is* one such implementation). Rather, by "implementation" we mean the "use" or "enactment" of the model to build a multi-perspective software specification.

communication and coordination. The relationships between inconsistencies, conflicts, contradictions and mistakes are also examined.

6.1.1. Method Integration: A Working Definition

We define method integration in the ViewPoints framework as the process of combining a collection of ViewPoint templates together in such a way that the ViewPoints instantiated from these templates are also integrated. Two ViewPoints are integrated if all the relationships that have been defined between them have been checked and are found to hold. Thus, integration in this context has a very specific meaning: it is the satisfaction of some relationship. Of course, such relationships themselves may be defined and managed collectively in such a way so as to achieve a variety of higher level objectives; e.g., process integration to coordinate and synchronise the concurrent development of multiple ViewPoints.

6.1.2. ViewPoint Interaction and Coordination

The ViewPoints framework achieves separation of concerns at both the method engineering and method use levels. During method engineering, methods are decomposed into their constituent development techniques, represented by ViewPoint templates. During method use, systems specifications are constructed from several partial specifications, represented by ViewPoints instantiated from ViewPoint templates. However, such a ViewPoint-oriented development process requires that the different ViewPoints representing a system specification are somehow combined. Method integration in this context is the process of combining the templates that constitute a method, in order that the ViewPoints instantiated from these templates are also combined (integrated).

In order to achieve integration in this setting, ViewPoints developed separately need to interact ("communicate") intermittently in order to coordinate their activities and check the consistency of their specifications against each other. These interactions are a function of the method integration relationships defined between methods' constituent ViewPoint templates.

6.1.3. Consistency Management

The process of integrating two or more ViewPoints requires checking and establishing any integration (consistency) relationships that method engineers have prescribed should hold between these ViewPoints. In this setting, *consistency management* is the process of (1) expressing inter-ViewPoint relationships, (2) checking that they hold, and (3) handling inconsistencies when any of these relationships do not hold. Checking consistency across ViewPoints subsumes local (in-ViewPoint) consistency checking, which therefore also falls under the umbrella of consistency management. In fact one could argue that the entire software development process is about building artefacts that can be subjected to various levels of consistency checking. Figure 6.1 below summarises the scope of consistency management activities, which begin during method engineering and extend to method use[18].

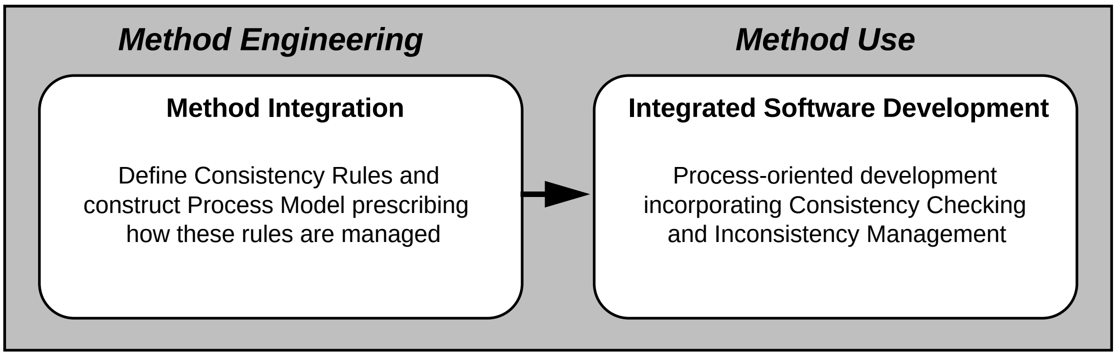

Method Engineering
Method Use
Method Integration
Define Consistency Rules and construct Process Model prescribing how these rules are managed
Integrated Software Development
Process-oriented development incorporating Consistency Checking and Inconsistency Management

Figure 6.1: *Consistency management.*

As a whole, this chapter addresses the management of consistency between ViewPoints. However, in order to manage consistency, the notion of inconsistency requires further clarification.

6.1.4. Inconsistency Management

Multi-perspective development, by definition, implies the existence of multiple views that are potentially inconsistent. The ViewPoints framework permits the independent development of such views (ViewPoints), and does not require the maintenance of consistency between them. Of course, the other extreme to universal consistency maintenance is a completely ad hoc development process - which can be chaotic, and therefore is also undesirable. A process of *inconsistency management* is thus needed in order to (1) detect and record inconsistencies, (2) facilitate continued development in the presence of such inconsistencies, and (3) work towards resolving or removing inconsistencies in the final delivered system.

Inconsistency management is part of the overall process of consistency management which is addressed by this chapter. In chapter 7, we explore techniques for handling inconsistencies, and in particular experiment with a temporal, action-based logic framework that supports software development in the presence of inconsistencies. In the remainder of this chapter however, we focus on expressing consistency relationships between ViewPoints, and examine the process by which ViewPoints interact in order to detect inconsistencies. Before we begin, some clarification of our terminology is due.

[18] Of course, some inconsistencies can only be defined during method use (since they are domain-specific inconsistencies that only occur when specifying an actual problem). We do not address these kinds of inconsistencies in this chapter on method integration for precisely that reason: they are *not* method integration consistency relationships. We do however, acknowledge (in chapter 10) that these *are* an important kind of inconsistency that needs to be managed in software development.

6.1.4.1. Terminology

Thus far, we have assumed an intuitive understanding of the term “inconsistency”, although we have not defined it precisely. In fact, a number of related terms also require clarification and include notions of “conflict”, “contradiction” and “mistake”. These are shown schematically in figure 6.2, which illustrates how inconsistencies manifest themselves in different ways. The figure also distinguishes between inconsistency and conflict: inconsistency is the main tool we have for revealing conflicts, but not all inconsistencies are the result of conflict.

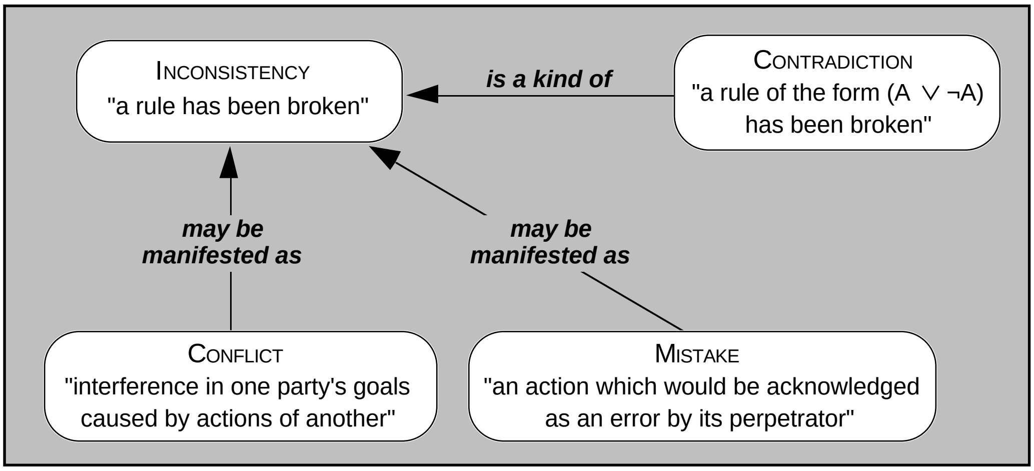

INCONSISTENCY
"a rule has been broken"
is a kind of
CONTRADICTION
"a rule of the form (A ∨ ¬A) has been broken"
may be manifested as
may be manifested as
CONFLICT
"interference in one party's goals caused by actions of another"
MISTAKE
"an action which would be acknowledged as an error by its perpetrator"

Figure 6.2: *Inconsistencies, conflicts, contradictions and mistakes.*

An *inconsistency* occurs if a rule has been broken. Such a rule is laid down by a method engineer, when specifying the correct use of an integrated method. Thus, what constitutes an inconsistency in any particular situation is entirely dependent on the rules defined during method design. Such rules may, for example, specify the correct use of a notation, and the relationships between different notations.

We treat *contradictions* as a special kind of inconsistency in which a logical rule of the form (A ∨ ¬A) has been broken. In some cases contradictions may be derived through logical inference. However, no logical inferences are drawn unless explicit consistency checking rules have been defined to carry them out. Hence, derived contradictions still adhere to our definition of inconsistency.

We define *conflict* in general as interference in the goals of one party caused by the actions of another party [Easterbrook et al. 1993]. Note that this definition refers to an action, in that conflict cannot occur unless somebody does something. Thus, two partial specifications may be inconsistent (according to some consistency rule), but they are only in conflict if the developers of either or both specifications attempt to combine them together somehow.

Finally we define a *mistake* as an action that would be acknowledged as an error by the perpetrator of that action. We deliberately blur the role of perception in that we are not interested in the circumstances that lead the perpetrator to acknowledge the error. Such an acknowledgement may, for example, result from the action being pointed out by another person, or through a process of argumentation.

6.1.4.2. Implications of Terminology on ViewPoints

Clearly then, inconsistencies (and contradictions) concern static situations, while conflicts and mistakes concern actions. In other words, a given collection of ViewPoints can be inconsistent (or contradictory), while actions on those ViewPoints may be mistaken or in conflict. Hence, we can test a collection of ViewPoints for the existence of inconsistency, but we cannot test for conflicts or mistakes. Conflicts and mistakes *may* manifest themselves as inconsistencies. Each inconsistency *must* have one or more mistakes and/or conflicts as the cause. Moreover, the existence of an inconsistency may lead to further mistakes and conflicts in subsequent actions.

If we examine the definitions of conflict and mistake more closely, it should be apparent that they are not mutually exclusive. For example, if a person is persuaded that their action was a mistake only after a lengthy argument, then the mistake would also appear to be a conflict. This apparent overlap between mistakes and conflicts arises from the fact that they may both manifest themselves as inconsistencies. In fact, there appears to be a continuum, with conflicts at one end and mistakes at the other. This continuum also expresses a scale of the amount of effort (typically argumentation) needed to resolve an inconsistency. Figure 6.3 illustrates the continuum and populates it with some examples.

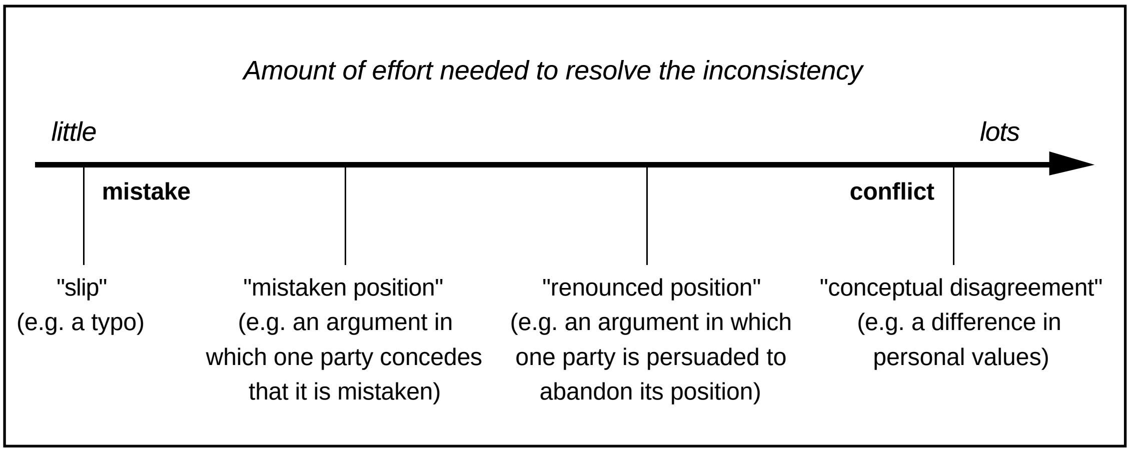

Amount of effort needed to resolve the inconsistency
little
lots
mistake
conflict
"slip"
(e.g. a typo)
"mistaken position"
(e.g. an argument in which one party concedes that it is mistaken)
"renounced position"
(e.g. an argument in which one party is persuaded to abandon its position)
"conceptual disagreement"
(e.g. a difference in personal values)

Figure 6.3: *Effort needed to resolve inconsistencies.*

In some cases, no amount of effort will detect a mistake, because the inconsistency was caused by a substantive conflict between ViewPoints. In other cases, one ViewPoint (*c.f.* alternative solution) is seen as better than the other, but a strict substitution is not appropriate. For example an “incorrect” ViewPoint may still contain important information that is missing from the “correct” one. In yet other cases, both ViewPoints may have separate merits. In such cases, alternative resolution strategies are needed, such as compromise, merging or reformulation.

6.1.5. Summary

Our objective in managing inconsistency is to produce an integrated (consistent) configuration of ViewPoints which represents a system specification or software system. During the development of such a configuration, inconsistencies may exist and need to be handled (although this does not necessarily mean immediate resolution of such inconsistencies). Inconsistency management is an activity that is part of consistency management which, in turn, is necessary to achieve ViewPoints integration. ViewPoints integration is part of a ViewPoint-oriented software development process which provides integrated methods with which ViewPoints integration may be achieved. Figure 6.4 summarises this schematically.

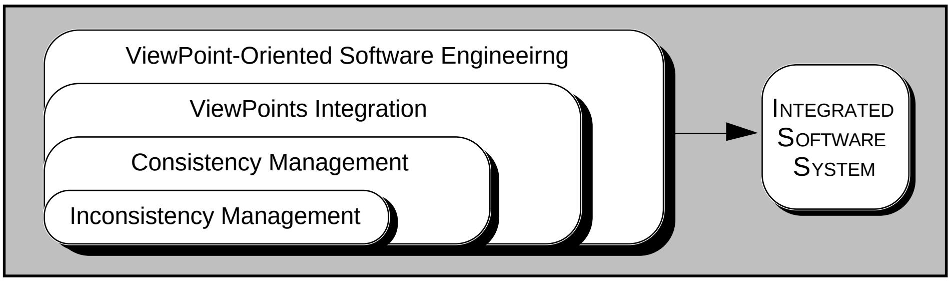


***Figure 6.4:** Positioning ViewPoints integration.*

In this section we have clarified our use of the terminology with regards to method integration in the ViewPoints framework. In the next section, we deploy the terms that we have defined to develop a model of ViewPoints integration, which itself supports the method integration we require.

6.2. A Model of ViewPoints Integration

In this section we examine the notion of ViewPoints integration - the goal of method integration. ViewPoints integration is a function of the relationships defined between ViewPoints in a configuration. An integrated multi-perspective specification, is an integrated configuration of ViewPoints in which the relationships between the constituent ViewPoints hold. These relationships are represented by *inter-ViewPoint rules,* which provide the integration “glue” that is required for integrated software development. Thus, there is a need to express these rules, to decide when they should be invoked, and to check whether or not the relationships expressed by these rules do in fact hold.

Figure 6.5 is a schematic model of such a ViewPoints integration process. This model is central to our approach, and is elaborated in this, and the next, chapters. Basically, it identifies the stages that inter-ViewPoint rules pass through, and illustrates the order of, and preconditions for, moving from one stage to the next. The activities modelled straddle both method engineering and method use, and therefore have an impact on both method integration and integrated software development.

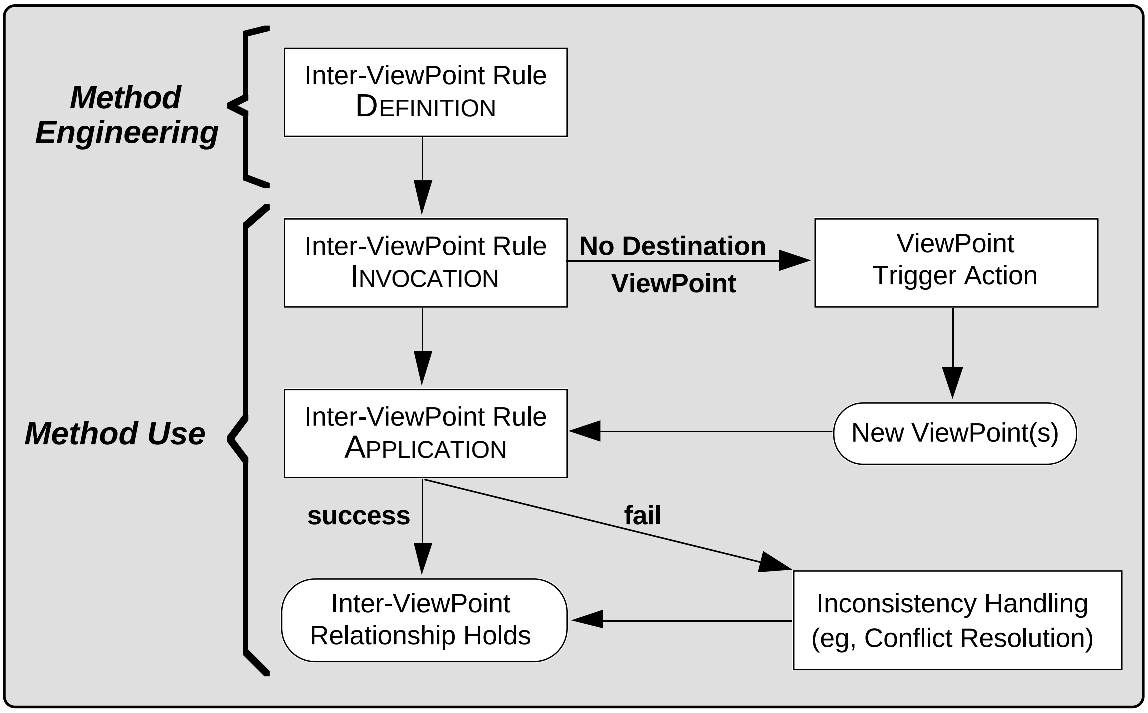

Method Engineering
Inter-ViewPoint Rule DEFINITION
Method Use
Inter-ViewPoint Rule INVOCATION
No Destination ViewPoint
ViewPoint Trigger Action
Inter-ViewPoint Rule APPLICATION
New ViewPoint(s)
success
fail
Inter-ViewPoint Relationship Holds
Inconsistency Handling (eg, Conflict Resolution)

***Figure 6.5:** A model of ViewPoints integration. Labelled arrows represent preconditions for the next step to be performed. Of course, inconsistency handling may not remove/resolve an inconsistency, but this is not shown in the diagram*

Recall that two important design objectives of the ViewPoints framework are to maximise decentralisation of, and handle inconsistencies between, ViewPoints. We therefore distribute the different inter-ViewPoint rules among the different ViewPoints (as opposed to maintaining them centrally), and we do not attempt to enforce total consistency as a matter of course. This has a number of consequences on our approach:

- In some cases, inter-ViewPoint rules need to be included in both the ViewPoints they relate[19].
- There is a danger in proliferation of inter-ViewPoint rules if ViewPoint templates are designed with large areas of overlap, or if a method engineer relates one template to too many other templates.
- An explicit inconsistency handling step is required in order to recover from situations in which inter-ViewPoint rules have been broken.
- In some cases where an “overall” development process is well understood, some effort is still needed in order to distribute this process among the different constituent ViewPoint process models.

[19] More on this in the next section, but clearly, because we distribute the rules between ViewPoints among these different ViewPoints, some relationships may need to be duplicated and “inverted” or “reversed”. Thus, if one ViewPoint contains the rule A > B, then the other should contain the rule B < A (of course, if the relationship is commutative, say A = B, then it need not be changed; i.e., B = A).

Before we address each of the stages of our ViewPoints integration model in detail, let us examine each briefly, in order to get an intuitive understanding of the consequences of managing inter-ViewPoint consistency in this way. For the purposes of this preliminary discussion, we treat an inter-ViewPoint rule as one which describes a relationship, $\mathfrak{R}$, between a *source* ViewPoint, VP_S, and a *destination* ViewPoint, VP_D. In other words, an inter-ViewPoint rule is of the general form:

$$\boxed{VP_S \; \mathfrak{R} \; VP_D}$$

The two related ViewPoints, VP_S, and VP_D, are called “source” and “destination” ViewPoints to convey the fact that the source ViewPoint *contains* the rule and *initiates* its invocation, while the “destination” ViewPoint *responds* to queries from the source ViewPoint. This is in line with our requirement of fully distributing inter-ViewPoint rules among ViewPoints. We now examine the interpretation such inter-ViewPoint rules, and their structural consequences.

6.2.1. Method Integration through Inter-ViewPoint Rule Definition

During method design, a method engineer identifies the constituent templates of a method, and identifies the relationships that need to hold between ViewPoints instantiated from these templates. Thus, inter-ViewPoint rules in this setting represent “hypothetical” relationships between ViewPoints that have not yet been created. This is because method engineering is at the “type” (template) rather than “instance” (ViewPoint) level. In other words, a method designer expressing the relationships between source and destination ViewPoints, VP_S and VP_D respectively, is in fact saying: *if* the ViewPoints VP_S and VP_D exist, *then* there should be a relationship $\mathfrak{R}$ that holds between them. The broken lines in figure 6.6 denote ViewPoints that have not yet been created, and therefore a relationship, $\mathfrak{R}$, that does not yet hold.

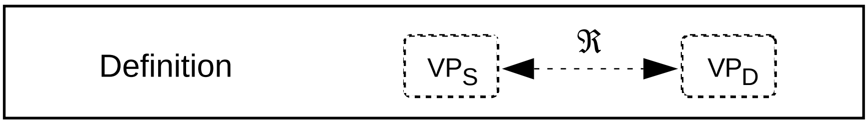


Figure 6.6: *Inter-ViewPoint rule definition.*

Therefore, inter-ViewPoint rule definition is the process of expressing intra- and inter-template relationships - the primary elements method integration.

6.2.2. Process Modelling through Inter-ViewPoint Rule Invocation

During method deployment, software developers elaborate individual ViewPoints separately. Each individual ViewPoint’s process model prescribes points at which inter-ViewPoint rules ought to be invoked and then checked. Invoking an inter-ViewPoint rule from within any individual ViewPoint, VP_S, is equivalent to saying: *if* there is a destination ViewPoint, VP_D, *then* the relationship, $\mathfrak{R}$, should hold between VP_S and VP_D. The solid line around VP_S in figure 6.7

indicates that VP_S already exists (it is the ViewPoint from within which the rule was invoked), while the broken lines indicate that the ViewPoint VP_D and the relationship $\Re$ have not yet been created and found to hold, respectively. Of course what the rule also says is that before we proceed to the next stage (inter-ViewPoint rule application), one or more destination ViewPoints need to be created. This is done by the performance of a ViewPoint trigger action (described in chapter 4, section 4.2.2.3) as shown in figure 6.5.

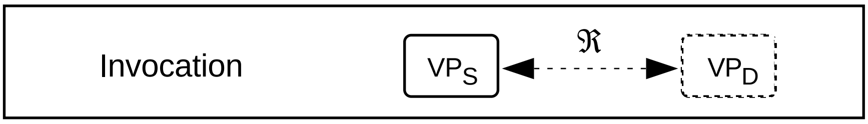


***Figure 6.7:** Inter-ViewPoint rule invocation.*

Therefore, inter-ViewPoint rule invocation is the process of identifying the points at which consistency (integration) needs to be checked.

6.2.3. System Development through Inter-ViewPoint Rule Application

Once two related ViewPoints, VP_S and VP_D, have been identified, then the inter-ViewPoint rule relating them needs to be checked. Inter-ViewPoint rule application is therefore the process of asking: does the defined relationship $\Re$ hold between VP_S and VP_D? The broken line in figure 6.8 represents the relationship $\Re$ - which has yet to be checked.

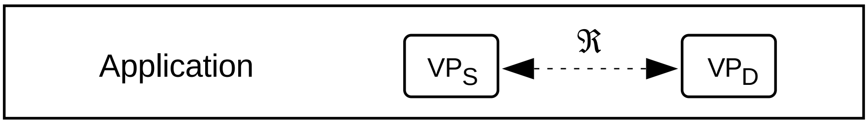


***Figure 6.8:** Inter-ViewPoint rule application.*

If the relationship $\Re$ is found not to hold, then a process of inconsistency handling may be entered, whose eventual objective is to make $\Re$ hold. Recall that the objective of the entire consistency management process, as represented by this ViewPoints integration model, is to achieve inter-ViewPoint consistency by making the appropriate inter-ViewPoint rules hold. Therefore, whether inter-ViewPoint rules hold immediately or after some inconsistency handling, the final desired state is shown in figure 6.9, in which the relationship $\Re$ is confirmed to hold between VP_S and VP_D (hence the solid lines in all parts of the diagram).

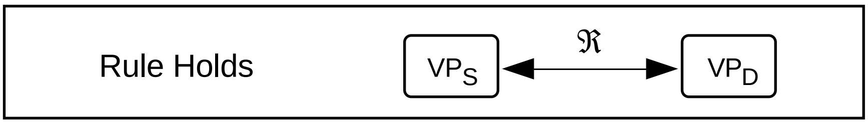


***Figure 6.9:** Inter-ViewPoint rule holds.*

Thus, inter-ViewPoint rule application is the process of consistency checking between two interacting ViewPoints with the objective of, eventually, making the relationship expressed by the applied rule hold.

6.2.4. Structural Consequences

Inter-ViewPoint rules reveal different structural information about methods, development processes and system specifications, depending on the stage at which they are observed. This information may be useful when constructing tool support for ViewPoint framework, and may be used to visualise the structure of methods, the state of development processes, and the structure of systems being developed.

6.2.4.1. Methods

During method engineering inter-ViewPoint rules describe intra- and inter-template relationships. Therefore, when combined with the templates they relate, these rules provide information about the structure of the methods they integrate. Recall that a method in the ViewPoints framework is a configuration (structured collection) of ViewPoint templates. This structure is determined by the relationships (expressed as rules) between the method's constituent templates. These relationships are denoted by arrows in figure 6.10 below.

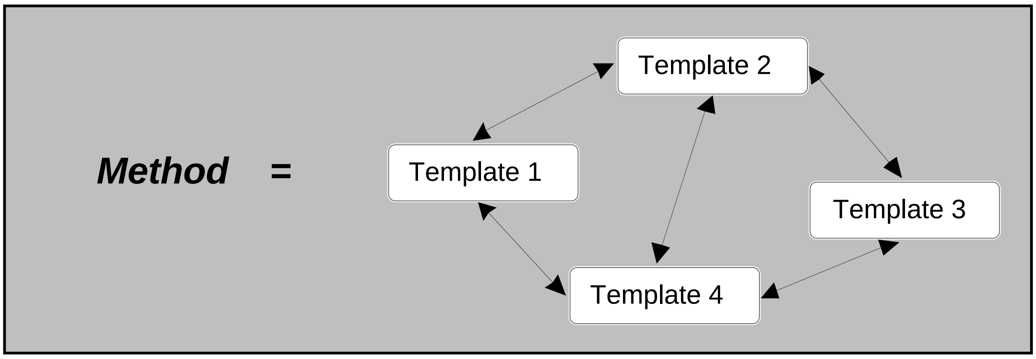


Figure 6.10: *ViewPoint-oriented method structure is determined by inter-template rules.*

Therefore, in our model of ViewPoints integration, inter-ViewPoint rule definition is also the step in which the structure of methods is defined.

6.2.4.2. Process

At any point during the development process, not all the required ViewPoints will have necessarily been created. Moreover, at any point during development, a number of ViewPoints *can* be directly created from the current configuration of ViewPoints. Which ViewPoints *ought* to be created is determined by the inter-ViewPoint rules that have been defined during method design. These rules represent relationships between ViewPoints in a configuration, and can therefore be used to determine, and "prompt" for, missing ViewPoints. Figure 6.11 is a "snapshot" of a ViewPoint-oriented development process. Broken lines represent ViewPoints that have not yet been created, but which are "reachable" from existing ViewPoints (they are reached by invoking the inter-ViewPoint rules shown in the diagram).

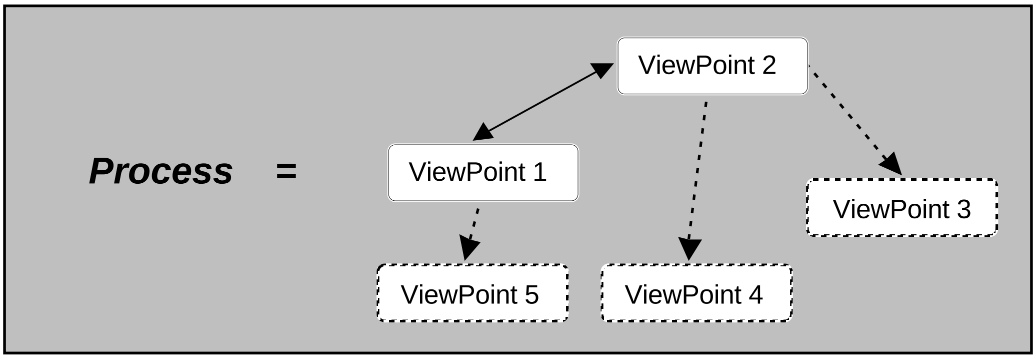


Figure 6.11: *ViewPoint-oriented development process is determined by inter-ViewPoint rules. ViewPoints 1 and 2 have been created and ViewPoints 3, 4 and 5 can be "spawned" directly from them.*

6.2.4.3. System

The objective of a ViewPoint-oriented development process is to produce a system specification (or software system) that is consistent (according to some inter-ViewPoint rules). In the ViewPoints framework, the objective is to produce an integrated configuration of ViewPoints, in which the relationships that were defined during method design, have been checked and found (or been made) to hold. Figure 6.12 illustrates how the structure of a system specification is determined by inter-ViewPoint rules. Broken lines indicate inter-ViewPoint relationships that are yet to be checked, and which do not necessarily hold.

The figure also highlights the potential problem of scale posed by having such a configuration of ViewPoints. This as an area for further work and will be discussed in more detail in chapter 10. In particular, ways of managing (and navigating) large configurations of related ViewPoints need to be implemented.

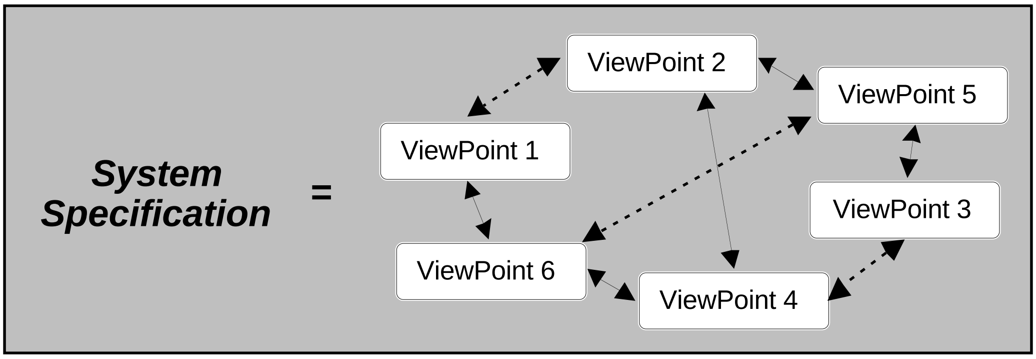


Figure 6.12: *ViewPoint-oriented system specification is determined by inter-ViewPoint rules.*

In our model of ViewPoints integration, inter-ViewPoint rule application is the step in which the relationships between ViewPoints are checked, and eventually made to hold. The structured collection (configuration) of ViewPoints produced at the end of this step is the system specification or software.

6.2.5. Summary

In order to support multi-perspective software development effectively, such development must be based on integrated software development methods. Thus, in the context of the ViewPoints framework, method integration is a vehicle for achieving ViewPoints integration. ViewPoints integration is the production of a consistent collection of ViewPoints by applying inter-ViewPoint consistency rules. These rules define pairwise relationships between ViewPoints and are the key to structuring and integrating methods, processes and specifications in this setting.

The ViewPoints integration model described in this section has identified a number of different activities that are necessary for achieving integration in this context. It is based on the notion of inter-ViewPoint rules - how they are expressed, when and how they are used, and the consequences of their use. The manipulation of inter-ViewPoint rules is the key to managing consistency between ViewPoints, and our model is as much a model of consistency management as it is a model of ViewPoints integration.

6.3. Expressing Inter-ViewPoint Relationships

In this section we propose a notation for expressing inter-ViewPoint rules, which facilitates the representation of relationships between partial specifications contained in different ViewPoints. What we aim to demonstrate is the feasibility of expressing pairwise ViewPoint relationships which may be used to integrate the different templates of a method. While the inter-ViewPoint rules described in this section express relationships between the representation schemes used by different ViewPoints, the next section discusses how these relationships may be used to coordinate the development processes of different ViewPoints by means of consistency checking protocols and associated mechanisms.

Expressing inter-ViewPoint rules is a process of method integration in which a number of development techniques are related and thus combined. To illustrate our approach, we present simple example fragments from the requirements method CORE (described in chapters 2 and 3, in sections 2.1.2 and 3.3.1 respectively)[20], data flow diagramming techniques and the formal method Z.

6.3.1. The Dot Notation

As discussed in chapter 4 (section 4.2.2.1), we define ViewPoint representation styles in terms of objects and relations, with associated typed attributes and values. This allows us to identify partial

[20] Since CORE uses the term “viewpoint” as part of its terminology, we substitute this with the term “agent” to avoid the clash in nomenclature.

representation knowledge (notations) and partial specification knowledge (specifications) which are needed to express inter-ViewPoint rules.

We use a dot (.) to separate (from left to right) relations, objects, attributes and values. Thus, the term:

Object1.Attribute1

identifies "the value of Attribute1 of Object1". For example, if a "Process" in a data flow diagram has a "Name" attribute, then to identify the value of that attribute one would write:

Process.Name

Similarly, the term:

Relation1(Object1, Object2). Object1.Attribute1

identifies "the value of Attribute1 of Object1 in the Relation1(Object1, Object2)".

A relationship itself may also have an attribute (e.g. the "label" of a transition arrow in a state transition diagram) which may be identified as follows:

Relation1(Object1, Object2).Attribute1

This identifies "the value of Attribute1 of Relation1 between Object1 and Object2". For example, in the state transition diagram in figure 6.13, the "Button-press" transition may be identified by the expression:

Transition(State, State).Name

assuming that Transition relations between two States have a Name attribute.

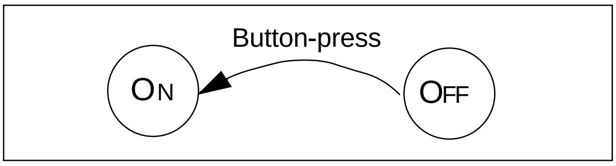


Figure 6.13: *A partially complete state-transition diagram.*

Particular values in a specification may also be identified by concatenating them to the above expressions. For example, in figure 6.13 particular "labels" in the state-transition diagram may be identified as follows:

State.Name.'ON'

This identifies the State named 'ON'. While:

Transition(OFF, ON). Name.'Button-press' identifies the Transition 'Button-Press' between the 'OFF' and 'ON' states. The next step is to include this kind of information in inter-ViewPoint rules, in order to relate partial specifications in different ViewPoints together.

6.3.2. Inter-ViewPoint Rules

An inter-ViewPoint rule describes a relationship between two ViewPoints. As our model in section 6.2 illustrated, expressing or defining inter-ViewPoint rules is a method engineering activity in which partial representation schemes are related. These representation schemes are defined in the style slots of their respective ViewPoint templates. Upon instantiation of the relevant ViewPoint templates, an inter-ViewPoint rule relates partial specifications in two ViewPoints.

6.3.2.1. General Form

Most inter-ViewPoint rules that traditional software engineering methods deploy, require some form of pattern matching to check that values of certain types of objects are related by simple binary relations (e.g., =, <, >). For example, it is frequently necessary to check that the string values of various named objects have been preserved or that integer values are within certain numerical limits. Other rules are more complex in that the relationships between the partial specifications are not simply a comparison between typed values. Instead the rules express a correspondence between different types of objects in different specifications.

In general, inter-ViewPoint rules are defined in ViewPoint *template* work plans, and therefore describe relationships between ViewPoints that have not yet been created. They are of the general form:

$$\boxed{\forall\ VP_S\ ,\ \exists\ VP_D \text{ such that } VP_S\ \Re\ VP_D}$$

where VP_S is the source ViewPoint in which the rule resides, VP_D is the destination ViewPoint with which the relationship $\Re$ holds, and $VP_S \neq VP_D$ (that is, the source and destination ViewPoints are different - otherwise the rule would be an *in*-ViewPoint rule). The source ViewPoint is universally quantified to indicate that the rule applies to every ViewPoint derived from the template in which the rule is defined. Once a ViewPoint has been instantiated from a template, this quantifier can be dropped, since the source ViewPoint will always contain the rule. Therefore, what remains is the existential quantification of the destination ViewPoint, VP_D. VP_D must somehow be "identified", that is, located from among a collection of other ViewPoints. A *ViewPoint identifier,* V_I, is used to "specify" the destination ViewPoint in an inter-ViewPoint rule (it may for example describe where that ViewPoint can be found in a local area network). Examples of "typical" ViewPoint identifier information include:

<Template, Domain>	e.g., <Data Flow Diagram, Library>
<Owner, Domain>	e.g., <Jeff, Patient>
Network address	e.g., 146.169.14.8

To express the kinds of inter-ViewPoint rule of the general form outlined above, in its appropriate ViewPoint template, we use ViewPoint identifiers defined by a tuple:

$$(t, d)$$

where t specifies the template from which the ViewPoint is (or will be) instantiated, and d is its domain (label) such that:

$$d \in \{D_p, D_a, D_s, D_d\}$$

where,

D_p denotes a *particular* (named) domain,

D_a denotes *any (arbitrary)* domain not known at template construction time,

D_s denotes the *same* domain as that of the *source* ViewPoint,

D_d denotes a *different* domain from the current (source) ViewPoint.

Moreover, since inter-ViewPoint rules, in general, relate partial specifications in different ViewPoints rather than ViewPoints as a whole, we can rewrite their general form as:

$$\boxed{\text{partial-spec-S} \ \Re\ V_I : \text{partial-spec-D}}$$

where partial-spec-S denotes a partial specification in VP_S - the source ViewPoint in which the rule resides (and which therefore does not require an identifier); and partial-spec-D denotes a partial specification in VP_D - the destination ViewPoint identified by V_I (and which therefore has domain d and is instantiated from template t).

Finally, the existential quantification in the general form of an inter-ViewPoint rule insists that a destination ViewPoint must exist, implying that if it does not, it ought to be created at some time in the future. This form of rule therefore combines an *existence* relationship ("another ViewPoint exists ...") and an *agreement* relationship ("... which is related in this way"). In some circumstances it may be necessary to express only one or other of these relationships, which we can write as:

Existence relationship	$\{\text{partial-spec-S}\} \rightarrow \exists\ VP_D : \{\text{partial-spec-D}\}$
Agreement relationship	$\forall\ VP_D\ \{\text{partial-spec-S} \ \Re\ VP_D : \text{partial-spec-D}\}$

where "→" denotes an "implication" relationship, which when applied performs a "ViewPoint trigger action" (i.e., it creates or "spawns" a new destination ViewPoint identified by VP_D). For many inter-ViewPoint rules, it is not productive to separate the two kinds of relationship, in which case the combined form is used.

6.3.2.2. Examples

We now present a number of sample rules that demonstrate the feasibility of our inter-ViewPoint rule definition approach. The examples illustrate the way in which relationships between specifications, developed using the same technique (intra-template) or different techniques (inter-template), are expressed.

We use our "dot" notation to describe partial specifications in ViewPoints, and we identify (destination) ViewPoints by the identifier VP(t, d) with t and d defined as before. Moreover, we label the different rules so that we can refer to them in our discussions[21]. These labels may also be referred to by ViewPoints' local process models, as we shall see briefly in the next section and in more detail in the next chapter.

We begin by demonstrating two kinds of existence relationship, which differ in the extent to which they make reference to the contents of the source ViewPoint. In the first kind, the mere existence of the source ViewPoint requires the existence of another related ViewPoint. For example, Z defines the following relationship: *"Every Z schema must have a textual description"*. This kind of relationship can be expressed as an inter-ViewPoint rule defined in the "Z Schema" template - that is, every ViewPoint containing a Z schema will contain this rule. Since the rule makes no reference to the actual content of the textual description associated with a Z schema, the partial specification for the destination ViewPoint can be omitted entirely. Therefore, the above rule may be written as:

R_1: $\text{self} \rightarrow \exists\ \text{VP}(\text{TD}, D_S)$

where self denotes the source ViewPoint, VP_S (containing *any* partial specification - including, possibly, an empty specification), TD is the template for "Textual Description" and D_S denotes the same domain as that of the source ViewPoint.

The second kind of existence relationship covers situations in which elements of the specification in one ViewPoint require other related ViewPoints to exist. An example from data flow techniques is: *"Every non-primitive process in a data flow diagram must have a decomposition data flow diagram associated with it"*. In this case only the partial specification of the destination ViewPoint is omitted:

[21] We have labelled the different rules consecutively for convenience. In practice, these rules do not necessarily appear in a single ViewPoint template, as they are drawn from different methods and refer to different notations.

R_2: {Process.Status.'Nonprimitive'} → ∃ VP(DFD, Process.Name)

where DFD is the template for "Data Flow Diagrams", Process is an object with attribute Status whose value is 'Nonprimitive', and Process.Name denotes the domain of the decomposition ("child") ViewPoint (i.e., D_p = Process.Name) because it is the name of the process which that ViewPoint represents (R_2 is shown schematically in figure 6.14).

