National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

A FORMAL METHOD FOR PARTIALLY TOLERATING INCOMPLETENESS IN SPECIFICATIONS: A PROPOSAL

Dimitrios Kourkopoulos

A Thesis
in
The Department of Computer Science

Presented in partial fulfillment of the requirements
for the degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

April 1993

Copyright © 1993 by Dimitrios Kourkopoulos

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-84625-9

Canada

Abstract

A Formal Method for Partially Tolerating Incompleteness in Specifications: A Proposal

Dimitrios Kourkopoulos

Completeness is usually listed as a desirable attribute of specifications; incompleteness, as a reason for the failure of software to satisfy its intended requirements. Unfortunately, these terms are rarely given anything but intuitive definitions, making it unclear how to achieve the former or, alternatively, avoid the latter. This thesis begins by examining various notions of (in)completeness in specifications, and introduces a pragmatic definition of incompleteness: a classification based on its potential sources. From this, it observes that completeness, though needed to properly reason about, and capture the behaviour of, the system, is undesirable in some cases. To reconcile these conflicting needs, this thesis proposes a novel formal method for (partially) tolerating incompleteness in specifications.

The method focuses on one of the classes. A connection is drawn between this class and a group of related problems involved in reasoning about time and action in artificial intelligence: the qualification, frame, and ramification problems. Both endeavors must contend with incomplete information. Since the techniques employed to deal with these problems usually involve non-monotonic logics, a number of such logics are considered, but most rejected. Shoham's logic of chronological ignorance, however, shows promise. Its shortcomings are addressed, and an extension of it defined. This serves as the formal basis for the specification language *KAT*, which is intended for real-time, concurrent systems. The thesis concludes with a description of the language, a discussion of pragmatic issues, including how it permits fairly easy modification of specifications, and a specification of a telephone system demonstrating its use.

Acknowledgements

I would like to thank Dr V.S. Alagar, my supervisor, for the kind support and gentle inspiration he has given me throughout the development of this thesis

Contents

List of Figures viii

1 Introduction 1
1.1 Tar Pits and Silver Bullets . . . 2
1.2 Requirements and Specifications . . . 3
1.2.1 Why Specify? . . . 6
1.3 Errors in a Specification . . . 8
1.3.1 Incomplete Specifications . . . 9
1.4 Problem Statement . . . 11
1.4.1 Hinting at the Proposed Solution . . . 12
1.5 The Rest of the Thesis . . . 13

2 Logic Background 15
2.1 Logics . . . 15
2.2 Classical Logics . . . 18
2.2.1 Propositional Logic . . . 18
2.2.2 First-Order Predicate Logic . . . 22
2.3 Modal Logics . . . 27
2.3.1 Modal Propositional Logic . . . 28
2.3.2 Applications . . . 30

3 (In)Completeness in Specifications 32
3.1 Building a Specification . . . 33
3.1.1 Software Specification is Theory Formation . . . 35

	3.2	Some Formal Considerations	39
		3.2.1 Complete Theories	40
		3.2.2 Complete Specifications With Respect to Implementations	42
		3.2.3 Comparison of the Notions of Completeness in Specifications	43
		3.2.4 Weaker Notions of Completeness	44
	3.3	Types of Incompleteness	45
		3.3.1 The Classification	47
		3.3.2 Discussion	56
4	**Incomplete Information and Non-Monotonicity**		**58**
	4.1	Commonsense Reasoning and Incomplete Information	59
		4.1.1 Non-Monotonic Reasoning	64
		4.1.2 Reasoning About Change	66
	4.2	Non-Monotonic Approaches	70
		4.2.1 Default Logic	70
		4.2.2 Autoepistemic Logic	73
		4.2.3 Circumscription	75
		4.2.4 Objections to Non-Monotonic Formalisms	79
		4.2.5 The Yale Shooting Problem	82
5	**Semantical Non-Monotonicity**		**88**
	5.1	Semantical Non-Monotonicity	89
	5.2	Shoham's Logics	91
		5.2.1 The Logic of Temporal Knowledge	92
		5.2.2 Chronological Ignorance	96
		5.2.3 Causal Theories	98
	5.3	Shortcomings of Shoham's Logics	102
6	**The Specification Language KAI**		**107**
	6.1	kFOTK: a Logic of Temporal Knowledge	108
		6.1.1 Syntax	108
		6.1.2 Semantics	110