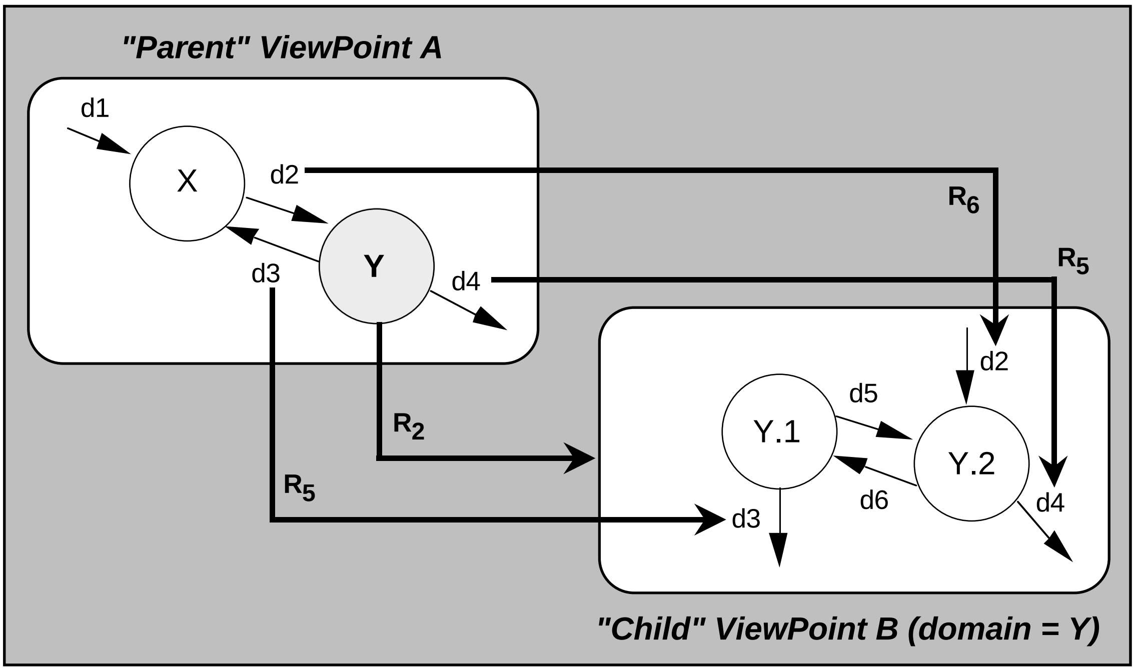


Figure 6.14: *Inter-ViewPoint relationships between data flow diagrams. Each diagram resides in the specification slot of its respective ViewPoint. Circles represent processes and arrows represent data flows. The shaded process, Y, in ViewPoint A is decomposed into the data flow diagram in ViewPoint B. The relationships between the two ViewPoints are shown as darker (and thicker) arrows and can be expressed by the labelled rules R_2, R_5 and R_6 in the text.*

Each type of rule described above can also be negated, for instance to specify that an element of a ViewPoint specification should not have another ViewPoint associated with it, or that a particular ViewPoint should be unique. For example, the rule *"A primitive process in a data flow diagram should not be decomposed"* can be written as:

R_3: {Process.Status.'Primitive'} → ¬ ∃ VP(DFD, Process.Name)

where 'Primitive' is the value of the Status attribute of a Process and D_p = Process.Name; while in CORE, the rule *"There should be only one agent hierarchy diagram"*[22] can be defined in an AH template as:

R_4: self → ¬ ∃ VP(AH, D_a)

[22] Recall that an "agent hierarchy" in CORE is simply an inverted tree that decomposes a problem or system into its information processing entities (which we call agents).

where D_a indicates that the domain of the destination ViewPoint can be anything, and AH denotes an “Agent Hierarchy” template. It is important to remember that inter-ViewPoint rules are always applied from a source ViewPoint, and that the source ViewPoint will never be checked for consistency with itself (i.e., VP_D will never be instantiated as VP_S). Local ViewPoint consistency is checked using a separate set of in-ViewPoint rules. Without this arrangement, rules like R_4 would always fail.

In general, agreement rules express relationships between the contents of two ViewPoints. An obvious example is the relationship between the flows connected to a process in a data flow diagram and the contextual (“hanging”) flows in the decomposition of that process (figure 6.14). An example of such a rule for the “parent” ViewPoint is: *“Every output from a process in a data flow diagram must appear as a contextual output in every decomposition of that process”* which can be written as:

R_5: {link(From, _).Flow.Name = VP(DFD, From.Name): link(_, To.Status.‘context’).Flow.Name}

where the underscore (“_”) is used to denote “any” value, and “link(A, B)” is a relation in the data flow notation linking process A to process B (this relation also has an attribute Flow representing the data flows between A and B). Note that there also is an implicit universal quantification of the destination ViewPoint; i.e., we are referring to all ViewPoints identified by VP(DFD, From.Name). This is in contrast with the existence rules defined previously. Hence an agreement rule does not require the related ViewPoint to exist: a separate existence relationship expresses this (e.g., R_2). It also allows for the possibility that several alternative ViewPoints exist, for example where two conflicting ViewPoints have been proposed (e.g., as alternative solutions to a problem).

Similarly, the rule *“Every input to a process in a data flow diagram must appear as a contextual input in every decomposition of that process”* can be expressed as:

R_6: {link(_, To).Flow.Name = VP(DFD, To.Name): link(From.Status.‘context’, _).Flow.Name}

In addition to expressing equality, relationships in inter-ViewPoint rules may also express exclusion. Such rules include, for example, the rule *“Process names must be unique across all data flow diagrams”* which can be written as:

R_7: ¬{Process.Name = VP(DFD, D_a): Process.Name}

Again, there is an implicit universal quantification of the destination ViewPoint identified by VP(DFD, D_a). Moreover, this rule does not exclude duplicate process names within a single data flow diagram, as the destination ViewPoint will never be instantiated to be the same as the source ViewPoint.

Even though all inter-ViewPoint rules are distributed among the different ViewPoints they relate, there are however some checks which are conceptually “global”. Such rules may require checking without knowing whether any of the ViewPoints being checked actually exist. An example of such a rule in CORE is *"There must be an agent hierarchy"*. One way to handle such rules is for the method designer to create a template for a ViewPoint which has as its specification a graph representing other ViewPoints and the relationships between them. Such a ViewPoint can then also used to "navigate" or "browse" a current collection of ViewPoints. There may in fact be any number of such ViewPoints which contain different project management information. For instance, there may be one for each template, to keep track of relationships between all ViewPoints instantiated from that template. Alternatively, there may be just one for the entire collection of ViewPoints (figure 6.15), with their relationships extracted from the individual ViewPoints themselves. The choice of management ViewPoint(s), if any, is an issue of method design.

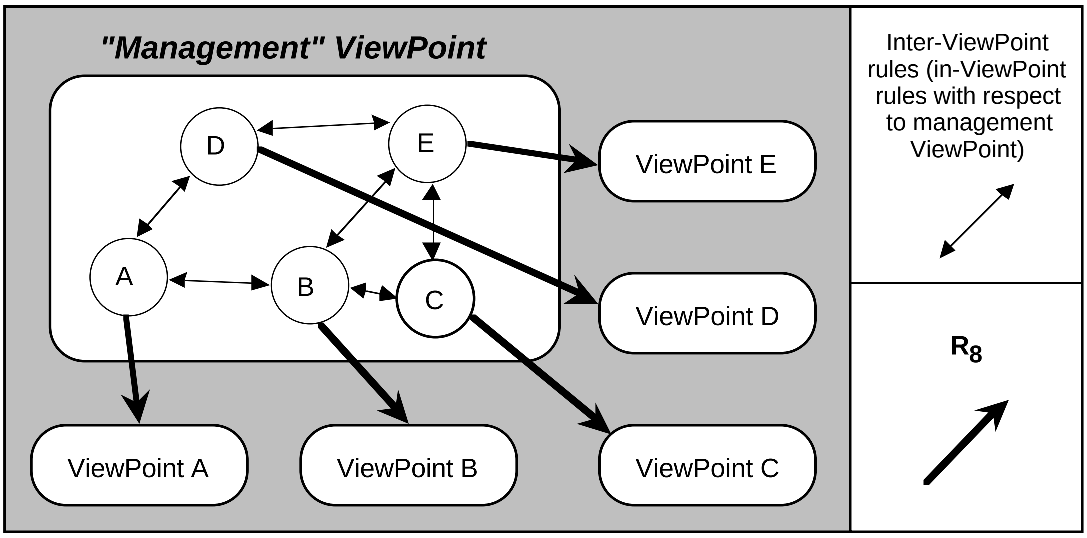


Figure 6.15: *A sample "management" ViewPoint. Such a ViewPoint's specification may contain nodes that represent other ViewPoints created in a project, and any relationships that exist between them (denoted by the double-ended arrows).*

Of course, because of the decentralised nature of the ViewPoints framework, there is no guarantee that the specification in such a "management" ViewPoint accurately represents the current set of ViewPoints. In other words, it can be inconsistent with other ViewPoints. Inter-ViewPoint rules define its relationships with other ViewPoints. In-ViewPoint rules in this type of ViewPoint define global checks over the set of ViewPoints represented.

Consider, for example, a ViewPoint that keeps track of all ViewPoints containing data flow diagrams. A simple inter-ViewPoint rule in this ViewPoint might be *"Every node in the graph represents a data flow diagram ViewPoint"*. This rule describes the kind of relationships denoted by the thick arrows on figure 6.15, and may be written as:

R_8: $\{\text{Node}\} \rightarrow \exists\ \text{VP(DFD, Node.Name)}$

Alternatively, individual ViewPoints may "update" management ViewPoints themselves. The rule that expresses this in the case of data flow diagrams is "*Every data flow diagram ViewPoint is* *represented as a node in the graph”*, which may be written in the DFD template as:

R_9: self → ∃ VP(MV, D_a): Node.Name.D_s

where MV is a “Management ViewPoint” template.

Such management ViewPoints allow us to express global rules as in-ViewPoint checks in these ViewPoints. For example, the rule *“There must be one top-level data flow diagram”* is an in-ViewPoint check to test that there is only one node (in the management ViewPoint) that has no parent.

Apart from our initial example of an inter-ViewPoint rule loosely relating a Z schema with a textual description, all our examples have been of intra-template rules; that is, relating ViewPoints instantiated from the same template. Inter-template rules may be analogously defined. For example, in CORE there is a rule which states that *“Every source in a tabular collection diagram should be a named agent in the agent hierarchy”* (figure 6.16). This rule is defined in the “tabular collection” template (TC) and may be simply written as:

R_{10}: {Source.Name = VP(AH, D_d): Agent.Name}

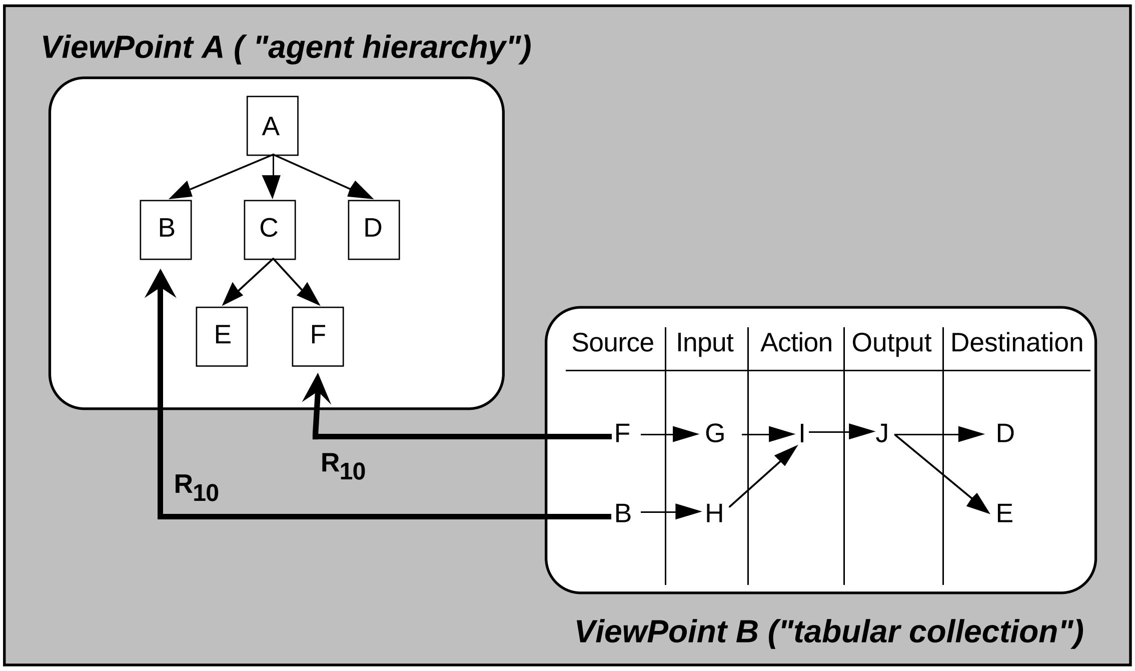

ViewPoint A ("agent hierarchy")
A
B
C
D
E
F
Source
Input
Action
Output
Destination
F
G
I
J
D
B
H
E
R10
R10
ViewPoint B ("tabular collection")

Figure 6.16: *An inter-ViewPoint rule (R_{10}) illustrating the relationship between an “agent” in an agent hierarchy and a “source” in a tabular collection diagram in CORE.*

Finally, as we outlined in section 6.2, in many cases the “converse” of a rule defined in a source ViewPoint must be defined in the destination ViewPoint (e.g., rules R_8 and R_9). This permits consistency checking from either of the two ViewPoints which the rule relates. Moreover, even though some overhead is incurred in defining basically the same rule twice, this usually results in a better understanding of the relationships between ViewPoints and therefore strengthens consistency checking between them. A particularly subtle example from CORE (illustrated schematically in figure 6.17) is the rule which asserts that *"Every output from a tabular collection diagram must be an input in another tabular collection diagram for another agent (the destination agent for the original input)"*. This rule may be written as:

R_{11}: {Connected-to(Output, Destination).Output.Name =
VP(TC, Destination.Name): Connected-to(D_S, Input).Input.Name}

where the relation "Connected-to" denotes the directed arrow between any two objects in a tabular collection diagram. The inverse of this rule asserts that *"Every input from a source in a tabular collection diagram must have been produced as an output by the tabular collection diagram of that source agent"* which can be written as:

R_{12}: {Connected-to(Source, Input).Input.Name =
VP(TC, Source.Name): Connected-to(Output, D_S).Output.Name}

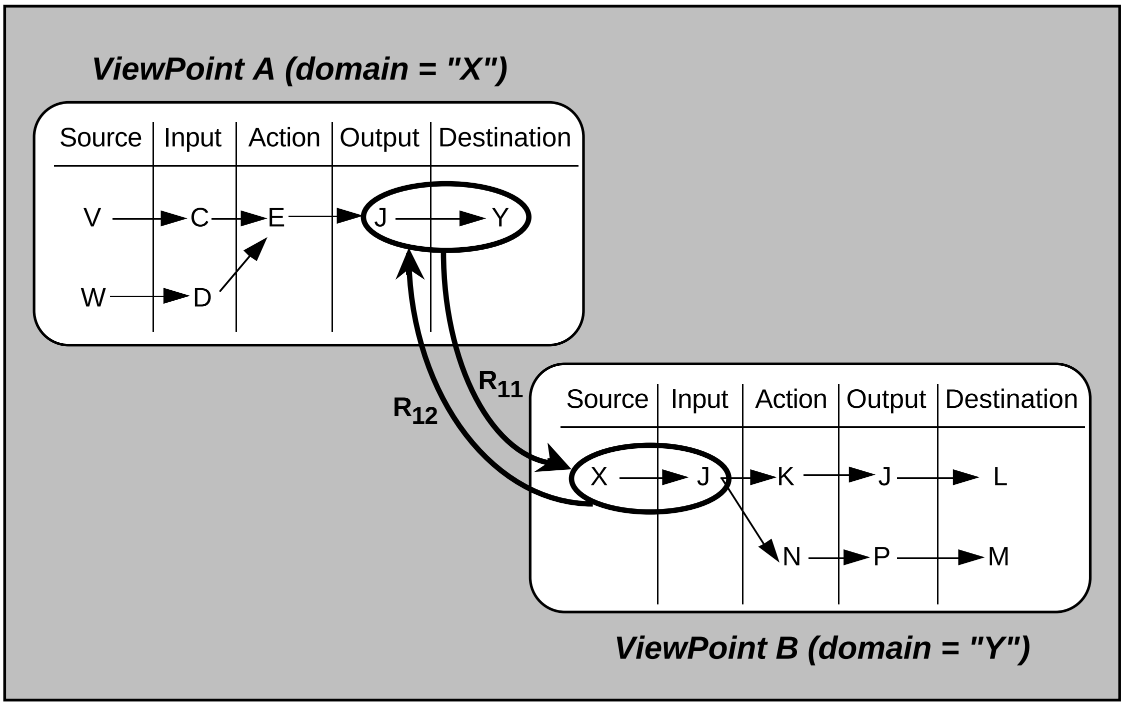


Figure 6.17: *The relationship between two tabular collection diagrams in CORE, expressed by the rule R_{11} and its converse R_{12}. Both rules express the same relationship between the two ViewPoints; however R_{11} is defined in ViewPoint A, while R_{12} is defined in ViewPoint B.*

Not every rule in CORE however has a valid inverse; e.g., every agent in an agent hierarchy *does not* necessarily have to be a named source or destination in a tabular collection diagram.

6.3.3. Summary

In this section we have addressed a critical step in the process of method integration, namely, *expressing* the relationships between multiple ViewPoints. In particular, we proposed a sample notation for representing partial specifications, and proposed the use of pairwise inter-ViewPoint rules to describe the relationships between these partial specifications. These rules are defined in individual ViewPoint templates, and are therefore distributed among the various ViewPoints as the development proceeds. Moreover, inter-ViewPoint rules are defined by method engineers at the *template* level, and are therefore a vehicle for method integration (since methods in the ViewPoints framework are simply structured collections of ViewPoint templates). In fact, the structure of methods, and subsequent specifications constructed using these methods, is also determined by inter-ViewPoint rules.

We concluded this section by presenting a collection of examples that illustrate a variety of inter-ViewPoint rules, which in turn describe the relationships used to combine different "development techniques" (templates) together. In particular, rules for describing relationships between different data flow diagrams (intra-template) and the different stages of CORE (inter-template) were presented.

However, expressing the integration (consistency) relationships between ViewPoints addresses only part of the problem. Defining inter-ViewPoint rules expresses the "intentions" of method integration. In order to "implement" such integration intentions - that is, in order to produce integrated software system specifications - procedures for "consistency checking" are also needed.

6.4. Consistency Checking

Two ViewPoints are consistent if all the inter-ViewPoint rules defined between them have been found to hold. Consistency in the ViewPoints framework is a relative notion: "total" consistency is the satisfaction *all* the rules, while "partial" consistency is the satisfaction of *some* of the rules that have been defined by the method designer. The process of systematically ascertaining whether or not one or more inter-ViewPoint rules hold, we call *consistency checking*[23].

In our model of ViewPoints integration (figure 6.5), inter-ViewPoint rule *invocation* is the step during which the process of consistency checking is started. Inter-ViewPoint rule *application* is the process of actually determining whether or not the relationship defined by an inter-ViewPoint rule holds.

[23] Of course, we can also perform *in*-ViewPoint consistency checking (which is often necessary *before* inter-ViewPoint checking is done). However, since we are interested in method (and hence ViewPoints) integration, we concentrate on inter-ViewPoint consistency checking only.

While a desirable outcome of inter-ViewPoint consistency checking is to confirm that an inter-ViewPoint rule holds, this does not always happen. In other words, inconsistencies are sometimes detected. Activities such as conflict resolution may be required to remove such inconsistencies. In general however, we prefer to tolerate inconsistency and therefore propose an *inconsistency handling* step, which does not necessarily remove inconsistencies between ViewPoints, but rather permits, and preferably provides "corrective" or "constructive" guidance for, the continued development of either or both ViewPoints in the presence of such inconsistencies.

In this section we address the process of consistency checking; that is, the process of inter-ViewPoint rule application. To this end, we propose a sample protocol and mechanism for interaction between the ViewPoints. *When* such consistency checking is initiated, and *how* one acts in the presence of detected inconsistencies, we regard as process modelling issues and discuss in more detail in chapter 7.

6.4.1. When to Check: Inter-ViewPoint Rule Invocation

An off-putting characteristic of first generation CASE tools was their immediate flagging of inconsistencies as soon as they occurred. In effect, consistency checks were continuously being invoked, constraining developers in their use of these tools. As a result, the users of such tools often resorted to developing their specifications off-line, only using the CASE tools to record the final outcome of the development process.

Second generation CASE tools took a more pragmatic approach by allowing developers to switch consistency checking on and off whenever it suited them to do so (and to specify the scope of the checks; e.g., local or project-wide). This introduced more freedom during the "creative" stage of development (that is, when developers were still "sketching" their ideas), but of course was entirely reliant on these individual developers' competence and judgement as to when consistency checking ought to be invoked.

The approach we favour, and adopt, in the ViewPoints framework is what we call a *process-driven* or *process-oriented* approach. Using this approach, the invocation of the appropriate consistency checks is guided by a process model which suggests appropriate times for invoking these checks. Since process modelling in our framework is a means of *guiding,* as opposed to automating, development, check invocation advice in this context can be ignored by developers, if they feel that it is constraining their development activities or creativity.

Consistency checking in a multi-perspective development environment, exemplified by the ViewPoints framework, is particularly problematic, since more often than not, ViewPoints will be inconsistent with each other. Therefore, the invocation of inter-ViewPoint rules, which are used to check consistency between ViewPoints, is especially critical. In this context, inter-ViewPoint rule invocation comprises of two steps:

- *Deciding when consistency checking should take place.* This is the critical process modelling step (chapter 7) which determines the timeliness of consistency checking. One heuristic that has proven to be particularly appropriate, is to invoke inter-ViewPoint consistency checking only after, at least some, in-ViewPoint consistency checking has been performed. Checking the consistency between two ViewPoints, both of which are internally inconsistent (or worse, whose internal state is completely unknown), makes the results of inter-ViewPoint consistency checking unclear and usually unusable.
- *Locating, or if necessary creating, the ViewPoints that will be involved in the consistency checking process, identified by an inter-ViewPoint rule.* Inter-ViewPoint rules are invoked from the ViewPoints in which they reside (source ViewPoints), and express relationships that should exist between these and other ("destination") ViewPoints. When such rules are invoked however, the relevant destination ViewPoints may not necessarily have been created. Inter-ViewPoint rule invocation attempts to ensure that both interacting ViewPoints specified by an inter-ViewPoint rule have been identified - if necessary by performing ViewPoint trigger actions to create one or more destination ViewPoints.

Of course, we also allow for the fact that an inter-ViewPoint rule invocation step may fail (for example, if we decide *not* to create a destination ViewPoint deemed necessary for the application the inter-ViewPoint rule). However, once both ViewPoints identified by an inter-ViewPoint rule *have* been located, the rule may then be *applied* in order to check the consistency of these two ViewPoints according to the relationship expressed by this rule.

6.4.2. How to Check: Inter-ViewPoint Rule Application

Consistency checking is the application of one or more inter-ViewPoint rules of the general form $VP_S \Re VP_D$. Inter-ViewPoint rule application attempts to answer the question: does the relationship $\Re$ hold between VP_S and VP_D? Alternatively, the same rule may be applied in an operational way, in which case it is equivalent to applying a transformation of information between VP_S and VP_D, in order to make the relationship $\Re$ hold between them. Such rule application is analogous to the application of logical rules in, say, Prolog. "Querying" (applying) a Prolog rule either returns a success/fail response or generates the solutions to make the rule succeed (hold).

We call the two modes of application of inter-ViewPoint rules described above, "check mode" and "transfer mode", respectively. In general, one would expect inter-ViewPoint rule application to start in check mode, and enter into transfer mode when the application of the rule fails (that is, when the relationship expressed by the rule is found not to hold). In some cases however, a developer may want to transfer information from one ViewPoint to another, in order to maintain the relationship that has been defined between those two ViewPoints (that is, enforce it to hold). In such cases, a transfer mode of inter-ViewPoint rule application is used, in which a function of the form $f(\Re, VP_S, VP_D)$ is performed in order to achieve consistent transfer and transformation of information between VP_S and VP_D.

Consistency checking between two ViewPoints requires the interacting ViewPoints to engage in a communication *protocol* in which information in either or both ViewPoints is exchanged and compared. In a distributed setting, this includes the physical transfer of information from one ViewPoint to another, and typically, the transformation of this information into a form understood by the other ViewPoint. The *mechanism* for such interaction therefore also needs to be specified. Bearing in mind that the objective of the interaction is to investigate the relationship $\Re$ between VP_S and VP_D, we outline in table 6.1 an inter-ViewPoint communication protocol for collaboratively interacting ViewPoints. We adopt the terminology used by [Werner 1992] in which *cooperation/collaboration* requires *communication*. Note that ultimately, our goal is to achieve consistency between the interacting ViewPoints; that is, our goal is for the relationship $\Re$ to hold between VP_S and VP_D. The process of achieving this begins during inter-ViewPoint rule invocation and continues after inter-ViewPoint rule application.

We make the following assumptions in relation to the protocol in table 6.1:

- There is a reliable communication network to support distributed inter-ViewPoint communication.
- There is a reliable (preferably distributed) name service [Comer & Peterson 1989] to help identify and locate ViewPoints in the network.
- There is a reliable concurrency control mechanism, which prevents ViewPoints being updated concurrently. Such a mechanism ensures that only two ViewPoints communicate at any one time. A fine-grain concurrency control mechanism such as that suggested by [Barghouti 1992] may be used. Alternatively, inter-ViewPoint communication may be treated as a “long transaction” [Hurson, Pakzad & Cheng 1993] which takes place over a long period of time, and during which other ViewPoint work plan actions are performed.
- In general, ViewPoints interact on an equal (“peer-to-peer”) footing - more in line with collaborative work architectures as opposed to strictly client-server. Thus, ViewPoints interact more to exchange information rather than to request (remote) computational services. The above protocol therefore follows more closely a “data shipping” approach in which partial specification information from the destination ViewPoint is transferred (shipped) to the source ViewPoint, and then checked locally. Its only explicit goal is the requirement to have the inter-ViewPoint rule, $VP_S \Re VP_D$, hold. This may appear to oversimplify a more realistic scenario involving many goals, but we submit that these may be described separately in ViewPoint work plan process models and related via one or more inter-ViewPoint rules.

STEP	COMMENT
Goal: $\boldsymbol{VP_S \Re VP_D}$	Achieving the goal $VP_S \Re VP_D$ or enacting the rule $VP_S \Re VP_D$ means performing an inter-ViewPoint action/check that engages VP_S and VP_D in the protocol outlined below.
At Source ViewPoint, $\boldsymbol{VP_S}$	
Invoke (locally) inter-ViewPoint rule: $VP_S \Re VP_D$	Guided in doing so by local process model of VP_S.
Send request for destination ViewPoint, VP_D, identified in the by the inter-ViewPoint rule	Locate VP_D.
Wait for identification acknowledgement from VP_D	Or message indicating failure to find VP_D.
Send request for (partial) specification information from VP_D (if specified by the rule)	Or perform ViewPoint Trigger Action(s) if identification step above failed.
Wait for response from VP_D	Partial specification or error message.
Apply $VP_S \Re VP_D$	“Check” rule using whatever local “engine” is provided by VP_S; e.g., pattern match.
Send (post) rule application result to VP_D	Fail, error, conflict, inconsistency, success, etc ...
Invoke inconsistency handling, if needed	E.g., conflict resolution.
At Destination ViewPoint, $\boldsymbol{VP_D}$	
Wait for identification request from VP_S	
Send identifier (acknowledgement) to VP_S	E.g. network address of VP_D.
Wait for (specification) information request from VP_S	
Access (and transform) requested (specification) information and send to VP_S	Extract requested partial specification from local specification slot.
Receive notification from VP_S of rule application result.	Receive notification that rule application has succeeded or failed (including failure message)

Table 6.1: *Sample protocol for inter-ViewPoint communication.*

While the protocol described outlines general interaction activities between ViewPoints, it does not address a number of distributed systems issues that are still the subject of much research but which are beyond the scope of this thesis. These include:

- The granularity of the information transferred between ViewPoints. It is problematic to transfer large partial specifications from VP_D to VP_S for checking. Our approach attempts to alleviate this problem by prescribing simple inter-ViewPoint rules that relate only small partial specifications between ViewPoints.

- Communication delays during transfer of information between physically distributed ViewPoints. These can be minimised by choosing to transfer the minimum of information to and from different ViewPoints. Our protocol attempts to compromise between the two approaches[24] of sending or receiving information to and from VP_D. On the one hand, VP_D (*c.f.* server) is responsible for supplying (finding, and transforming if necessary) partial specification information. While VP_S (*c.f.* client) is responsible for applying (checking) the rule locally.

To summarise, interacting ViewPoints need to exchange information. The inter-ViewPoint communication protocol outlined in table 6.1 specifies the nature of this exchange. VP_S needs to obtain a partial specification from VP_D, and if necessary transform it into a form it can understand and manipulate (so that pattern matching, for example, can be performed). If the relationship $\Re$ fails to hold, then VP_D may need to be made aware of this failure (i.e., another transfer), and some form of inconsistency handling needs to be performed. In a typical software engineering setting, time constraints on such transfers are insignificant, but if the ViewPoints are deployed in a real-time distributed environment (following a client-server model for example), then traditional problems such as communication load overhead or a high rate of change of fetched server information may become much more significant, and need to be considered.

Clearly, the basic infrastructure of ViewPoints needs to be extended to handle the various transfers and transformations that will occur during typical inter-ViewPoint communication. One such modification is the addition of ViewPoint *interfaces* to provide information hiding and other transformation services. These interfaces may also provide “mailboxes” to which information from other ViewPoints is “posted” rather than forcibly transferred into destination ViewPoint specifications. It is then left to the discretion of individual ViewPoint owners to incorporate information and/or guidance residing in their ViewPoint mailboxes into their local ViewPoint specifications. Table 6.2 outlines the conceptual structure and role of a ViewPoint interface (first introduced in chapter 4, section 4.2.5), and figure 6.18 provides a example scenario of how such an interface may be used.

[24] Recall that these two approaches are the “data shipping” and “function shipping” protocols described in chapter 3, section 3.4.3.

Components of a ViewPoint interface	Role
ViewPoint Identifier	Contains sufficient information to enable the unique identification of a ViewPoint in a configuration of other ViewPoints. Typically, a ViewPoint identifier may include information about the template from which the ViewPoint is instantiated, the domain which it describes, its owner and/or its network address.
ViewPoint Mailbox	Contains information "posted" to it from other ViewPoints engaged in inter-ViewPoint communication. This information may include notes or messages in natural language or more structured partial specification information. Such information may, for example, be used by the ViewPoint's owner during inconsistency handling or conflict resolution.
Merge Methods	Are the functions, procedures and heuristics for comparing, merging and resolving conflicts between partial specifications. Typically, merge methods analyse information in a ViewPoint mailbox, with a view to incorporating this information into the ViewPoint's local development process and specification. Again, merge methods may be formal merge procedures (such as those discussed in chapter 3, section 3.4.3) or informal "rules-of-thumb".
Access Methods (Interface Definition)	Are the procedures for extracting and returning partial specification information from a ViewPoint's specification slot. Typically, this information will have been requested by a source ViewPoint engaged in some form of interaction with this ViewPoint. If different ViewPoints use different representation schemes, then the role of access methods is also to translate or transform local specification information into a form understood by other communicating ViewPoints (e.g., a "standard" interchange form).
Working Memory	Can be a conceptual or physical container for holding an inter-ViewPoint rule and the associated partial specification information it requires during inter-ViewPoint rule application. In analogous computer hardware terminology, it is a volatile short-term memory, compared to the non-volatile longer-term memory represented by a ViewPoint's mailbox.

Table 6.2: *Components of a ViewPoint interface.*

The ViewPoint interaction scenario shown in figure 6.18 describes the following:

- The source ViewPoint's work plan places the appropriate inter-ViewPoint rule ($VP_S \; \Re \; VP_D$: ps_D) in that ViewPoint's working memory;
- The source ViewPoint, VP_S, locates/identifies the destination ViewPoint, VP_D (from the latter's ViewPoint identifier), and partial specification information in it (ps_D) is transferred back to VP_S (possibly after some transformation by the access methods of VP_D);
- The relationship ($\Re$) expressed by the inter-ViewPoint rule is checked in the working memory of VP_S, and the result is posted to the mailbox of VP_D *and* recorded locally in the work record of VP_S;
- VP_D can then use its merge methods to incorporate (if necessary) its mailbox information into its specification.

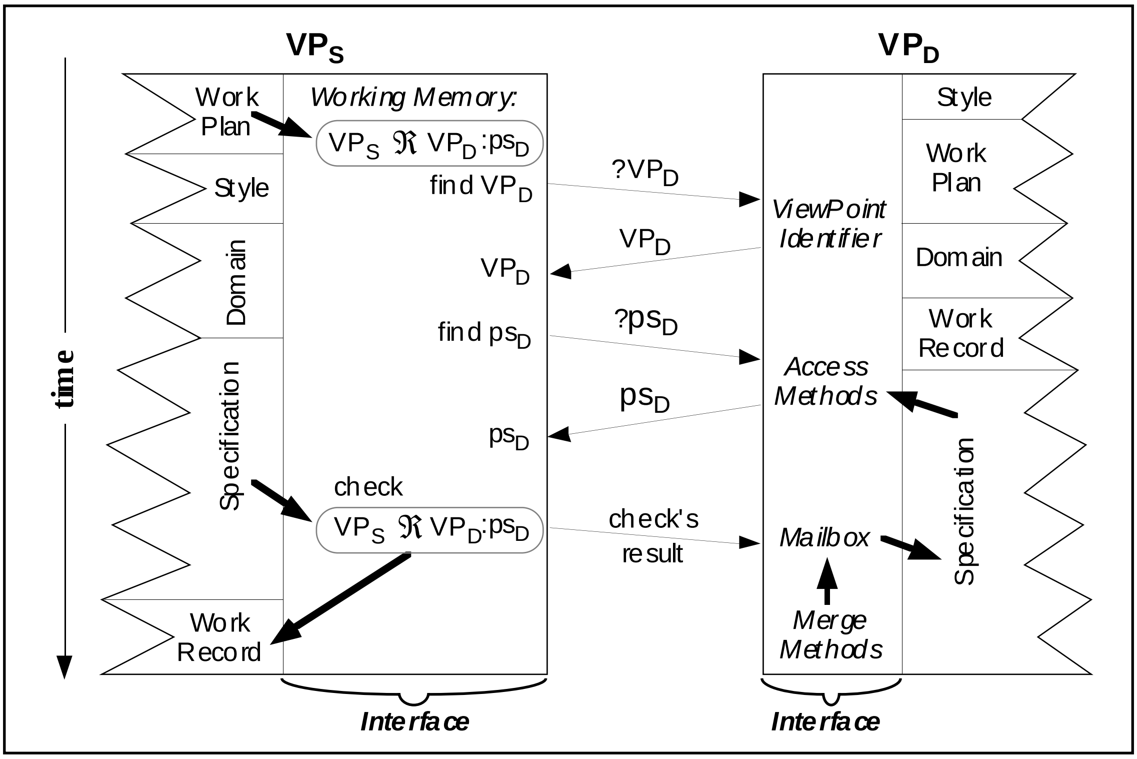


Figure 6.18: *The role of ViewPoint interfaces in ViewPoint interaction.*

6.4.3. And When a Check Fails: Inconsistency Handling

Not only can consistency checking reveal inconsistencies between ViewPoints, but the process itself may fail before the relationships expressed by inter-ViewPoint rules themselves are checked. The scenario described in the previous section for example, assumes that both “find VP_D” and “find ps_D” succeed. In fact this is not always the case: the appropriate destination ViewPoint may not have been created, and even if it has, it may not contain a partial specification against which the source ViewPoint can check its local specification. In the presence of such failure, the entire interaction may be aborted. Alternatively, a variety of *inconsistency handling actions* may be available to the ViewPoint developer.

For example, failure to locate an appropriate destination ViewPoint may be resolved by performing a ViewPoint trigger action to create such a ViewPoint. Alternatively, further development actions in the source ViewPoint may reveal that no such destination ViewPoint need be created. Thus, *immediate* resolution of this kind of inconsistency is *not* necessary. Similarly, if no appropriate partial specification is available to check with the specification of the source ViewPoint, then a transfer of partial specification information from the source to the destination ViewPoint may resolve this inconsistency. Again however, further development of either or both interacting ViewPoints may no longer necessitate such a transfer.

The role of a ViewPoint’s local process model in such cases is to advise and guide the ViewPoint developer in handling inconsistencies. The process model must be at as fine a granularity as possible, to be able to recommend specific actions that facilitate the handling of inconsistencies. Chapter 7, addresses the role of ViewPoint process models in handling inconsistencies that are identified by the inter-ViewPoint rule application process.

Finally, assuming that neither of the above inconsistencies arise, the actual application of the check may reveal other inconsistencies between ViewPoint specifications, such as name clashes, name mismatches and other logical inconsistencies. Again, a process modelling framework within which these inconsistencies may be handled is needed. Chapter 7 proposes such a framework.

6.4.4. Goal: Method Integration

In examining the detail of ViewPoint interaction in the previous sections, it is possible to lose track of our declared goal, that of achieving method integration in a multi-perspective development setting. Let us re-examine how our account so far addresses this objective.

We have defined method integration in the ViewPoints framework to have a very specific meaning. An integrated method is a combination of ViewPoint templates whose relationships have been explicitly expressed. When such a method is deployed in a software development exercise, the resultant ViewPoints that are created must be checked against each other for consistency. Consistency checking in this setting is the instantiation of the relationships that were defined during a method engineering phase, and successful integration in this setting is denoted by the successful application of these checks. A check succeeds, if the relationship that the check tests is found to hold.

Defining the relationships between templates is therefore a declaration of intentions: these relationships (described as inter-ViewPoint rules) express the integration links that should exist between ViewPoints instantiated from these templates. When actual ViewPoints are created, these relationships are instantiated and checked, and if they are found to hold then we can say that method integration has been successfully “implemented”, because valid relationships now hold between ViewPoints.

The notion of consistency is therefore central to our integration objectives. Integration *is* achieving consistency. Moreover, in the same way as we support “partial” consistency, we also support “partial” integration. “Partial integration” is the satisfaction of *some* inter-ViewPoint rules, as opposed to “total integration” which denotes the satisfaction of *all* the inter-ViewPoint rules that have been defined between ViewPoint templates that constitute a method. Furthermore, since inconsistencies are tolerated (by allowing partial consistency) during development, inconsistency handling may therefore be used to achieve *incremental integration* between ViewPoints.

Figure 6.19 summarises, semi-formally, the role of a consistency check in achieving integration in the ViewPoints framework (the inconsistency handling rule shown is discussed in more detail in the next chapter). Integration in the framework is further illustrated by our model of ViewPoints integration which is centred around inter-ViewPoint rules - their definition, invocation and application. In this model, inter-ViewPoint checks are the vehicle for achieving this integration. From a method user's point of view, ViewPoint integration is synonymous with integrated specification and software (deliverables). In fact, the litmus test for effective method integration is the degree to which the software system developed by the integrated method, is itself integrated.

Inter-ViewPoint checking may be regarded as the execution of a tuple <R, I>,

where, R = Inter-ViewPoint rule, $VP_S \Re VP_D$,
and, I = Inconsistency handling rule, $\perp \Rightarrow A$,
where, $\perp$ = inconsistency,
and, A = Action.

In this way, for every inter-ViewPoint rule, R, there is an associated rule, I, in the tuple <R, I>, which specifies how to handle inconsistencies resulting from the failure of the rule.

Figure 6.19: *Inter-ViewPoint checks.*

Chapter 9 presents a number of examples that demonstrate method integration in this setting.

6.5. Chapter Summary

This chapter has presented an approach to method integration in the ViewPoints framework. The approach is based on managing consistency between multiple ViewPoints, according to appropriately defined inter-ViewPoint relationships and their controlled (process-oriented) invocation and application. These relationships are defined as pairwise inter-ViewPoint rules during method design, which are then instantiated and checked during method use. Integration in this context is achieving consistency between ViewPoints according to these rules.

The chapter presented a notation for expressing inter-ViewPoint rules, and outlined a protocol for controlling interaction between ViewPoints during consistency checking. A varied selection of examples demonstrated the feasibility of our approach to managing consistency between ViewPoints in this way.

Integrating methods using pairwise inter-ViewPoint rules impacts upon the way in which software development proceeds in this setting. The "radical" decentralisation of software development knowledge, provided by the ViewPoints framework, raises the issue of how to model the software development process in such a distributed environment. Furthermore, equating integration with inter-ViewPoint consistency checking suggests the need to handle or manage inconsistencies between multiple ViewPoints when they arise. These two issues are addressed in the next chapter.

Chapter 7 Process Modelling and Inconsistency Handling

This chapter addresses two separate but related issues, namely, the use of process models to provide method-based development guidance *and* inconsistency handling in multi-perspective specifications. The particular relationship examined is that between process modelling and inconsistency handling, where process models recommend what actions a method user may take in the presence of inconsistencies. The chapter presents a sub-framework for inconsistency handling within the ViewPoints framework, and explores the use of local ViewPoint process models to provide method guidance during method-based software development. These process models are used as a means of partially ordering ViewPoint development (work plan) actions, which subsume inconsistency handling actions and the consistency checks described in chapter 6. Process modelling in this context is therefore used to guide local ViewPoint development and coordinate ViewPoint interaction. As such it is an important step towards method integration.

We begin the chapter by exploring the scope of ViewPoint-oriented development activities, and then address the role of process modelling in this setting. The notion of fine-grain process modelling in particular is defined, and examined through a series of illustrative examples. Modelling inconsistency handling in a ViewPoint-oriented development process is then addressed, and an action-based temporal logic framework is used to illustrate how such activities may be modelled.

7.1. ViewPoint-Oriented Software Development

As discussed in chapter 2, the development of large complex systems typically involves dealing with at least four different kinds of knowledge:

- domain knowledge - “the world”
- representation knowledge - “the language”
- development process knowledge - “the strategy”
- specification knowledge - “the product”

These have traditionally been identified and partitioned independently of each other, leading to possible mismatches between the partitions (figure 7.1a). This has made the task of identifying and expressing the relationships and inter-dependencies between partitions of different kinds of knowledge much more difficult. In the ViewPoints framework described in this thesis, we have proposed that these different kinds of knowledge be partitioned along the same lines, and then grouped together into objects called ViewPoints, whose boundaries are defined by those lines (figure 7.1b).

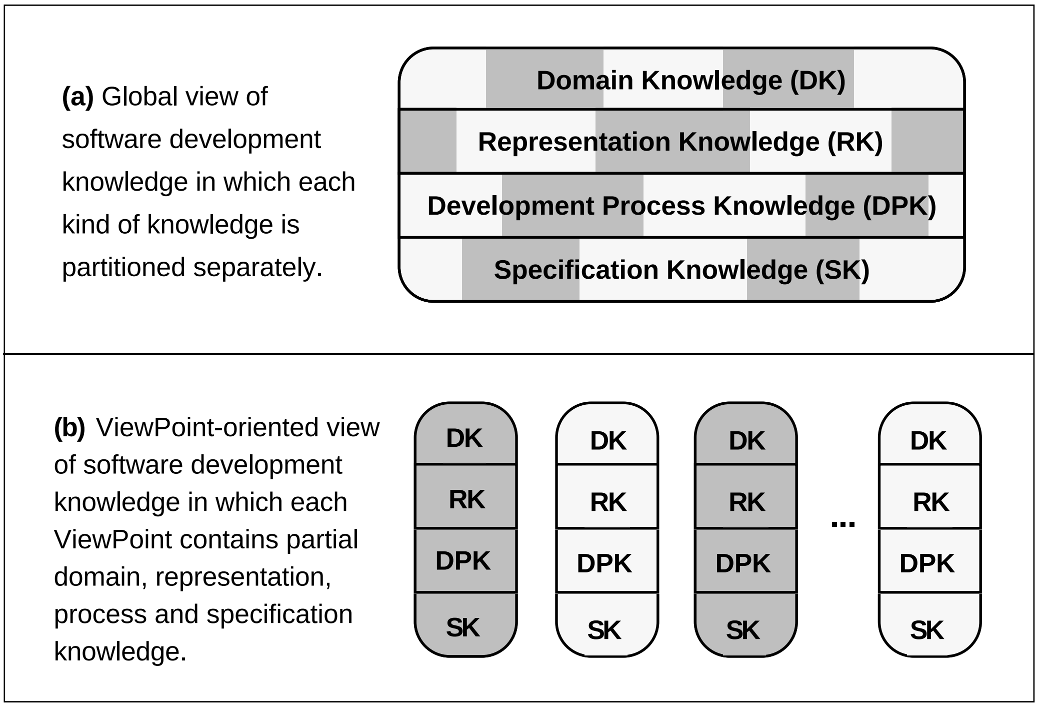

(a) Global view of software development knowledge in which each kind of knowledge is partitioned separately.
Domain Knowledge (DK)
Representation Knowledge (RK)
Development Process Knowledge (DPK)
Specification Knowledge (SK)
(b) ViewPoint-oriented view of software development knowledge in which each ViewPoint contains partial domain, representation, process and specification knowledge.
DK
RK
DPK
SK
...