6.1.3 Chronological Ignorance . . . 111
6.2 KAT Knowledge of Action and Time . . . 115
6.2.1 Language Issues . . . 115
6.2.2 P-Causal Theories . . . 121
6.2.3 KAT Specifications . . . 124
6.2.4 Pragmatic Issues . . . 125
6.2.5 Example: Plain Old Telephone System . . . 131

7 Conclusions and Future Considerations **143**
7.1 Summary of Thesis . . . 143
7.2 Features of KAT . . . 145
7.3 Future Considerations . . . 147

Bibliography **149**

List of Figures

3.1 Theories in the software development process . . . 36

6.1 Relationships Between Logics 113

6.2 POTS Specification: Language Declarations . . 110

6.3 POTS Specification: Axioms 111

6.4 POTS Specification: Axioms, continued 112

Chapter 1

Introduction

> If we find those who are engaged in metaphysical pursuits, unable to come to an understanding as to the method which they ought to follow; if we find them, after the most elaborate preparations, invariably brought to a stand before the goal is reached, and compelled to retrace their steps and strike into fresh paths, we may then feel quite sure that they are far from having attained to the certainty of scientific progress and may rather be said to be merely groping about in the dark. In these circumstances we shall render an important service to reason if we succeed in simply indicating the path along which it must travel, in order to arrive at any result — even if it should be found necessary to abandon many of those aims which, without reflection, have been proposed for its attainment.
>
> **Immanuel Kant**, *The Critique of Pure Reason*,
> in Preface to Second Edition, 1787, *transl.* J.M.D. Meiklejohn

Software development, though more mundane a pursuit than metaphysics, can also be said to be "far from having attained to the certainty of scientific progress." Whether it can ever experience a scientific revolution in the Kuhnian [41] sense remains uncertain, but it is clear from the haphazard and *ad hoc* nature of the attempts to find suitable paradigms that it has yet to achieve such a status. Hoping for a strong scientific foundation, however, may be a little unrealistic; most developers would gladly accept the systematic rigour of an engineering discipline. Unfortunately, only the most optimistic ones would claim that that has already been achieved. Heeding Kant's advice then, we must continue to look for new paths, new perspectives, new goals ... .

1.1 Tar Pits and Silver Bullets

Frederick Brooks, drawing on his experience of managing the development of OS 360 in the early 1960's, wrote one of the first critical examinations of the (then fledgling) field of software engineering, *The Mythical Man-Month* [7]. In this seminal text he likened the development of large system software to a great beast thrashing in a tar pit. The "beast" was uncontrollable, its complexity surprisingly overwhelming and unyielding. Early software developers and project managers, who had little experience in managing such complexity, and even fewer tools, found themselves quite unprepared to handle it. Getting a hold on any particular component of the software was not a difficult problem, but pulling the whole out of the tar was nearly impossible. As a result, products were delivered late or not at all, cost many times more than initially estimated, and did not perform exactly as expected. Moreover, the developers and managers found it disconcerting that the application of common engineering principles or management techniques proved to be inadequate for these conceptual structures. For example, they could no longer simply add more people to a late project to increase production. Surmounting all these difficulties was often such a futile task that Brooks warned us to be prepared to throw away the first version, perhaps then to use the experience gained in making the first to build the second.

Two decades later, Brooks [8] tells us that not only has there been little change, but that there is not even a "silver bullet" in sight: a method, software tool, development in technology or management technique that would dramatically result in improved productivity. Any gains, he claims, were made against the *accidental* difficulties of software technology, those difficulties that are concomitant to the production of software. While high-level languages, object oriented programming, CASE tools and environments, artificial intelligence, expert systems, and powerful workstations have all contributed to easing the task of the developer, they have not significantly addressed the *essential* difficulties, those difficulties that are inherent in the nature of software:

complexity The complexity of software is probably greater than that of any other human construct. One complicating factor is that it increases nonlinearly with

the size of the product, mainly because there does not exist a well-established notion of repeatable or reusable components.

changeability Software is constantly subject to change, not simply because of enhancements and other changes to requirements, but also because it is so easy to change.

invisibility Software does not have a physical nature: it is not visualizable. It is an abstract entity that does not yet have a universally accepted way of representing it.

conformity Software lacks a set of unifying principles, forcing the developer to conform to a variety of arbitrary principles, interfaces, and standards.