Figure 7.1: *Traditional versus multi-perspective software development knowledge.*

Figure 7.1b reflects the nature of multi-perspective software development of complex systems involving many development participants who hold different views of the world and the software system they wish to construct. These development participants describe and elaborate their views using different representation schemes and follow different development strategies. Each participant has his or her own agenda of activities and goals and is only occasionally, if ever, concerned with the overall system development goals and objectives. In fact, even those development participants concerned with some global properties of a software system, do not need to be concerned with *all* properties of that system.

Unlike traditional software development, it is not always practical to construct a global model of a multi-perspective development process. ViewPoint-oriented software development is either local to individual ViewPoints, or involves the controlled interaction of ViewPoints engaged in some form consistency checking or cooperation. The role of a method engineer designing a method in this setting is therefore to define local ViewPoint development processes in individual ViewPoint templates, and then to specify the coordination between these local processes in the different

ViewPoints. Such coordination is achieved by controlling the invocation and application of inter-ViewPoint consistency checks, and modelling inconsistency handling between ViewPoints in this setting.

Clearly then, method engineering and integration in the ViewPoints framework is largely a process modelling exercise. This is not to say that other kinds of software engineering knowledge are not important, but the way in which such knowledge is deployed (i.e., the process) is crucial to successful multi-perspective software development. Therefore, we now discuss in more detail the role of process modelling in the ViewPoints framework.

7.2. Process Modelling

In chapter 3, we defined process modelling as the construction of abstract descriptions of the activities by which software is developed. In the area of software development environments, the focus is on models that are enactable; that is, executable, interpretable or amenable to automated reasoning. Modelling the software development process is a means of understanding the ways in which complex software systems are designed, constructed, maintained and improved.

Software process modelling is a complex activity that can span the entire software development life cycle, from requirements analysis and specification to system implementation, evolution and maintenance. From an organisational point of view, it is desirable to model the overall development process including the coordination and interaction of a large number of development participants. From the individual developer's point of view, while coordination and interaction are still important, the focus is on modelling development activities that fall within the domain of concern or responsibility of that individual developer.

7.2.1. Guidance

We believe that the primary role of a software process model is to provide a method user with method guidance. That is, a process model should be used as a vehicle for answering questions such as "what should I do next?", or more importantly, "how do I get out of this mess I'm now in?".

Clearly then, the role of the software engineering *method* in this context is paramount. Effective method guidance can only be provided if the method being used has been well-defined, and is subsequently used correctly. What does this mean in practical terms? Well, it does *not* mean that a method should prescribe all possible actions that a method user may take. Rather, it should support flexible development strategies, and provide guidelines and heuristics for effective development.

To achieve this, the process prescribed by a method should be sufficiently fine-grain to provide more than simply a recommendation to invoke some external tool. It should be able to advise a method use, about what, for example, he should do next in terms of very concrete guidelines about the actual actions he ought to perform. This requires knowledge about the domain in which the method is used, and knowledge about the representation scheme or language employed by the method user. Computer-aided learning or instruction systems attempt to deploy these principles [Tomek 1992].

7.2.2. Coordination

Another important role of process modelling is to coordinate or synchronise development activities. This includes the coordination of different developers working concurrently on developing a single software system, or the coordination of the activities of an individual developer with a local agenda of actions to perform. Again, the goal here is not to tightly couple developers, or to prescribe a rigid agenda for any single developer, but to provide a guiding framework within which multiple developers can make progress.

The decentralised architecture of the ViewPoints framework, including the explicit separation of the different kinds of software development knowledge, facilitates the provision of both the method guidance and coordination we require. This is achieved by using the framework to construct fine-grain process models.

7.3. Fine-Grain Process Modelling

So what do we mean by fine-grain process modelling, and how is it achieved in the ViewPoints framework?

In this section, we discuss two levels of process granularity, and examine their impact on ViewPoint-oriented software development. In particular, we present a number of examples of fine-grain process modelling, illustrating their benefits in the ViewPoints framework. The effects of fine-grain process modelling on inconsistency handling are also examined.

7.3.1. Developer Level Granularity

Traditionally, software process modelling has focused on global development processes, such as organisational or business processes. These have been useful to understand and improve organisational practices, but have not usually benefited individual developers engaged in the constituent activities of such processes. For example, attempts to specify software development processes programatically (“process programming”) have not in general been successful, largely because they have been difficult to evolve dynamically as the processes they represent evolve and change.

The ViewPoints framework promotes the division of software development knowledge into many smaller units as the first step towards achieving a finer level of granularity of software process modelling. This decomposition into small units of knowledge results in many “smaller” process models each of which is typically associated with a single developer, representation or both. Such models may represent processes that deal with individual activities or participants in the development life cycle. Fine-grain process modelling at this level of granularity is modelling at the *developer level.* This is in contrast with the more coarse-grain modelling that is concerned with more managerial and organisational activities such as the synchronisation of tool invocations.

Modelling multiple developer processes in this way is an inherently simpler exercise since the complexity of the global process has been reduced by separation of concerns through decomposition. Moreover, traditional process modelling activities such as the synchronisation of tool invocations can still be achieved by using inter-ViewPoint rule invocation as a means of invoking tools or coordinating the activities of one ViewPoint (developer) with another.

“Process integration” in this setting is then more than the usual (but non-trivial) task of producing a single, coherent process model for the overall development life cycle. It is also concerned with “gluing” together many individual process models that must interact and cooperate in a coordinated manner in order to achieve the overall objectives of the development. Such coordination requires more than just synchronisation and concurrency control or communication between agents in a cooperative setting which are challenging enough, but also the reconciliation of fundamentally different development strategies encapsulated in different process models. This latter reconciliation may not in fact be entirely necessary in the ViewPoints framework, since different individual process models should be able to coexist, and only those areas of overlap need to be reconciled.

Figure 7.2 schematically illustrates the nature of developer level fine-grain process modelling. A ViewPoint-oriented development process is represented by a configuration of ViewPoints each with their own local process model. These models are integrated via inter-ViewPoint rules that are used to control, coordinate and synchronise interaction between ViewPoints. While a global process model will exist (and may or may not be specified or documented), fine-grain process modelling at the developer level assumes that the different ViewPoints represent “slices” of, or “views” on, that global process. Again, such views may be totally, partially or non-overlapping.

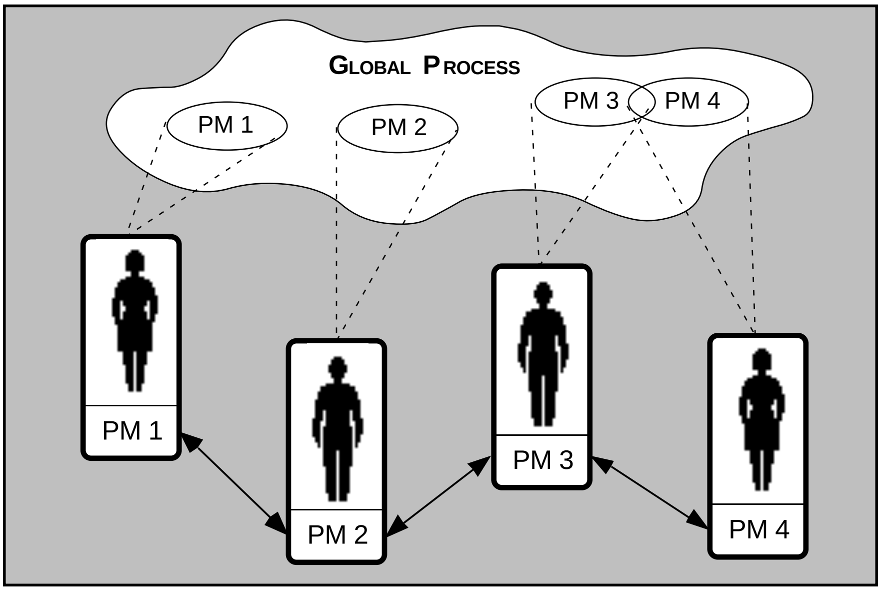


***Figure 7.2:** Developer level fine-grain process modelling. Arrows denote inter-ViewPoint rules, and "PM" denotes a "process model" that is a subset of an overall global process. ViewPoints in the diagram are denoted by boxes with thicker borders.*

7.3.2. Representation Level Granularity

Fine-grain process modelling at the developer level, facilitates modelling at an even finer level of granularity, namely, *representation level* process modelling. Here, modelling a development process is at the level of actions or activities that relate to, or manipulate elements of, a representation scheme. Representation level fine-grain process modelling is therefore concerned with providing links between development process knowledge and representation knowledge which may then be used to produce a specification for a particular problem domain (this is informally denoted by the phrase "process meets representation" in figure 7.3).

A process model at the representation level is useful to the individual developer because it may be used to provide development *guidance* expressed in a language the developer understands best - the language he or she is using! Thus, guidance can take the form of a recommendation on what to do next in order to advance the development process, or advice on handling existing specification inconsistencies. This is in contrast with processes that are "unaware" of the representation schemes they manipulate and therefore treat them as coarse-grain "vanilla" objects with no internal structure or semantics.

ViewPoint process models that specify when and under what circumstances development actions or inter-ViewPoint rule invocations ought to be performed, may be regarded as representation level fine-grain process models. Such models may be used to coordinate ViewPoint interaction during consistency checking and inconsistency handling for example. Since ViewPoint process models are also at the developer level, they are simpler and consequently easier to coordinate and integrate.

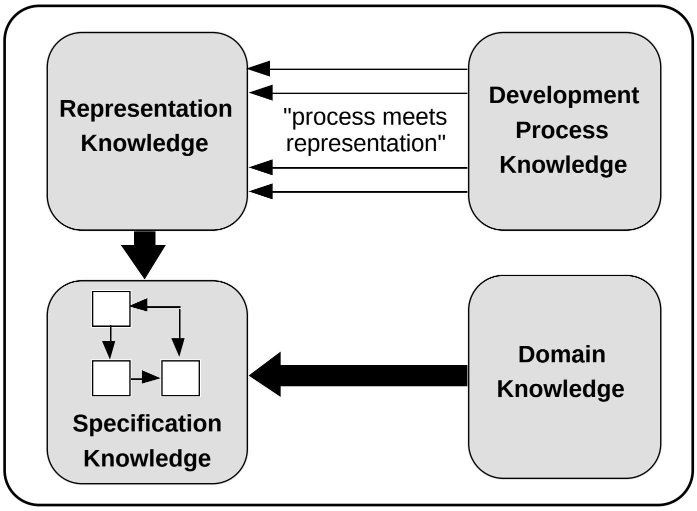


***Figure 7.3:** Representation level fine-grain process modelling. This provides a means of describing the relationships between development process knowledge and representation knowledge within a single ViewPoint. This, together with any domain knowledge may be used to construct a specification.*

Developer and representation level process modelling are complementary activities. Creating developer level process models results in simpler representation schemes and process models, which are easier to relate, which in turn makes representation level modelling easier. Conversely, representation level process models are generally easier to integrate, since the elements of overlap are mostly syntactic and are thus easier to transform, translate and check.

7.3.3. Examples

We now present a number of, mostly generic, examples that illustrate fine-grain process modelling in the ViewPoints framework. These examples are presented in two parts. The first part illustrates how basic process model entries may be described in individual ViewPoints. The second part describes interactions between different inter-ViewPoint rules, in order to illustrate how process models can also be used to coordinate interactions between different ViewPoints.

7.3.3.1. Process Models

In chapter 4, we suggested that ViewPoint process models may be expressed in terms preconditions, actions, agents and postconditions. We adopt this notation in our examples below in which an entry in a process model is of the general form:

preconditions ⇒ [agent, action] postconditions

For any ViewPoint (VP_S) development actions (e.g., assembly actions), the general form of a process model entry is:

specification-1 ⇒ [VP_S, assembly-action] specification-2

Similarly, the invocation of a consistency check may be expressed by:

$$\text{specification} \Rightarrow [\mathrm{VP}_S, \text{check}]\ \ \text{consistent-specification} \vee (\text{inconsistent-specification} \wedge \text{inconsistencies})$$

The above kinds of entries are at a developer level of granularity. They specify (recommend) actions that a developer may perform if very general preconditions hold, and specify the postconditions that hold if those actions are actually performed. To achieve an even finer-level - that is, representation level - process modelling granularity, specific actions and more detailed pre- and postconditions need to be included in process model entries. Inter-ViewPoint consistency checks provide illustrative examples of this, and also demonstrate how ViewPoint process models may control interactions between ViewPoints. In such cases, an "action" in the process model entry is the application of an inter-ViewPoint rule, which is always applied by an "agent" which is the source ViewPoint, VP_S. Pre- and postconditions are lists of predicates.

The preconditions to an entry in the process model come from two sources. The first source is information about the state of the ViewPoint. These are used to restrict the stages of development in which an action can be performed. For example, it might be necessary to prevent an inter-ViewPoint rule being applied unless certain *in*-ViewPoint rules have also been applied. The second source of preconditions is information about relationships with other ViewPoints. These will normally be generated as postconditions of other rules.

In general, the postconditions of applying rule R_n will be a list of instantiations of the relationship $\Re_n$ contained in the rule, expressed as a set of predicates of the form $\Re_n(\sigma, \delta)$, where σ and δ are the specification items ("instances") that match ps_S and ps_D - the partial specifications in the source and destination ViewPoints, respectively. If no partial specifications in the source and destination ViewPoints match the rule, then this set of relationships will be empty. We denote the set of p elements by:

$$\{\Re_n(\sigma_1, \delta_1), ..., \Re_n(\sigma_p, \delta_p)\}.$$

It is important to record the contents of this set of relationships in a ViewPoint's work record, because it is a record of the relationships between ViewPoints that have actually been found to hold at a particular moment in time. Further development may later cause the two related ViewPoints to diverge again, but analysis of the work record may help understand, and possibly resolve or reconcile, this divergence.

Of course, the application of an inter-ViewPoint rule may fail, and we therefore need predicates to represent the different failure states. We represent the possible failure states of *agreement rules* (chapter 6, section 6.3.2.1) using predicates of the form $\text{inconsistent}(\sigma, \delta, \mathrm{R}_n)$. Thus, the result of applying rule R_n to the partial specifications ps_S and ps_D, is a set of items for which the relationship $\Re_n$ holds, and a set of items for which the relationship should hold but does not. This is shown in the default entry in the process model for any agreement rule, R_n, as follows:

$$\{\text{preconditions}\} \Rightarrow [VP_S, R_n] \quad \{\Re_n(\sigma_1, \delta_1),\ldots, \Re_n(\sigma_p, \delta_p)\} \cup$$
$$\{\text{inconsistent}(\sigma_1, \delta_1, R_n),\ldots, \text{inconsistent}(\sigma_q, \delta_q, R_n)\}$$

An *existence rule* (chapter 6, section 6.3.2.1) fails when a destination ViewPoint is missing, or when no ViewPoint is found which contains the pattern ps_D. We can show this in the default process model entry as:

$$\{\text{preconditions}\} \Rightarrow [VP_S, R_n] \quad \Re_n(\sigma, \psi) \vee$$
$$\text{missing}(\sigma, VP_{D(t,\, d)}, R_n) \vee \text{missing}(\sigma, VP_{D(t,\, d)}{:}ps_D, R_n)$$

where σ is the actual item in the source ViewPoint that matched ps_S, and ψ is the destination ViewPoint that satisfied the existence criteria. The $\text{missing}(\sigma, VP_{D(t,\, d)}{:}ps_D, R_n)$ predicate can be read as "no (destination) ViewPoint of template t and domain d (and containing partial specification ps_D) was found to meet the existence criteria associated with partial specification σ as required by the rule R_n".

7.3.3.2. Interactions Between Rules

In many cases, the applicability of some rule will be affected by some other rule. Links between rules allow method designers to express complex interactions between ViewPoints such as *"Rule B should be invoked only if relationship A holds"* and *"Relationship A should hold unless rule B is applicable"*. Note the distinction between a relationship holding and a rule being applicable. Each rule R_n expresses a relationship $\Re_n$. Hence to say that $\Re_n$ holds at a particular instance is to say that the rule has been applied and was satisfied. To say that a rule is applicable only states that any necessary preconditions have been met.

We use ViewPoints' process models, to identify such dependencies between rules. Recall for example rule R_5 (described in chapter 6) which prescribed that "Every *output from a process in a data flow diagram must appear as a contextual output in every decomposition of that process*". This was expressed as:

R_5: {link(From, _).Flow.Name = VP(DFD, From.Name): link(_, To.Status.'context').Flow.Name}

Now consider the inverse of R_5 for data flow diagrams which relates a decomposition ViewPoint to its parent. Firstly, we cannot take the existence of the parent for granted, and we cannot assume that only one version of the parent exists. Thus, the existence rule *"There must be a parent data flow diagram containing the process represented by this data flow diagram"* deals with this by ensuring that there is at least one parent[25] ViewPoint. We express this as:

[25] We ignore for now the possibility that the source ViewPoint might be the top level ViewPoint, and hence have no parent.

R_{13}: self → ∃ VP(DFD, D_d): Process.Name.D_s

where D_s is the domain of the source ViewPoint, which we assume to be the name of the process which the source ViewPoint decomposes.

We then define separately any rules which express relationships between the decomposition and its parent, such as *"Contextual outputs in a data flow diagram must have the same names as the outputs from the process in the parent data flow diagram":*

R_{14}: {link(_, To.Name.'context').Flow.Name = VP(DFD, D_d): link(D_s, _).Flow.Name}

The relation between this rule (R_{14}) and R_{13}, is that R_{14} should only be applied when the destination ViewPoint is also the parent, as established by applying R_{13}. Thus, the process model must specify that R_{14} should only be applied to destination ViewPoints for which R_{13} has also been successfully applied. This may be expressed as:

$$\Re_{13}(\text{self}, \psi) \Rightarrow [VP_S, R_{14}] \quad \{\Re_{14}(\sigma_1, \psi{:}\delta_1), \ldots, \Re_{14}(\sigma_n, \psi{:}\delta_n)\} \cup$$
$$\{\text{inconsistent}(\sigma_1, \psi{:}\delta_1, R_{14}), \ldots, \text{inconsistent}(\sigma_m, \psi{:}\delta_m, R_{14})\}$$

where the precondition $\Re_{13}$ defines the destination ViewPoint, ψ, to which rule R_{14} applies. The postcondition is a set of partial specifications for which $\Re_{14}$ holds and a set of partial specifications that are inconsistent. Both sets could be empty, if nothing in the ViewPoints' specifications matched the patterns in the rule.

A more complex example is provided by the rule *"Data flow names must be unique across all data flow diagrams unless related across a decomposition"*. As we have seen above, there are several ways that data flow names can be related across a decomposition, including those specified in R_5, R_{13} and R_{14}, and some others to deal with input flows (such as R_6) which we will ignore for now. As a first step, we express the uniqueness of all data flow names across different ViewPoints by the rule:

R_{15}: ¬{link(_, _).Flow.Name = VP(DFD, D_d): link(_, _).Flow.Name)}

We then link it in the process model to the rules that specify exceptions. When applying R_{15} however, we are not interested in the set of partial specifications for which the rule holds, as this is just an exhaustive list of pairs of different data flow names. However, we are interested in any partial specifications for which the relationship does *not* hold. Hence, we link R_{15} in the process model using the entry:

$$[VP_S, R_{15}]\ \{\text{breaks}(\sigma_1, \delta_1, R_{15}), \ldots, \text{breaks}(\sigma_p, \delta_p, R_{15})\}$$

with no preconditions necessary to proceed with the actions specified. The predicate breaks denotes partial specifications for which a relationship should hold but does not. This is because partial specifications for which a relationship ($\Re_{15}$ in this case) does not hold, are not necessarily inconsistent, as one of the exceptions may apply. We then introduce another entry in the process model, that allows us to detect an inconsistency once we have also checked all the exceptions:

$$\text{breaks}(\sigma, \delta, R_{15}) \Rightarrow [VP_S, (R_5 \;\&\; R_{14})]$$
$$\text{irrelevant}(\sigma, \delta, R_5) \wedge \text{irrelevant}(\sigma, \delta, R_{14}) \rightarrow \text{inconsistent}(\sigma, \delta, R_{15})$$

where irrelevant(σ, δ, R_n) is true if neither $\Re_n(\sigma, \delta)$ nor inconsistent(σ, δ, R_n) are true (that is, the irrelevant predicate denotes exceptions).

To summarise, the last two entries in the process model above state that we cannot conclude there is an inconsistency between σ and δ just from applying R_{15}. The inconsistency can only be demonstrated if we know that relationship $\Re_{15}$ does not hold, and we have tested to confirm that none of the rules containing exceptions apply.

7.3.4. Inconsistency Handling

Consistency checking in the ViewPoints framework often results in the detection of inconsistencies. These inconsistencies may be local to individual ViewPoints, or involve two or more ViewPoints. In either of these two cases, immediate resolution may not always be necessary. In fact, the whole philosophy of the ViewPoints approach is to tolerate transient inconsistencies within ViewPoints, and to support the development of multiple ViewPoints which may be inconsistent with each other (be they conflicting requirements, alternative design solutions or actual mistakes in one or more ViewPoints). Clearly, the systematic support of inconsistency handling in this context is a process modelling issue.

How does one act in the presence of inconsistencies? Well, process modelling is a means of providing development guidance, and therefore inconsistency handling should also be a means of providing guidance about how and when to identify, circumvent, resolve or even ignore inconsistencies. This is particularly problematic when inconsistencies arise between ViewPoints, as opposed to being local to any one ViewPoint. Typically, the problem is to detect and identify an inconsistency, to decide if that inconsistency is in either or both of the two ViewPoints being checked, and then to perform the appropriate actions in order to handle that inconsistency. We now propose an inconsistency handling framework within which these issues may be addressed.

7.4. Inconsistency Handling in Multi-Perspective Specifications

The philosophy and requirements of our approach are as follows. We do not believe that it is possible, in general, to maintain absolute consistency between ViewPoints at all times. Indeed, it is often not even desirable to enforce consistency, particularly when this constrains the specification unnecessarily or entails loss of design freedom by enforcing an early resolution. We therefore require inconsistency handling techniques in which inconsistency is tolerated and used to trigger further actions.

We begin this section by providing some background and further motivation for (1) "making inconsistency respectable" [Gabbay & Hunter 1991] (via inconsistency handling techniques), and (2) the use of logic for this purpose. We then propose a simple framework for inconsistency handling using logic, and illustrate it with examples. A number of open issues in this context are also briefly outlined.

7.4.1. Background

One of the drawbacks of distributed development and specifications is the problem of managing consistency. It is generally more difficult to check and maintain consistency in a distributed environment. We believe however, that we should re-examine our attitude to consistency, and make more provision for inconsistency. Inconsistency is an inevitable part of the development process. Forcing consistency may restrict the development process and stifle novelty and invention. Thus, consistency should only be checked between particular parts or views of a design or specification, and at particular stages, rather than enforced as a matter of course. Addressing the problems of inconsistency however, raises many questions including: What exactly does consistency checking across multiple partial specifications mean? When should consistency be checked? How do we handle inconsistency?

In chapter 6, we described inter-ViewPoint relationships using inter-ViewPoint rules, and checked consistency between ViewPoints by applying these rules. In the last section, we addressed process modelling issues and examined the way in which process models can be used to guide inter-ViewPoint rule invocation and interaction. We now examine inconsistency handling in this context.

Given that inconsistency is often viewed as a logical concept (e.g., "contradictions" as defined in chapter 6, section 6.1.3.1), we can also base inconsistency handling on logic. The problem of inconsistency handling in the ViewPoints framework can then be viewed as being equivalent to inconsistency handling in distributed logical databases. In this way, we can then define rewrites (translations) from specification information - and inter-ViewPoint information - in a ViewPoint, to a set of logical formulae. Not all logics however, can be used for this purpose.

Classical logic, and intuitionistic logic, take the view that anything follows from an inconsistency. Effectively, when an inconsistency occurs in a database, it becomes unusable. This has prompted the logic community to study such logics as relevant [Anderson & Belnap 1976] and paraconsistent logics [da Costa 1974] that allow reasoning with inconsistent information. These isolate inconsistency by various means, but do not offer strategies for dealing with the inconsistency. Therefore there still remains the question of what do we do when we have two contradictory items of information in a database. Do we choose one of them? How do we make the choice? Do we leave them in and find a way “around” them?

Other logics, such as certain non-monotonic logics (for a review see [Bell 1990]), resolve some forms of inconsistency, but do not allow the representation of certain forms of inconsistent data, or give no answer when such data is present. There are also attempts at paraconsistent non-monotonic logics [Blair & Subrahmanian 1989; Pequendo & Buschbaum 1991; Wagner 1991], but these again do not answer the questions of handling inconsistency in software development.

The logic programming and deductive database communities have explored alternative approaches to handling of inconsistencies in data. These include integrity constraints (for example [Sadri & Kowalski 1986]) and truth maintenance systems [Doyle 1979]. In these approaches, any attempt to introduce inconsistency in the database causes rejection of input, or amendment of the database. This makes them unsuitable solutions for the ViewPoints framework, since they do not allow us to represent and reason with inconsistent information, nor do they allow us to formalise the desired actions that should result from identifying such inconsistent information.

In summary, “traditional” logic-based approaches constitute a significant shortfall in the ability required to handle inconsistency in formal knowledge representation. In our approach, we attempt to shift the view of inconsistency from being necessarily “bad” to being acceptable, or even desirable, if we know how to deal with it [Gabbay & Hunter 1991; Gabbay & Hunter 1992]. Moreover, when handling inconsistencies in a database, we analyse them within the larger context of the environment of the database and its use. Thus, when viewed locally, an inconsistency may seem undesirable, but within the larger environment surrounding the data, it may be desirable and useful, if we prescribe appropriate actions to handle it. Moreover, we do not necessarily deal with inconsistencies by restoring consistency, but by supplying rules telling one how to act when such inconsistencies arise.

7.4.2. Framework

Our approach is to formally consider both the data in the database, and the use of the database in the environment. The latter is usually not formalised, though for many database applications there are informal procedures, or conventions, that are assumed by the user. If we formalise the link between the database and the environment, this allows us to handle inconsistency in data in terms of these procedures and conventions. Furthermore, it also allows us to consider the inconsistencies resulting from contradictions between the data and the use of the data. For example, it is not uncommon for inconsistencies to occur in accounting systems. Consider the use of credit cards in a department store, where an inconsistency may occur on some account. In this case the store may take one of a series of actions such as writing off the amount owed, or leaving the discrepancies indefinitely, or invoking legal action. Another example is in government tax databases where inconsistencies in a taxpayer's records are "desirable" (at least from the tax inspectors point of view!), and are used to invoke an investigation of that taxpayer.

In our approach we capture in a logical language the link between the data and the usage of the data. In particular we analyse inconsistencies in terms of a pair of logical formulae (D, E) where D is a database of logical formulae representing some of the information in one or more ViewPoints, and E is a logical representation of some of the implicit assumptions and integrity constraints used in controlling and coordinating the use of a set of ViewPoints. We can view E as the environment in which the database operates and thus also include some information on inter-ViewPoint relationships. We further assume that for the purposes of our discussions the information expressed in E is consistent - that is, we use E as the reference against which consistency is checked. Using (D, E), we can perform partial or full consistency checking in an attempt to elucidate the "sources" of inconsistency in our logical database.

We handle inconsistencies in (D, E), by adopting a meta-language approach that captures the required actions to be undertaken when discovering an inconsistency, where the choice of actions is dependent on the larger context. Using a meta-language allows handling of a database in an environment by encoding rules of the form:

INCONSISTENCY IN (D, E) SYSTEM implies ***ACTION*** IN (D, E) SYSTEM

These rules may be physically distributed among the various ViewPoints under development, and invoked by the ViewPoint that initiates the consistency checking. Some of the actions in these rules may make explicit internal database actions such as invoking a truth maintenance system, while others may require external actions such as "seek further information from the user" or invoke external tools. To support this formalisation of data handling, we need to consider the nature of the external and internal actions that result from inconsistencies in the context of the ViewPoints framework.

Figure 7.4 schematically summarises the stages of rewriting (translation), identification, and handling of inconsistency when checking two ViewPoints in the framework. To check the consistency of specifications in two ViewPoints, partial specification knowledge in each is translated into classical logic. Together with the inter-ViewPoint rules in each ViewPoint (which are also translated into logic), inconsistencies between the two ViewPoints may be identified. Meta-level rules are then invoked which prescribe how to act on the identified inconsistencies.

Note that we are *not* claiming that classical logic is a universal formalism into which any two representations may be translated. Rather, we argue that for any two partial specifications *a* common (logical) representation may be found and used to detect and identify inconsistencies.

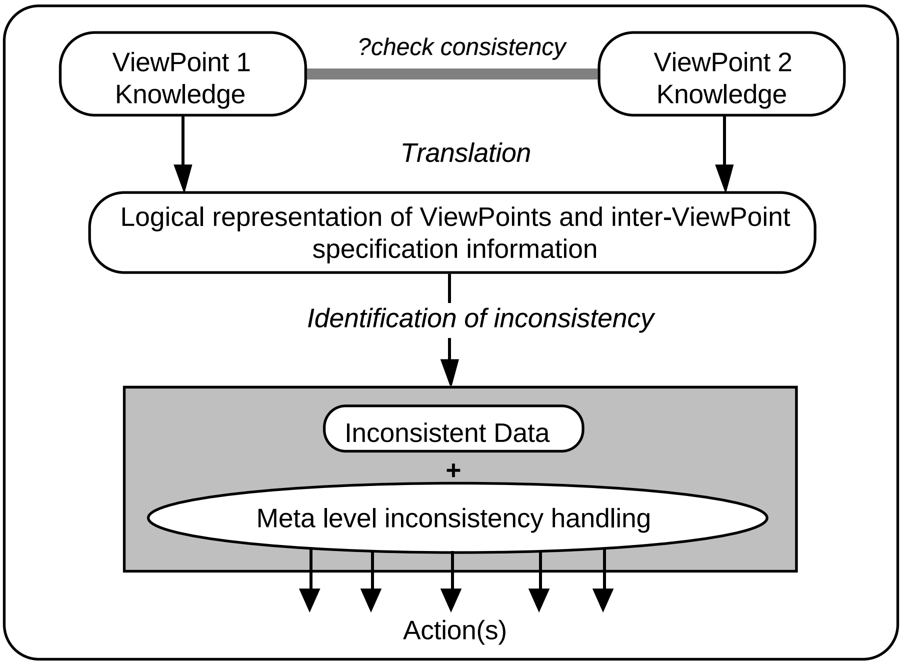

ViewPoint 1 Knowledge
?check consistency
ViewPoint 2 Knowledge
Translation
Logical representation of ViewPoints and inter-ViewPoint specification information
Identification of inconsistency
Inconsistent Data
+
Meta level inconsistency handling
Action(s)

Figure 7.4: *ViewPoint interaction and inconsistency handling in the ViewPoints framework. Selected ViewPoint knowledge in each of the interacting ViewPoints is translated into logical formulae and used to detect and identify inconsistencies. The meta-level rules can then be used to act upon these inconsistencies.*

7.4.3. Examples: Identification of Inconsistency

To demonstrate our approach, we use a simplified library example specified using a subset of the representation schemes deployed by the requirements analysis method CORE, namely, agent structuring (figure 7.5) and tabular collection (figures 7.6, 7.7 and 7.8). Recall that an agent structure diagram (agent hierarchy) decomposes a problem domain into information processing entities or roles called “agents”. Agents may be “direct”, if they process information, or “indirect”, if they only generate or receive information without processing it. For each direct agent in the hierarchy, we construct a tabular collection diagram showing the actions performed by that agent, the input data required for the actions to occur and the output data produced by those actions. The destination and source agents to and from which the data flows are also shown (tabular collection diagrams may therefore be regarded as a structured form of data flow diagram). An arc drawn between two lines in a tabular collection diagram indicates the conjunction of the two terms preceding the joining lines[26].

In the context of the ViewPoints framework, figures 7.5 to 7.8 denote the contents of the specification slots of the different ViewPoints in the overall system specification. Thus, the specification shown in figure 7.8 for example, would appear in the ViewPoint outlined schematically in figure 7.9.

[26] This is an extension of the “standard” CORE [Mullery 1985].

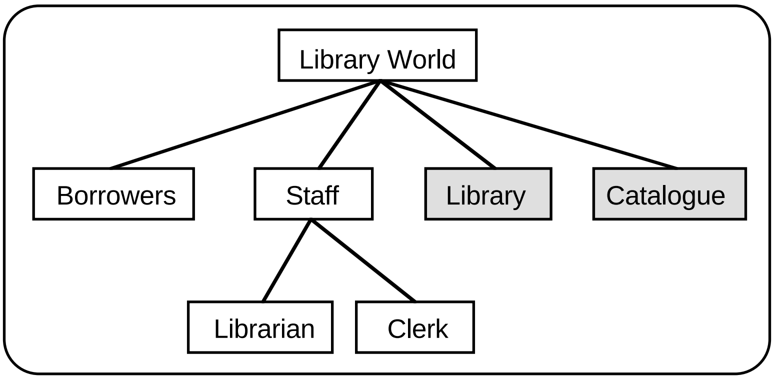


Figure 7.5: *An agent hierarchy decomposing the library problem domain into its constituent agents. Shaded boxes indicate indirect agents which require no further decomposition or analysis.*

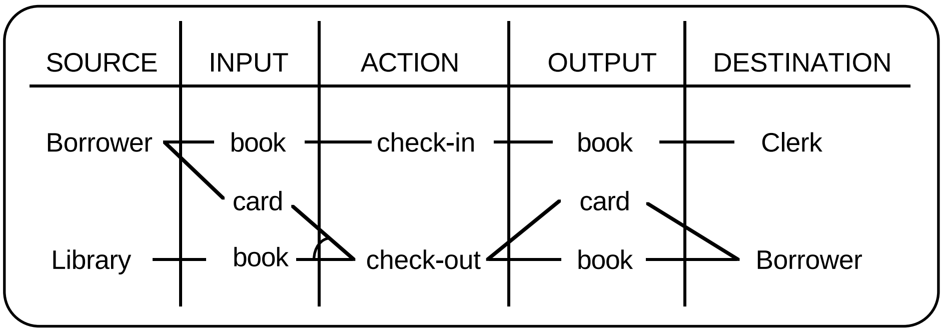


Figure 7.6: *A tabular collection diagram elaborating the agent "Borrower". In ViewPoints terminology, the tabular collection diagram is part of a ViewPoint specification where the ViewPoint domain is "Borrower".*

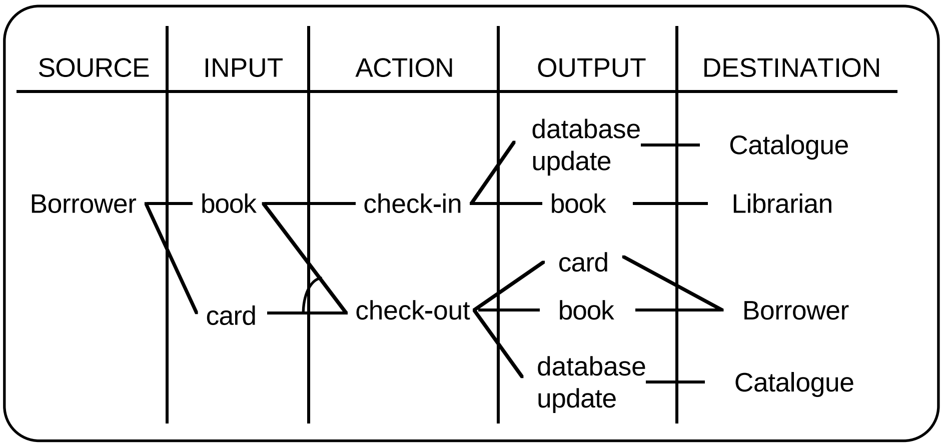


Figure 7.7: *A tabular collection diagram elaborating the agent "Clerk". In ViewPoints terminology, the tabular collection diagram is part of a ViewPoint specification where the ViewPoint domain is "Clerk".*

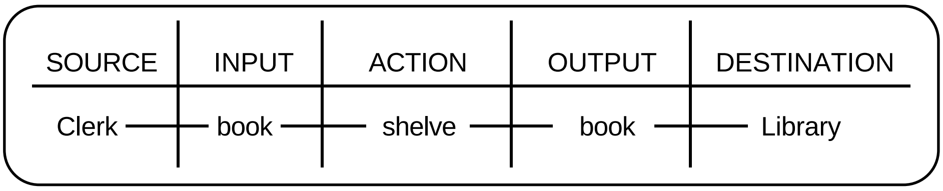


Figure 7.8: *A tabular collection diagram elaborating the agent "Librarian". In ViewPoints terminology, the tabular collection diagram is part of a ViewPoint specification where the ViewPoint domain is "Librarian".*

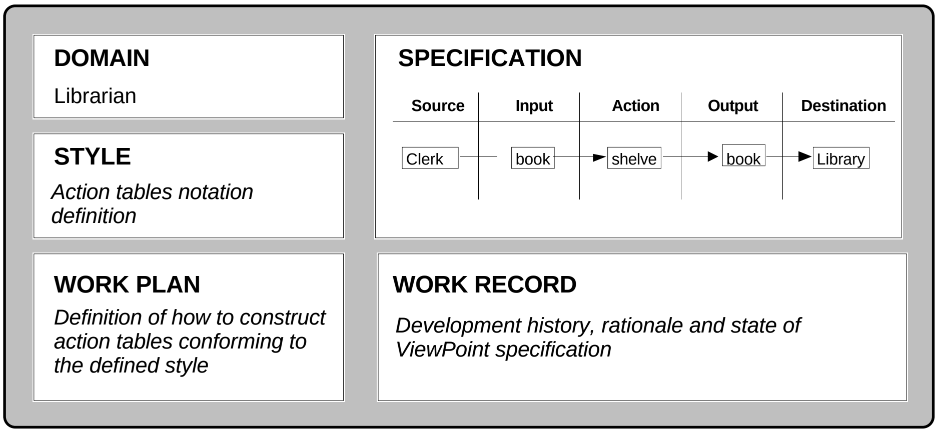

DOMAIN
Librarian
SPECIFICATION
Source
Input
Action
Output
Destination
Clerk
book
shelve
book
Library
STYLE
Action tables notation definition
WORK PLAN
Definition of how to construct action tables conforming to the defined style
WORK RECORD
Development history, rationale and state of ViewPoint specification

Figure 7.9: *A schematic outline of a ViewPoint containing the specification shown in figure 7.8.*

The agent structuring and tabular collection representation schemes are related in a number of ways (as specified by the CORE method designer). Two such relations are described informally by the following rules:

Rule 1 (between an agent hierarchy and actions tables): Any “source” or “destination” in an action table, must appear as a leaf agent in the agent hierarchy (rule R_{10} described in chapter 6 is within the scope of this rule).

Rule 2 (between action tables): The output (Z) produced by an action table for an agent (X) to a destination (Y), must be consumed as an input (Z) from a source (X) by the action table for the original destination (Y) (rule R_{11} described in chapter 6).

Note that both the library example and the ViewPoints used to describe it have been greatly simplified to illustrate our consistency handling mechanism. The library world in fact involves many more transactions and may require richer notations to specify it fully. This can be done by defining the desired notation in the appropriate ViewPoint style slot.

To perform a consistency check between two or more ViewPoints, we form a logical database (D, E), where D contains formulae representing the partial specifications in these ViewPoints, and E contains formulae representing environmental information such as inter-ViewPoint rules. In our examples, we assume the language for (D, E) is first-order classical logic.

We start by looking at the tabular collection diagrams in figures 7.6, 7.7 and 7.8. For these we consider the preconditions and postconditions to an action. We use pre(X, Y) to denote that X is a source of an input, Y, consumed by some action. Similarly, post(Y, X) denotes that X is the destination of an output, Y, produced by some action. Now we denote the part of the tabular collection diagram associated with an action B as follows:

table(A, P, B, Q)

where, A is an agent name (the content of the ViewPoint domain slot),

P is a "conjunction" of preconditions for action B, and

Q is a "conjunction" of postconditions for action B.

Note that for this definition we are using "conjunction" as a function symbol (denoted by "&") within first-order classical logic.

We can now represent each table in figures 7.6, 7.7 and 7.8 using the above predicates. First we consider the information in figure 7.6 and represent it as:

(1) table(borrower, pre(borrower, book), check-in, post(book, clerk)).

(2) table(borrower, pre(borrower, card) & pre(library, book), check-out,

post(book&card, borrower)).

Similarly, we represent the information in figure 7.7 as:

(3) table(clerk, pre(borrower, book), check-in,

post(database-update, catalogue) & post(book, library)).

(4) table(clerk, pre(borrower, book&card), check-out,

post(book&card, borrower) & post(database-update, catalogue)).

Finally we represent the information in figure 7.8 as:

(5) table(library, pre(clerk, book), shelve, post(book, library)).

Obviously there are many ways we could present the information in figures 7.6-7.8 in logic. However, it is relatively straightforward to define a rewrite that can take any such table and return logical facts of the above form. We can also capture the relevant inter-ViewPoint rules in classical logic. So rule 2 (between action tables) may be expressed as:

(6) $\forall$ A, B_i , C_i

[[table(A, _, _, post(C_1, B_1) & .. & post(C_m, B_m))]

$\rightarrow$ [table(B_1, X_1, _, _) and ... and table(B_m, X_m, _, _)]]

where $1 \leq i \leq m$ and pre(A, C_i) is a conjunct in X_i.

where the underscore symbol "_" denotes that we are not interested in this part of the argument for this rule, and that it can be instantiated without restriction.

The formulae (1) - (5) are elements in D, and the formula (6) is an element in E. Since we have represented the information in D as a set of classical logic facts, we can use the Closed World

Assumption [Reiter 1978] to capture the facts that do not hold in each ViewPoint specification. The Closed World Assumption (CWA) essentially captures the notion that if a fact A is not a member of a list of facts then ¬A holds. So for example, using the CWA with the borrower domain we can say that the following arbitrary formula does not hold:

(7) table(borrower, pre(borrower, book), check-in, post(card, clerk)).

In other words we can infer the following:

(8) ¬table(borrower, pre(borrower, book), check-in, post(card, clerk)).

Using this assumption we can identify ViewPoints that are inconsistently specified. We show this for the relation between the borrower and the clerk. Suppose that formula (3) was not represented, then by the CWA we would have its negation as follows:

(9) ¬table(clerk, pre(borrower, book), check-in,
post(database-update, catalogue) & post(book, library)).

Using a classical logic theorem prover with the CWA (or by simple observation in this case), we can show that (D, E) is inconsistent and that the formulae (1), (6) and (9) cause the inconsistency - since (1) and (6) give:

table(clerk, pre(borrower, book), check-in, X, Y)

where X and Y can be instantiated with any term in the language, whereas the CWA gives:

¬table(clerk, pre(borrower, book), check-in, X, Y)

for any terms X, Y in the language. We address the handling of this situation in the next section.

In a similar fashion, we can represent the agent hierarchy in figure 7.5 by the following set of logical facts:

(10) tree(library-world, borrower)

(11) tree(library-world, staff)

(12) tree(library-world, library)

(13) tree(library-world, catalogue)

(14) tree(staff, librarian)

(15) tree(staff, clerk)

with the first argument to the predicate tree represents a parent in the agent hierarchy, and the second argument represents its child. To these facts we add “standard” axioms of reflexivity, transitivity, anti-symmetry, up-linearity and leaf; e.g., (leaf) $\forall$ X, Y [tree(X, Y) and $\neg$($\exists$ Z such that tree(Y, Z)) $\rightarrow$ leaf(Y)]

These axioms then allow us to capture the appropriate reasoning with the hierarchy so that from (11), (15) and the transitivity axiom for example we can infer the following:

(16) tree(library-world, clerk).

Another inter-ViewPoint rule (rule 1 - between ViewPoints deploying different notations) is captured as follows:

(17) $\forall$ A_i [[table(_, pre(A_1, _) & .. & pre(A_m, _), _, _)]

$\rightarrow$ $\exists$ X such that [leaf(X) and X = A_i]]

where $1 \leq i \leq m$

(18) $\forall$ B_i [[table(_, post(_, B_1) & .. & post(_, B_m))]

$\rightarrow$ $\exists$ X such that [leaf(X) and X = B_i]]

where $1 \leq i \leq m$

As before, formulae (10) - (15) plus the axioms of reflexivity, transitivity, antisymmetry, leaf and up-linearity are elements in D, while (17) and (18) are elements in E. Remember that we are also assuming that rules 1 and 2, expressed in logic as (6), (17) and (18) above, are correctly defined by the method designer.

Thus, if formula (13), say, had not been inserted in the relevant ViewPoint, then again using the CWA together with a classical logic theorem prover, we can see how insufficient information in (D, E) leads to inconsistency.

We now turn our attention to a situation where we have too much information in a pair of partial specifications. Inconsistencies in such a situation are in fact easier to detect. Suppose in one ViewPoint, we have the following information,

(19) reference-book(childrens-dictionary)

(20) $\forall$ X, reference-book(X) $\rightarrow$ $\neg$lendable(X)

and suppose in another ViewPoint, we have the following information,

(21) childrens-book(childrens-dictionary)

(22) $\forall$ X, childrens-book(X) $\rightarrow$ lendable(X)

Taking the formulae in (19) - (22) as elements in D, we have an inconsistency in (D, E) resulting from too much information. Such a situation is common in developing specifications, though the causes are diverse.

To summarise our examples thus far, we have used classical logic together with the CWA to provide a systematic and well-understood way of finding inconsistencies in specifications. In general, it will not be possible to provide a universal and meaningful translation (rewrite) from any software engineering representation scheme into classical logic. However, for partial consistency checking it is often possible to compare some of the specification information, plus other information such as inter-ViewPoint relationships, for two, or maybe more, ViewPoints.

7.4.4. Acting on Inconsistency

For the meta-level inconsistency handling part of our framework, we have explored the use of an action-based meta-language [Gabbay & Hunter 1992] based on linear-time temporal logic. We use a first-order form where we allow quantification over formulae[27]. Furthermore, we use the usual interpretation over natural numbers - each number denotes a point in time. Using this interpretation we can define operators such as LAST^n and NEXT^n where LAST^n A holds at time t if A holds at t-n, and NEXT^n A holds at time t if A holds at t+n.

Using temporal logic, we can specify how databases (“ViewPoints”) should evolve over time. In this way, we can view the meta-level handling of inconsistent ViewPoint specifications in terms of satisfying temporal logic specifications. So if during the course of a consistency check between two ViewPoints an inconsistency is identified, then one or more of the meta-level action rules will be fired (invoked). Furthermore, since we use temporal logic, we can use recorded information about how we have handled consistency checks between these ViewPoints in the past.

The meta-level axioms specify how to act according to the context of the inconsistency. This context includes the history behind the inconsistent data being put into the ViewPoint specification - as recorded in the ViewPoint work record - and the history of previous actions to handle the inconsistency. The meta-level axioms should also include implicit and explicit background information on the nature of certain kinds of inconsistencies, and how to deal with them.

To illustrate the use of actions at the meta-level, we now return to the examples introduced in the last section. For handling the inconsistency resulting from formulae (1), (6) and (9), a simplified solution would be to incorporate the kind of meta-level axiom (23) into our framework (e.g., as an entry in a ViewPoint process model). For this we provide the following informal definitions of the key predicates:

- $\text{data}(\text{vp1}, \Delta_1)$ holds if the formulae in the database Δ_1 are a logical rewrite of selected information in ViewPoint vp1.

[27] Note that we are not using a second-order logic - rather we are treating object-level formulae as objects in the semantic domain of the meta-level.

- union(Δ_1, Δ_2) $\vdash$ false holds if the union of the databases Δ_1 and Δ_2 implies inconsistency.
- inconsistency-source(union(Δ_1, Δ_2), S) holds if S is a minimal inconsistent subset of the union of Δ_1 and Δ_2.
- likely-spelling-problem(S) holds if the cause of the inconsistency is likely to result from typographical errors in S. Since we are using a temporal language at the meta-level, we can also include conditions in our rule that we haven't checked this problem at previous points S in time. This means that we can specify how our history affects our future actions.
- tell-user("is there a spelling problem?", S) is an action in which the message "is there a spelling problem", together with the data in S, is outputted to the user. In software process modelling terminology, this is equivalent to, say, a tool invocation, such as a spell-checker or other tool.

(23) data(vp1, Δ_1) and data(vp2, Δ_2)
and union(Δ_1, Δ_2) $\vdash$ false
and inconsistency-source(union(Δ_1, Δ_2), S)
and likely-spelling-problem(S)
and $\neg$LAST^1 likely-spelling-problem(S)
and $\neg$LAST^2 likely-spelling-problem(S)
$\rightarrow$ NEXT tell-user("is there a spelling problem?", S).

Essentially, this rule captures the action that if S is the source of the inconsistency and that if the likely reason that S is inconsistent is a typographical error, then the user must be told to handle the problem. We assume that the user can usually deal with this kind of problem once informed. However, we should include further meta-level axioms that provide alternative actions, in case the user cannot deal with the inconsistency on this basis. Indeed, it is likely that for handling inconsistency between different notations such as in (1), (6) and (9), there will be a variety of possible actions. This meta-level axiom also has the condition that this action is blocked if likely-spelling-problem(S) has been identified in either of the two previous two steps. This is to stop the same rule firing if the user wants to ignore the problem for a couple of steps.

Similarly, we can define appropriate meta-level axioms for handling the inconsistencies resulting from formulae (9), (10), (17) and (18) in the examples of the last section.

For handling the problem of too much information occurring in formulae, such as (19) - (22) for example, a simplified solution would be to incorporate the kind of meta-level axiom (24) into our framework, where likely-conflict-between-specs-problem(Δ_1, Δ_2) holds if the inconsistency arises from just information in the specification. In other words, this inconsistency does not arise because the method or tools have been used incorrectly, but rather, it arises from incorrectly specifying the system.

(24) data(vp1, Δ_1) and data(vp2, Δ_1)
and union(Δ_1, Δ_2) $\vdash$ false
and likely-conflict-between-specs-problem(Δ_1, Δ_2)
$\rightarrow$ NEXT tell-user("is there a conflict between specifications?", (Δ_1, Δ_2)).

7.4.5. Research Agenda

Since the examples in our proposed approach use temporal logic, they have a well-developed theoretical basis[28]. Assuming that time corresponds to a linear sequence of natural numbers, we have all the usual temporal operators including NEXT^n, LAST^n, SOMETIME-IN-THE-FUTURE, SOMETIME-IN-THE-PAST and ALWAYS. Similarly, if we assume that time corresponds to a linear sequence of real numbers, we still have many of these operators.

The meta-level axioms presented here however, have ignored many difficult technical problems, including the general problems of decidability and complexity of such axioms, and the more specific problems of, say, defining predicates like inconsistency-source, likely-spelling-problem and likely-conflict-between-specs-problem. Moreover, we have ignored the many ways that our approach builds on a variety of existing work on database updates, integrity constraints, database management systems and meta-level reasoning. Nevertheless, we have illustrated how a sufficiently rich meta-level logic can be used to formally capture intuitive ways of handling inconsistencies in our (D, E) databases. Our meta-level axioms can also be used to describe, guide and manage parts of a multi-ViewPoint development process in this setting. The advantage over traditional approaches to process modelling in such cases, is that our approach allows very fine-grain modelling - at the level of individual inconsistencies between ViewPoints.

A number of other open issues however, remain to be addressed. Many of these centre around conflict (or inconsistency) *resolution* versus *handling*. Handling conflicts or inconsistencies subsumes resolution and therefore subsumes the following activities:

- *Local resolution.* An inconsistency is detected (within a ViewPoint) which can be, and is, resolved by the ViewPoint's owner.
- *Deferring action.* An inconsistency is detected (within a ViewPoint) which can be resolved, but the ViewPoint's owner decides to defer such resolution; e.g., because more information may be available later to make resolution easier.
- *Note posting.* An inconsistency between two ViewPoints is detected, but the source ViewPoint's owner believes that the destination ViewPoint is in error, and subsequently

[28] It can be shown that this meta-level language inherits desirable properties of first-order until-since (US) temporal logic such as a complete and sound proof theory, and of semi-decidability.

"posts" information necessary for conflict resolution to that ViewPoint.

- *Using the work record and previous checks' results.* The results of previous consistency checks (contained in a ViewPoint's work record) are used to facilitate identifying the possible causes of inconsistencies.
- *Using domain relationships between ViewPoints.* Inconsistencies between ViewPoints are often domain-specific, and therefore detecting them relies on checking domain-specific inter-ViewPoint relationships or rules.
- *Amelioration.* Inconsistencies between ViewPoints are not always easy to resolve immediately or completely, and therefore intermediate steps that "improve" the situation may be performed.
- *Reasoning in the presence of inconsistency.* Inconsistencies within or between ViewPoints should not prevent further development of either or both ViewPoints, including reasoning about and/or analysis of the specifications contained within those ViewPoints.

There also appears to be another temporal aspect to inconsistency management, which may help identify and prioritise inconsistencies. An illustrative example of this is the distinction between an inconsistency reflecting an error in development, and an inconsistency that only exists temporarily because certain development steps have not been performed yet. The latter inconsistency is a part of every development and, we submit, is less important than the former (which reflects a more fundamental failure in development). If at all, current process modelling technology provides guidance for "normal" development (e.g., "what should I do next?"), whereas we are attempting to handle inconsistencies that are usually labelled as "undesirable" in such a development process (e.g., "how do I get out of the mess I'm now in?").

Yet another interesting temporal consideration is what we might call the "age" of an unresolved consistency failure. This is a measure of, say, the number of work plan actions that were performed since the last time a consistency failure was introduced by an action. An upper bound on this is the number of work plan actions that were performed since the last time a consistency rule was found to hold. It may be useful to explore the correlation, if any, between the age of an unresolved consistency failure and the degree of difficulty by which it may be handled or resolved. Intuitively, one would expect that the greater the age of the last consistency check, the higher the risk becomes, and that there is a trade-off between the cost of consistency checking and the cost of resolution (we measure risk in this context as the likelihood of consistency failures multiplied by the cost of resolving them).

Despite the open issues discussed above, even modest progress towards achieving effective inconsistency handling facilitates the method integration we require. Inconsistency handling is an important step in a (ViewPoint-oriented) development process, and also facilitates process integration. While a number of technical issues have yet to be resolved, we feel that a framework is in place within which these may be addressed.

7.5. Chapter Summary

This chapter has outlined how process modelling, and in particular inconsistency handling, may be performed in the ViewPoints framework. The notion of fine-grain process modelling was introduced as a means of providing more effective method guidance for individual developers in a software development process. The use of process modelling to control, coordinate and synchronise inter-ViewPoint rule invocation and application was also proposed and demonstrated using a number of illustrative examples. The chapter also emphasised the need for inconsistency handling (in contrast with consistency maintenance), and proposed an action-based framework to demonstrate the feasibility of this in the context of the ViewPoints framework.

Clearly, computer-based software tools are needed to support the kinds of processes described in this chapter. While we have not implemented the logic-based inconsistency handling framework outlined above, the next chapter addresses the scope, role and our implementation of tool support for large parts of the ViewPoints framework.

Chapter 8 Tool Support - The Viewer

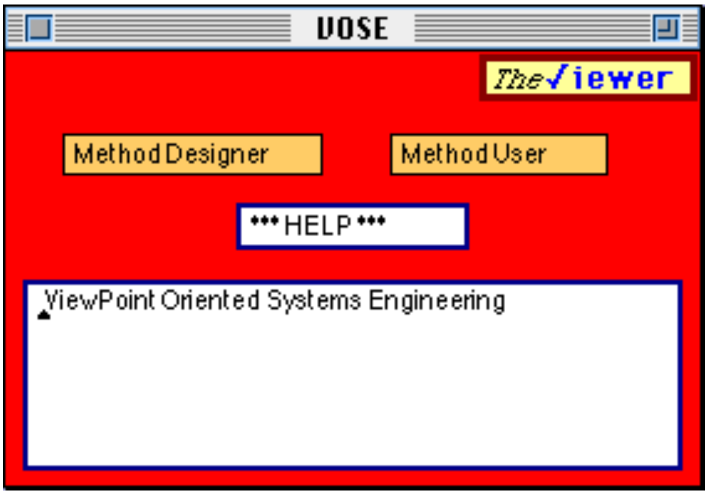
This chapter outlines the scope of computer-based tool support for the ViewPoints framework. It describes *The Viewer,* a prototype environment and tool set, constructed to support the framework, and to demonstrate its feasibility for multi-perspective software development. The functionality and architecture of *The Viewer* is described, with particular emphasis on its object-oriented implementation that reflects the object-based nature of the ViewPoints framework itself. The role of *The Viewer* as an agenda of software engineering concerns is discussed, and its extensibility is explored.

8.1. Scope

The ViewPoints framework we have described in this thesis addresses a wide range of software engineering concerns that result from adopting a decentralised, multi-perspective software development approach. Consequently, our design of computer-based tool support for the framework primarily focuses on demonstrating "proof-of-concept", rather than on optimising support for a particular aspect or technique used in the framework. Our approach was to build a generic support environment for the framework, and systematically populate it with specialised tools and techniques that either demonstrate the feasibility of the approach or address a particular software engineering issue.

The Viewer environment and associated tools, was developed in Smalltalk (Objectworks version 4.0) on an Apple Macintosh IIci. The nature of the Smalltalk environment, and our portable implementation, allows *The Viewer* to run, without modification, on UNIX workstations running X-Windows, and on IBM PCs running MS-Windows. While *The Viewer* is largely developed as a standalone environment, familiarity with the Smalltalk development environment and its WIMPS user interface is helpful to the novice user.

The scope of *The Viewer's* functionality ranges from method engineering and design, to method deployment and use. The startup window of *The Viewer*, shown in figure 8.1, reflects this. *The Viewer* may be used by method designers or engineers constructing software engineering methods from their constituent ViewPoint templates. Alternatively, it may be used by system developers (method users) who create, develop and manage ViewPoints instantiated from methods designed and defined by method engineers.



ViewPoint Oriented Systems Engineering

Figure 8.1: *The startup window of The Viewer.*

While *The Viewer* combines both of the above activities in the same tool, in fact this is only done for convenience. A method engineer usually works separately from a method user (although methods do evolve as a result of feedback from users). Including both method engineering and method use in *The Viewer* allows us to demonstrate some of the relationships between the two activities. In particular, if we treat the definition of a method (in terms of templates) as a declarative description of tools supporting that method, then using *The Viewer's* meta-CASE facilities, this definition can be transformed to produce a large part of the CASE tools that support the development of ViewPoints instantiated from those templates (during method use).

In the context of *The Viewer*, we therefore consider two kinds of tools. The first are *support tools* that form part of the infrastructure of *The Viewer*, and which facilitate the development and management of ViewPoints and ViewPoint templates. These include the TemplateBrowsers, ViewPointConfigurationBrowsers and ViewPointInspectors described in the sections that follow. The second kind are *CASE tools* that have been developed using *The Viewer's* (meta-CASE) support tools to automate or support particular templates or techniques (we also assume in what follows that "external" tools - such as a variety of UNIX tools - can be represented as *The Viewer's* CASE tools).

8.2. Method Engineering and Integration

Method engineering in *The Viewer* is the process of selecting, designing and defining the constituent ViewPoint templates of a chosen method. The choice and design of templates is a process that relies on creativity, experience and the availability of templates in a reuse library. Defining these templates is a more structured process that is therefore more amenable to computer-based tool support. Thus, what *The Viewer* provides is a so-called TemplateBrowser to facilitate the assembly and reuse of templates and their structured definition (editing, customising, modifying, etc.).

The reuse capabilities of the TemplateBrowser are simple. ViewPoint templates are stored in a flat Smalltalk database, either individually or as parts of a software engineering method. The TemplateBrowser effectively acts as a DBMS, allowing users to access and modify the templates database.

The primary role of the TemplateBrowser is to facilitate the definition of ViewPoint templates. Figure 8.2 shows a TemplateBrowser when it is first invoked from *The Viewer's* startup window.

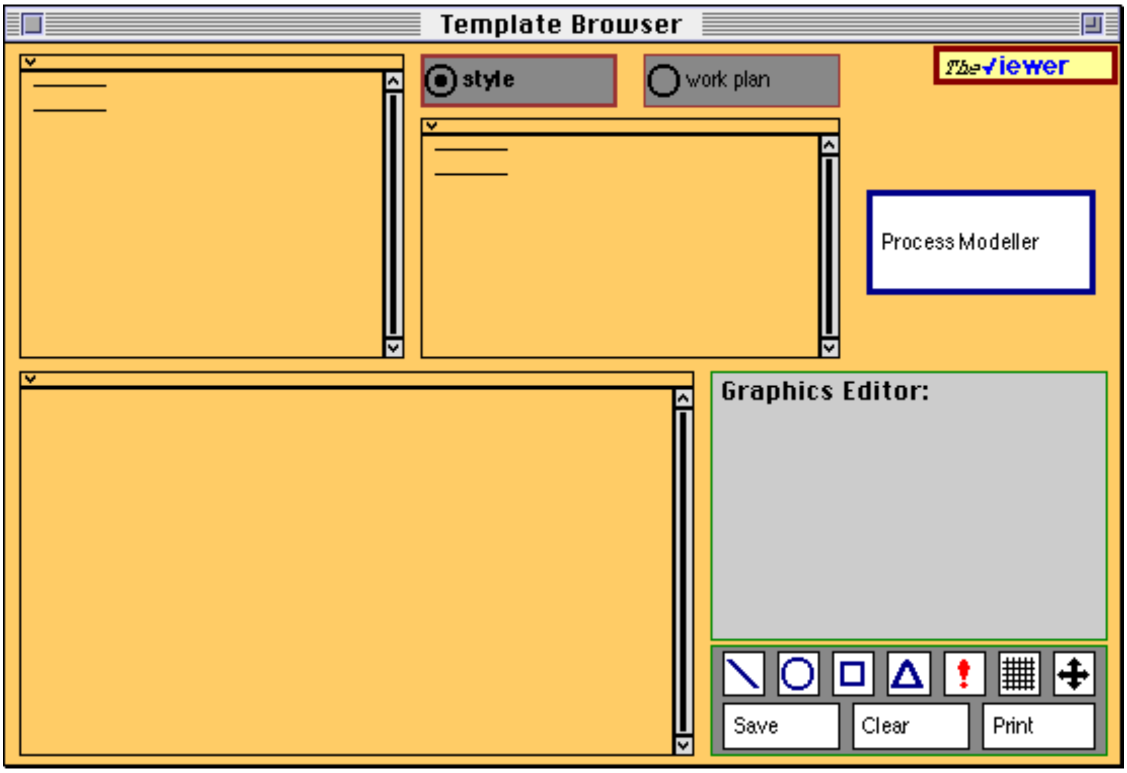
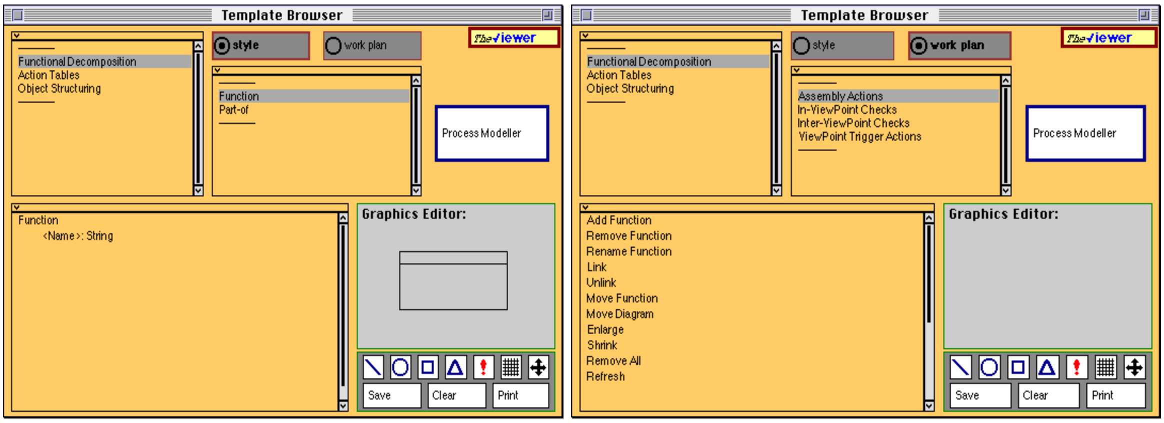


Figure 8.2: *The Viewer's TemplateBrowser.*

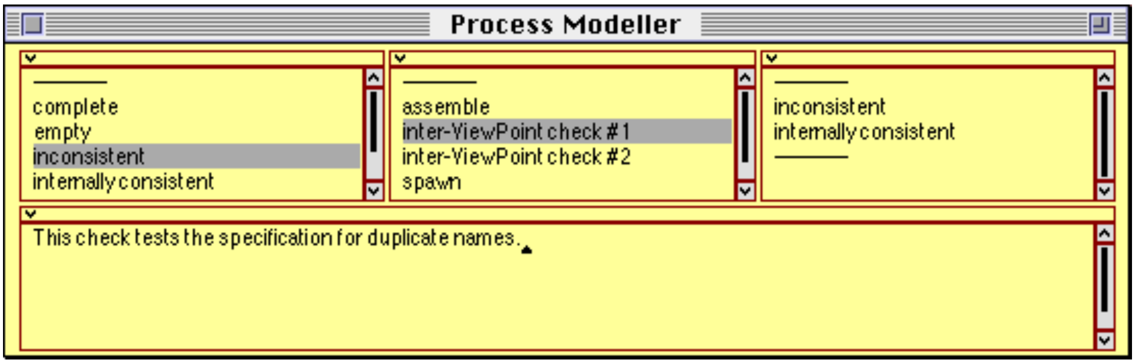
The top left-hand window pane of the TemplateBrowser lists the ViewPoint templates that are currently loaded into the tool. For any selected template, either the "Style" or the "Work Plan" button/switch may be selected. This allows the definition of the two slots of a ViewPoint template. If the "Style" switch is selected (figure 8.3a), then the method engineer can graphically and textually define the objects and relations of the representation style of the template (as described in chapter 4, section 4.2.2.1). If the "Work Plan" switch is selected (figure 8.3b), then the various work plan actions can be textually described (as outlined in chapter 4, section 4.2.2.2). In the current prototype implementation, the different actions, such as check actions, are described in freehand text for convenience. The implementations of these actions are hard-coded into *The Viewer*, and require some programming in Smalltalk. We have structured the implementation of *The Viewer* to facilitate such programming, as described in section 8.5. Since integration in this context is the definition of inter-ViewPoint rules, these rules have to be textually described in the TemplateBrowser, and implemented in Smalltalk.



(a) **(b)**

Figure 8.3: *(a) Style and (b) Work Plan definitions in a TemplateBrowser.*

Part of the definition of a template work plan, is the definition of the template's local process model. We have prototyped a simple process modelling tool (figure 8.4) to do this, which can be invoked by clicking on the "Process Modeller" button on the TemplateBrowser. Note that we have located the "Process Modeller" button separately to emphasise its importance in method engineering. Process modelling however, is a work plan activity, and therefore the ProcessModeller tool can only be invoked when the "Work Plan'" switch has been selected.



This check tests the specification for duplicate names.

Figure 8.4: *A ViewPoint template's ProcessModeller. The top three panes, from left to right, contain preconditions, actions and postconditions, respectively.*

The ProcessModeller tool facilitates the definition of simple process models using preconditions, actions and postconditions (which can be added, removed, modified and selected in the three top window panes of the tool). Each precondition or action can be annotated with free text (in the bottom pane), which is a useful way of defining context-sensitive help (guidance) for the method user (see figure 8.11 in section 8.4.4).

Once one or more ViewPoint templates have been defined, they can be saved individually (e.g., for later reuse) or collectively (as a method) in the templates database - ready to be instantiated by method users.

8.3. Tool Engineering with Meta-CASE

Before we describe method use in *The Viewer*, we first briefly examine how CASE tools to support the templates defined in the TemplateBrowser can be constructed. The architecture of *The Viewer* (section 8.5) facilitates tool engineering in this context by using Smalltalk's object-oriented framework to "plug-in" and extend tools in the program code. In this way, CASE tools to support particular templates may be implemented in Smalltalk, taking advantage of inheritance to avoid duplicating "standard" editing and syntax checking activities - particularly for similar templates.

The Viewer however, also provides a small meta-CASE tool kit to demonstrate the role of meta-CASE technology in the ViewPoints framework, and to illustrate the relationship between method engineering and method use. A particularly illustrative example is the graphical definition of objects in a template's style slot. The graphical editor provided by the TemplateBrowser allows a method engineer to sketch graphical icons of different shapes and sizes. These icons are stored as part of their respective templates' definitions, and are then translated by *The Viewer* into icons available to the method user during ViewPoint specification development.

Similarly, the various work plan actions are translated by *The Viewer* into development menus. For example, the list of assembly actions defined in a template work plan, becomes part of an "edit menu" provided for a method user; and, the lists of consistency rules in ViewPoint template work plans, become lists of selectable items in the ConsistencyChecker tools provided for ViewPoint specification developers.

Clearly then, a ViewPoint template definition can be regarded as a specification of the CASE tool required to support the development of ViewPoints instantiated from that template. The more precise and formal one can make such a template definition, the more a meta-CASE tool can generate supporting tools for that template. Since we have decided to describe parts of ViewPoints relatively informally, *The Viewer* only partly automates CASE tool generation. The remainder must be implemented in Smalltalk (but the architecture of *The Viewer* facilitates such an implementation).

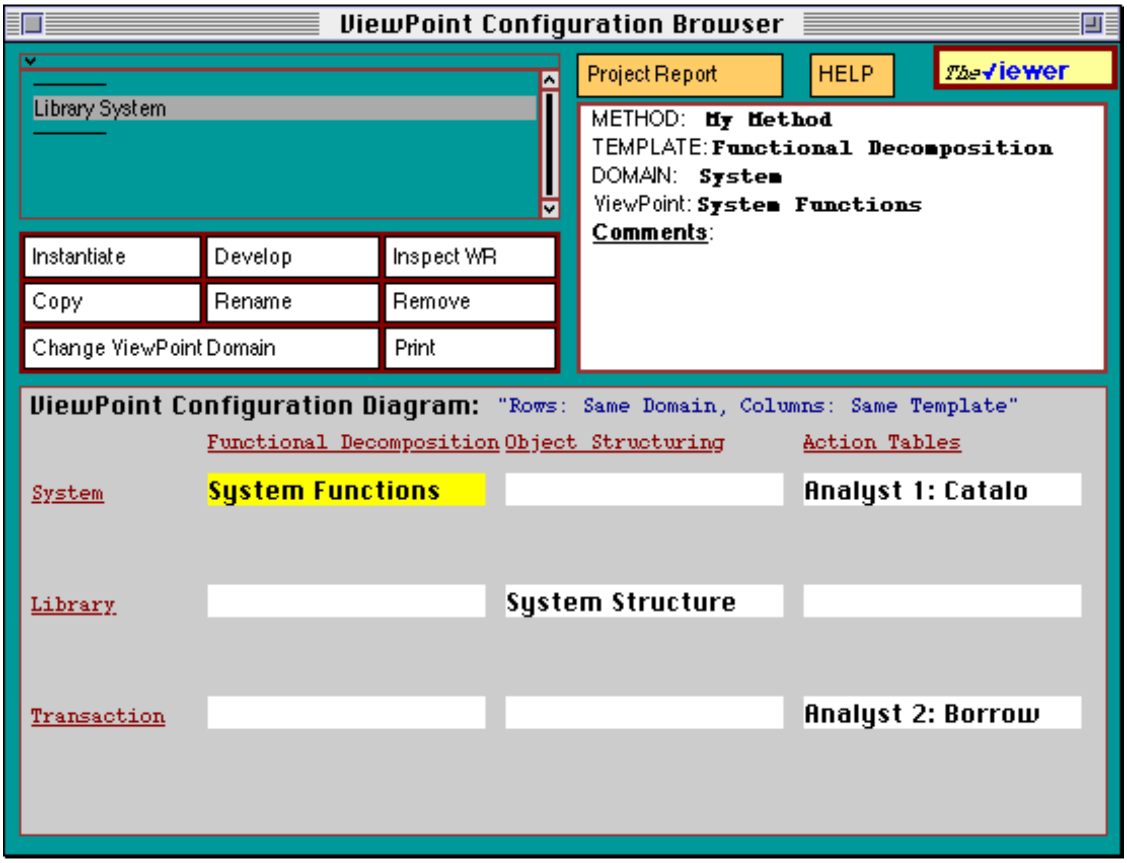
8.4. Method Use

The Viewer provides a wide range of tools for managing, developing and monitoring ViewPoints. We now describe the main tools briefly, and examine their scope, functionality and limitations.

8.4.1. ViewPoint Management and Development

Two main high level tools are provided for method users in *The Viewer* development environment, namely, a ViewPointConfigurationBrowser and a ViewPointInspector. A

ViewPointConfigurationBrowser (figure 8.5) is invoked from the startup window of *The Viewer*, and can be regarded as a simple management tool for manipulating and monitoring collections of ViewPoints in a software development project.



Library System

METHOD: My Method
TEMPLATE: Functional Decomposition
DOMAIN: System
ViewPoint: System Functions
Comments:

ViewPoint Configuration Diagram: "Rows: Same Domain, Columns: Same Template"

	Functional Decomposition	Object Structuring	Action Tables
System	System Functions		Analyst 1: Catalo
Library		System Structure	
Transaction			Analyst 2: Borrow

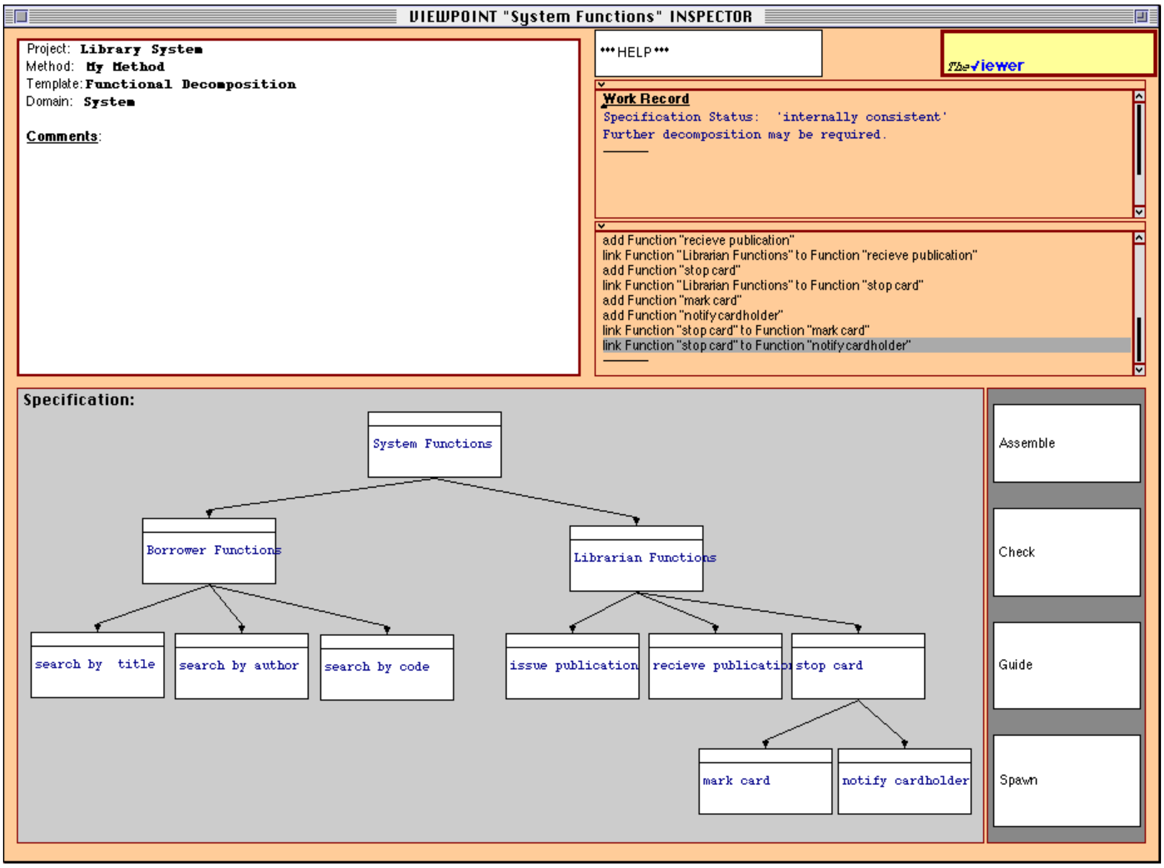
Figure 8.5: *The Viewer's ViewPointConfigurationBrowser.*

A ViewPointConfigurationBrowser lists loaded development projects in its top left-hand window pane, and displays the configuration of ViewPoints in a selected project in the bottom pane. The configuration is presented as a simple rectangular lattice, with columns denoting ViewPoints instantiated from different templates, and rows denoting ViewPoints concerned with different domains[29].

ViewPoints created in a development project are labelled and placed in their appropriate position in the rectangular lattice (this position is determined by the template from which these ViewPoints were instantiated, and the domain with which they are concerned). ViewPoints that can be, but have not been, created - given the current collection of templates and domains - are marked by blank rectangles in the lattice.

[29] This is somewhat of a simplification. There may be different "levels" of ViewPoints which need not be viewed all at once. Moreover, for large projects involving hundreds, even thousands, of ViewPoints, such rectangular lattices do not scale-up. These problems may be alleviated by "layering" ViewPoint configuration structures, and using, say, hypertext tools to navigate around these large structures.

Figure 8.5 shows a ViewPointConfigurationBrowser loaded with a single ("Library System") project which has been selected. The ViewPoint configuration for that project is also shown. Any ViewPoint in the configuration may be selected and developed further. This is done by selecting the appropriate ViewPoint and clicking on the "Develop" button. This in turn invokes a ViewPointInspector, an example of which is shown in figure 8.6.



Project: **Library System**
Method: **My Method**
Template: **Functional Decomposition**
Domain: **System**

Comments:

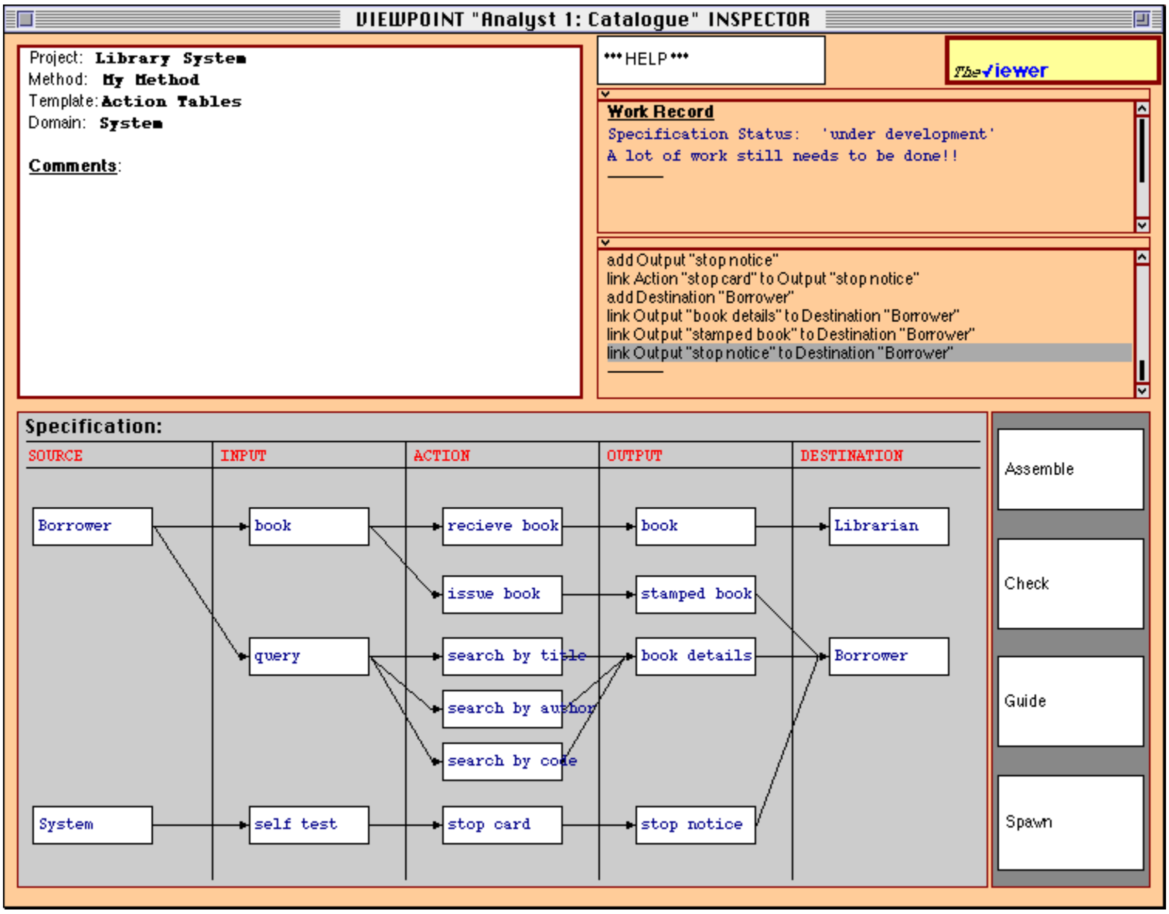
Work Record
Specification Status: 'internally consistent'
Further decomposition may be required.

add Function "recieve publication"
link Function "Librarian Functions" to Function "recieve publication"
add Function "stop card"
link Function "Librarian Functions" to Function "stop card"
add Function "mark card"
add Function "notify cardholder"
link Function "stop card" to Function "mark card"
link Function "stop card" to Function "notify cardholder"

Specification:

***Figure 8.6:** A sample ViewPointInspector for a ViewPoint instantiated from a functional decomposition template.*

For consistency of user interface and convenience, we have used the same kind of ViewPointInspector for all ViewPoints developed by *The Viewer*. Naturally, one may want to invoke a completely different kind of development tool for different kind of ViewPoints; e.g., a "text editor" ViewPoint based on vi or emacs. The only variations we provide for in the current prototype of *The Viewer*, are in the development buttons shown on the bottom right-hand side of the ViewPointInspector (these variations result in different menus generated by *The Viewer's* meta-CASE functions). Thus, the ViewPointInspector shown in figure 8.7, has the same basic structure as that shown in figure 8.6, except that the various editing (assembly), checking and help (guidance) buttons, pop-up different menus.



Project: **Library System**
Method: **My Method**
Template: **Action Tables**
Domain: **System**

Comments:

Work Record
Specification Status: 'under development'
A lot of work still needs to be done!!

add Output "stop notice"
link Action "stop card" to Output "stop notice"
add Destination "Borrower"
link Output "book details" to Destination "Borrower"
link Output "stamped book" to Destination "Borrower"
link Output "stop notice" to Destination "Borrower"

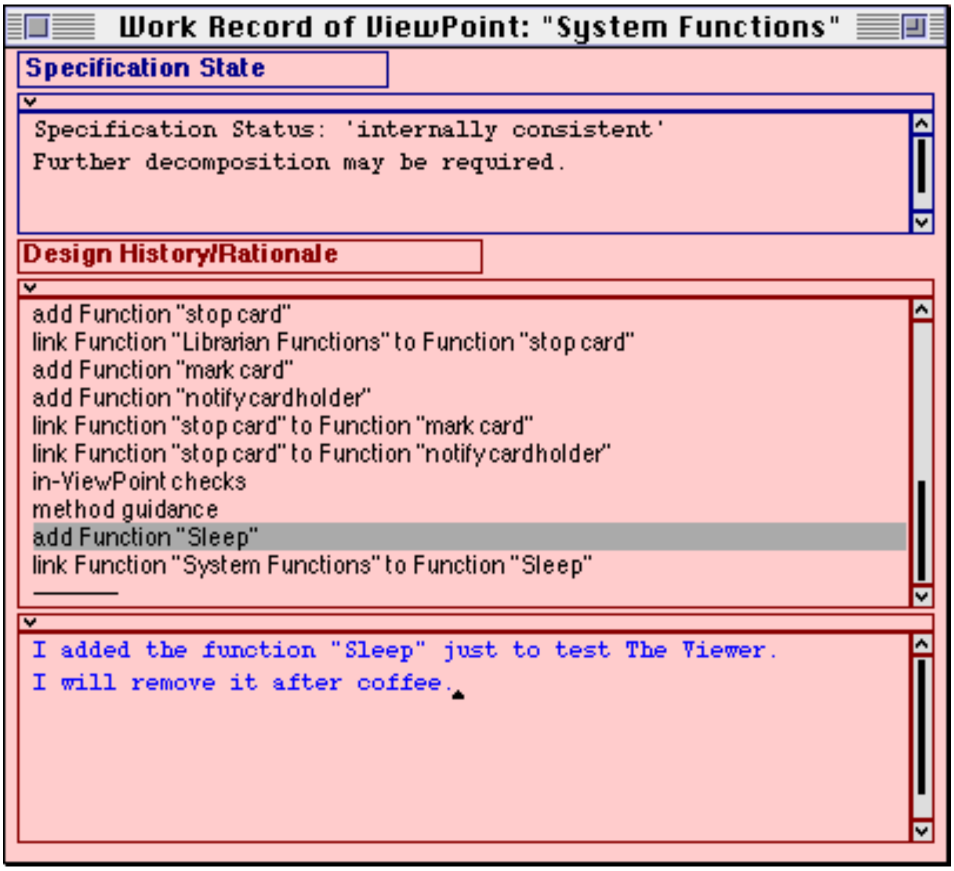
Figure 8.7: *A sample ViewPointInspector for a ViewPoint instantiated from an action tables template.*

8.4.2. Development Record and Monitoring

The top right-hand window panes of a ViewPointInspector display the work record of the ViewPoint being developed. The basic elements of a work record are the actions that have been performed in reaching the current specification state. These actions may be annotated individually with text, to provide a ViewPoint development rationale. These annotations may be made by a ViewPoint owner, or may be added automatically by *The Viewer*, whenever "useful" information is gathered during development (for example, the results of consistency checks are automatically annotated to "check actions", time stamps are annotated to ViewPoint trigger actions, and any guidance that is given to developers is also recorded). Work records may also be globally annotated (top right-hand window pane of ViewPointInspector) should the developer wish to make general comments about the status of a ViewPoint.