All is not entirely bleak, however, as Brooks does point to a number of promising attacks on the essential difficulties. One of them is of particular interest, not because of the approach advocated, but because of the reason given for adopting it: it deals with the problem of gathering and specifying the requirements of a software product.

1.2 Requirements and Specifications

Before continuing further, some definitions and distinctions must be made. The term *product* will sometimes be used to refer to the software entity being developed; it does not have any commercial connotations. At other times, when the context is clear, the simple term *system* will be used. The generic designations *client* and *developer* will usually be used to refer to the individual(s) who commission and develop the product, respectively. This does not imply that they are separate groups, for the developer may also be the client, nor that the client is the eventual *user* of the product. When necessary, further distinctions, such as analyst, programmer, user, and so on, will be made. The *requirements* of a product are the things that the client would like the product to do. If they are compiled into a document, the *requirements document*, it is assumed that it is an informal one, written in a natural language. The *specifications*, on the other hand, are the formal or semi-formal translations of the requirements.

They are contained in the *specification document*, or simply the *specification* when it is clear. Formal means that it is written in a specification language that has a firm mathematical basis. Ideally, the language should have a well defined syntax, semantics and inference mechanism. A semi-formal language typically relaxes the definition of the latter two components. Finally, since the tasks of gathering and specifying requirements are often intertwined, the term *specification phase*, unless otherwise noted, will refer to the phase of development in which both occur.

Getting back to Brooks [8], he identifies the specification phase as the most critical point in the development of software. He describes it as follows (pg. 17).

> The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

The claims are not extravagant; similar sentiments are echoed regularly in texts and articles on software engineering. They are usually buttressed with the well known fact that errors committed during the specification phase are usually the last to be discovered, and, consequently, the hardest and most costly to repair (see for example, [6], [11], and [17]). In fact, in the traditional life-cycle or "waterfall" model of software development - in which development proceeds through a sequence of several phases the cost rises exponentially through each phase. A specification error discovered during the system testing phase may cost as much as 100 times more to correct than it would during the specification phase. Therefore, one can safely add an additional claim to the above quote:

> *No other part affords the greatest potential for significant improvement in the time, cost and quality of software.*

Hence, software engineers and researchers should concentrate their efforts on this aspect of software development. A greater effort must be made to investigate ways

of making this phase easier to realize and less error-prone. The types of errors that occur during this phase must be identified, and methods and tools for dispatching them suggested. Since the errors eventually end up in the specification document, the problem can alternatively be stated in the following manner: how can we eliminate, or at least minimize, the errors that are made in writing a specification? Let us begin by briefly looking at why the task is such a hard one.

The difficulties stem primarily from having to deal with a fuzzy domain, the requirements. In attempting to make some sense of them, to formalize them, the task is not unlike the whole development process itself, only on a reduced scale. Indeed, the specification and product (source code) attempt to express the same thing, but in different languages. As such, it suffers, albeit not to the same degree, from the same essential difficulties listed above. The biggest difference is a reduction in complexity, since implementation details are abstracted away. However, the conceptual complexity of the product—often called the "what" of the product—remains. It must not be simplified because abstracting it away would abstract away the product's essence. The rest of the difficulties are largely unaffected. In particular, the changeability or volatility of requirements remains a serious problem. Brooks [8] contends that the task is so difficult that it is impossible to arrive at a complete and correct specification of the requirements in the manner advocated by the life-cycle model of software development—*i.e.* done on paper and completed before going on to the rest of the phases. His pessimism stems from the observation that clients are rarely capable of providing a complete and consistent set of (unambiguous) requirements. Instead, he suggests that the requirements should be extracted and refined in an iterative fashion, perhaps throughout the development of the product. This would make it easier for the developer to handle any changes in the requirements, while giving the client the opportunity to flesh them out. As a potential solution, he points to *rapid prototyping*, a model of software development that employs just such an iterative approach. Briefly, a prototype is a quickly produced mock-up of the intended product that the client can try out. The client's feedback is then used to repeatedly refine it. Given the importance of the interactive aspect of this approach, it is especially useful for developing products that have a significant user interface. Conversely, embedded systems

with no user interface are probably not appropriate.