From an individual developer's point of view, having this information in the work record is useful for explaining why part actions were performed (perhaps by other developers who were responsible for the ViewPoint in the past). Work record information may also potentially be analysed to help handle new inconsistencies that arise. For example, if a ViewPoint specification was found to be consistent some time in the past, but is now no longer consistent, then analysis of actions performed since the last state of consistency may be helpful in identifying the cause of the new inconsistency. Identifying the *cause* of an inconsistency, as we saw in chapter 7, is an important step in the process of inconsistency handling.

From a manager's point of view, the work record also provides useful monitoring information about individual developers. A manager (at the level of the ViewPointConfigurationBrowser) can inspect individual ViewPoint work records as shown in figure 8.8. The list of actions performed by developers of the selected ViewPoint are listed (in the middle window pane), and each may be selected and its annotation (bottom window pane) inspected to see what the developers "have been up to". Of course, not all managers may want to inspect ViewPoints at this fine level of granularity, and so one can envisage "collapsing" a work record into high level actions such as "edit", "check" and so on.



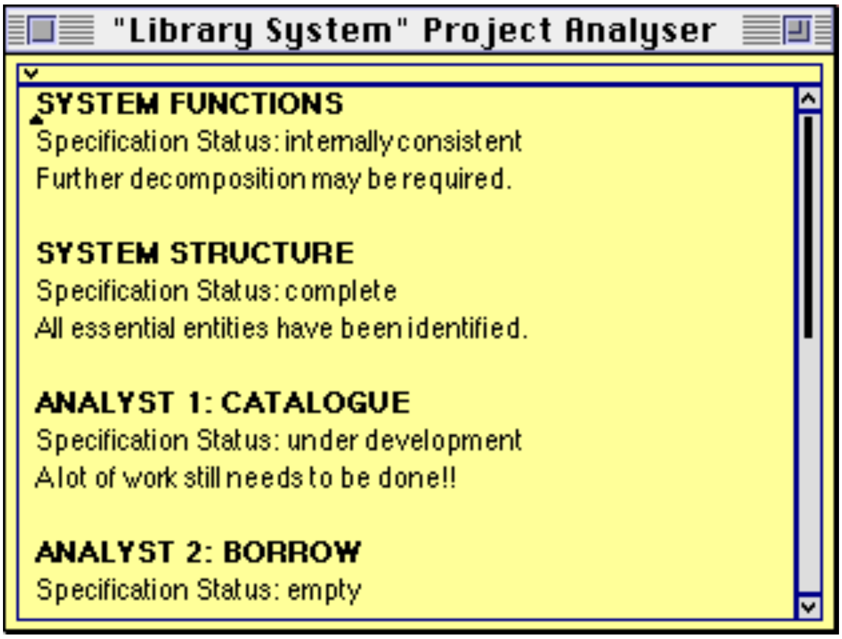
```
Specification Status: 'internally consistent'
Further decomposition may be required.
```

add Function "stop card"
link Function "Librarian Functions" to Function "stop card"
add Function "mark card"
add Function "notify cardholder"
link Function "stop card" to Function "mark card"
link Function "stop card" to Function "notify cardholder"
in-ViewPoint checks
method guidance
add Function "Sleep"
link Function "System Functions" to Function "Sleep"

I added the function "Sleep" just to test The Viewer.
I will remove it after coffee.

Figure 8.8: *Monitoring a ViewPoint work record.*

A project manager may also wish to generate project reports based on information residing in a collection of ViewPoints. This can also be done at the ViewPointConfigurationBrowser level, a simple example of which is shown in figure 8.9. The report in this case lists all ViewPoints in the project configuration and their global work record annotations. One can envisage a hierarchically numbered report, with ViewPoint specifications included, and more sophisticated formatting capabilities provided.



SYSTEM FUNCTIONS
Specification Status: internally consistent
Further decomposition may be required.

SYSTEM STRUCTURE
Specification Status: complete
All essential entities have been identified.

ANALYST 1: CATALOGUE
Specification Status: under development
A lot of work still needs to be done!!

ANALYST 2: BORROW
Specification Status: empty

Figure 8.9: *Generating reports.*

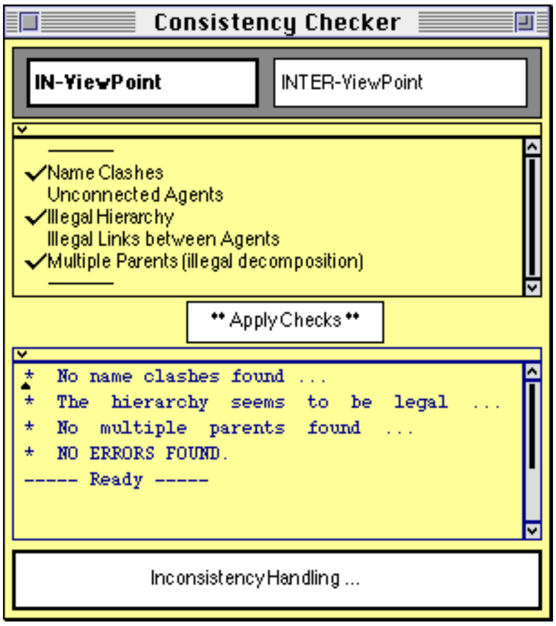
8.4.3. Consistency Checking

As we have emphasised throughout this thesis, we believe that a major part of any development process is checking and achieving different levels of consistency. In the ViewPoints framework, consistency may be checked internally within a ViewPoint, or externally across ViewPoints. This is reflected by the simple ConsistencyChecker provided by *The Viewer*, shown in figure 8.10.

The ConsistencyChecker is used to check the in- and inter-ViewPoint rules described in the selected ViewPoint's template. It is important to highlight that the checks are not performed automatically, can be checked selectively (i.e., all or some), and development can continue even if one or more of the checks fail.

The ConsistencyChecker allows users to select the scope of checking (in- or inter-ViewPoint), and then select the rules to be checked. Clicking the "Apply Checks" button starts the checking process and the results are displayed in the bottom window pane.

In our current prototype implementation of *The Viewer*, we have clearly separated the application of consistency rules from the process of inconsistency handling. We have *not* implemented the logical action-based inconsistency handling framework described in chapter 7 (section 7.4.2), but we have implemented prototype demonstrators of the kinds of inconsistency handling tools we expect to produce. These tools may be invoked by clicking on the "Inconsistency Handling" button located at the bottom of the ConsistencyChecker tool - when one or more of the applied consistency checks have failed.



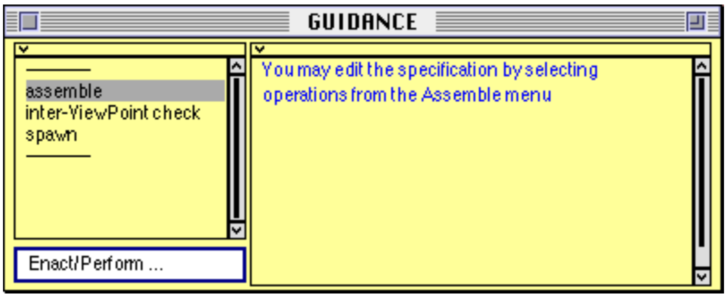
✓Name Clashes
Unconnected Agents
✓Illegal Hierarchy
Illegal Links between Agents
✓Multiple Parents (illegal decomposition)

```
*  No name clashes found ...
*  The hierarchy seems to be legal ...
*  No multiple parents found ...
*  NO ERRORS FOUND.
----- Ready -----
```

Figure 8.10: *The Viewer's ConsistencyChecker.*

8.4.4. Method Guidance

Rather than allow a process model to drive the ViewPoint development process automatically, *The Viewer* uses it to *guide* ViewPoint developers. A ViewPoint developer can request guidance at any point during development, and is provided with context-sensitive help of the form shown in figure 8.11. Depending on the state of the ViewPoint (a function of the preconditions that hold at any particular moment), a number of possible or recommended actions are suggested by the guidance tool. An explanation of each action is also available (since this was provided by the method engineer during process modelling (figure 8.4).



You may edit the specification by selecting operations from the Assemble menu

Figure 8.11: *Method guidance.*

The role of guidance in this context is to support development and not necessarily to automate it. Therefore, the guidance provided may be totally ignored by a ViewPoint developer. However, should a developer wish to adopt the recommendations suggested, then the tool also proves a means of guiding the developer through the process of implementing any of the suggested actions. For example, a developer may select a particular recommended action and ask the guidance tool to "Enact/Perform" that action, whereupon the appropriate menu or tool is invoked from within the guidance tool.

8.4.5. Dynamic Evolution

Finally, *The Viewer* allows new ViewPoints to be "spawned" or created from within individual ViewPoints. This may be useful if, for example, one ViewPoint becomes too large and complicated and needs to be decomposed, or if, say, analysis using a different development technique is required. Alternatively, another perspective on the problem may exist and a new ViewPoint may be necessary to explore this different perspective.

A manager, operating at the ViewPointConfigurationBrowser level, may instantiate a template to create a new ViewPoint if required, however, individual developers also have the option of doing so by means of the "Spawn" button on the ViewPointInspector. New ViewPoints can therefore be created dynamically from within any ViewPoint, and developers have the option of moving on to develop these further, or to simply leave them blank for later or for other developers to elaborate.

Further work is needed in order to combine, in *The Viewer*, the existence rules described in chapter 6 (section 6.3.2.1), with the kinds of ViewPoint process models described in chapter 7 (section 7.3). Such a combination would provide ViewPoint developers with context-sensitive guidance that advises them on the appropriate time and kind of new ViewPoints they need to create. It is also important because most methods will deploy systematic approaches to creating new ViewPoints (rather than leaving this to the discretion of individual developers).

8.5. Architecture and Implementation

We implemented *The Viewer* in Objectworks/Smalltalk for a number of conceptual and pragmatic reasons:

- As we discussed in chapter 4 (section 4.4.3.1), the ViewPoints framework is object-based, with ViewPoint templates corresponding to object classes and ViewPoints corresponding to objects. This allowed the convenient representation of ViewPoints as objects in the Smalltalk programming environment.
- The Smalltalk environment is particularly suitable for rapid prototyping and exploratory programming, which facilitated the evolutionary development of the ViewPoints framework in conjunction with *The Viewer* implementation.
- The availability of Smalltalk on a number of different platforms enhanced portability. Thus, *The Viewer* was developed on an Apple Macintosh, and runs on UNIX workstations running X-Windows and IBM PCs running MS-Windows.
- For users of *The Viewer*, the Smalltalk environment provides a relatively easy way to extend and add (implement) additional CASE tools.
- We already had experience in implementing graphical CASE tools and consistency checking mechanisms in Smalltalk [Nuseibeh 1989].

The Viewer uses a very simple approach to tool implementation. Currently all the ViewPoint template support tools are implemented in, and depend upon, Smalltalk - with the architecture allowing for their "easy" addition, modification and extension. Abstract classes are available for rapid development of skeleton tools, with meta-CASE mechanisms implemented for the incorporation of textual and graphical information from template descriptions into these tools.

The Viewer is based on a "conventional" Smalltalk implementation architecture. Most of the tools have a Model-View-Controller (MVC) [Krasner & Pope 1988] architecture to model their behaviour, presentation and user-interface, respectively. In an MVC application, the Model represents the application engine that contains, and operates upon, the application's data. The View denotes the visual representation of the data generated by the Model. Finally, the Controller provides the interface between users and an application (Model), and is normally associated with one or more Views. Generally speaking, an application model has multiple Views and Controllers - although "non-conventional" combinations are also possible.

The objects which the various tools manipulate are also of particular importance. A class Template was implemented as a subclass of Object, with two principle instance variables style and workPlan. Class ViewPoint was implemented as a subclass of Template with the addition of three more instance variables domain, specification and workRecord (figure 8.12). Clearly, these classes directly model the ViewPoints framework - a widely accepted benefit of object-oriented programming. Moreover, the direct mapping of ViewPoints onto Smalltalk objects facilitates the incorporation of exploratory modifications of the ViewPoints framework into *The Viewer* environment (such as the extensions for integration support discussed in section 8.7).

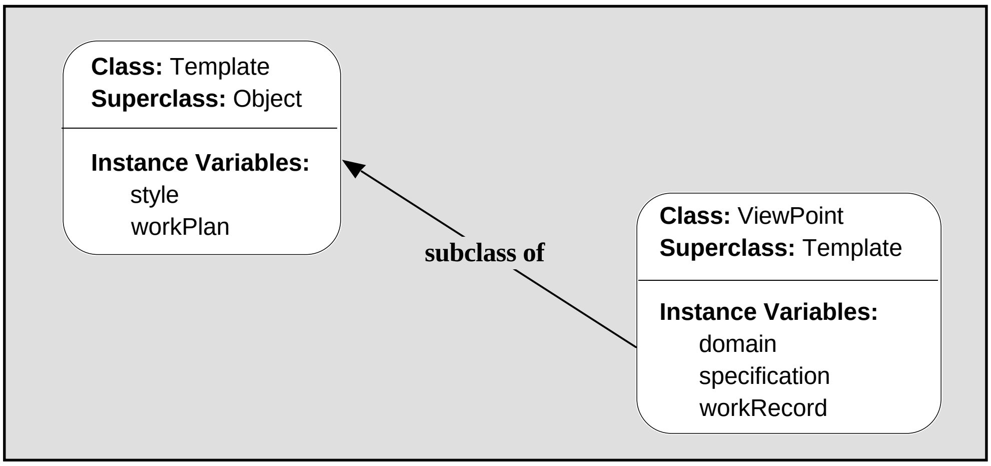


Figure 8.12: *The Viewer implements the class "Template" and its subclass "ViewPoint" which correspond directly to objects manipulated in the ViewPoints framework.*

Templates and ViewPoints are stored in separate Smalltalk databases which may be selectively accessed by TemplateBrowsers and ViewPointConfigurationBrowsers, respectively. These databases may also be saved to files and exported across Smalltalk images (i.e., between copies of *The Viewer*).

The Viewer is a large prototype Smalltalk application implementing over 45 classes and 15,000 lines of code. It employs a number of different clusters of classes and MVC triads - the most common being a single Model and Controller with multiple Views (window panes). Frequently however, Views also carry a substantial part of the Model functionality, to allow many Views to share a single Model (e.g., the specification Views of ViewPointInspectors). The Controller is sometimes also combined with its View, particularly when direct interaction with the View is required; e.g., in the case of ViewPointInspectors which allow users to manipulate specifications directly using the mouse (e.g., moving diagrams around). Figure 8.13 shows the top level architecture of *The Viewer.*

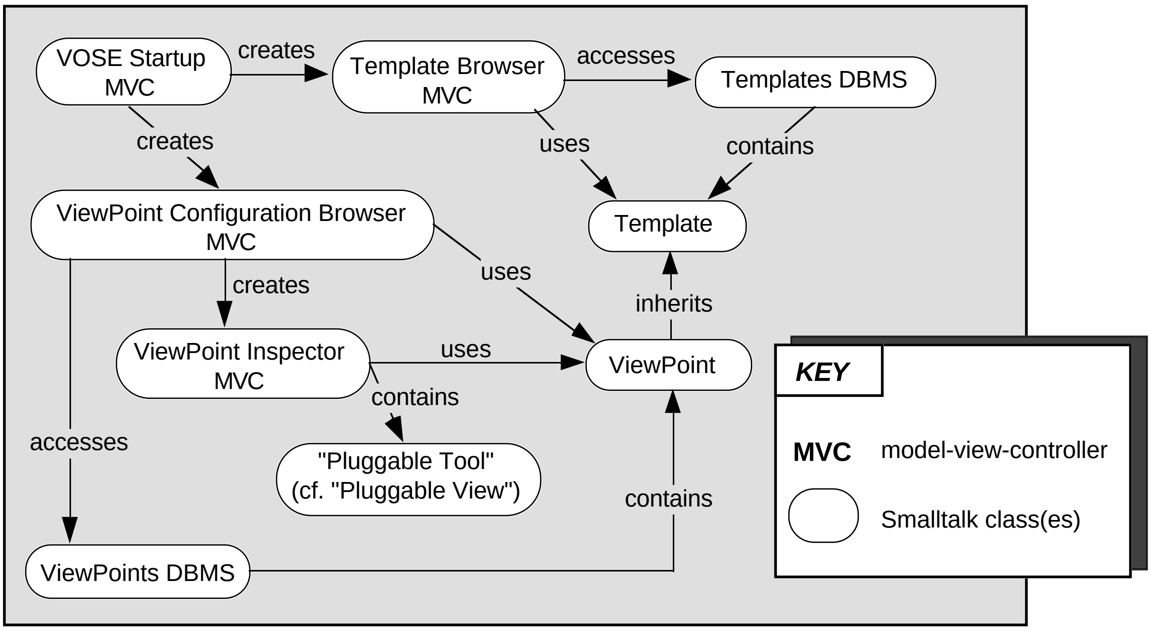


Figure 8.13: *The top level architecture of The Viewer.*

Note that *The Viewer's* implementation approach to tool integration, is analogous to Smalltalk's approach to having "pluggable views". Effectively, adding or integrating a tool into *The Viewer*, is achieved by "plugging" a View into the MVC of the ViewPointInspector.

8.6. Limitations

The primary objective of implementing *The Viewer* was to use it as a vehicle for demonstrating the feasibility, and consequences, of adopting a multi-perspective software development approach, exemplified by ViewPoint-oriented development. As such, *The Viewer* was constructed as a "proof-of-concept" environment, which was gradually populated with a variety of simple tools to support the ViewPoints framework and illustrate the benefits of using it to develop complex systems. *The Viewer* clearly achieves its objectives and demonstrates how method engineering and method use take place in this setting.

The Viewer however, has a number of limitations which we briefly outline below.

- *Problems of scale.* It is not clear how *The Viewer*, and possibly the ViewPoints framework as a whole, will scale up to industrial size projects. Currently *The Viewer* is geared towards managing tens rather than hundreds or even thousands of ViewPoints, so clearly the visualisation of ViewPoint configurations needs to be reviewed. This also requires better navigation tools to inspect large ViewPoint configurations structures.
- *Distribution.* The current implementation of *The Viewer*, while demonstrating a decentralised, multi-perspective software development approach, is nevertheless a single-user implementation. Implementing *The Viewer* across a distributed system, while beyond the scope of this thesis, is necessary to for realistic use of the tool. Of course, *The Viewer* itself need not be distributed, but rather a specialised distributed system could be extended to support ViewPoint-oriented architectures and other ViewPoint structuring concepts.
- *Tool integration. The Viewer* is currently a standalone environment (as is the Smalltalk environment), and integration with other (e.g., UNIX or CASE) tools is not specifically supported. *The Viewer* needs to be extended to be able to communicate and exchange information with external tools. A pragmatic approach to addressing this limitation however, may be able to use one of many “standard” interchange formats to import or export data between tools. This, for example, could be achieved by having an “interchange template” whose inter-ViewPoint rules translate between formats (effectively, one can “simulate” a centralised architecture as one possible approach to achieving tool integration).
- *Process Modelling. The Viewer’s* process modelling capabilities are primitive - as one would expect from essentially a simplified state-transition approach. Only simple processes can be modelled, and there is no provision for inevitable process evolution. Process modelling in this context is clearly an area that requires further work. Currently, even the simple process models described by the ProcessModeller tool have been hard-coded into *The Viewer* for convenience. This is something we should clearly be avoiding, and we already exploring ways of dynamically “deriving” ViewPoint states which can then be used to coordinate internal ViewPoint development and inter-ViewPoint communication.
- *Method Engineering.* In general method engineering is a meta-modelling process in which a method is constructed from a number of modular “types” (the ViewPoint templates). Currently, *The Viewer* does not provide any specialised meta-modelling tools such as graphical editors to build configurations of templates and to specify their relationships at the “meta” (type) level.
- *Development Rationale. The Viewer* adopts a simple approach to recording development rationales. It relies on recording development actions and annotating them with “explanations”. This is clearly the simplest approach and we may want to extend this so that actions are annotated with more structured “rationale languages” (as in gIBIS for example). The ability to annotate an actual specification (as opposed to the actions used to develop it) may also be useful. The only danger here however, is that removing an (annotated) item from

a specification also eliminates the annotation itself (i.e., the rationale is also removed).

- *Version Control. The Viewer* has no version control. In fact it is not clear at this stage what role, if any, of version control is in the context of the ViewPoints framework. For example, the should a new version of a ViewPoint be regarded as an alternative version or a new ViewPoint altogether? The latter option does not appear to be problematic, since we can always create new ViewPoints. To manage versions of a single ViewPoint however is more difficult. One solution would be to use the work record to store different "traces" of development representing different ViewPoint versions. These traces could then be "replayed" to access the required version[30].

Despite the above limitations, *The Viewer* illustrates the feasibility of populating the ViewPoints framework with more sophisticated or emerging tools and techniques if and when they are required or available. In fact, what *The Viewer* does provide is a series of "hooks" or "slots" into which a whole range of software engineering technologies may be placed and addressed independently. As such, the above limitations may be regarded as a software engineering research agenda, which we believe has been clarified and simplified when viewed in the context of the ViewPoints framework.

8.7. Towards Improved Integration Support: Inconsistency Handling

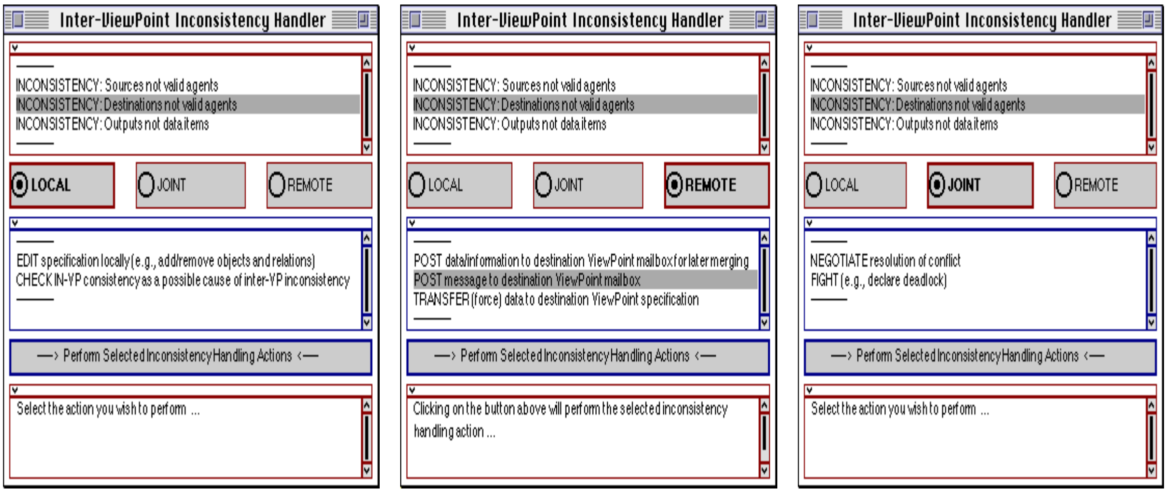
The key to building an integrated system from multiple perspectives in the ViewPoints framework, is the ability to express, invoke and apply relationships between ViewPoints. These relationships represent the integration "glue", used to integrate methods during method design, and actual partial specifications during method use. Since integration in our framework are achieved incrementally, inconsistencies are tolerated and managed.

In this context, automated tool support for integration in the framework must be the capable of categorising ViewPoints and their associated checks, then checking consistency and managing inconsistencies between them. The former capabilities are provided by our current implementation of *The Viewer*, while the latter may be achieved by providing inconsistency handling tools that guide developers in the presence of inconsistencies.

We have not implemented the inconsistency handling mechanisms described earlier in this thesis (section 7.4). We have however produced user interface prototypes of the kinds of tools we believe developers would find desirable in this setting. One such tool is the inter-ViewPoint InconsistencyHandler shown in figure 8.14. The figure illustrates the scope and some of the

[30] Pushing the notion of replay even further, one could imagine replaying different portions of a work record to examine alternative solutions. In such cases however, it becomes more difficult to justify not creating alternative ViewPoints to explore these alternative solutions.

difficulties involved in managing inconsistencies between ViewPoints. One or more actions may be required to *handle* (which subsumes *resolve*) inconsistencies between ViewPoints. The handling of such inconsistencies may be local to either of the two inconsistent ViewPoints (figure 8.14a & b), or may require joint handling (figure 8.14c). Some actions may be "corrective" (i.e., they *resolve* an inconsistency completely), while others may be "constructive" (i.e., they *improve* the current situation or "reduce" the extent of an inconsistency). External tools may also be needed, or simply some off-line negotiation.


Inter-ViewPoint Inconsistency Handler
● LOCAL ○ JOINT ○ REMOTE
○ LOCAL ○ JOINT ● REMOTE
○ LOCAL ● JOINT ○ REMOTE
—> Perform Selected Inconsistency Handling Actions <—

INCONSISTENCY: Sources not valid agents
INCONSISTENCY: Destinations not valid agents
INCONSISTENCY: Outputs not data items

EDIT specification locally (e.g., add/remove objects and relations)
CHECK IN-VP consistency as a possible cause of inter-VP inconsistency

Select the action you wish to perform ...

POST data/information to destination ViewPoint mailbox for later merging
POST message to destination ViewPoint mailbox
TRANSFER (force) data to destination ViewPoint specification

Clicking on the button above will perform the selected inconsistency handling action ...

NEGOTIATE resolution of conflict
FIGHT (e.g., declare deadlock)

Select the action you wish to perform ...

***(a)** Local handling* ***(b)** Remote handling* ***(c)** Joint handling*

***Figure 8.14:** An Inter-ViewPoint Inconsistency Handler.*

Inconsistency handling in this context is clearly a rich area for future work. Its role is to manage the "links" (consistency relationships) between ViewPoints, and is therefore critical to achieving closer integration between these ViewPoints.

8.8. Chapter Summary

This chapter has described *The Viewer*, a prototype environment supporting the ViewPoints framework. *The Viewer's* support of method engineering, method integration and method use was also examined, and its meta-CASE capabilities demonstrated. *The Viewer's* role as an agenda of software engineering concerns was also described, and its contributions to resolving a number of these issues was explored. Modular extensions to *The Viewer*, such as an inter-ViewPoint InconsistencyHandler, were also proposed and used to highlight a potentially fruitful research path to achieving improved integration in the framework.

Chapter 9 Case Study and Evaluation

This chapter demonstrates the use of the ViewPoints framework by both method engineers and method users. While various aspects of the framework have already been illustrated by fragmentary examples in previous chapters, the goal of this chapter is to bring these fragments together in the form of a single case study. The requirements specification method CORE is used to demonstrate method engineering in the framework, while its application to the development of a computer-based library system is used to demonstrate method deployment. The case study, and a variety of further examples, are used to build a body of experiences in using the framework. We critically evaluate these experiences and discuss the scope for technology transfer of ViewPoint concepts into an industrial setting. We review other examples, case studies and tools that also demonstrate the applicability of our approach.

The ViewPoints framework addresses a wide range of software engineering concerns. It is therefore difficult for a single case study to capture all aspects of the framework or to evaluate all the claimed benefits of the approach. While previous chapters have examined individual concerns addressed by the framework, this chapter serves to demonstrate that these concerns are collectively pertinent to multi-perspective software development as exemplified by the ViewPoints approach.

9.1. Case Study

The nature and scope of the ViewPoints framework is such that different methods and domains illustrate different concepts better than others. Nevertheless, we have chosen a method, CORE, and a problem domain, a library, that broadly cover the range of issues under investigation. Since the ViewPoints framework is aimed at supporting a multi-perspective development process in which multiple inconsistent ViewPoints may co-exist, one way to represent such a process is to present ViewPoints (*c.f.* partial specifications) at various stages of elaboration.

9.1.1. Choice of Method

The CORE method is a particularly interesting, and we argue typical, example of a software engineering method for a number of reasons:

- It prescribes a number of distinct stages of development, based on traditional notions of decomposition and separation of concerns;
- It deploys a number of different representation schemes that are used to describe the different stages or aspects of development;
- It prescribes a range of consistency relations within and between stages of development;
- Many of the notations are highly redundant resulting in overlaps between descriptions, and increasing the scope for consistency checking;
- The representation schemes deployed are mostly graphical and the notion of a client authority or stakeholder is explicit.

In the case study that follows, we only describe and deploy three graphical stages of CORE in detail, namely, agent structuring, tabular collection and data structuring. We do however, outline how they are related to the other stages of CORE. The problem domain we specify is a simplified "library system".

9.1.2. Problem Definition

The specification of libraries is a popular software engineering exemplar because the domain contains a variety of users, stakeholders, resources, events and actions. A library contains publications (such as books and journals), that may be borrowed by users of the library. A single publication title may have multiple copies that may be borrowed. Users also include library staff who manage the library.

A computer-based library system is required to record all publications in the library, to facilitate publication searches, and to record the issuing and returning of publications to and from borrowers, respectively. Overdue publications incur fines, after reminders have been issued, while other publications are not permitted to be borrowed at all.

We have left the problem definition deliberately short and incomplete, to demonstrate how a requirements engineering method such as CORE is used to refine and clarify the problem definition and systems requirements. While CORE deploys a number of stages and notations to facilitate the requirements specification and analysis process, we also demonstrate in this case study that the organisational structure added by our ViewPoint-Oriented Software Engineering (VOSE) framework, facilitates multi-perspective development even further.

9.1.3. ViewPoint-Oriented Method Engineering: CORE

To describe CORE in the ViewPoints framework, we must chose and elaborate a collection of ViewPoint templates that comprise the method. Since CORE itself comprises seven clearly defined stages, we also chose our templates based on these different stages. These stages (and consequently templates) are:

- *Problem Definition (PD),* in which a statement of objectives is described textually. This includes the identification of the customer authority, business objectives, current system problems, new systems features, costs, time scales and so on.
- *Agent Structuring (AS),* in which the problem boundaries are identified by identifying information processing entities (“agents”), which are then organised into a hierarchy[31].
- *Tabular Collection (TC),* in which the actions performed by each agent are identified, including the source agents of their inputs and the destination agents of their outputs. This stage provides a means of analysing information on data flows and data transformations, presented in a tabular form.
- *Data Structuring (DS),* in which the outputs produced by each agent are analysed by identifying these data items, deriving the order in which they are produced, and the number of times and the conditions under which they are produced. These data items are also decomposed into their constituent components and organised into a hierarchy.
- *Single Agent Modelling (SAM),* in which information from the tabular collection stage is directly mapped into an alternative (non-tabular) representation scheme that additionally describes the control and ordering of actions, and the internal data flows within each agent.
- *Combined Agent Modelling (CAM),* in which the SAM representation scheme above is used to describe “scenarios” or transactions that involve more than one agent. This allows the analysis of particularly critical or important aspects of a system, rather than examining all possible transactions.
- *Constraints Analysis (CA),* in which a variety of non-functional constraints or requirements are described textually. While the understanding gained from previous specification and analysis stages is helpful in clarifying system constraints, there is no direct mapping between any of the previous stages and constraints analysis.

The PD and CA templates may be simply described as “text” templates (i.e., the representation scheme they deploy is “English” and their work plans simply describe the usage (grammar) of the English language. It is clearly not feasible nor useful to define detailed templates for these stages in our case study. However, one may want to specify a rule that ensures that at least one ViewPoint from each template is created in the life-time of a project. A greatly simplified description of the text-based PD template, is shown in figure 9.1. Note in particular the inter-ViewPoint rule that specifies the need for an agent structuring diagram to exist, once any form of problem definition has been produced.

[31] Recall our use of the term “agents” in place of CORE’s “viewpoints” to avoid the clash in nomenclature.

TEMPLATE: PD
STYLE
English language alphabet, α
WORK PLAN
Assembly Actions add(α), remove(α)
In-ViewPoint Check Actions English language grammar, γ
Inter-ViewPoint Check Actions (1) self $\rightarrow \exists$ VP(AS, D_S)
ViewPoint Trigger Actions instantiate(AS)
Inconsistency Handling Actions inter-ViewPoint rule (1) fails $\rightarrow$ NEXT(instantiate(AS))

Figure 9.1: *An outline description of a Problem Definition (PD) template in CORE.*

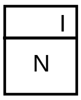
Figures 9.2, 9.3 and 9.4 describe the agent structuring (AS), tabular collection (TC) and data structuring (DS) templates, respectively. Some of the representation schemes used have been simplified for clarity (for example, CORE actually has four different kinds of agents: direct, indirect, parallel direct and parallel indirect), and a number of consistency checks have also been omitted. The SAM and CAM templates may similarly be defined, but are not included in our case study.

Recall that the process of identifying and defining a method's constituent ViewPoint templates, is a process of method engineering. Only after such a process has been completed can we actually specify the library system using our method. Also note that the template descriptions shown are not complete, but illustrative of the different kinds of information one would expect to define in each ViewPoint template. We will elaborate and explain the contents of the different templates in the following sections, when these templates are deployed to specify the library system.

TEMPLATE: AS

STYLE

Object: Agent

Attributes	*Types*	*Values*
Name (N)	String	
Identifier (I)	Number	
Icon	Bitmap	

Relation: part-of(Agent, Agent)

Attributes	*Types*	*Values*
Icon	Bitmap	↘

WORK PLAN

Assembly Actions

add-agent, remove-agent, link(part-of(Agent, Agent)), unlink(part-of(Agent, Agent))

In-ViewPoint Check Actions

One root agent, No agent name clashes, Child agent has only one parent, Unique agent identifiers

Inter-ViewPoint Check Actions

(1) self $\rightarrow \neg\exists$ VP(AS, D_a)

(2) Agent $\rightarrow \exists$ VP(TC, Agent.Name)

(3) Agent $\rightarrow \exists$ VP(DS, Agent.Name)

(4) Agent $\rightarrow \exists$ VP(SAM, Agent.Name)

ViewPoint Trigger Actions

instantiate(TC), instantiate(DS), instantiate(SAM)

Inconsistency Handling Actions

name clash → NEXT(ask user to check spelling)

name clash and ¬LAST[1](spelling error) → NEXT(ask user to resolve conflict)

inter-ViewPoint rule (2) fails → NEXT(instantiate(TC))

inter-ViewPoint rule (3) fails → NEXT(instantiate(DS))

inter-ViewPoint rule (4) fails → NEXT(instantiate(SAM))

Figure 9.2: *An outline description of an Agent Structuring (AS) template in CORE.*

TEMPLATE: TC

STYLE

Attributes	*Types*	*Values*
Object: Input		
Name (N)	String	
Icon	Bitmap	$\boxed{N}$
Object: Input		
Name (N)	String	
Icon	Bitmap	$\boxed{N}$
Object: Action		
Name (N)	String	
Icon	Bitmap	$\boxed{N}$
Object: Output		
Name (N)	String	
Icon	Bitmap	$\boxed{N}$
Object: Destination		
Name (N)	String	
Icon	Bitmap	$\boxed{N}$
Relation: connected-to(Object, Object)		
Icon	Bitmap	⟶

WORK PLAN

Assembly Actions

add-source, add-input, add-action, add-output, add-destination, remove-source, remove-input, remove-action, remove-output, remove-destination, link(connected-to(Object, Object)), unlink(connected-to(Object, Object))

In-ViewPoint Check Actions

No name clashes, No internal flows, source-connected-to-input, input-connected-to-action, action-connected-to-output, output-connected-to-destination

Inter-ViewPoint Check Actions

(1) Source.Name = VP(AH, D_d): Agent.Name

(2) Destination.Name = VP(AH, D_d): Agent.Name

(3) connected-to(Output, Destination).Output.Name =
VP(TC, Destination.Name): connected-to(D_S, Input).Input.Name

(4) connected-to(Source, Input).Input.Name =
VP(TC, Source.Name): connected-to(Output, D_S).Output.Name

(5) Output.Name = VP(DS, D_S): Data-Item.Name

ViewPoint Trigger Actions

instantiate(TC), instantiate(DS)

Inconsistency Handling Actions

name clash → NEXT(ask user to check spelling)

inter-ViewPoint rule (3) fails → NEXT(instantiate(TC))

inter-ViewPoint rule (3) fails → NEXT(remove-output)

inter-ViewPoint rule (1) fails → NEXT(remove-source)

Figure 9.3: *An outline description of a Tabular Collection (TC) template in CORE.*

TEMPLATE: DS		
STYLE		
Object: Data-Item		
Attributes	*Types*	*Values*
Name (N) Annotation (A) Icon	String Symbol Bitmap	 {*, , o} A N
Relation: part-of(Data-Item, Data-Item)		
Attributes	*Types*	*Values*
Icon	Bitmap	/
WORK PLAN		
Assembly Actions add-data-item, remove-data-item, link(part-of(Data-Item, Data-Item)), unlink(part-of(Data-Item, Data-Item)), add-annotation(Data-Item), remove-annotation(Data-Item)		
In-ViewPoint Check Actions One root, No data item name clashes, Child data item has only one parent		
Inter-ViewPoint Check Actions (1) Data-Item.Name = VP(TC, D_S): Output.Name		
ViewPoint Trigger Actions instantiate(TC), instantiate(DS)		
Inconsistency Handling Actions name clash → NEXT(ask user to check spelling) inter-ViewPoint rule (1) fails → NEXT(instantiate(TC)) inter-ViewPoint rule (1) fails → NEXT(remove-data-item)		

Figure 9.4: *An outline description of an Data Structuring (DS) template in CORE.*

The development process of individual ViewPoint specifications using each of the above templates is relatively simple, where in general, any of the work plan actions may be performed at any stage during development. We can however, define simple process models, one in each template work plan, to prescribe a recommended ordering of these actions, and to provide simple method guidance.

The general entries that apply to any of the above ViewPoint templates include:

{ } → [assembly actions] {partial specification}

which simply says "if the specification is empty, then perform some assembly actions to create a partial specification". The "agent" that performs such assembly actions has been omitted, because it will always be the owner of the ViewPoint which is currently being developed. Another standard entry is:

{partial specification} → [assembly actions] {partial specification}

which states that the specification may always be edited. An important variation between the process models of the different templates is the timing of consistency checks. In general, we can have:

{partial specification} → [check actions] {partial specification}

which says that we can check consistency at any time. However, we may want to be more specific about the scope and kind of check, and the results of applying it. So, for the TC template for example, we may want to include the entry:

{consistent partial specification} → [inter-ViewPoint check action 1]
{consistent partial specification according to rule 1}
or {inconsistent partial specification according to rule 1}

Again, we have described the resultant specification states relatively informally and hence incompletely. The notion of a "partial specification state" may be derived from the set of actions already performed (i.e., those contained in the ViewPoint's work record). If we prescribe a number of such states (e.g., "internally consistent", "internally inconsistent according to rule 3", etc.), then we can be more specific about our process modelling entries and our inconsistency handling rules. A case study in which the work record is used to derive a ViewPoint specification state is described in [Leonhardt 1994], but is beyond the scope of this thesis.

One way to test for a "complete" specification may be to successfully perform the process model entry:

{all in- and inter-ViewPoint checks successfully performed on partial specification} → [ ] end.

In other words, if all consistency relationships are found to hold, then we can stop developing this ViewPoint. Of course, developing other ViewPoint specifications may result in inconsistencies with a "complete" ViewPoint, which may then have to be developed further in order to handle these inconsistencies. So in fact, completion in the framework is only a relative term - relative with respect to the list of consistency rules defined within each local ViewPoint.

9.1.4. Building Tool Support

Before we can deploy the templates defined above to specify the library system, we need to build CASE tools, to support development using these templates.

The template definitions produced at the end of the ViewPoint-oriented method engineering process can be used as declarative specification of the CASE tools required to support development using these templates. For example, *The Viewer* translates the bitmap attributes into icons that may be added, removed or modified during ViewPoint development. Moreover, lists of actions such as assembly actions are transformed into basic editing menus.

Thus, supported by these simple meta-CASE capabilities of *The Viewer*, we implemented simple CASE tools to support the three stages of CORE described by the AS, TC and DS templates. The graphical specification diagrams shown in the next section describing the library system are all screen dumps of specifications generated by *The Viewer's* tools (ViewPointInspectors).

Since these tools either describe hierarchical or tabular notations, they were simply constructed by sub-classing abstract classes in *The Viewer* that implemented hierarchical and tabular diagramming techniques. Terminology specific to CORE was then added, by modifying the appropriate Smalltalk methods. Moreover, in- and inter-ViewPoint checks specific to CORE were hard-coded into these subclasses. Again, the architecture of *The Viewer* allowed checks that tested for similar inconsistencies (such as name clashes of object names) to be shared in abstract classes.

A detailed description of how the various tools were implemented in *The Viewer* would detract from the usefulness of this case study, and is therefore omitted.

9.1.5. ViewPoint-Oriented Method Use: Library Specification

Once the CORE templates, and their supporting tools, have been defined, they can then be deployed to specify different systems. To specify our library system, we begin with an informal statement of requirements like the one described in section 9.1.2. In CORE, this statement may be documented by instantiating the initial problem definition template, PD, to produce a ViewPoint "LibrarySystem-PD", whose domain is "Library". An outline of this ViewPoint is shown in figure 9.5.

ViewPoint: LibrarySystem-PD

Template: PD

Domain: Library World

Specification:

Problem:
The library contains publications (such as books and periodicals), that may be borrowed by users of the library. A single publication title may have multiple copies that may be borrowed. Users also include library staff who manage the library.

A computer-based library system is required to record all publications in the library, to facilitate publication searches, and to record the issuing and returning of publications to and from borrowers, respectively. Overdue publications incur fines, after reminders have been issued, while other publications may not be borrowed at all.

(1) Customer Authority: Mr. Smith, Head Librarian at the University of Excellence, has final say about technical requirements and the budget available for the project.
(2) Business Objectives: To provide an efficient facility to manage a large, and expanding, library collection of publications.
(3) Current System Problems: The existing card-index system is slow, error prone, and increasingly unmanageable with the increasing number of publications and users.
(4) Principal New Features: Automated publication searches (keyword, author, title), checking out (borrowing), checking in (returning), updating (adding to collection, corrections), issuing reminders and fine letters.
(5) Future Expansions: The new systems may be required to operate in a self service mode, in which initially users can reserve publications, access catalogue remotely, and perhaps sometime in the future issue and return publications.
(6) Costs and Time Scales: The new system must be operational by the 1st January 1995, and should cost no more than 15,000 ECUs.

Work Record: **"**Above specification was developed on 10th June 1994 and modified on the 11-13th June 1994. Customer approved specification on 15th June 1994..."

Figure 9.5: *Library system problem definition ViewPoint, instantiated from PD template.*

Checking inter-ViewPoint consistency at this point invokes the only inter-ViewPoint rule in the PD template:

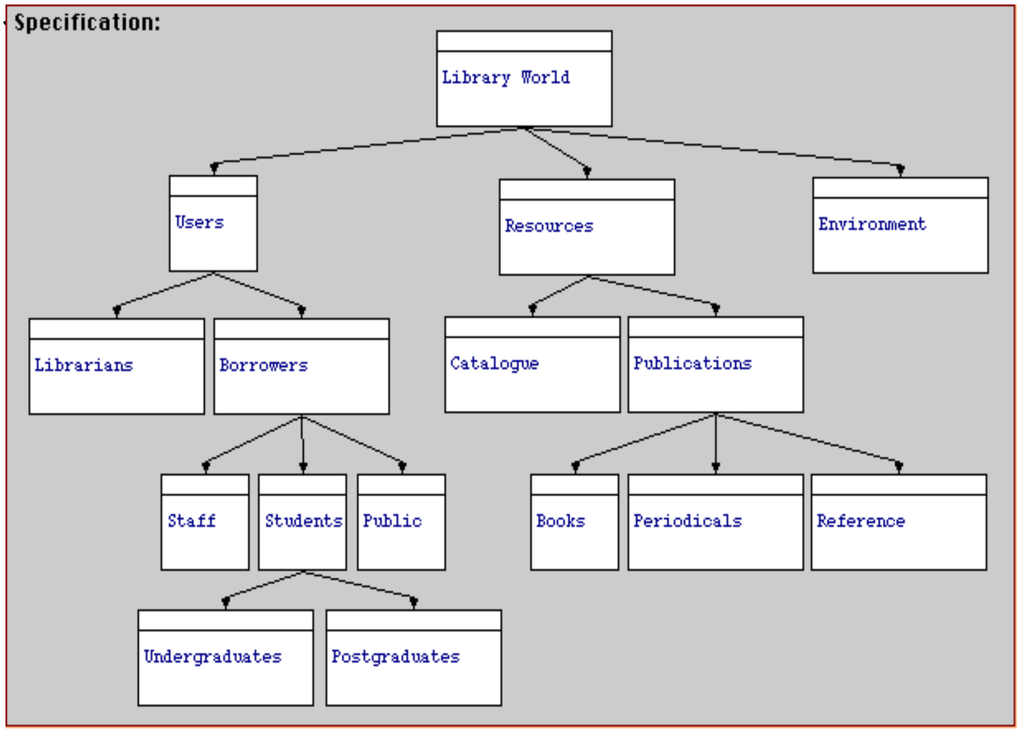
$$\text{self} \rightarrow \exists\, \text{VP}(\text{AS}, \text{D}_\text{S})$$

which in turn prompts for the creation of a new ViewPoint instantiated from the AS template for the same domain, “Library”. The partial specification of this ViewPoint can then be developed as shown in figure 9.6.

ViewPoint: LibrarySystem-AS

Template: AS

Domain: Library World



Work Record: Add "Library World", add "Users", add "Resources", add "Environment", add "Librarians", add "Borrowers", add "Staff", add "Students", add "Public", add "Undergraduate", add "Postgraduate", add "Catalogue", add "Publications", add "Books", add "Periodicals", add "Reference", link (Library World, Users), link (Library World, Resources, link(Library World, Environment), link(Users, Borrowers), ...

Figure 9.6: *Library system agent structuring ViewPoint, instantiated from AS template.*

Applying all the in-ViewPoint checks to the specification of ViewPoint LibrarySystem-AS, shown in figure 9.6, yields no inconsistencies. However, applying inter-ViewPoint rules, such as:

$$\text{Agent} \rightarrow \exists\ \text{VP(TC, Agent.Name)}$$

and:

$$\text{Agent} \rightarrow \exists\ \text{VP(DS, Agent.Name)}$$

requires that ViewPoints instantiated from the TC and DS templates, respectively, be created. At this stage, we choose to only partly address those inconsistencies at this stage by creating two “blank” tabular collection ViewPoints for the “Borrowers” and “Librarians” agents in the AS hierarchy.

Figure 9.7 shows the ViewPoints that have been created thus far, and the order (from left to right) in which they were created.

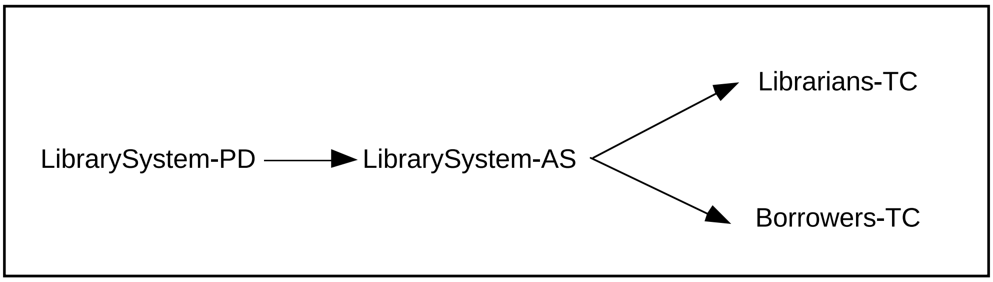


Figure 9.7: *ViewPoints created in the library system case study (order of creation shown from left to right). Each ViewPoint name is hyphenated, the first part denoting the ViewPoint's domain, and the second part denoting the template from which the ViewPoint was instantiated.*

Each tabular collection ViewPoint may now be elaborated separately, concurrently or in series. For example, the Borrowers-TC ViewPoint is shown in figure 9.8. This ViewPoint's domain, "Borrowers", is relatively well understood and is therefore a good starting point for the tabular collection stage.

Note however, that in developing the specification for this ViewPoint, we have introduced a source ViewPoint called "Library". Thus when we apply the inter-ViewPoint rule:

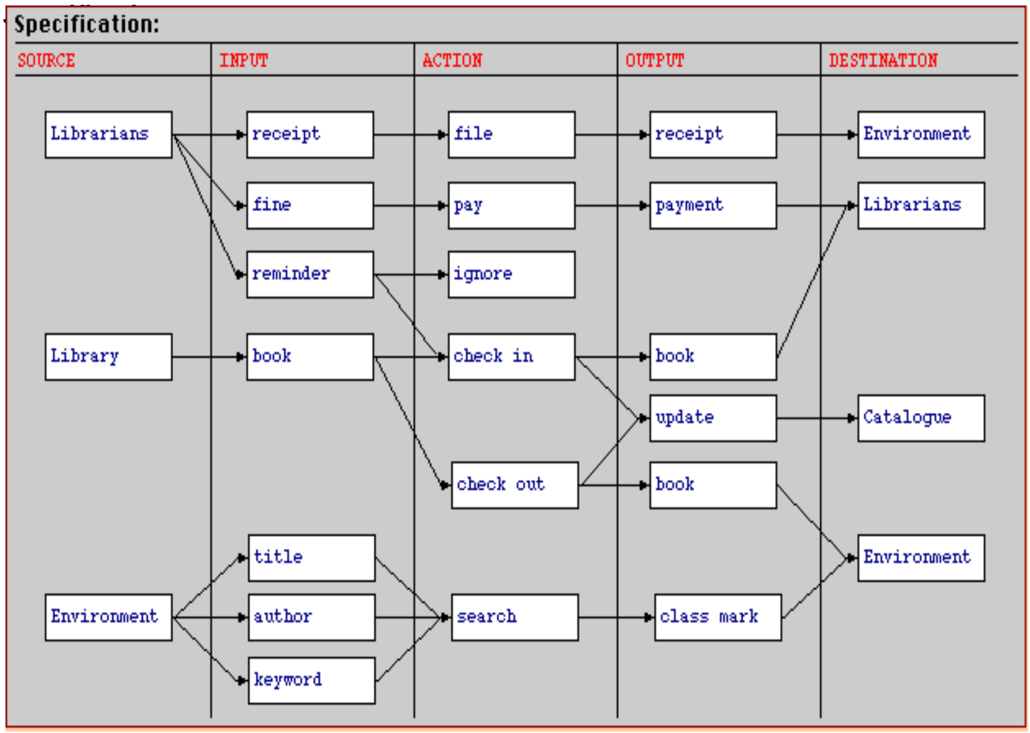
Source.Name = VP(AH, D_d): Agent.Name

we discover that we either need to remove the "Library" source from our tabular collection diagram or add it as an agent in our agent hierarchy. We decide to add "Library" to our hierarchy at this point, and also remove the sub-tree under "Publications" (since it is subsumed within "Library"). Thus, our LibrarySystem-AS ViewPoint now appears as shown in figure 9.9.

ViewPoint: Borrowers-TC

Template: TC

Domain: Borrowers



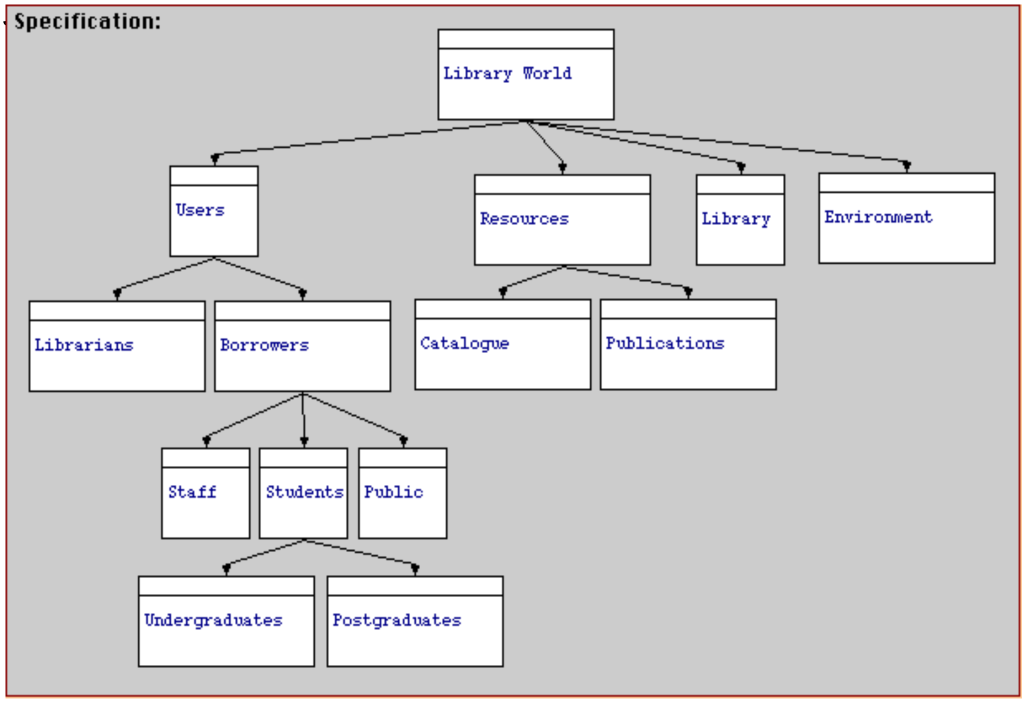
Work Record: Add "check in", add "check out", add "search", add "pay", ..., add "fine", add "reminder", ..., add "Librarians", add "Library", add "Environment", add "receipt", add "money", ..., add "Catalogue", ..., link(Library, book), link(book, check in), link(check in, update), link(update, Catalogue), ...

Figure 9.8: *ViewPoint describing borrower interactions within the library.*

ViewPoint: LibrarySystem-AS

Template: AS

Domain: Library World



Work Record: Add "Library World", add "Users", add "Resources", add "Environment", add "Librarians", add "Borrowers", add "Staff", add "Students", add "Public", add "Undergraduate", add "Postgraduate", add "Catalogue", add "Publications", add "Books", add "Periodicals", add "Reference", link (Library World, Users), link (Library World, Resources, link(Library World, Environment), link(Users, Borrowers), ..., add "Library", link(Library World, Library), unlink(Publications, Books), ..., remove "Books", remove ""Periodicals", remove "Reference".

Figure 9.9: *New state of agent hierarchy ViewPoint describing library world .*

Returning to the Borrowers-TC ViewPoint, invoking and applying the inter-ViewPoint rule:

connected-to(Source, Input).Input.Name =

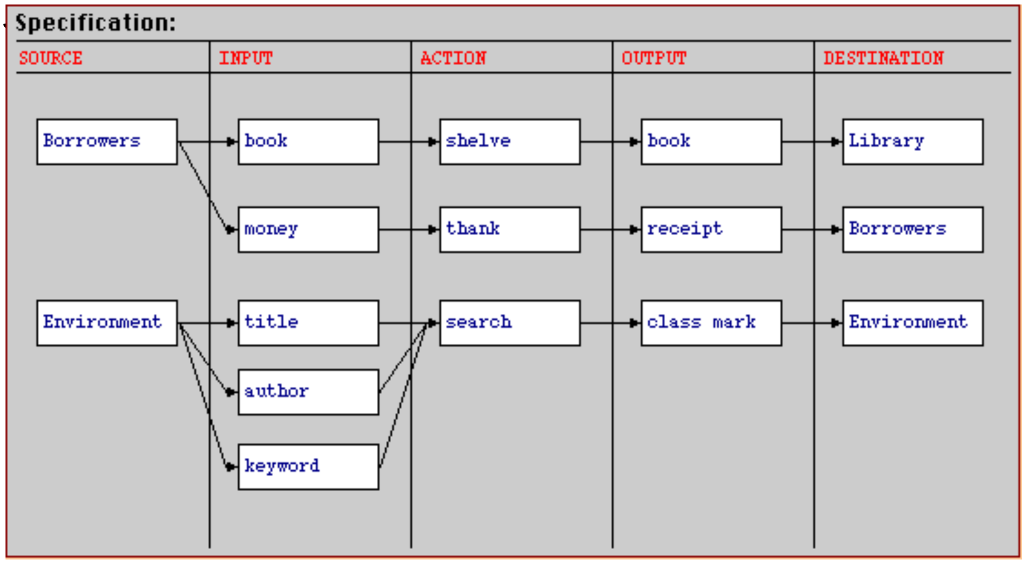
VP(TC, Source.Name): connected-to(Output, D_S).Output.Name

yields two inconsistencies. The first is with the Librarians-TC ViewPoint (which is empty). The rule specifies that the specification in this ViewPoint should produce an outputs “receipt”, “fine” and “reminder” which are inputs in the Borrowers-TC ViewPoint specification. To handle this inconsistency these outputs may be added, and the remainder of the specification elaborated as shown in figure 9.10.

ViewPoint: Librarians-TC

Template: TC

Domain: Librarians



Work Record: Add "book", add "money", ..., add "shelve", add "thank", add "search", add "Borrowers", add, "Environment", add "receipt", add "class mark", ..., add "Library", add "Borrowers", link(Borrowers, book), link(book, shelve), ...

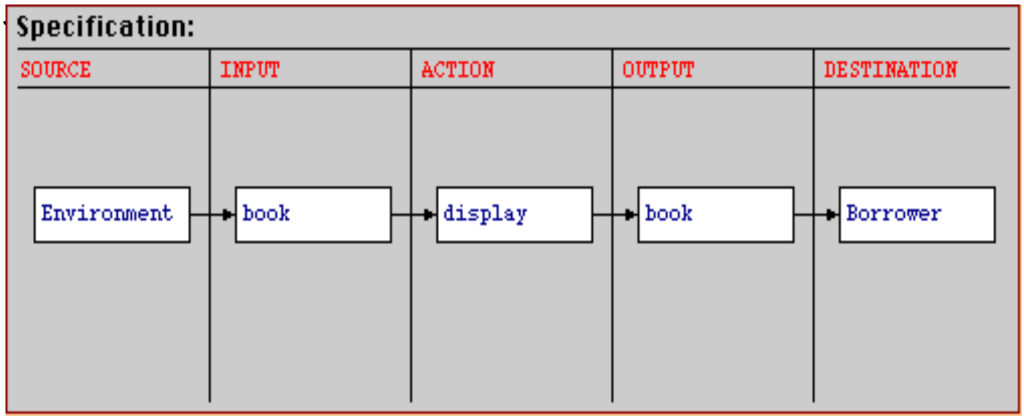
Figure 9.10: *ViewPoint describing librarian behaviour within the library.*

The second inconsistency resulting from the application of the same rule requires a ViewPoint for the “Library” agent to exist, one of whose outputs is “book”. This may now be created and elaborated as shown in figure 9.11.

ViewPoint: Library-TC

Template: TC

Domain: Library



Work Record: Add "book", add "display", ..., add "book", add "Environment", add "Borrowers", link(Environment, book), link(book, display), link(display, book), link (book, Borrowers)

Figure 9.11: *ViewPoint partially describing the "Library" agent.*

Similarly, invoking and applying the rule:

connected-to(Output, Destination).Output.Name =

VP(TC, Destination.Name): connected-to(D_S, Input).Input.Name

from the Borrowers-TC ViewPoint, triggers the creation of a new TC for "Catalogue", and confirms the consistency of the Librarians-TC and Borrowers-TC ViewPoints.

We can now elaborate TC ViewPoints for the various categories of borrowers (staff, undergraduates, postgraduates and public), but this does not add any new insight into the use of the ViewPoints framework, so we omit them from this chapter.

To illustrate yet another step in the unfolding of the CORE method for specifying the library, we invoke and apply the following rule from the Borrowers-TC ViewPoint:

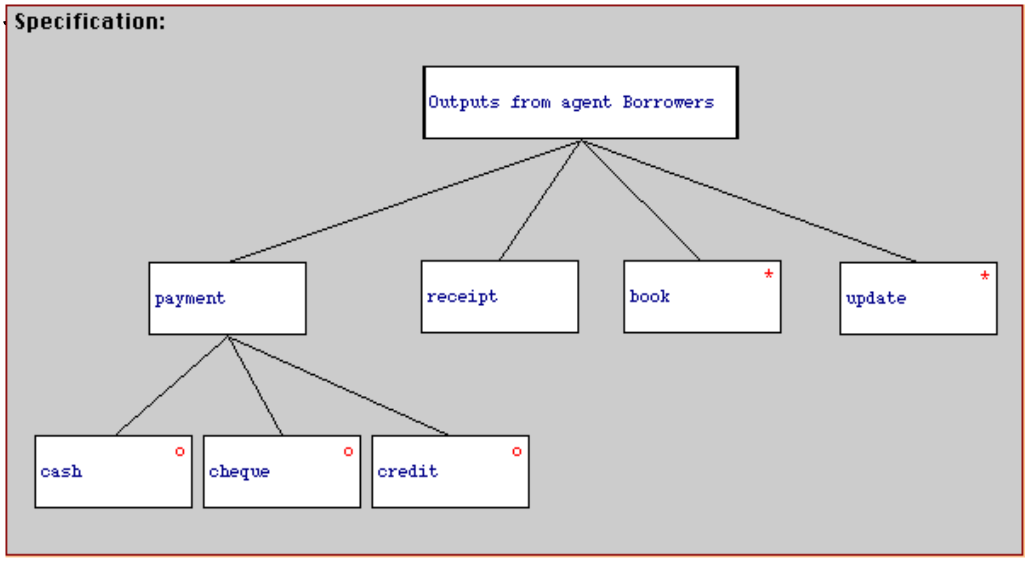
Output.Name = VP(DS, D_S): Data-Item.Name

This rule fails on invocation because there have been no DS ViewPoints created thus far. A ViewPoint trigger action at this point creates a new ViewPoint, "Borrowers-DS", for the same domain, "Borrowers". Next, all the outputs from the Borrowers-TC ViewPoint are transferred across to this new ViewPoint, and may be organised as shown in figure 9.12.

ViewPoint: Borrowers-DS

Template: DS

Domain: Borrowers



Work Record: Add "outputs from agent Borrowers", add "payment", ..., add "receipt", add "book", add-annotation(book, *), add "update", add-annotation(update, *), link(Output from agent Borrowers, payment), ..., add "cash", add-annotation(cash, o), link(payment, cash), add "cheque", add-annotation(cheque, o), ...

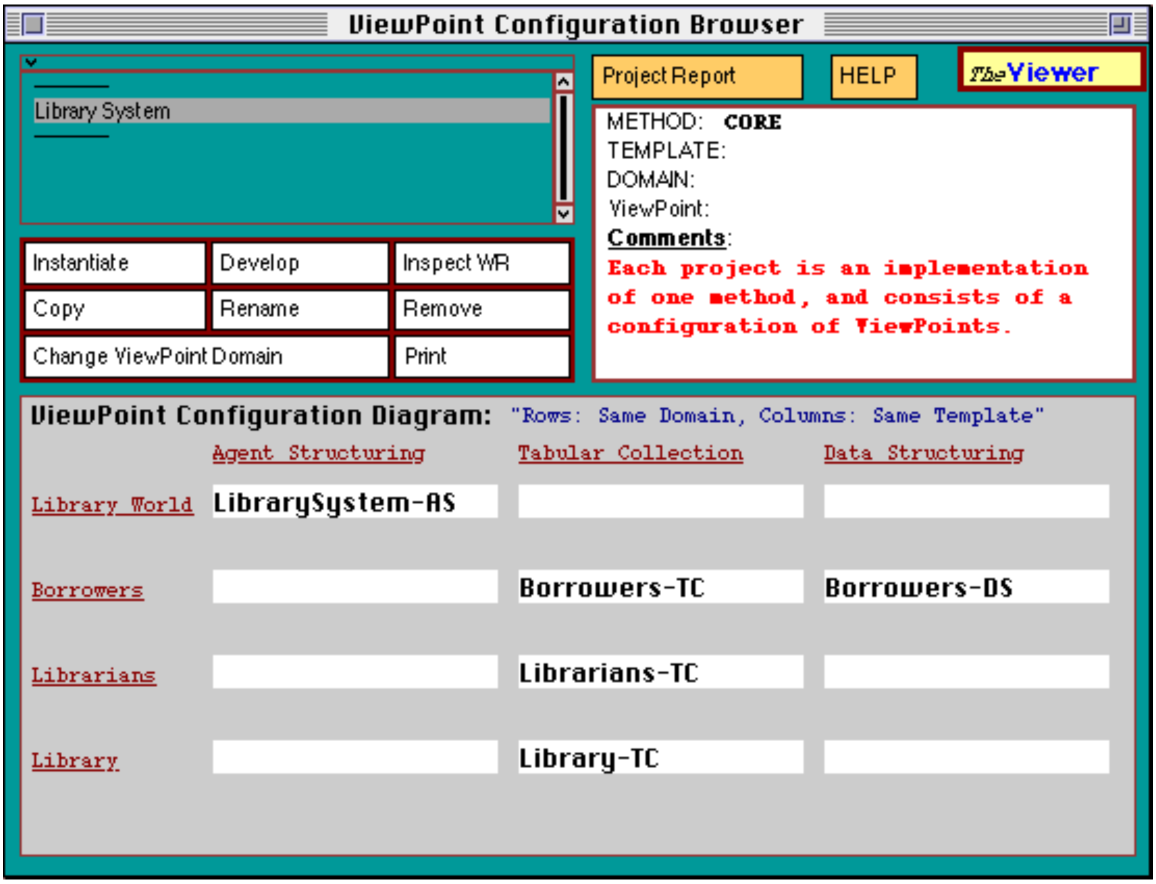
Figure 9.12: *A data structuring ViewPoint describing outputs from the "Borrowers" agent. An "o" denotes mutual exclusion, while an "*" denotes iteration. No annotation indicates that the data item is produced only once.*

Similarly, DS ViewPoints may be created for all the agents for which a TC ViewPoint has been created and elaborated. Development using the method then continues further, with the creation of SAM ViewPoints for each elaborated agent. At each point during development, in-ViewPoint checks test for local consistency of individual ViewPoints, while inter-ViewPoint checks test for consistency across ViewPoints, and drive the development further when inconsistencies are found.

9.1.6. Using Tool Support

The case study described in this chapter was developed using *The Viewer*, and is documented using the tool. The AS, TC and DS templates were defined in the TemplateBrowser, and tools to support development of ViewPoints instantiated from these templates, were implemented in Smalltalk (and using *The Viewer's* meta-CASE tool kit). The library system itself was then specified as described in the previous section, with several additional iterations performed to refine the various partial specifications and to remove inconsistencies. All development steps were automatically logged on to the work record, and either manually or automatically annotated with explanations and rationale.

Figure 9.13 is a screen dump of the ViewPointConfigurationBrowser, with the “Library System” project selected and the configuration of ViewPoints created displayed in the bottom pane.



METHOD: CORE
TEMPLATE:
DOMAIN:
ViewPoint:
Comments:
Each project is an implementation of one method, and consists of a configuration of ViewPoints.

ViewPoint Configuration Diagram: "Rows: Same Domain, Columns: Same Template"

	Agent Structuring	Tabular Collection	Data Structuring
Library World	LibrarySystem-AS		
Borrowers		Borrowers-TC	Borrowers-DS
Librarians		Librarians-TC	
Library		Library-TC	

Figure 9.13: *The ViewPointConfigurationBrowser with the “Library System” project selected.*

9.2. Critical Evaluation

To evaluate the ViewPoints framework based on the results of our case study, the two activities of method engineering and method use are considered separately. Clearly however, the impact of having a well designed method affects the way in which it is deployed.

What we are *not* trying to do is to evaluate the CORE method itself or its suitability for specifying software requirements in general, or a library system in particular. Moreover, we are not trying to address all issues pertaining to the ViewPoints framework, rather only those highlighted by our case study. A more detailed evaluation of our work is presented in the concluding chapter of this thesis.

9.2.1. Method Engineering Issues

Our case study demonstrates the difficulty of precisely and comprehensively describing an entire software development method. CORE consists of a wide range of notations and procedures, from informal natural language representations and guidelines, to more precise representation schemes and development steps. This kind of knowledge is difficult to document under any circumstances, and so what the ViewPoints framework provides is a means of rigorously analysing and carefully structuring methods so that they can be used more effectively.

Thus, we were able to describe the different stages of CORE separately (facilitating their reuse later in a different context or method), and concentrated on the relationships or overlaps between these stages. Moreover, describing CORE in terms of its constituent ViewPoint templates, provides a better understanding of the method itself and the inter-working of its different stages. CORE proved to be particularly well suited to the kind of partitioning described, however, methods with less well-defined stages are more problematic and require experience and careful design.

Nevertheless, the difficulties of describing the development processes deployed by methods remain, although the decomposition of a method into smaller and simpler methods (templates) does make this task easier.

Describing CORE using a number of ViewPoint templates also facilitated the development of tool support for the method. The infrastructure of *The Viewer* provided many of the automatic CASE tools generation capabilities, but the modular way in which the method was described also facilitated the development of individual tools to support the individual CORE stages/templates. This was clearly reflected in the speed with which a supporting tool kit for CORE was constructed using *The Viewer.*

Moreover, the ViewPoint templates themselves acted as specifications for the tools required to support them, and therefore describing CORE in terms of templates naturally facilitated tool development in this setting. In this context then, the CASE tools are (or should be) the automated realisation (implementation) of methods. Nevertheless, the current status of the framework excludes the notion of “external tools” such a variety of editors, debuggers and analysis tools, since, strictly speaking, they need to be described by ViewPoint templates first, before they may be used by ViewPoint developers.

9.2.2. ViewPoint-Oriented Development Issues

Developing any large specification is a challenging managerial and organisational exercise. ViewPoint-oriented development attempts to manage complexity by imposing consistent structure upon the range of partial specifications that proliferate during the development process. As the case study demonstrates however, problems of scale remain, although managing a collection of uniformly structured ViewPoints is easier than managing a disconnected set of partial specifications. Clearly, the current flat structure of ViewPoint configurations is inadequate for managing very large collections of ViewPoints, but again we maintain that this is a not a problem unique to the ViewPoints framework, and is typical of any development project involving large amounts of documentation (specification and code). A variety of configuration management and version control techniques may therefore be appropriately used in this setting, and certainly, tool support is essential.

Our library system case study has highlighted a number of interesting issues arising from a ViewPoint-oriented software development approach. The first is that the approach is indeed tolerant of inconsistencies during development. Our ViewPoints represent partial specifications that may themselves be partially consistent, internally or with other ViewPoints. Consistency is achieved incrementally by gradually applying the list of consistency rules defined in these ViewPoints' work plans. In fact, development in the framework proceeds by applying a variety of rules and then acting on their results. A second observation is that a "stable" structure for the ViewPoint configuration (that is, the system specification) emerges, regardless of the different ways in which individual developers decide to deploy the method. Thus, while local ViewPoint process models prescribe recommended development actions and procedures, these again are flexible, and the final goal of a consistent collection of ViewPoints is not lost.

The role of tool support of ViewPoint-oriented development in our case study is central. *The Viewer* facilitated rapid, evolutionary development of ViewPoint-oriented specifications by managing collections of ViewPoints and facilitating consistency checking within and between ViewPoints. Development progress was recorded, and documentation of this was partially automated. *The Viewer* therefore provides a smooth transition of method engineering to effective method deployment in this context, without imposing unreasonable development procedures on method users. Nevertheless, a fully distributed implementation of *The Viewer* is needed for effectively supporting multiple (distributed) development participants.

Finally, what our case study has demonstrated is that a method such as CORE, if loosely integrated using a number of simple consistency relationships, can produce a well structured, organised and integrated system specification as a result of applying it within the ViewPoints framework.

9.3. Technology Transfer

We now briefly outline two facets of technology transfer in the context of the ViewPoints framework. The first is the role of the framework as a vehicle for inter-disciplinary technology transfer, and the second is the transfer of ViewPoint concepts and technology from the university laboratory into an industrial setting.

In a collaborative project (SEED) with the School of Electrical Engineering at The City University [Finkelstein et al. 1992b], we examined technology transfer between the disciplines of software engineering and electronic engineering design. While our experiments [Finkelstein et al. 1992c;

Finkelstein et al. 1992d] illustrated the feasibility of applying some software engineering techniques, such as structured and formal methods (CORE and Z, respectively), we found that analysts and designers prefer to work with their own preferred techniques and languages to develop their own areas of concern and responsibility. This was a large motivating factor for elaborating the ViewPoints framework, in order to support this mixing and matching of methods and techniques.

In a further set of projects, we explored the transfer of our ViewPoint concepts to industrial application domains. For example, Siemens (Munich, Germany), used ViewPoint templates to describe an editor and a simulator for an in-house method based on a version of Petri Nets [Graubmann 1992a; Graubmann 1992b]. Using ViewPoints in this way provided useful organisation and structuring of the techniques used, and facilitated the development of a specialised support tool called HyperView.

At Hewlett-Packard (HP) Laboratories (Bristol, UK) an in-house method called FUSION [Coleman et al. 1993] was described using ViewPoint templates and implemented in *The Viewer* [Ballesteros 1992]. This proved to be useful for HP in improving and documenting their method, and useful to us in enhancing *The Viewer* to support custom (e.g., industrial) methods.

A number of prototype instantiations of *The Viewer* were also constructed by Masters project students. For example, Thanitsukkarn [Thanitsukkarn 1993] developed tool support for the Constructive Design Approach (described in chapter 2, section 2.1.2), while Lai [Lai 1993] developed editors to support the latter stages of CORE.

Other researchers have also proposed integration techniques that either directly build upon or use our ViewPoint concepts. For example, Zave and Jackson [Zave & Jackson 1993] suggest that their technique for composing partial specification can be deployed within the ViewPoints framework without modification. Barroca and McDermid [Barroca & McDermid 1993] on the other hand, adopt the organisational and structuring characteristics of our framework in describing a view-oriented approach to the specification of real-time systems.

9.4. Chapter Summary

This chapter has demonstrated ViewPoint-Oriented Software Engineering (VOSE) for the partial development of a greatly simplified library system. The case study described, demonstrates the incremental process by which a multi-perspective specification is developed within the ViewPoints framework. The case study, describing the specification of a library system's requirements using the CORE method, illustrates how alternative, contradictory or complementary, partial specifications may be produced, and how an incremental process of consistency checking can achieve incremental integration in this setting. The development of the case study was supported by *The Viewer* environment, which demonstrated both the usefulness of the environment itself, and the need for tool support for ViewPoint-oriented development in general. Other experiences of adopting a VOSE approach were also outlined, which further demonstrated the feasibility of deploying the ViewPoints framework in alternative (e.g., industrial) settings.

Chapter 10 Summary, Conclusions and Future Work

10.1. Summary

This thesis has addressed a range of issues arising from distributed, multi-perspective software specification and design. In particular, it has addressed the problem of method integration in this setting - an instance of the multiple perspectives problem.

The thesis elaborated a software development framework based on loosely coupled, locally managed, distributable objects, called “ViewPoints”, which encapsulate partial representation, process and specification knowledge about a system and its domain. These ViewPoints collectively represent the system specification, and may be concurrently developed by multiple development participants.

ViewPoint templates - denoting ViewPoint types - were also introduced as the building blocks of software development methods. Such methods, and the ViewPoints that are created as a result of deploying them, are integrated via inter-ViewPoint rules, which express the consistency relationships that are required to hold between ViewPoints in order to achieve integration in this setting. Thus, inter-ViewPoint rules denote the method integration “glue” provided by method engineers, and their application by method users checks the consistency of multiple partial specifications.

The thesis proposed a model of ViewPoint interaction in which inter-ViewPoint rules are defined during method design, and invoked and applied during method deployment. Fine-grain process modelling is used as the means of providing local method guidance and of coordinating ViewPoint interaction. In this setting then, each ViewPoint contains its own local process model which guides specification development within that ViewPoint, and suggests appropriate moments for consistency checking, and appropriate actions for inconsistency handling.

In the spirit of multi-perspective development, inconsistencies are tolerated and managed by prescribing rules that specify how to act in the presence of these inconsistencies. An action-based temporal logic for describing inconsistency handling rules of the general form “*inconsistency* *implies action"* is used to illustrate this approach.

The thesis also described *The Viewer*, a prototype environment supporting the entire ViewPoints framework, implemented to demonstrate the scope and feasibility of the approach. *The Viewer* illustrates the activities of method engineering and method use, and provides a vehicle for demonstrating the utility of the ViewPoints framework for addressing a number of software engineering concerns. Thus for example, it provides the opportunity of recording and analysing development rationale by means of a semi-automatically maintained work record.

10.2. Analysis of Contributions

This thesis makes a novel contribution to the organisation and management of multi-perspective software development, and illustrates the role of method engineering, process modelling and consistency management in this setting. We now examine individual contributions in more detail, highlighting the benefits and limitations of our approach, and the scope for future work to address these limitations.

10.2.1. Contributions

The ViewPoints framework provides an infrastructure within which multiple development participants can develop their own perspectives or areas of concern in a system. These participants may do so concurrently, guided by local (fine-grain) process models that facilitate individual ViewPoint development and control the invocation and application of consistency rules. This constitutes the central contribution of the research, and is a significant elaboration of previous work on ViewPoints [Finkelstein et al. 1989], which focused on the concept of a ViewPoint as a partial specification, and did not address the "dynamics" of multi-perspective software development in this setting.

The framework provides a means of reducing software development complexity by separation of concerns at a number of levels. During method design, complex development methods are decomposed into their constituent ViewPoint templates by identifying (and separating) the different representation and process knowledge they deploy. During method deployment, an additional kind of knowledge is added separately, namely, specification knowledge - which is a domain-specific description in the representation scheme specified by the ViewPoint's template. Thus, the ViewPoints framework achieves separation of concerns at both method engineering and method use levels, by deploying ViewPoint templates and ViewPoints, respectively. Further separation is also achieved by separating representation, process and specification knowledge within each ViewPoint.

A model of ViewPoint interaction identifies inter-ViewPoint rule definition, invocation and application as the central activities in managing consistency and achieving integration in this setting. The feasibility of each of these activities is demonstrated by a sample inter-ViewPoint rule notation, process modelling approach and inter-ViewPoint communication protocol, respectively. The model also demonstrates a primary objective of this thesis, namely, method integration in a multi-perspective development setting. Thus, inter-ViewPoint rule definition expresses method integration intentions, while the invocation and application of these rules implement these intentions.

Inconsistency handling supports the simultaneous development of multiple overlapping ViewPoints that may be inconsistent with each other. Rather than attempt to eradicate inconsistencies as a matter of course, they are tolerated by specifying how to act in their presence.

The Viewer environment - together with the case study described in chapter 9 - has supported the demonstration and informal evaluation of the ViewPoints framework. Both *The Viewer* and the case study show the potential of using ViewPoints as a vehicle for (computer-supported) multi-perspective software development. *The Viewer* illustrates the distinction between method design, method use, and meta-CASE technology in this setting. It also demonstrates the feasibility of deploying a wide range of CASE technologies despite the radically decentralised approach adopted by the framework. Thus, “traditional” CASE capabilities are provided for each ViewPoint, including for example, diagram editing and syntactic consistency checking. The case study on the other hand, provides a realistic (albeit somewhat “sanitised”) example of multi-perspective development guided, and sometimes automated, by *The Viewer* environment.

Finally, an advance that the ViewPoints framework achieves, which has not been explored in this thesis, has been the insight it has given us into what constitutes a “good” software engineering method. The research strategy we have adopted in developing the ViewPoints framework has been to construct *The Viewer* as an evolutionary prototype, in parallel with our development of the framework. Together with a series of examples, and finally the case study, our experiences appear to support the view that method engineers should design methods that deploy many, simple, highly redundant, preferably graphical, representation schemes. We expect that specifications developed using such methods will be easier to construct, understand and maintain. However, this is more of a conjecture than a scientific conclusion, and what is needed is a systematic study of a wide range of method engineering activities in this context. This would also require access to “real” method designers - either to elicit the process by which they have developed methods in the past, or to observe them during a current method engineering process in which they are engaged. Such a study would also provide additional insight into their needs for tool support, which could be exploited in the future development of *The Viewer*. Nevertheless, our simple extension and customisation of “traditional” methods, together with the case study and *The Viewer*, have demonstrated the potential of the ViewPoints framework as a vehicle for method engineering, while also clarifying its limitations and scope for future work.

10.2.2. Limitations and Future Work

We identify below a number of limitations of the ViewPoints framework, some of which may be addressed by future work in the area. General limitations of *The Viewer*, and the scope for improving it are also outlined.

10.2.2.1. The ViewPoints Framework

- The framework does not provide any mechanisms for integrating methods that are conceptually difficult to integrate. However, integrating two methods that have unclear areas of overlap, or no overlap at all, may be intellectually difficult under any circumstances, and is not necessarily facilitated by using the framework. What the ViewPoints framework does provide, is a means of systematically examining the relationships between techniques deployed by one or more methods, which may then facilitate their definition in terms of inter-ViewPoint rules. Thus, integrating methods that deploy the Z and CSP notations for example, is more difficult then integrating, say, SSADM (deploying data flow diagrams) with an object-oriented design method. Method integration of radically different methods therefore remains problematic, although our framework provides the integration infrastructure, if the relationships between the methods being integrated can be identified.
- It is still not clear with the size of examples and case studies we have performed, how the framework will scale-up to industrial size projects. The proliferation of ViewPoints clearly needs to be carefully managed and controlled, for which a layered or hierarchical structuring of ViewPoint configurations may be needed. Better navigation around such large ViewPoint configuration structures may also be needed, and hypertext capabilities may be useful in this context. Moreover, some form of standardisation of the kind of information transferred between such large numbers of ViewPoints is clearly desirable - even necessary.
- The role of the ViewPoints framework as a vehicle for facilitating computer-supported cooperative work (CSCW) also needs to be examined. While the framework, as it stands, models the scenario in which multiple development participants hold multiple views, there is a large body of work on CSCW, including the use of multi-agent systems for cooperative problem solving, that is relevant in this setting. The role of inter-ViewPoint rules as the vehicle for interaction between cooperating agents for example, needs to be examined more carefully.
- Finally, the role of domain knowledge has not been explored in any detail in this thesis, and is clearly of fundamental importance to realistic development. Two facets of this are pertinent. The first is the inclusion of domain knowledge in methods themselves, if one knows beforehand that these method will be used in a particular domain. So for example, methods used in the telecommunications industry deploy telecommunications terminology that is part of the methods used. This allows alternative terms for the same concept to be used. The second

kind of domain knowledge is that which emerges during development, such as dependencies or relationships that are identified dynamically as the development proceeds. These need to be recorded to further guide development, and facilitate inconsistency handling and conflict resolution. One approach that may address this is to use analogous inter-ViewPoint rules that are domain-specific. These rules, rather than relating objects, relations, attributes and types, actually relate *values* of the former. In such cases we can use our same notation to express such rules.

Where these rules reside is another question. The domain slot appears a natural location to place domain knowledge, however, domain-specific inter-ViewPoint rules have an impact on the development process, and there also is an argument for including them in ViewPoint work plans. Of course, such rules differ from the other method integration rules in that they dynamically change and evolve during development, an therefore a mechanism for updating them is also needed.

10.2.2.2. The Viewer

- While *The Viewer* demonstrates the feasibility of using a variety of CASE technologies to support ViewPoint-oriented development, there is ample scope for extension and improvement. Ideally, further development should be driven by studies of the way that methods are developed and used; however, the need for some enhancements has become evident in the course of the informal evaluation described in chapter 9. The use of the work record is one example. The work record is a temporal record of activities and is therefore useful for inconsistency handling. A record of activities performed in the past, may provide insight into, or even identify the cause of, current inconsistencies. The work record also embodies the notion of a ViewPoint “state”, which can be used to provide better means of process modelling: if we have a precise knowledge about the current state of a ViewPoint, we are in a better position to know what is to be done next. Thus, process modelling in *The Viewer,* which is currently oversimplified, may be improved. In general however, achieving fully decentralised, cooperating process models is still a problematic area that requires further work.
- While the ViewPoints framework is distributable, *The Viewer* implementation is not distributed. It is implemented in Objectworks/Smalltalk which is a single-user, centralised environment. Thus, to fully test our approach with automated support, a distributed, multi-user version of *The Viewer* needs to be implemented. The consequences of our radically decentralised approach may then be more realistically assessed. Moreover, the consequences of real-time concurrent development may then also be explored.
- The current implementation of *The Viewer* allows method engineers to specify in- and inter-ViewPoint rules textually, and relies on the tool builder to hard-code, these rules in terms of Smalltalk methods. Clearly, high-level rule definition languages are more desirable than

having to program these rules manually every time. While we have not found it a particularly useful exercise to devise such a language for this thesis, it is in fact necessary if *The Viewer* is to be used on a wider scale. In the short term, a Prolog-like notation may be practical for expressing in-ViewPoint rules, while the notation proposed in this thesis (together with an appropriate interpreter) may be used to describe (and check) inter-ViewPoint rules.