While the suggestion has merit, and early observational and experimental data appear to lend support, leading Boehm and Papaccio [6] to make a similar recommendation, it should be noted that prototyping does not actually make the specification phase or document unnecessary. On the contrary, the prototype is often used simply as an analytical tool to discover all the requirements of the product; once that is done, it is usually discarded, harking back to Brooks' old throw-away rule. According to Connell and Shafer [12], documents must still be produced, even when employing an *evolutionary* prototyping approach, in which the prototype is not discarded but instead iteratively transformed into the final product. An initial, and admittedly incomplete, specification must be produced before the initial prototype is made. Then, with each suggested refinement to the prototype, the specification must also be updated to reflect the changes. At the end of the process, one has not just the product, but a final specification of it as well. Thus, for them, it is not really a mutually exclusive choice between specifying or prototyping; rather, the two are considered complementary. A similar argument can be made for other iterative approaches, such as *incremental development* which incidentally is also recommended by [8] and [6]

1.2.1 Why Specify?

A formal specification method, however, is not a panacea. It will not, just by itself, insure that no errors will be made, nor that the final product will be perfect. After all, the specification of incorrect requirements will result in a specification that is wrong. Determining the requirements is clearly a necessary task, but can the same be said about producing a specification? Why not work from the requirements? Why make the effort to formalize them? Or at most, why not adopt a prototyping approach but write only the initial specification? Unfortunately, unless the project is small and the developer has no intention of further enhancements, one cannot avoid writing a specification. And despite the slow acceptance of formal specification methods by industry, the advantages of using them far outweigh the disadvantages.

The main advantage is precision. A specification states in a clear and concise manner what the product does, without the unnecessary implementation details that

clutter the source code. It is not just supposed to serve as the basis for building the product, but as a communication tool as well. It is expected that different people using the specification should arrive at the same interpretation. This is important because the people who write it are usually not the ones who design, code and test the product. People come and go in organizations; the original development team may no longer be around when further enhancements to the product are requested. Furthermore, given its formal nature, it should be possible theoretically to make the transition to source code in a mechanical manner, thus eliminating the substantial number of errors that are made in going from one phase to the next. Although still a pipe dream, this is the underlying notion of a variety of methods that fall under the labels of *automatic programming*, *very-high level programming* and *transformational techniques* (see [75] and [82] for examples of these).

Requirements, on the other hand, are ambiguous. Even ones that appear very clear may have many interpretations. Gause and Weinberg demonstrate this quite forcefully in [20] by showing how many interpretations can be derived from simple English sentences. Basili and Perricone [5] demonstrate this by presenting empirical data from a large development project that showed that the majority of errors (48%) were due to incorrect or misinterpreted specifications. Equally significant is their observation that more errors appeared in modules that were re-used than in new ones. The developers were hoping to save some time and work by re-using some modules from a previous project, but their specifications were so poorly expressed that the modules were used inappropriately. Therefore, a requirements document is a poor communication tool and using it as the basis for development is, quite frankly, reckless. Why would anyone spend a substantial amount of money on development and then allow the use of something that adds considerable uncertainty to the project?

Another significant benefit of a specification is the role it plays in determining whether or not the final product is the correct one. Delivering the correct product is arguably the most important duty of the developer—correct in the sense that it exhibits all the behaviours that the client had requested. By definition, therefore, correctness is a relative term: correctness can only be established with respect to some frame of reference. If that frame of reference is a specification, then correctness

can be firmly established; if it is a requirements document, there is no such certainty.

One commonly expressed disadvantage of formal methods is that they are difficult to write because they involve complex mathematics. Most languages, however, involve either set theory or first-order logic. These can hardly be deemed complex. Besides, all computer science programs include a course that covers them. Another disadvantage cited is that they may be incomprehensible to the clients. This is a concern because the specification is, in some respects, considered a contract between the client and developer. Here again, the notation can be explained to anyone with at least high-school mathematics training. However, it would be helpful to include natural language explanations with the formal section. Hall in [23] dispels these and other myths concerning formal methods.