More generally, and in pursuit of a more thorough evaluation of the ViewPoints framework, a sustained effort to "productise" *The Viewer* may be appropriate. This would facilitate its deployment in an industrial setting in which the ViewPoints framework may be fully evaluated for feasibility and usability. In the shorter term, controlled software engineering experiments are planned to evaluate the use of the ViewPoints framework and *The Viewer*. These may take the form of examples and case studies developed: (i) "traditionally", (ii) within the ViewPoints framework, and (iii) supported by *The Viewer.*

- Finally, there is a need to display two or more ViewPoints concurrently and to be able to compare them visually. Clearly, high resolution bit-mapped displays are necessary, but further control of the amount of information displayed in each ViewPoint may also be needed. This is particularly important when a multi-party conflict resolution process is taking place, where reliance on a simple one-to-one communication protocol is not sufficient to handle the bandwidth of the information exchanged.

10.3. Conclusions

Have we been successful in meeting our objective of elaborating a multi-perspective software development framework for method integration? The broad coverage of software engineering issues that the ViewPoints framework provides, makes the wholesale illustration of its feasibility difficult. In this thesis, we have therefore chosen to address a variety of these issues individually, examining and illustrating each independently.

The ViewPoints framework presented in this thesis, supported by *The Viewer* environment, has demonstrated the nature and feasibility of adopting a decentralised approach to multi-perspective software development. The framework provides a means for incrementally building, customising and integrating software engineering methods from their constituent development techniques (templates). These techniques are integrated by identifying and defining the areas of overlap between them using so-called inter-ViewPoint rules. These rules are distributed among the different templates (and upon instantiation, ViewPoints), resulting in a totally distributable configuration of ViewPoints, which we have argued is a more realistic reflection of the actual development of complex systems. While in reality some centralisation may be necessary, or even desirable, we have chosen in our framework to explore the consequences of a "radically" decentralised approach.

We have also advanced the notion of tolerating inconsistency, which one would expect to do when supporting multiple views during software development. Our approach allows the guided development of individual ViewPoints, whose consistency against each other is checked by the application of pre-defined inter-ViewPoint rules. The invocation of these rules is again guided by local ViewPoint process models, and the interaction between any two ViewPoints is within the scope of a flexible communication protocol. Inconsistencies that are detected need not be resolved immediately, and development need not stop. Rather, inconsistency handling rules prescribe possible actions that may be performed in the presence of these inconsistencies. These actions either resolve conflicts, reduce inconsistencies or simply avoid them altogether. In this way development can proceed despite any conflicts or contradictions that inevitably arise during any concurrent engineering process.

We believe that our framework, and the philosophy it promotes (multiple views, integration via distributed one-to-one checks driven by decentralised process models and inconsistency handling), is a realistic approach to concurrent software engineering. It avoids the bottleneck of centralised architectures, strict consistency maintenance, global process models and representation schemes, and inflexible tool support. While further case studies and examples are needed to test and refine the framework, the evidence we have collected from the use of our concepts and tools is encouraging, and has demonstrated that ViewPoint-oriented development is a novel and practical approach to software specification and design.

References

[Acly 1988] Acly, E. (1988); "Looking Beyond CASE"; *Software*, 5(2): 39-43, March 1988; IEEE Computer Society Press.

[ACM 1992] ACM (1992); *Proceedings of International Conference on Computer-Supported Cooperative Work: Sharing Perspectives*, Toronto, Ontario, Canada, 31st October - 4th November, ACM SIGCHI & SIGOIS.

[Ahmed et al. 1991] Ahmed, R., P. De Smedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii and M. Shan (1991); "The Pegasus Heterogeneous Multidatabase System"; *Computer*, 24(12): 19-27, December 1991; IEEE Computer Society Press.

[Aho, Kernighan & Weinberger 1988] Aho, A. V., B. W. Kernighan and P. J. Weinberger (1988); *The Awk Programming Language*; Addison-Wesley, Reading, Massachusetts, USA.

[Aho & Ullman 1977] Aho, A. V. and J. D. Ullman (1977); *Principles of Compiler Design*; Addison-Wesley Publishing Company, Reading Massachusetts, USA.

[Ainsworth et al. 1994] Ainsworth, M., A. H. Cruickshank, L. G. Groves and P. J. L. Wallis (1994); "Viewpoint Specification and Z"; *Information and Software Technology*, 36(1), February 1994; Butterworth-Heinemann.

[Akima & Ooi 1989] Akima, N. and F. Ooi (1989); "Industrializing Software Development: A Japanese Approach"; *Software*, 6(2): 13-21, IEEE Computer Society Press.

[Alderson 1991] Alderson, A. (1991); "Meta-CASE Technology"; *Proceedings of European Symposium on Software development Environments and CASE Technology*, Königswinter, Germany, June 1991, 81-91; LNCS, 509, Springer-Verlag.

[Alford 1994] Alford, M. (1994); "Attacking Requirements Complexity Using a Separation of Concerns"; *Proceedings of 1st International Conference on Requirements Engineering*, Colorado Springs, Colorado, USA, 18-22nd April 1994, 2-5; IEEE Computer Society Press.

[Alford 1977] Alford, M. W. (1977); "A Requirements Engineering Methodology for Real Time Processing Requirements"; *Transactions on Software Engineering*, 3(1): 60-69, January 1977; IEEE Computer Society Press.

[Alford 1985] Alford, M. W. (1985); "SREM at the Age of Eight: The Distributed Computing Design System"; *Computer*, 18(4): 36-46, April 1985; IEEE Computer Society Press.

[Anderson & Belnap 1976] Anderson, A. R. and N. D. Belnap (1976); *The Logic of Entailment*; Princeton University Press, USA.

[Arnold et al. 1991] Arnold, P., S. Bodoff, D. Coleman, H. Gilchrist and F. Hayes (1991); "An Evaluation of Five Object-Oriented Design Methods"; *Technical Report*, HP-91-52; Information Management Laboratory, Hewlett-Packard Research Laboratories, Bristol, UK.

[Ashworth & Goodland 1990] Ashworth, C. and M. Goodland (1990); *SSADM: A Practical Approach*; McGraw-Hill Book Company Europe, Maidenhead, UK.

[Backhurst 1993] Backhurst, N. (1993); "Meta-CASE"; *Personal Communication (email),* 24th December 1993; IE Services, Coalville LE67 2HB, UK.

[Ballesteros 1992] Ballesteros, L. A. R. (1992); "Using ViewPoints to Support the FUSION Object-Oriented Method"; *M.Sc. Thesis*, Department of Computing, Imperial College, London, UK.

[Balzer 1985] Balzer, R. (1985); "A 15 Year Perspective on Automatic Programming"; *Transactions on Software Engineering*, 11(11): 1257-1268, November 1985; IEEE Computer Society Press.

[Balzer 1991] Balzer, R. (1991); "Tolerating Inconsistency"; *Proceedings of 13th International Conference on Software Engineering (ICSE-13)*, Austin, Texas, USA, 13-17th May 1991, 158-165; IEEE Computer Society Press.

[Balzer, Cheatham & Green 1983] Balzer, R., T. E. Cheatham and C. Green (1983); "Software Technology in the 1990's: Using a New Paradigm"; *Computer*, 16(11): 39-45, November 1983; IEEE Computer Society Press.

[Barghouti 1992] Barghouti, N. (1992); "Supporting Cooperation in the MARVEL Process-Centered Environment (Proceedings of ACM SIGSOFT Symposium on Software Development Environments)"; *Software Engineering Notes*, 17(5): 21-31, December 1992; SIGSOFT & ACM Press.

[Barroca & McDermid 1993] Barroca, L. and J. McDermid (1993); "Specification of Real-Time Systems: A View-Oriented Approach"; *Proceedings of XIII Congresso da Sociedade Brasileira de Computação*, Florianópolis, Brazil, Sociedade Brasileira de Computação, XX SEMISH, Seminário Integrado de Software e Hardware.

[Barstow 1993] Barstow, D. (1993); "Should we Specify Systems or Domains"; *Proceedings of International Symposium in Requirements Engineering (RE '93)*, Coronado Island, San Diego, USA, 4-6th January 1993, 79; IEEE Computer Society Press.

[Barstow 1985] Barstow, D. R. (1985); "Domain-Specific Automatic Programming"; *Transactions on Software Engineering*, 11(11): 1321-1336, November 1985; IEEE Computer Society Press.

[Batini, Lenzerini & Navathe 1986] Batini, C., M. Lenzerini and S. B. Navathe (1986); "A Comparative Analysis of Methodologies for Database Schema Integration"; *ACM Computing Surveys*, 18(4): 323-364, December 1986; ACM Press.

[Bell 1990] Bell, J. (1990); "Non-Monotonic Reasoning, Non-Monotonic Logics, and Reasoning About Change"; *Artificial Intelligence Review*, 4: 79-108, Blackwells.

[Benjamin 1989] Benjamin, M. (1989); "A Message Passing System: An Example of Combining CSP and Z; Proceedings of Z Users Meeting, December 1989, Oxford.

[Birrel & Nelson 1984] Birrel, A. and B. Nelson (1984); "Implementing Remote Procedure Calls"; *Transactions on Computer Systems*, 2(1): 38-59, February 1984; ACM Press.

[Blair & Subrahmanian 1989] Blair, H. and V. Subrahmanian (1989); "Paraconsistent Logic Programming"; *Theoretical Computer Science*, 68: 135-154.

[Boehm 1981] Boehm, B. (1981); *Software Engineering Economics*; Prentice-Hall, Engelwood Cliffs, New Jersey, USA.

[Boehm 1988] Boehm, B. W. (1988); "A Spiral Model of Software Development and Improvement"; *Computer*, 31(5): 61-72, May 1988; IEEE Computer Society Press.

[Boloix, Sorenson & Tremblay 1992] Boloix, G., P. G. Sorenson and J. P. Tremblay (1992); "Transformations Using a Meta-system Approach to Software Development"; *Software Engineering Journal*, 7(6): 425-437, November 1992; IEE on behalf of the BCS and the IEE.

[Booch 1991] Booch, G. (1991); *Object-Oriented Design with Applications*; Benjamin Cummings, Redwood City, California, USA.

[Borgida, Greenspan & Mylopoulos 1985] Borgida, A., S. Greenspan and J. Mylopoulos (1985); "Knowledge Representation as the Basis for Requirements Specification"; *Computer*, 18(4): 82-90, April 1985; IEEE Computer Society Press.

[Bott 1989] Bott, M. F. (1989); *The ECLIPSE Integrated Project Support Environment*; Peter Perigrinus, Stevenage, UK.

[Boudier et al. 1988] Boudier, G., F. Gallo, R. Minot and I. Thomas (1988); "An Overview of PCTE and PCTE+"; *SIGPLAN Notices (Proceedings of SDE3)*, 24(2): 248-257, February 1988, SIGPLAN & ACM Press.

[Bouzeghoub & Comyn-Wattiau 1991] Bouzeghoub, M. and I. Comyn-Wattiau (1991); "View Integration by Semantic Unification and Transformation of Data Structures"; *(In) Entity-Relationship Approach: The Core of Conceptual Modelling;* H. Kangassalo (Ed.); 381-398; Elsevier Science Publishers B.V. (North Holland), Holland.

[Bright, Hurson & Pakzad 1992] Bright, M. W., A. R. Hurson and S. H. Pakzad (1992); "A Taxonomy and Current Issues in Multidatabase Systems"; *Computer*, 25(3): 50-60, March 1992; IEEE Computer Society Press.

[Brinkkemper 1993] Brinkkemper, S. (1993); "Integrating Diagrams in CASE Tools Through Modelling Transparency"; *Information and Software Technology*, 32(2): 101-105, February 1993; Butterworth-Heinemann Ltd.

[Brooks 1987] Brooks, F. P. (1987); "No Silver Bullet: Essence and Accidents of Software Engineering"; *Computer*, 20(4): 10-19, April 1987; IEEE Computer Society Press.

[Brown 1989] Brown, A. W. (1989); *Database Support for Software Engineering*; Kogan Page Ltd., London, UK.

[Brown & McDermid 1992] Brown, A. W. and J. A. McDermid (1992); "Learning from IPSEs Mistakes"; *Software*, 35(3): 23-28, March 1992; IEEE Computer Society Press.

[Buxton 1980] Buxton, J. (1980); “Requirements for Ada Programming Support Environments: Stoneman”; *Technical Report,* US Department of Defense, Washington DC, USA.

[Byte 1989] Byte (1989); “Making the Case for CASE”; *Byte Magazine*, 14(12): 154-171, December 1989.

[CACM 1992] CACM (1992); “Special issue on CASE in the ‘90s”; *Communications of the ACM,* 35(4): 27-64, April 1992; ACM Press.

[Cagan 1990] Cagan, M. R. (1990); “The HP SoftBench Environment: An Architecture for a New Generation of Software Tools”; *Hewlett-Packard Journal*, 41(3): 36-47, June 1990; Hewlett-Packard Inc.

[Cameron 1989] Cameron, J. (1989); *JSP & JSD: The Jackson Approach to Software Development*; 2nd Edition, IEEE Computer Society Press, Washington, USA.

[CASE 1987] CASE (1987); “The State of Automatic Code Generation”; *CASE Outlook*, 1(3): 1-13, September 1987; CASE Consulting Group, Inc., Oregon 97035, USA.

[Castelfranchi, Miceli & Cesta 1992] Castelfranchi, A., M. Miceli and A. Cesta (1992); “Dependence Relations Among Autonomous Agents”; *(In) Decentralized A.I. 3 (Proceedings of 3rd Workshop on Modelling Autonomous Agents in a Multi-Agent World, Kaiserlauten, Germany, 5-7th August 1991);* E. Werner and Y. Demazeau (Eds.); 215-227; Elsevier Science Publishers B.V. (North-Holland), Amsterdam, Holland.

[CDIF 1993] CDIF (1993); “CDIF - CASE Data Interchange Format: Overview”; *Draft document,* PN 3065; Electronic Industries Association (EIA), Washington, DC 20006, USA.

[Checkland 1981] Checkland, P. B. (1981); *Systems Thinking, Systems Practice*; John Wiley & Sons Ltd. (reprinted with corrections 1984), Chichester, UK.

[Chen & Norman 1992] Chen, M. and R. J. Norman (1992); “A Framework for Integrated CASE”; *Software,* 35(3): 18-22, March 1992; IEEE Computer Society Press.

[Chen 1976] Chen, P. (1976); “The entity relationship model - towards a unified view of data”; *Transactions on Database Systems*, 1(1): 9-36, ACM Press.

[Cleaveland 1988] Cleaveland, J. C. (1988); “Building Application Generators”; *Software,* 5(4): 25-34, July 1988; IEEE Computer Society Press.

[Clemm & Osterweil 1990] Clemm, G. and L. Osterweil (1990); “A Mechanism for Environment Integration”; *Transactions on Programming Languages and Systems*, 12(1): 1-25, January 1990; ACM Press.

[Coad & Yourdon 1991] Coad, P. and E. Yourdon (1991); *Object-Oriented Analysis*; 2nd Edition, Yourdon Press, Prentice-Hall, Englewood Cliffs, New Jersey, USA.

[Coleman et al. 1993] Coleman, D., P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremaes (1993); *Object-Oriented Development: The Fusion Method*; Prentice-Hall, Engelwood Cliffs, NJ, USA.

[Comer & Peterson 1989] Comer, D. E. and L. L. Peterson (1989); “Understanding Naming in Distributed Systems”; *Distributed Computing,* 3: 51-60, Springer-Verlag.

[Computer 1983] Computer (1983); "Special Issue on Knowledge Representation"; *Computer*, 16(10): 12-123, October 1983; IEEE Computer Society Press.

[Conklin & Begeman 1988] Conklin, J. and M. L. Begeman (1988); "gIBIS: A Hypertext Tool for Exploratory Policy Discussion"; *Transactions on Office Information Systems*, 6(4): 281-318, October 1988; ACM Press.

[Curtis, Kellner & Over 1992] Curtis, B., M. I. Kellner and J. Over (1992); "Process Modelling"; *Communications of the ACM,* 35(9): 75-90, September 1992; ACM Press.

[Cutkosky et al. 1993] Cutkosky, M. R., R. S. Engelmore, R. E. Fikes, M. R. Genesereth, T. R. Gruber, W. S. Mark, J. M. Tenenbaum and J. C. Weber (1993); "PACT: An Experiment in Integrating Concurrent Engineering Systems"; *Computer*, 26(1): 29-37, January 1993; IEEE Computer Society Press.

[da Costa 1974] da Costa, N. D. (1974); "On the Theory of Inconsistent Formal Systems"; *Notre Dame Journal of Formal Logic*, 15: 497-510.

[Dardenne, Fickas & van Lamsweerde 1993] Dardenne, A., S. Fickas and A. van Lamsweerde (1993); "Goal-directed Requirements Acquisition"; *Science of Computer Programming,* 20: 3-50, 1993.

[Dayal & Hwang 1984] Dayal, U. and H. Hwang (1984); "View Definition and Generalization for Database Integration in a Multidatabase System"; *Transactions on Software Engineering*, 10(6): 629-645, November 1984; IEEE Computer Society Press.

[De Bono 1978] De Bono, E. (1978); *Teaching Thinking*; Pelican Books, Penguin, London, UK.

[De Champeaux & Faure 1992] De Champeaux, D. and P. Faure (1992); "A Comparative Study of Object-Oriented Analysis Methods"; *Journal of Object-Oriented Programming*, 5(2): 21-33, March/April 1992; SIG Publications, Inc.

[DeMarco 1978] DeMarco, T. (1978); *Structured Analysis and System Specification*; Yourdon Press, New York, USA.

[Dictionary 1987] Dictionary (1987); *The Collins Dictionary and Thesaurus*; Collins, UK.

[Digital 1991] Digital (1991); "How to get the most out of CASE applications on Digital workstations"; *Application Guide,* EU-CH278-00; Digital Equipment Company Ltd., Reading, UK.

[Doerry et al. 1991] Doerry, E., S. Fickas, R. Helm and M. Feather (1991); "A Model for Composite System Design"; *Proceedings of 6th International Workshop on Software Specification and Design (IWSSD-6)*, Como, Italy, 25-26 October 1991, 216-219; IEEE Computer Society Press.

[Dowson 1987] Dowson, M. (1987); "Integrated Project Support with ISTAR"; *Software*, 4(6): 6-15, IEEE Computer Society Press.

[Doyle 1979] Doyle, J. (1979); "A Truth Maintenance System"; *Artificial Intelligence*, 12: 231-272, Elsevier.

[Easterbrook et al. 1994] Easterbrook, S., A. Finkelstein, J. Kramer and B. Nuseibeh (1994); "Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check"; *(to appear in) Concurrent Engineering: Research and Applications*, 1994; CERA Institute, West Bloomfield, USA.

[Easterbrook 1991] Easterbrook, S. M. (1991); "Elicitation of Requirements from Multiple Perspectives"; *Ph.D. Thesis,* Department of Computing, Imperial College, London, UK.

[Easterbrook et al. 1993] Easterbrook, S. M., E. E. Beck, J. S. Goodlet, L. Plowman, M. Sharples and C. C. Wood (1993); "A Survey of Empirical Studies of Conflict"; *(In) CSCW: Cooperation or Conflict?;* S. M. Easterbrook (Ed.); 1-68; Springer-Verlag, London.

[Eherer & Jarke 1991] Eherer, S. and M. Jarke (1991); "Knowledge-based Support for Hypertext Co-authoring"; *Proceedings of 2nd International Conference on Database and Expert Systems Applications (DEXA '91),* Berlin, Germany, 465-470; LNCS, Springer-Verlag.

[Evans 1992] Evans, N. (1992); "What are Client Server Systems?"; *The Client Server Group Journal*, 1: 6-7, Tesseract Publications, Farnham, Surrey, UK.

[Feather 1987] Feather, M. S. (1987); "Language Support for the Specification and Development of Composite Systems"; *Transactions on Programming Languages and Systems*, 9(2): 198-234, April 1987; ACM Press.

[Feather 1989a] Feather, M. S. (1989a); "Constructing Specifications by Combining Parallel Elaborations"; *Transactions on Software Engineering*, 15(2): 198-208, February 1989; IEEE Computer Society Press.

[Feather 1989b] Feather, M. S. (1989b); "Detecting Interference When Merging Specification Evolutions"; *Proceedings of 5th International Workshop on Software Specification and Design (IWSSD-5)*, Pittsburgh, Pennsylvania, USA, 19-20th May 1989, 169-176; IEEE Computer Society Press.

[Fernström, Närfelt & Ohlsson 1992] Fernström, C., K.-H. Närfelt and L. Ohlsson (1992); "Software Factory Principles, Architecture and Experiments"; *Software*, 35(3): 36-44, March 1992; IEEE Computer Society Press.

[Finkelstein 1992] Finkelstein, A. (1992); "A Software Process Immaturity Model"; *Software Engineering Notes*, 17(4): 22-23, October 1992; SIGSOFT & ACM Press.

[Finkelstein & Fuks 1989] Finkelstein, A. and H. Fuks (1989); "Multi-party Specification"; *Proceedings of 5th International Workshop on Software Specification and Design (IWSSD-5)*, Pittsburgh, Pennsylvania, USA, 19-20th May 1989, 185-195; IEEE Computer Society Press.

[Finkelstein et al. 1993] Finkelstein, A., D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh (1993); "Inconsistency Handling in Multi-Perspective Specifications"; *Proceedings of 4th European Software Engineering Conference (ESEC '93)*, Garmisch-Partenkirchen, Germany, 13-17 September 1993, 84-99; LNCS, 717, Springer-Verlag.

[Finkelstein et al. 1994] Finkelstein, A., D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh (1994); "Inconsistency Handling in Multi-Perspective Specifications"; *Transactions on Software Engineering*, 20(8): 569-578, August 1994; IEEE Computer Society Press.

[Finkelstein, Goedicke & Kramer 1990] Finkelstein, A., M. Goedicke and J. Kramer (1990); "ViewPoint Oriented Software Development"; *Proceedings of 3rd International Workshop on Software Engineering and its Applications*, Toulouse, France, 3-7th December 1990, 337-351; Cigref EC2.

[Finkelstein et al. 1989] Finkelstein, A., M. Goedicke, J. Kramer and C. Niskier (1989); "ViewPoint Oriented Software Development: Methods and Viewpoints in Requirements Engineering"; *(In) Algebraic Methods II: Theory, Tools and Applications (Proceedings of 2nd METEOR Workshop);* J. A. J. A. Bergstra and L. M. G. Feijs (Eds.); 29-54; LNCS, 490, Springer-Verlag, Germany.

[Finkelstein & Kramer 1991] Finkelstein, A. and J. Kramer (1991); "TARA: Tool-Assisted Requirements Analysis"; *(In) Concept Modelling, Databases and CASE: An Integrated View of Information Systems;* P. Loucopoulous and R. Zicari (Eds.); 413-432; Wiley, UK.

[Finkelstein, Kramer & Hales 1992] Finkelstein, A., J. Kramer and M. Hales (1992); "Process Modelling: A Critical Analysis"; *(In) Integrated Software Engineering with Reuse: Management and Techniques;* P. Walton and N. Maiden (Eds.); 137-148; Chapman & Hall and UNICOM, UK.

[Finkelstein, Kramer & Nuseibeh 1994] Finkelstein, A., J. Kramer and B. Nuseibeh (Eds.) (1994); *Software Process Modelling and Technology*, Advanced Software Development Series, Research Studies Press Ltd. (Wiley), Somerset, UK.

[Finkelstein et al. 1992a] Finkelstein, A., J. Kramer, B. Nuseibeh, L. Finkelstein and M. Goedicke (1992a); "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development"; *International Journal of Software Engineering and Knowledge Engineering*, 2(1): 31-58, March 1992; World Scientific Publishing Co.

[Finkelstein et al. 1992b] Finkelstein, A., B. Nuseibeh, L. Finkelstein and J. Huang (1992b); "Technology Transfer: Software Engineering and Engineering Design"; *Computing and Control Engineering Journal*, 3(6): 259-265, November 1992; IEE.

[Finkelstein et al. 1992c] Finkelstein, L., J. Huang, A. Finkelstein and B. Nuseibeh (1992c); "Using Software Specification Methods for Measurement Instrument Design - Part 1: Structured Methods"; *Measurement - Journal of the International Measurement Confederation*, 10(2): 79-86, April-June 1992; IMEKO.

[Finkelstein et al. 1992d] Finkelstein, L., J. Huang, A. Finkelstein and B. Nuseibeh (1992d); "Using Software Specification Methods for Measurement Instrument Design - Part 2: Formal Methods"; *Measurement - Journal of the International Measurement Confederation*, 10(2): 87-92, April-June 1992; IMEKO.

[Fischer, McCall & Morch 1989] Fischer, G., R. McCall and A. Morch (1989); "Design Environments for Constructive and Argumentative Design"; *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '89)*, Austin, Texas, USA, 30th April-4th May 1989, 269-275; ACM Press.

[Forte & Norman 1992] Forte, G. and R. J. Norman (1992); "A Self-Assessment by the Software Engineering Community"; *Communications of the ACM*, 35(4): 28-32, April 1992; ACM Press.

[Gabbay & Hunter 1991] Gabbay, D. and A. Hunter (1991); “Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Part 1 - A Position Paper”; *Proceedings of the Fundamentals of Artificial Intelligence Research ‘91*, 19-32; LNCS, 535, Springer-Verlag.

[Gabbay & Hunter 1992] Gabbay, D. and A. Hunter (1992); “Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning: Part 2”; *(In) Symbolic and Quantitative Approaches to Reasoning and Uncertainty;* 129-136; LNCS, Springer-Verlag.

[Gane 1990] Gane, C. (1990); *CASE: the methodologies, the products and the future*; Prentice-Hall International, Engelwood Cliffs, New Jersey, USA.

[Gane & Sarson 1979] Gane, C. and T. Sarson (1979); *Structured Analysis and Systems Analysis: Tools and Techniques*; Prentice-Hall International, Engelwood Cliffs, New Jersey, USA.

[Garlan, Cai & Nord 1992] Garlan, D., L. Cai and R. L. Nord (1992); “A Transformational Approach to Generating Application-Specific Environments”; *Software Engineering Notes (Proceedings of ACM SIGSOFT Symposium on Software Development Environments)*, 17(5): 63-77, 9-11th December 1992; SIGSOFT & ACM Press.

[Garlan & Shaw 1993] Garlan, D. and M. Shaw (1993); “An Introduction to Software Architecture”; *(In) Advances in Software Engineering and Knowledge Engineering;* V. Ambriola and G. Tortora (Eds.); 1-39; Series on Software Engineering and Knowledge Engineering, Volume 2, World Scientific, Singapore.

[Gasser 1992] Gasser, L. (1992); “Boundaries, Identity, and Aggregation: Plurality Issues in Multiagent Systems”; *(In) Decentralized A.I. 3 (Proceedings of 3rd Workshop on Modelling Autonomous Agents in a Multi-Agent World, Kaiserlauten, Germany, 5-7th August 1991);* E. Werner and Y. Demazeau (Eds.); 199-213; Elsevier Science Publishers B.V. (North-Holland), Amsterdam, Holland.

[Gera et al. 1992] Gera, M., B. Hirsch, B. Holkamp, J.-P. Moularde, G. Samuel and H. Weber (1992); “CoRe: A Conceptual Reference Model for Software Factories”; *Technical report*, V1.1; ESF, Berlin, Germany.

[Ghezzi, Jazayeri & Mandrioli 1991] Ghezzi, C., M. Jazayeri and D. Mandrioli (1991); *Fundamentals of Software Engineering*; Prentice-Hall, Inc., Engelwood Cliffs, New Jersey, USA.

[Gibson 1989] Gibson, M. L. (1989); “The CASE Philosophy”; *Byte Magazine*, 14(4): 209-218, April 1989.

[Gladden 1982] Gladden, G. R. (1982); “Stop the life cycle - I want to get off”; *Software Engineering Notes,* 7(2): 35-39, SIGSOFT & ACM Press.

[Godart 1993] Godart, C. (1993); “COO: A transaction model to support cooperating software developers coordination”; *Proceedings of 4th European Software Engineering Conference (ESEC ‘93)*, Garmisch-Partenkirchen, Germany, September 1993, 361-379; LNCS, 717, Springer-Verlag.

[Goedicke 1990] Goedicke, M. (1990); “Paradigms of Modular System Development”; *(In) Managing Complexity in Software Engineering;* R. J. Mitchell (Ed.); 1-20; IEE Computing Series, 17, Peter Peregrinus, Stevenage, UK.

[Göhner 1991] Göhner, P. (1991); “Building IPSE’s by Combining Heterogeneous CASE Tools”; *Proceedings of European Symposium on Software Development Environments and CASE Technology*, Königswinter, Germany, 18-30; LNCS, 509, Springer-Verlag.

[Goldberg 1984] Goldberg, A. (1984); *Smalltalk-80: The Interactive Programming Environment*; Addison-Wesley, Reading, Massachusetts, USA.

[Gotel & Finkelstein 1994a] Gotel, O. C. Z. and A. C. W. Finkelstein (1994a); “An Analysis of the Requirements Traceability Problem”; *Proceedings of International Conference on Requirements Engineering (ICRE)*, Utrecht, Holland, April 1994, IEEE Computer Society Press.

[Gotel & Finkelstein 1994b] Gotel, O. C. Z. and A. C. W. Finkelstein (1994b); “Modelling the Contribution Structure Underlying Requirements”; *Proceedings of the 1st International Workshop on Requirements Engineering (REFSQ-94)*, June 1994, Springer-Verlag.

[Graubmann 1992a] Graubmann, P. (1992a); “The HyperView Tool Standard Methods”; *REX technical report*, REX-WP3-SIE-021-V1.0; Siemens, Germany.

[Graubmann 1992b] Graubmann, P. (1992b); “The Petri Net Method ViewPoints in the HyperView Tool”; *REX technical report*, REX-WP3-SIE-023-V1.0; Siemens, Germany.

[Habermann & Notkin 1986] Habermann, A. N. and D. Notkin (1986); “Gandalf: Software Development Environments”; *Transactions on Software Engineering*, 12(12): 1117-1127, IEEE Computer Society Press.

[Hagensen & Kristensen 1992] Hagensen, T. M. and B. B. Kristensen (1992); “Consistency in Software System Development: Framework, Model, Techniques & Tools”; *Software Engineering Notes (Proceedings of ACM SIGSOFT Symposium on Software Development Environments)*, 17(5): 58-67, 9-11th December 1992; SIGSOFT & ACM Press.

[Hahn, Jarke & Rose 1991] Hahn, U., M. Jarke and T. Rose (1991); “Teamworks Support in a Knowledge-based Information Systems Environment”; *Transactions on Software Engineering*, 17(5): 467-482, IEEE Computer Society Press.

[Hailpern 1986] Hailpern, B. (1986); “Special issue on multiparadigm languages”; *Software*, 3(1): 6-77, January 1986; IEEE Computer Society Press.

[Hall 1992a] Hall, P. A. V. (1992a); “Overview of Reverse Engineering and Reuse Research”; *Information and Software Technology*, 34(4): 239-249, April 1992; Butterworth-Heinemann.

[Hall 1992b] Hall, P. A. V. (Ed.) (1992b); *Software Reuse and Reverse Engineering in Practice*, UNICOM Applied Information Technology 12, Chapman & Hall, London, UK.

[Hammersley & Atkinson 1983] Hammersley, M. and P. Atkinson (1983); *Ethnography: Principles in Practice*; Tavistock Publications, London, UK.

[Harmsen & Brinkkemper 1993] Harmsen, F. and S. Brinkkemper (1993); “Computer Aided Method Engineering based on existing Meta-CASE Technology”; *Proceedings of 4th European Workshop on the Next Generation of CASE Tools (NGCT ‘93)*, Sorbonne, Paris, France, 7-8th June 1993, Memorandum Informatica 93-32, University of Twente, Holland.

[Hayes & Schlichting 1987] Hayes, R. and R. D. Schlichting (1987); “Facilitating Mixed Language Programming in Distributed Systems”; *Transactions on Software Engineering*, 13(12): 1254-1264, December 1987; IEEE Computer Society Press.

[Henderson-Sellers & Edwards 1990] Henderson-Sellers, B. and J. M. Edwards (1990); “The Object-Oriented Systems Life Cycle”; *Communications of the ACM*, 33(9): 142-159, September 1990; ACM Press.

[Hoare 1985] Hoare, C. A. R. (1985); *Communicating Sequential Processes*; Prentice-Hall International, Engelwood Cliffs, New Jersey, USA.

[Horowitz, Kemper & Narasimhan 1985] Horowitz, E., A. Kemper and B. Narasimhan (1985); “A Survey of Application Generators”; *Software*, 2(1): 40-54, IEEE Computer Society Press.

[Humphrey 1989] Humphrey, W. S. (1989); *Managing the Software Process*; Addison-Wesley, Reading, Massachusetts, USA.

[Humphrey, Snyder & Willis 1991] Humphrey, W. S., T. R. Snyder and R. R. Willis (1991); “Software Process Improvement at Hughes Aircraft”; *Software*, 8(4): 11-23, IEEE Computer Society Press.

[Hurson, Pakzad & Cheng 1993] Hurson, A. R., S. H. Pakzad and J.-B. Cheng (1993); “Object-Oriented Database Management Systems: Evolution and Performance Issues”; *Computer*, 26(2): 48-60, February 1993; IEEE Computer Society Press.

[IDE 1991] IDE (1991); “Software through Pictures”; *Products & Services Overview*, Interactive Development Environments, Inc., Versailles, France.

[IEEE 1992] IEEE (1992); “IEEE Trial-Use Standard Reference Model for Computing System Tool Interconnections”; *IEEE Std 1175*, ISBN 1-55937-197-8; Standards Coordinating Committee, IEEE Computer Society Press.

[Iverson 1980] Iverson, K. E. (1980); “Notations as a Tool of Thought”; *Communications of the ACM*, 23(8): 444-465, August 1980; ACM Press.

[Jackson 1990] Jackson, M. (1990); “Some Complexities in Computer-Based Systems and Their Implications for System Development”; *Proceedings of International Conference on Computer Systems and Software Engineering (CompEuro ‘90)*, Tel-Aviv, Israel, 8-10th May 1990, 344-351; IEEE Computer Society Press.

[Jackson 1991] Jackson, M. (1991); “Description is Our Business”; *Proceedings of 4th International Symposium of VDM Europe (VDM ‘91), Volume 1: Conference Contributions*, Noordwijkerhout, Holland, October 1991, LNCS, 551, Springer-Verlag.

[Jackson 1993] Jackson, M. (1993); “The Nature of Software Development Method”; *Draft Report*, London, UK.

[Jackson & Zave 1993] Jackson, M. and P. Zave (1993); “Domain Descriptions”; *Proceedings of International Symposium on Requirements Engineering (RE ‘93)*, Coronado Island, San Diego, USA, 4-6th January 1993, 56-64; IEEE Computer Society Press.

[Jackson 1983] Jackson, M. A. (1983); *System Development*; Prentice-Hall International, London, UK.

[Jarke 1992] Jarke, M. (1992); “Strategies for Integrating CASE Environments”; *Software*, 35(3): 54-61, March 1992; IEEE Computer Society Press.

[Jarke & Peters 1993] Jarke, M. and P. Peters (1993); “Method Modelling with ConceptBase: Principles and Experiences”; *Proceedings of 2nd International Summerschool on Method Engineering and Meta-Modelling*, Twente, Holland, 12-16th May 1993, University of Twente.

[Jennings & Mamdani 1992] Jennings, N. R. and E. H. Mamdani (1992); “Using Joint Responsibility to Coordinate Collaborative Problem Solving in Dynamic Environments”; *Proceedings of 10th National Conference on Artificial Intelligence (AAAI ‘92),* San Jose, California, USA, AAAI.

[Johnson 1975] Johnson, S. C. (1975); “Yacc: Yet Another Compiler-Compiler”; *Computer Science Technical Report,* 32; Bell Laboratories, Murray Hill, New Jersey, USA.

[Jones 1990] Jones, G. W. (1990); *Software Engineering*; John Wiley & Sons, Inc., New York, USA.

[Kaplan et al. 1992] Kaplan, S. M., W. J. Tolone, A. M. Carrill, D. P. Bogia and C. Bignoli (1992); “Supporting Collaborative Software Development with ConversationBuilder”; *Software Engineering Notes (Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments)*, 17(5): 11-20, December 1992; SIGSOFT and ACM Press.

[Karam & Casselman 1993] Karam, G. M. and R. S. Casselman (1993); “A Cataloging Framework for Software Development Methods”; *Computer*, 26(2): 34-46, February 1993; IEEE Computer Society Press.

[Kernighan & Pike 1984] Kernighan, B. W. and R. Pike (1984); *The UNIX Programming Environment*; Prentice-Hall, Engelwood-Cliffs, New Jersey, USA.

[Korson & McGregor 1990] Korson, T. and J. D. McGregor (1990); “Understanding Object-Oriented: A Unifying Paradigm”; *Communications of the ACM*, 33(9): 41-60, September 1990; ACM Press.

[Kotonya & Sommerville 1992] Kotonya, G. and I. Sommerville (1992); “Viewpoints for Requirements Definition”; *Software Engineering Journal*, 7(6): 375-387, November 1992; IEE on behalf of the BCS and the IEE.

[Kramer 1991] Kramer, J. (1991); “Configuration Programming - A Framework for the Development of Distributed Systems”; *Proceedings of International Conference on Computer Systems and Software Engineering (CompEuro 90)*, Tel Aviv, Israel, May 1990, 374-384; IEEE Computer Society Press.

[Kramer & Finkelstein 1991] Kramer, J. and A. Finkelstein (1991); "A Configurable Framework for Method and Tool Integration"; *Proceedings of European Symposium on Software development Environments and CASE Technology,* Königswinter, Germany, June 1991, 233-257; LNCS, 509, Springer-Verlag.

[Kramer, Magee & Finkelstein 1990] Kramer, J., J. Magee and A. Finkelstein (1990); "A Constructive Approach to the Design of Distributed Systems"; *Proceeding of 10th International Conference on Distributed Computing Systems*, Paris, France, 28th May-1st June, 580-587; IEEE Computer Society Press.

[Krasner & Pope 1988] Krasner, G. E. and S. T. Pope (1988); "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80"; *Journal of Object Oriented Programming*, 1(3): 26-49, August/September 1988; SIGS Publications, Inc.

[Kronlöf 1993a] Kronlöf, K. (Ed.) (1993a); *Method Integration: Concepts and Case Studies,* Wiley Series in Software-Based Systems, John Wiley & Sons Ltd., Chichester, UK.

[Kronlöf 1993b] Kronlöf, K. (1993b); "Review of Experience"; *(In) Method Integration: Concepts and Case Studies;* K. Kronlöf (Ed.); 309-322; Wiley Series in Software-Based Systems, John Wiley & Sons Ltd., Chichester, UK.

[Kronlöf & Ryan 1993] Kronlöf, K. and K. Ryan (1993); "Conclusions and Recommendations"; *(In) Method Integration: Concepts and Case Studies;* K. Kronlöf (Ed.); 323-340; Wiley Series in Software-Based Systems, John Wiley & Sons Ltd., Chichester, UK.

[Kronlöf, Sheehan & Hallmann 1993] Kronlöf, K., A. Sheehan and M. Hallmann (1993); "The Concept of Method Integration"; *(In) Method Integration: Concepts and Case Studies;* K. Kronlöf (Ed.); 1-18; Wiley Series in Software-Based Systems, John Wiley & Sons Ltd., Chichester, UK.

[Kumar & Welke 1992] Kumar, K. and R. Welke (1992); "Methodology Engineering: A Proposal For Situation-Specific Methodology Construction"; *(In) Challenges and Strategies for Research in Systems Development;* W. W. Cotterman and J. A. Senn (Eds.); 257-269; John Wiley Series in Information Systems, John Wiley & Sons Ltd., Chichester, UK.

[Lai 1993] Lai, F. K. (1993); "CORE in The Viewer"; *M.Sc. Thesis,* Department of Computing, Imperial College, London, UK.

[Larkin & Simon 1987] Larkin, J. H. and H. A. Simon (1987); "Why a Diagram is (Sometimes) Worth Ten Thousand Words"; *Cognitive Science*, 11: 65-99, Ablex.

[Lee & Lai 1991] Lee, J. and K.-Y. Lai (1991); "What's in design Rationale?"; *Human-Computer Interaction*, 6(3-4): 251-280, Lawrence Erlbaum Associates, Inc.

[Lehman 1987] Lehman, M. M. (1987); "Process Models, Process Programs, Programming Support"; *Proceedings of 9th International Conference on Software Engineering (ICSE-9)*, Monterey, California, USA, 30th March - 2nd April 1987, 14-16; IEEE Computer Society Press.

[Lehman 1994] Lehman, M. M. (1994); "Software Evolution"; *(In) Encyclopaedia of Software Engineering;* J. J. Marciniak (Ed.); 1202-1208; 2, Wiley, UK.

[Lehman & Belady 1985] Lehman, M. M. and L. A. Belady (1985); *Program Evolution - Processes of Software Change*; Academic Press, London, UK.

[Leite 1989] Leite, J. C. S. P. (1989); “Viewpoint Analysis: A Case Study”; *Proceedings of 5th International Workshop on Software Specification and Design (IWSSD-5)*, Pittsburgh, Pennsylvania, USA, 19-20th May 1989, 111-119; IEEE Computer Society Press.

[Leite & Freeman 1991] Leite, J. C. S. P. and P. A. Freeman (1991); “Requirements Validation Through Viewpoint Resolution”; *Transactions on Software Engineering*, 12(12): 1253-1269, December 1991; IEEE Computer Society Press.

[Leonhardt 1994] Leonhardt, U. (1994); “Process Modelling in The Viewer”; *M.Eng. thesis*, Department of Computing, Imperial College, London, UK.

[Levy & Silberschatz 1990] Levy, E. and A. Silberschatz (1990); “Distributed File Systems: Concepts and Examples”; *Computing Surveys*, 22(4): 321-374, December 1990; ACM Press.

[Lewis 1990] Lewis, T. (1990); “Code Generators”; *Software*, 7(3): 67-70, May 1990; IEEE Computer Society Press.

[Lindstrom 1993] Lindstrom, D. R. (1993); “Five Ways to Destroy a Development Project”; *Software*, 10(5): 55-58, September 1993; IEEE Computer Society Press.

[Lippe & van Oostrom 1992] Lippe, E. L. and L. van Oostrom (1992); “Operation-Based Merging”; *Software Engineering Notes (Proceedings of ACM SIGSOFT Symposium on Software Development Environments)*, 17(5): 78-87, 9-11th December 1992; SIGSOFT & ACM Press.

[Litwin, Mark & Roussopoulos 1990] Litwin, W., L. Mark and N. Roussopoulos (1990); “Interoperability of Multiple Autonomous Databases”; *ACM Computing Surveys*, 22(3): 267-293, September 1990; ACM Press.

[Luqi 1989] Luqi (1989); “Software Evolution Through Rapid Prototyping”; *Computer*, 22(5): 13-25, May 1989; IEEE Computer Society Press.

[Maddison 1983] Maddison, R. N. (1983); *Information System Methodologies*; Wiley Heyden Ltd. on behalf of the BCS, UK.

[Maiden 1992] Maiden, N. (1992); “Analogical Specification Reuse During Requirements Analysis”; *Ph.D. Thesis,* Department of Business Computing, The City University, London, UK.

[Maiden & Sutcliffe 1992] Maiden, N. and A. Sutcliffe (1992); “Exploiting Reusable Specifications Through Analogy”; *Communications of the ACM*, 35(4): 55-64, April 1992; ACM Press.

[Maiden & Sutcliffe 1993] Maiden, N. and A. Sutcliffe (1993); “Requirements Engineering by Example: An Empirical Study”; *Proceedings of International Symposium on Requirements Engineering (RE ‘93)*, Coronado Island, San Diego, USA, 4-6th January 1993, 104-112; IEEE Computer Society Press.

[Martin 1988] Martin, C. F. (1988); “Second Generation CASE Tools: A Challenge to Vendors”; *Software*, 5(2): 46-49, March 1988; IEEE Computer Society Press.

[McCabe 1993] McCabe, F. G. (1993); “Multi-Agent Systems”; *Seminar*, Department of Computing, Imperial College, London, UK.

[McClure 1989] McClure, C. (1989); “The CASE Experience”; *Byte Magazine*, 14(4): 235-244, April 1989.

[McCracken & Jackson 1982] McCracken, D. D. and M. A. Jackson (1982); “Life cycle concept considered harmful”; *Software Engineering Notes*, 7(2): 28-32, SIGSOFT & ACM Press.

[MEMM 1993] MEMM (1993); *Proceedings of 2nd International Summerschool on Method Engineering and Meta-Modelling*, Twente, Holland, 12-16th May 1993, University of Twente.

[Meyers 1991] Meyers, S. (1991); “Difficulties in Integrating Multiview Development Systems”; *Software*, 8(7): 49-57, January 1991; IEEE Computer Society Press.

[Meyers 1993] Meyers, S. (1993); “Representing Software Systems in Multiple-View Development Environments”; *Ph.D. Thesis*, CS-93-18; Department of Computer Science, Brown University, USA.

[Meyers & Reiss 1991] Meyers, S. and S. P. Reiss (1991); “A System for Multiparadigm Development of Software Systems”; *Proceedings of 6th International Workshop on Software Specification and Design (IWSSD-6)*, Como, Italy, 25-26th October 1991, 202-209; IEEE Computer Society Press.

[Meyers & Reiss 1992] Meyers, S. and S. P. Reiss (1992); “An Empirical Study of Multiple-View Software Development”; *Software Engineering Notes (Proceedings of 5th ACM SIGSOFT Symposium on Software Development Environments)*, 17(5): 47-57, December 1992; SIGSOFT & ACM Press.

[Mi & Scacchi 1992] Mi, P. and W. Scacchi (1992); “Process Integration in CASE Environments”; *Software*, 35(3): 45-53, March 1992; IEEE Computer Society Press.

[Mikes 1990] Mikes, S. (1990); *X Window System Technical Reference*; Addison-Wesley, Reading, Massachusetts, USA.

[Milner 1989] Milner, R. (1989); *Communication and Concurrency*; Prentice-Hall International, Hemel Hempstead, Herts., UK.

[Minsky 1975] Minsky, M. (1975); “A Framework for Representing Knowledge”; *(In) The Psychology of Computer Vision;* P. Winston (Ed.); McGraw-Hill, New York, USA.

[Misra & Jalics 1988] Misra, S. and P. Jalics (1988); “Third-generation versus Fourth-generation Software Development”; *Software*, 5(4): 8-14, July 1988; IEEE Computer Society Press.

[Mödl 1991] Mödl, R. (1991); “A Formal Language for the Style Definition Part of a Template”; *REX project report*, REX-WP3-SIE-017-V1.0; Siemens, Munich, Germany.

[Mullender 1989] Mullender, S. (Ed.) (1989); *Distributed Systems*, Frontier Series, ACM Press (Addison-Wesley Publishing Company), New York, USA.

[Mullery 1979] Mullery, G. (1979); “CORE - a method for controlled requirements expression”; *Proceedings of 4th International Conference on Software Engineering (ICSE-4)*, 126-135; IEEE Computer Society Press.

[Mullery 1985] Mullery, G. (1985); "Acquisition - Environment"; *(In) Distributed Systems: Methods and Tools for Specification;* M. Paul and H. Siegert (Eds.); LNCS, 190, Springer-Verlag.

[Mullery & Newbury 1991] Mullery, G. P. and M. R. Newbury (1991); "Method Engineering: Methods via Methodology"; *Technical Report,* Systemic Methods Ltd., Farnborough, Hants. GU14 6SR, UK.

[Munck, Oberndorf & Ploedereder 1989] Munck, R., P. Oberndorf and E. Ploedereder (1989); "An overview of DOD-STD-1838A (proposed), the common APSE interface set, revision A"; *Software Engineering Notes,* 13(5): 235-247, SIGSOFT & ACM Press.

[Mylopoulos et al. 1990] Mylopoulos, J., A. Borgida, M. Jarke and M. Koubarakis (1990); "Telos: A Language for Representing Knowledge about Information Systems"; *Transactions on Information Systems,* 8(4): 325-362, ACM Press.

[Narayanaswamy & Goldman 1992] Narayanaswamy, K. and N. Goldman (1992); ""Lazy" Consistency: A Basis for Cooperative Software Development"; *Proceedings of International Conference on Computer-Supported Cooperative Work (CSCW '92),* Toronto, Ontario, Canada, 31st October - 4th November, 257-264; ACM SIGCHI & SIGOIS.

[Navathe, Elmasri & Larson 1986] Navathe, S. B., R. Elmasri and J. Larson (1986); "Integrating User Views in Database Design"; *Computer,* 19(1): 50-62, January 1986; IEEE Computer Society Press.

[Neighbours 1984] Neighbours, J. M. (1984); "The Draco Approach to Constructing Software from Reusable Components"; *Transactions on Software Engineering,* 10(5): 564-574, September 1984; IEEE Computer Society Press.

[Niskier, Maibaum & Schwabe 1989a] Niskier, C., T. Maibaum and D. Schwabe (1989a); "A Look Through PRISMA: Towards Pluralistic Knowledge-Based Environments for Specification Acquisition"; *Proceedings of 5th International Workshop on Software Specification and Design (IWSSD-5),* Pittsburgh, Pennsylvania, USA, 19-20th May 1989, 128-136; IEEE Computer Society Press.

[Niskier, Maibaum & Schwabe 1989b] Niskier, C., T. Maibaum and D. Schwabe (1989b); "A Pluralistic Knowledge-Based Approach to Software Specification"; *Proceedings of 2nd European Software Engineering Conference (ESEC '89),* University of Warwick, Coventry, UK, 11-15th September 1989, 411-423; LNCS, 387, Springer-Verlag.

[Norman & Chen 1992] Norman, R. J. and M. Chen (1992); "Guest Editors' Introduction to special issue on Integrated CASE"; *Software,* 35(3): 13-16, March 1992; IEEE Computer Society Press.

[Nuseibeh 1989] Nuseibeh, B. (1989); "CoreDemo: An Investigation into the Use of Object-Oriented Techniques for the Construction of CASE Tools"; *M.Sc. thesis,* Department of Computing, Imperial College, London, UK.

[Nuseibeh 1994] Nuseibeh, B. (1994); "Towards a Protocol for Inter-ViewPoint Communication"; *Draft report,* Department of Computing, Imperial College, London, UK.

[Nuseibeh & Finkelstein 1992] Nuseibeh, B. and A. Finkelstein (1992); “ViewPoints: A Vehicle for Method and Tool Integration”; *Proceedings of 5th International Workshop on Computer-Aided Software Engineering (CASE ‘92),* Montreal, Canada, 6-10th July 1992, 50-60; IEEE Computer Society Press.

[Nuseibeh, Finkelstein & Kramer 1993] Nuseibeh, B., A. Finkelstein and J. Kramer (1993); “Fine-Grain Process Modelling”; *Proceedings of 7th International Workshop on Software Specification and Design (IWSSD-7)*, Redondo Beach, California, USA, 6-7 December 1993, 42-46; IEEE Computer Society Press.

[Nuseibeh, Finkelstein & Kramer 1994] Nuseibeh, B., A. Finkelstein and J. Kramer (1994); “Method Engineering for Multi-Perspective Software Development”; *(to appear in) Information and Software Technology*; Butterworth-Heinemann.

[Nuseibeh, Kramer & Finkelstein 1993] Nuseibeh, B., J. Kramer and A. Finkelstein (1993); “Expressing the Relationships Between Multiple Views in Requirements Specification”; *Proceedings of 15th International Conference on Software Engineering (ICSE-15),* Baltimore, Maryland, USA, 17-21 May 1993, 187-200; IEEE Computer Society Press.

[Nuseibeh, Kramer & Finkelstein 1994] Nuseibeh, B., J. Kramer and A. Finkelstein (1994); “A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification”; *Transactions on Software Engineering*, 20(10), October 1994, IEEE Computer Society Press.

[Obbink 1991] Obbink, H. (1991); “Systems Engineering Environments of ATMOSPHERE”; *Proceedings of European Symposium on Software Development Environments and CASE Technology,* Königswinter, Germany, June 1991, 1-17; LNCS, 509, Springer-Verlag.

[Oberndorf 1988] Oberndorf, P. A. (1988); “The Common Ada Programming Support Environment (APSE) Interface Set (CAIS)”; *Transactions on Software Engineering*, 14(6): 742-748, June 1988; IEEE Computer Society Press.

[ObjectMaker 1991] ObjectMaker (1991); “ObjectMaker: A Technical Overview”; *Technical document,* Mark V Systems Limited, Encino, CA 91436, USA.

[Oddy 1990] Oddy, G. C. (1990); “Development Support Environments”; *(In) Managing Complexity in Software Engineering;* R. J. Mitchell (Ed.); 201-215; IEE Computing Series, 17, Peter Peregrinus, Stevenage, UK.

[Olle et al. 1991] Olle, T. W., J. Hagelstein, I. G. Macdonald, H. G. Sol, F. J. M. Van Assche and A. A. Verrijn-Stuart (Eds.) (1991); *Information Systems Design Methodologies: A Framework For Understanding,* Addison-Wesley Publishing Company, UK.

[Olle, Sol & Tully 1983] Olle, T. W., H. G. Sol and C. J. Tully (Eds.) (1983); *Information Systems Design Methodologies: A Feature Analysis*, Comparative Review of Information Systems design Methodologies, Elsevier Science Publishers B.V. (North-Holland), Amsterdam, Holland.

[OSF 1990] OSF (1990); “OSF/Motif Revision 1.1”; *Technical Manual,* Open Software Foundation, Cambridge, MA 02142, USA.

[Osterweil 1987] Osterweil, L. (1987); "Software Processes Are Software Too"; *Proceedings of 9th International Conference on Software Engineering (ICSE-9)*, Monterey, California, USA, 3rd March - 2nd April 1987, 2-13; IEEE Computer Society Press.

[Pequendo & Buschbaum 1991] Pequendo, T. and A. Buschbaum (1991); "The Logic of Epistemic Inconsistency"; *Proceedings of 2nd International Conference on the Principles of Knowledge Representation and Reasoning*, 453-460; Morgan Kaufmann.

[Perry 1990] Perry, D. E. (1990); "Policy and Product-Directed Process Instantiation"; *Proceedings of 6th International Software Process Workshop (ISPW-6)*, Hakodate, Japan, October 1990, IEEE Computer Society Press.

[Perry 1991] Perry, D. E. (1991); "Policy-Directed Coordination and Cooperation"; *Proceedings of 7th International Software Process Workshop (ISPW-7)*, Yountville, USA, October 1991, IEEE Computer Society Press.

[Perry 1992] Perry, D. E. (1992); "Humans in the Process: Architectural Implications"; *Proceedings of 8h International Software Process Workshop (ISPW-8)*, IEEE Computer Society Press.

[Petre & Winder 1988] Petre, M. and R. Winder (1988); "Issues Governing the Suitability of Programming Languages for Programming Tasks"; *(In) People and Computers IV;* D. M. Jones and R. Winder (Eds.); 199-215; Cambridge University Press, UK.

[Pocock 1991a] Pocock, J. N. (1991a); "Distributed Software Development and VSF"; *Proceedings of Colloquium on Architectures for Distributed Development Support Environments*, IEE, Savoy Place, London, UK, 4th November 1991, 6/1-6/5; Digest Number 1991/162, Computing and Control Division, Institution of Electrical Engineers (IEE).

[Pocock 1991b] Pocock, J. N. (1991b); "VSF and its Relationship to Open Systems and Standard Repositories"; *Proceedings of European Symposium on Software development Environments and CASE Technology*, Königswinter, Germany, June 1991, 53-68; LNCS, 509, Springer-Verlag.

[Polack, Whiston & Mander 1993] Polack, F., M. Whiston and K. C. Mander (1993); "The SAZ Method - Version 1.0"; *Technical report*, YCS 207 (1993); Department of Computer Science, University of York, York, UK.

[Potts 1989] Potts, C. (1989); "A Generic Model for Representing Design Methods"; *Proceedings of 11th International Conference on Software Engineering (ICSE-11)*, Pittsburgh, USA, 15-18th May 1989, 217-226; IEEE Computer Society Press.

[Potts & Bruns 1988] Potts, C. and G. Bruns (1988); "Recording the Reasons for Design Decisions"; *Proceedings of 10th International Conference on Software Engineering (ICSE-10)*, Singapore, 11-15th April 1988, 418-427; IEEE Computer Society Press.

[Pressman 1992] Pressman, R. S. (1992); *Software Engineering: A Practitioner's Approach*; 3rd Edition, McGraw-Hill, Inc., New York, USA.

[Prieto-Díaz 1990] Prieto-Díaz, R. (1990); "Domain Analysis: An Introduction"; *Software Engineering Notes*, 15(2): 47-54, April 1990; SIGSOFT & ACM Press.

[Prieto-Díaz 1991] Prieto-Díaz, R. (1991); “Making Software Reuse Work: An Implementation Model”; *Software Engineering Notes*, 16(3): 61-68, July 1991; SIGSOFT & ACM Press.

[Puncello et al. 1988] Puncello, P. P., P. Torrigiani, F. Pietri, R. Burlon, B. Cardile and M. Conti (1988); “ASPIS: A Knowledge-Based CASE Environment”; *Software*, 5(2): 58-65, March 1988; IEEE Computer Society Press.

[Ram 1991] Ram, S. (1991); “Heterogeneous Distributed Database Systems”; *Computer*, 24(12): 7-10, December 1991; IEEE Computer Society Press.

[Rational 1992] Rational (1992); “Rose: Rational Object-Oriented Software Engineering”; *Product Overview*, D-66B; Rational Technology Ltd., Brighton, UK.

[Reiss 1985] Reiss, S. P. (1985); “PECAN: Program Development Systems that Support Multiple Views”; *Transactions on Software Engineering*, 11(3): 276-285, March 1985; IEEE Computer Society Press.

[Reiss 1987] Reiss, S. P. (1987); “Working in the Garden Environment for Conceptual Programming”; *Software*, 4(6): 16-27, November 1987; IEEE Computer Society Press.

[Reiss 1990] Reiss, S. P. (1990); “Connecting Tools using Message Passing in the FIELD Program Development Environment”; *Software*, 7(4): 57-67, July 1990; IEEE Computer Society Press.

[Reiter 1978] Reiter, R. (1978); “On Closed World Databases”; *(In) Logic & Databases;* H. Gallaire and J. Minker (Eds.); Plenum Press.

[Rekers 1993] Rekers, J. (1993); “Graphical Definition of Graphical Syntax”; *Technical report*, Department of Computer Science, Leiden University, Holland.

[Rich 1981] Rich, C. (1981); “Multiple Points of View in Modelling Programs”; *SIGPLAN Notices*, 16(1): 177-179, January 1981; SIGPLAN & ACM Press.

[Rich & Waters 1990] Rich, C. and R. C. Waters (1990); *The Programmer’s Apprentice*; ACM Press, New York, USA.

[Robinson 1989] Robinson, W. (1989); “Integrating Multiple Specifications Using Domain Goals”; *Proceedings of 5th International Workshop on Software Specification and Design (IWSSD-5)*, Pittsburgh, Pennsylvania, USA, 19-20th May 1989, 219-226; IEEE Computer Society Press.

[Robinson 1990] Robinson, W. (1990); “Negotiation Behavior During Requirements Specification”; *Proceedings of 12th International Conference on Software Engineering (ICSE-12)*, Nice, France, 26-30th March 1990, 268-276; IEEE Computer Society Press.

[Robinson & Fickas 1994] Robinson, W. and S. Fickas (1994); “Supporting Multi-Perspective Requirements Engineering”; *Proceedings of 1st International Conference on Requirements Engineering*, Colorado Springs, Colorado, USA, 18-22nd April 1994, 206-215; IEEE Computer Society Press.

[Ross 1977] Ross, D. T. (1977); “Structured Analysis (SA): A Language for Communicating Ideas”; *Transactions on Software Engineering*, 3(1): 16-34, January 1977; IEEE Computer Society Press.

[Ross 1985] Ross, D. T. (1985); “Applications and Extensions of SADT”; *Computer*, 18(4): 25-34, April 1985; IEEE Computer Society Press.

[Ross & Schoman 1977] Ross, D. T. and K. E. Schoman (1977); “Structured Analysis for Requirements Definition”; *Transactions on Software Engineering*, 3(1): 6-15, January 1977; IEEE Computer Society Press.

[Rossi 1993] Rossi, M. O. (1993); “Meta-CASE”; *Personal Communication (email),* 30th December 1993; University of Jyvaskyla, Finland.

[Royce 1970] Royce, W. W. (1970); “Managing the development of large software systems: concepts and techniques”; *Proceedings of WESCON*, San Francisco, California, USA, August 1970, 1-9; IEEE Press.

[Rumbaugh et al. 1991] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen (1991); *Object-Oriented Modelling and Design*; Prentice-Hall, Inc., Engelwood Cliffs, New Jersey, USA.

[Rusinkiewicz, Sheth & Karabatis 1991] Rusinkiewicz, M., A. Sheth and G. Karabatis (1991); “Specifying Interdatabase Dependencies in a Multidatabase Environment”; *Computer,* 24(12): 46-53, December 1991; IEEE Computer Society Press.

[Sadri & Kowalski 1986] Sadri, F. and R. Kowalski (1986); “An Application of General Theorem Proving to Database Integrity”; *Technical report,* Department of Computing, Imperial College, London, UK.

[Satyanarayanan 1991] Satyanarayanan, M. (1991); “An Agenda for Research in Large-Scale Distributed Data Repositories”; *Proceedings of International Workshop on Operating Systems of the 90s and Beyond*, Germany, July 1991, 2-12; LNCS, 563, Springer-Verlag.

[Schwanke & Kaiser 1988] Schwanke, R. W. and G. E. Kaiser (1988); “Living With Inconsistency in Large Systems”; *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, Germany, 27-29 January 1988, 98-118; B. G. Teubner, Stuttgart.

[SEI 1991a] SEI (1991a); “Capability Maturity Model for Software”; *Technical report*, CMU/SEI-91-TR-24; Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, USA.

[SEI 1991b] SEI (1991b); “Key Practices of the Capability Maturity Model”; *Technical report*, CMU/SEI-91-TR-25; Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, USA.

[SEJ 1991] SEJ (1991); “Special Issue on Software Process and its Support”; *Software Engineering Journal*, 6(5): 229-371, September 1991; IEE on behalf of the BCS and the IEE.

[Semmens, France & Docker 1992] Semmens, L. T., R. B. France and T. W. G. Docker (1992); “Integrated Structured Analysis and Formal Specification Techniques”; *Computer Journal*, 35(6): 600-610, 1992; Cambridge University Press on behalf of the BCS.

[Shilling & Sweeney 1989] Shilling, J. J. and P. F. Sweeney (1989); “Three Steps to Views: Extending the Object-Oriented Paradigm”; *SIGPLAN Notices (Proceedings of OOPSLA ‘89, New Orleans, Louisiana, USA)*, 24(10): 353-361, October 189; ACM Press.

[Shoham 1990] Shoham, Y. (1990); “Agent-Oriented Programming”; *Technical report,* STAN-CS-1335-90; Robotics Laboratory, Computer Science Department, Stanford University, USA.

[Simon 1981] Simon, H. A. (1981); *The Sciences of the Artificial*; 2nd Edition, MIT Press, Cambridge, Massachusetts, USA.

[Smith & Davis 1981] Smith, R. G. and R. Davis (1981); “Frameworks for Cooperation in Distributed Data Repositories”; *Transactions on Systems, Man and Cybernetics*, 11(1): 61-70, January 1981; IEEE Computer Society Press.

[Smolander 1993] Smolander, K. (1993); “Meta-CASE”; *Internet News (article 17708 on comp.software-eng news group),* 28th December 1993; University of Jyvaskyla, Finland.

[Software 1988] Software (1988); “Special issue on CASE”; *Software*, 5(2): 8-72, March 1988; IEEE Computer Society Press.

[Software 1992] Software (1992); “Special issue on Integrated CASE”; *Software*, 35(3): 12-76, March 1992; IEEE Computer Society Press.

[Sommerville 1992] Sommerville, I. (1992); *Software Engineering*; Fourth Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, USA.

[Sommerville 1993] Sommerville, I. (1993); “Environment Repositories Considered Harmful”; *Proceedings of International Conference of Software Engineering Environments (SEE ‘93)*, Reading, UK, 7-10 July 1993, IEEE Computer Society Press.

[Sommerville et al. 1992] Sommerville, I., R. Bentley, T. Rodden, P. Sawyer, J. Hughes, D. Shapiro and D. Randell (1992); “Ethnographically-informed Systems Design for Air Traffic Control”; *Proceedings of International Conference on Computer-Supported Cooperative Work (CSCW ‘92)*, Toronto, Ontario, Canada, 31st October - 4th November, ACM SIGCHI & SIGOIS.

[Sommerville et al. 1993] Sommerville, I., T. Rodden, P. Sawyer, R. Bentley and M. Twidale (1993); “Integrating Ethnography into the Requirements Engineering Process”; *Proceedings of International Symposium on Requirements Engineering (RE ‘93)*, Coronado Island, San Diego, USA, 4-6th January 1993, 165-173; IEEE Computer Society Press.

[Sommerville, Welland & Beer 1987] Sommerville, I., R. C. Welland and S. Beer (1987); “Describing Software Design Methodologies”; *Computer Journal*, 30(2): 128-133, April 1987; Cambridge University Press on behalf of the BCS.

[Sorenson & Tremblay 1993] Sorenson, P. and J.-P. Tremblay (1993); “Using a Metasystem Approach to Support and Study the Design Process”; *Proceedings of the Workshop on Studies of Software Design (Technical Report),* ISSN-0836-0227-93-352; Department of Computing and Information Science, Queen’s University, Kingston, Ontario, Canada.

[Sorenson, Tremblay & McAllister 1988] Sorenson, P., J.-P. Tremblay and A. J. McAllister (1988); “The Metaview System for Many Specification Environments”; *Software*, 5(2): 30-38, March 1988; IEEE Computer Society Press.

[Spinellis 1994] Spinellis, D. (1994); “Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming”; *Ph.D. Thesis,* Department of Computing, Imperial College, London, UK.

[Spivey 1989] Spivey, J. M. (1989); *The Z Notation: A Reference Manual*; Prentice-Hall International, UK.

[Sriram & Logcher 1993] Sriram, D. and R. Logcher (1993); “The MIT Dice Project”; *Computer*, 26(1): 64-65, January 1993; IEEE Computer Society Press.

[Svoboda 1993] Svoboda, C. (1993); “Flexible Automation with Meta-CASE”; *Results from the 5th International Workshop on Computer-Aided Software Engineering (CASE ‘92),* Montréal, Québec, Canada, 26; OCRI Publications.

[Symonds 1988] Symonds, A. J. (1988); “Creating a Software-Engineering Knowledge Base”; *Software*, 5(2): 50-56, March 1988; IEEE Computer Society Press.

[Taylor et al. 1989] Taylor, R. N., R. W. Selby, M. Young, F. C. Belz, L. A. Clarke, J. C. Wileden, L. Osterweil and A. L. Wolf (1989); “Foundations for the Arcadia Environment Architecture”; *SIGPLAN Notices (Proceedings of SDE3, Boston, 28-30th November 1988)*, 24(2): 1-13, February 1989; SIGPLAN & ACM Press.

[Teitelman & Masinter 1981] Teitelman, W. and L. Masinter (1981); “The Interlisp Programming Environment”; *Computer*, 14(4): 25-34, April 1981; IEEE Computer Society Press.

[Thanitsukkarn 1993] Thanitsukkarn, T. (1993); “The Constructive Viewer”; *M.Sc. Thesis,* Department of Computing, Imperial College, London, UK.

[Thomas 1989a] Thomas, I. (1989a); “PCTE Interfaces: Supporting Tools in Software Engineering Environments”; *Software*, 6(6): 15-23, November 1989; IEEE Computer Society Press.

[Thomas 1989b] Thomas, I. (1989b); “Tool Integration in the Pact Environment”; *Proceedings of 11th International Conference on Software Engineering (ICSE-11)*, Pittsburgh, USA, 15-18 May 1989, 13-22; IEEE Computer Society Press.

[Thomas & Nejmeh 1992] Thomas, I. and B. Nejmeh (1992); “Definitions of Tool Integration for Environments”; *Software*, 35(3): 29-35, IEEE Computer Society Press.

[Tomek 1992] Tomek, I. (Ed.) (1992); *Proceedings of 4th International Conference on Computer Assisted Learning (ICCAL ‘92),* LNCS, 602, Springer-Verlag, Wolfville, Nova Scotia, Canada.

[Tontsch 1990] Tontsch, F. (1990); “Methods and Tools”; *(In) Managing Complexity in Software Engineering;* R. J. Mitchell (Ed.); 181-199; IEE Computing Series, 17, Peter Peregrinus, Stevenage, UK.

[van Vliet 1993] van Vliet, H. (1993); *Software Engineering*; John Wiley & Sons Ltd., Chichester, UK.

[Vivarès & Durieux 1992] Vivarès, F. and J. L. Durieux (1992); “Method Modelling”; *Technical Report*, ONERA-CERT-DERI, Toulouse, France.

[Voelcker 1988] Voelcker, J. (1988); “Automating Software: proceed with caution”; *Spectrum*, 25(7): 25-27, July 1988; IEEE Press.

[Wagner 1991] Wagner, G. (1991); “Ex contradictione nihil sequitor”; *Proceedings of 12th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann.

[Wand & Weber 1990] Wand, Y. and R. Weber (1990); “An Ontological Model of an Information System”; *Transactions on Software Engineering*, 16(11): 1282-1292, November 1990; IEEE Computer Society Press.

[Wasserman 1990] Wasserman, A. I. (1990); “Tool Integration in Software Engineering Environments”; *Proceedings of International Workshop on Environments*, Chinon, France, September 1989, 137-149; LNCS, 467, Springer-Verlag.

[Wasserman, Freeman & Porcella 1983] Wasserman, A. I., P. Freeman and M. Porcella (1983); “Characteristics of Software Development Methodologies”; *(In) Information Systems Design Methodologies: A Feature Analysis;* T. W. Olle, H. G. Sol and C. J. Tully (Eds.); 37-57; Elsevier Science Publishers B.V. (North-Holland), Amsterdam, Holland.

[Wegner 1987] Wegner, P. (1987); “Dimensions of Object-Based Language Design”; *SIGPLAN Notices (Proceedings of OOPSLA ‘87, Orlando, Florida, USA, 4-8th October 1987)*, 22(12): 168-182, December 1987; SIGPLAN & ACM Press.

[Weihmayer & Brandau 1990] Weihmayer, R. and R. Brandau (1990); “Cooperative Distributed Problem Solving for Communication Network Management”; *Computer Communications*, 3(9): 547-557, November 1990; Butterworth-Heinemann Ltd.

[Welke 1989] Welke, R. J. (1989); “Meta Systems on Meta Models”; *CASE Outlook*, 3: 35-45, December 1989; CASE Consulting Group, Inc., Oregon 97035, USA.

[Welke 1992] Welke, R. J. (1992); “The CASE Repository: More than another database application”; *(In) Challenges and Strategies for Research in Systems Development;* W. W. Cotterman and J. A. Senn (Eds.); 181-218; John Wiley Series in Information Systems, John Wiley & Sons Ltd., Chichester, UK.

[Welland, Beer & Sommerville 1990] Welland, R. C., S. Beer and I. Sommerville (1990); “Method Rule Checking in a Generic Design Editing System”; *Software Engineering Journal*, 5(2): 105-115, March 1990; IEE on behalf of the BCS and the IEE.

[Werner 1992] Werner, E. (1992); “The Design of Multi-Agent Systems”; *(In) Decentralized A.I. 3 (Proceedings of 3rd Workshop on Modelling Autonomous Agents in a Multi-Agent World, Kaiserlauten, Germany, 5-7th August 1991);* E. Werner and Y. Demazeau (Eds.); 3-28; Elsevier Science Publishers B.V. (North-Holland), Amsterdam, Holland.

[White & Purtilo 1992] White, E. L. and J. M. Purtilo (1992); “Integrating the Heterogeneous Control Properties of Software Modules”; *Software Engineering Notes (Proceedings of 5th ACM SIGSOFT Symposium on Software Development Environments)*, 17(5): 99-108, December 1992; SIGSOFT & ACM Press.

[Wile 1986] Wile, D. S. (1986); “Local Formalisms: Widening the Spectrum of Wide-Spectrum Languages”; *(In) Program Specification and Transformation (Proceedings of IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation, Bed Tölz, 15-17th April 1986);* L. G. L. T. Meertens (Ed.); 459-482; Elsevier Science B.V., Holland.

[Wile 1992] Wile, D. S. (1992); “Integrating Syntaxes and their Associated Semantics”; *Technical report,* RR-92-297; USC/Information Sciences Institute, University of Southern California, Marina del Rey, California, USA.

[Wileden et al. 1990] Wileden, J. C., A. I. Wolf, W. R. Rosenblatt and P. L. Tarr (1990); “Specification Level Interoperability”; *International Conference on Software Engineering (ICSE-12),* Nice, France, 26-30th March 1990, 74-85; IEEE Computer Society Press.

[Wileden et al. 1992] Wileden, J. C., A. I. Wolf, W. R. Rosenblatt and P. L. Tarr (1992); “Specification-Level Interoperability”; *Communications of the ACM*, 34(5): 72-87, May 1991; ACM Press.

[Woods 1983] Woods, W. A. (1983); “What’s Important About Knowledge Representation?”; *Computer*, 16(10): 22-27, October 1983; IEEE Computer Society Press.

[Wybolt 1991] Wybolt, N. (1991); “Perspectives on CASE Tool Integration”; *Software Engineering Notes*, 16(3): 56-60, July 1991; SIGSOFT & ACM Press.

[Yin 1994] Yin, S. G. (1994); “Tool Integration Through Repositories”; *Software*, 11(1): 123-126, January 1994; IEEE Computer Society Press.

[Yourdon & Constantine 1979] Yourdon, E. and L. L. Constantine (1979); *Structured Design*; Prentice-Hall International, Engelwood Cliffs, New Jersey, USA.

[Zahniser 1993] Zahniser, R. A. (1993); “Design by Walking Around”; *Communication of the ACM*, 36(10): 115-123, October 1993; ACM Press.

[Zave 1989] Zave, P. (1989); “A Compositional Approach to Multiparadigm Programming”; *Software*, 5(5): 15-25, September 1989; IEEE Computer Society Press.

[Zave & Jackson 1993] Zave, P. and M. Jackson (1993); “Conjunction as Composition”; *Transactions on Software Engineering and Methodology*, 2(4): 379-411, October 1993; ACM Press.

[Zave & Schell 1986] Zave, P. and W. Schell (1986); “Salient Features of an Executable Specification Language and its Environment”; Transactions on Software Engineering, 12(2): 312-325, February 1986; IEEE Computer Society Press.

Appendix A Tool Documentation

A.1. System Requirements

Program:	*The Viewer*
Author:	B. A. Nuseibeh
Version:	0.9999
Language:	Objectworks/Smalltalk R4.0

The Viewer is implemented in ParcPlace Objectworks/Smalltalk R4.0 and requires a valid Smalltalk runtime licence. It also operates under R4.1, but cannot be extended under that release (since the Smalltalk class hierarchy of R4.0 was substantially changed in R4.1).

The Viewer requires 4-5 MB of RAM and about 5 MB hard disk space after installation.

It runs on a variety of platforms including the Apple Macintosh, UNIX workstations running X-Windows, and IBM PCs and compatibles running MS-Windows. It was implemented on an Apple Macintosh IIci and tested on a Sun SparcStation running X-Windows.

It requires a 12” monitor and prefers a 16” monitor or larger (although it is usable with smaller, 9’’, monitors).

It runs on black and white monitors, although again, a colour monitor is preferred for optimal use. Presentation using a black and white monitor can be improved by running a utility developed by the author, invoked by selecting and executing the expression:

```
Color setPlatform: ‘B/W’
```

A.2. User’s Guide

Familiarity with the Objectworks/Smalltalk environment is assumed. This includes opening and closing windows, popping-up menus, selecting items from menus and text, and starting and quitting the system. Chapter 8 of this thesis describing *The Viewer* should also be read as essential background. Familiarity with the ViewPoints framework and its underlying concepts is also required. If *The Viewer* has not already been installed, then consult the next section (A.3. Programmer's Guide) for installation details.

If the startup window of *The Viewer* is not already open, then one can be opened by selecting and executing the expression:

```
VoseView open
```

A TemplateBrowser may then be invoked by clicking on the "Method Designer" button, while a ViewPointConfigurationBrowser may be invoked by clicking on the "Method User" button.

The Viewer is fairly robust and tolerant of user mistakes. Extensive error handing is built-in, and error messages are explicit in describing the nature of their corresponding errors.

Operating *The Viewer* on an Apple Macintosh requires the "option" and "command" keys to replace the other two mouse buttons on three-button-mouse machines.

If an X-Windows version is being used, it is easier to set X-windows to bring forward windows only when a mouse button is clicked on them. Otherwise, smaller windows and pop-up menus may inadvertently "disappear" into the background, if the mouse pointer is moved accidentally off these windows and menus (they can easily be recovered however, by moving or collapsing the top-level windows).

The Viewer occasionally invokes a garbage collector which is indicated by an "garbage can" icon in place of the normal cursor.

ViewPoint states are not automatically updated/changed. This has to be done manually by selecting the appropriate state via the top work record menu in a ViewPointInspector.

Finally, the debate over my choice of colours still rages on!

A.3. Programmer's Guide

Familiarity with the Objectworks/Smalltalk language and programming environment is assumed, including the use of tools such as the Class Hierarchy Browser, inspectors, debuggers and so on.

If a running "image" of *The Viewer* is not available, then a "fresh" image may be created by "filing-in" the following class categories:

Class Category	Classes
VOSE-Utilities	ErrorMessenger Color MultiInputDialogView MessagesView TitleView TextCodeController TextSavingController LogoController LogoView Setup SetupDataModel
VOSE-Graphics	GraphicsEditorView GraphicsEditorController VPConfigurationView VPConfigurationController Picture
VOSE-Style	VoseObject VoseRelation VoseSpecDiagram
VOSE-Work Record	WRInspector WRInspectorModel ProjectAnalyser ProjectAnalyserModel AnnotationEditor AnnotationEditorController
VOSE-Process Modelling	Guide Guidance ProcessModel ProcessModeller
VOSE-Templates	Template TemplateBrowser TemplateBrowserController TemplateBrowserModel TemplatesDB
VOSE-Tools	SpecificationController SpecificationView ObjectStructuring FunctionalDecomposition ActionTables TestTool
VOSE-ViewPoints	ViewPoint ViewPointBowser ViewPointBrowserController ViewPointBrowserModel ViewPointAnnotator ConsistencyChecker
VOSE-Projects	ProjectBrowser ProjectBrowserController ProjectBrowserModel
VOSE-Main	VoseView VoseModel

Table A1: *Smalltalk classes that implement The Viewer.*

Further installation details can then be obtained by selecting and executing the following expression:

```
Setup open
```

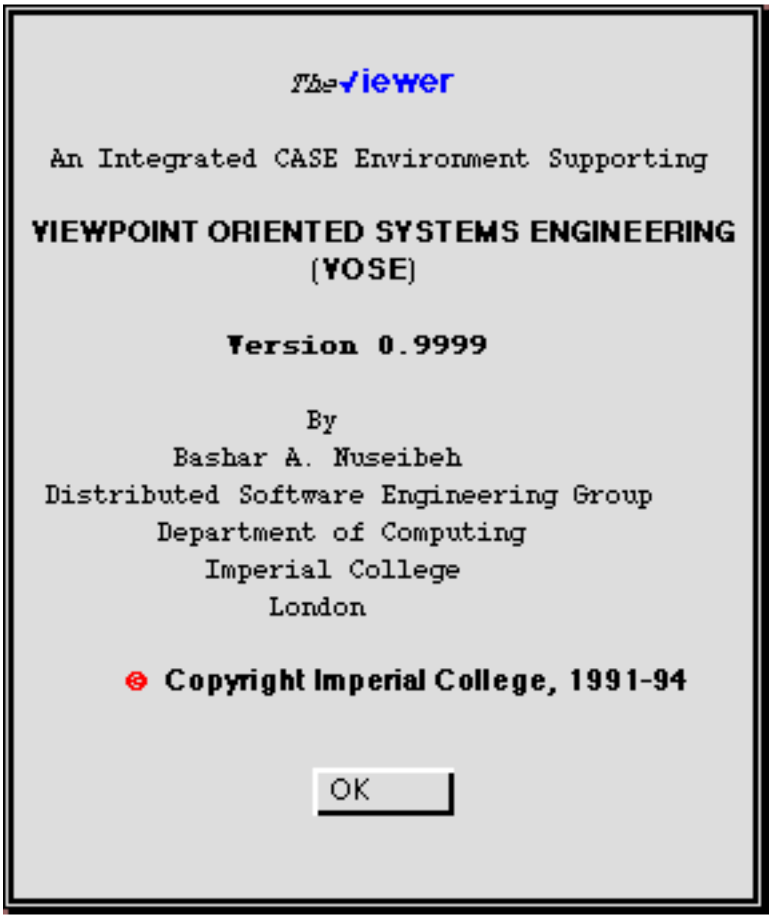
This opens a window with details about how to initialise a VOSE database and how to set *The Viewer* to black and white, or colour operation and so on.

Every class category, class and method in *The Viewer* is extensively commented. Browsing through the various categories provides a quick way of identifying the major functions of the various classes and their constituent “method protocols”. Switching on the “comments mode” from a Class Hierarchy Browser displays the descriptions (comments) associated with the various selected classes of *The Viewer*.

A.4. Historical Evolution

An early (black and white) version of *The Viewer* was implemented in Objectworks/Smalltalk R2.3, which was then upgraded to run under R4.0. An upgrade to R4.1 was not deemed useful, since it offered no additional functionality and would have involved major re-design and implementation (especially of those graphics classes that were dependent on R4.0’s original class hierarchy).

The separation of method design and method use activities in *The Viewer* has been stable for some time, although the functionality of the TemplateBrowser, ViewPointConfigurationBrowser and other tools has evolved with the framework. Major changes in *The Viewer* are indicated by an additional “9” in the version number which is currently 0.9999! A version number less than one, indicates the prototype status of *The Viewer*. Figure A1 is a screen-dump of the “about box” of *The Viewer*, which may be invoked from most windows by clicking on “The Viewer” button (located in the top right-hand corner of these windows).



The Viewer

An Integrated CASE Environment Supporting

VIEWPOINT ORIENTED SYSTEMS ENGINEERING
(VOSE)

Version 0.9999

By
Bashar A. Nuseibeh
Distributed Software Engineering Group
Department of Computing
Imperial College
London

© Copyright Imperial College, 1991-94

Figure A1: *“About The Viewer” pop-up window.*

Modifications and extensions to *The Viewer* by various students have been performed and documented independently of the version described in this thesis.

Appendix B Abbreviations

This appendix contains a key of the most common acronyms and abbreviations used in this thesis. These include both technical terms and popular product names.

AI	=	Artificial Intelligence
APSE	=	Ada Programming Support Environment
CAIS	=	Common APSE Interface Set
CAME	=	Computer-Aided Method Engineering
CASE	=	Computer-Aided Software Engineering
CDA	=	Constructive Design Approach
CDIF	=	CASE Data-Interchange Format
CMM	=	Capability Maturity Model
CORE	=	Controlled Requirements Expression
CSCW	=	Computer-Supported Cooperative Work
CSP	=	Communicating Sequential Processes
DAI	=	Distributed Artificial Intelligence
DBMS	=	Database Management System
DFD	=	Data Flow Diagram
DRL	=	Decision Representation Language
ER	=	Entity-Relationship
ESF	=	Eureka Software Factory
ESPRIT	=	European Strategic Programme for Research and Development in Information Technology
I-CASE	=	Integrated CASE
IPSE	=	Integrated Project Support Environment
JSD	=	Jackson System Design
MLP	=	Mixed Language Programming
OMS	=	Object Management System
OOA	=	Object-oriented Analysis
OOD	=	Object-Oriented Design
OOP	=	Object-Oriented Programming

PCTE	=	Portable Common Tool Environment
PTI	=	Public Tool Interface
RCS	=	Revision Control System
RML	=	Requirements Modelling Language
RPC	=	Remote Procedure Call
SADT	=	Structured Analysis and Design Technique
SCCS	=	Source Code Control System
SCS	=	Structured Common Sense
SD	=	Structured Design
SEI	=	Software Engineering Institute
SLI	=	Specification Level Interoperability
SREM	=	Software Requirements Engineering Methodology
SSADM	=	Structured Systems Analysis and Design Methodology
TBK	=	Tool Builder’s Kit
UIMS	=	User-Interface Management System
UTS	=	Universal Type System
VOA	=	Viewpoint-Oriented Analysis
VOSE	=	ViewPoint-Oriented Software/Systems Engineering
VSF	=	Virtual Software Factory
YACC	=	Yet Another Compiler-Compiler

Appendix C Trademarks

Ada is a registered trademark of the U.S. Government, Ada joint Program Office.

Excelerator and Customizer are trademarks of Intersolv Limited.

IBM is a trademark of International Business Machines Corp.

IPSYS Tools Builder Kit is a trademark of IPSYS Software Plc.

Macintosh is a trademark of Apple Computers Inc.

MetaEdit is a trademark of MetaCASE Consulting Oy.

ObjectMaker and ObjectMaker Tool Development Kit are trademarks of Mark V Systems Limited.

Objectworks/Smalltalk is a trademark of ParcPlace Systems.

Rational and Rational Rose are trademarks of Rational.

SADT is a trademark of SofTech, Inc.

SoftBench is a trademark of Hewlett-Packard Inc.

Software Bus is a trademark of the Eureka Software Factory.

Software through Pictures is a trademark of Interactive Development Environments Inc.

Sun is a trademark of Sun Microsystems Inc.

UNIX is a trademark of AT & T Bell Laboratories.

VSF is a trademark of VSF Limited.

Windows is a trademark of Microsoft Corp.

X Window System is a trademark of the Massachusetts Institute of Technology.