Therefore, regardless of the particular methodology used, the task of specifying the requirements should be considered just as necessary as the task of gathering them. The latter task involves, among others things, good interpersonal skills and the ability to pick out ambiguities in the client's statements. This thesis will not cover it, except perhaps in an indirect fashion after all, the specification is based on the requirements. The reader is directed towards [20] which provides very good (practical) advice.

1.3 Errors in a Specification

The salient point made above is that given the potentially crippling effect of errors in a specification, greater effort must be made to eliminate or, at least, minimize them. Now, in order to properly tackle this problem, one must first be aware of the sources of error. Once these are known and classified, it might then become possible to determine which ones can be eliminated, which ones can be minimized, and which ones are inevitable and unavoidable.

Knuth [38] provides an interesting classification of errors made during the development of TEX. Of the 15 classes listed, about 8 can probably be traced back to an error in the specification. Of these, 6 represent enhancements, either to add more features or improve some aspect of its execution, such as efficiency; the other two were due

to poor exception handling (robustness) and unexpected interactions between various parts of the program. Some software engineers may object to the term "error" being applied to enhancements, but Knuth resists calling them simply "changes" because they were needed to correct deficient designs. The remaining 7 were due to the usual errors one encounters in programming, such as incorrect algorithms, typos, mental errors, not referring to the specification, misusing the programming language, *etc.* Focusing on the specification errors, unexpected interactions can be largely avoided if a formal specification language is used, since the inference mechanism permits one to discover most of the consequences of the specification; the rest, however, fall under the general category of incompleteness. Ideally, the specification should have included them, but what this means is not exactly clear. Perhaps more interesting than the classification is an anecdote he recounts on the history of TeX. He had given the first specification — I use this term because it is the word Knuth uses, but it is not clear whether it was a formal one — to a couple of graduate students to implement over the summer of 1977. When he returned in the fall, he found that only 15% of the language had been implemented. Taking over the task himself, he soon realized what a difficult task he had set out for them. The document that he had initially thought was complete, was not; he found many loose ends, requiring him to make major decisions constantly. As both the client and developer of the product, he was able to recognize the document's incompleteness during implementation. It convinced him that the specifier of a new product should participate fully in its development. But is this a realistic expectation? Incompleteness is often attributed as a source of errors in a specification, so having some way of dealing with it should prove very useful. Is there any way to determine when a specification is incomplete? What does it mean for a specification to be incomplete?

1.3.1 Incomplete Specifications

Throughout this introduction I have used the term *complete* and its opposite, *incomplete*, in relation to specifications, without having defined them. This was deliberate; I was merely following convention. It seems that in software engineering they are often used in such a casual manner, as if they have well understood and universally

accepted meanings. Unfortunately, a quick perusal of the relevant literature will reveal that these terms are vague concepts which are rarely defined, forcing us to rely on our own intuitive interpretations. The dangers of this are obvious, especially given the emotionally-charged nature of these terms. Who would possibly accept an incomplete set of something, regardless of what it is? The case is not different for specifications. Thus, it is not surprising to find that *completeness* is treated as a "motherhood" issue in most software engineering texts: we are told that — aside from being concise, unambiguous, correct, consistent, *etc.* — a specification *should* be complete ([17] is one example). Since this is a desirable property, incompleteness is either tacitly discouraged or accompanied by vague admonishments about 'not satisfying customer needs.' A further complication of this inadequate treatment is that, when completeness is defined in these texts, there is no consensus on what it is. Different texts define it in different ways. For example, a complete specification is often defined by some as one that contains all the facts (about the system); others define it as one that has no "loose ends," meaning that it also indicates all the facts that do not apply; still others that all the functions and operations specified within it are so fully defined that they return a value for any possible set of inputs. At the very least, the variation in meaning results in a bit of miscommunication among professionals in the field; at worst, it becomes a case of something that has too many meanings ending up having none. Furthermore, despite its supposed importance, few guidelines are given in these texts on how to achieve it. Thus, merely saying a specification is incomplete does not elicit any insight into the types of errors involved.

The question, therefore, still remains: what does it mean for a specification to be (in)complete? Or, in more practical terms, what are the sources of incompleteness? It seems to encompass a wide range of errors. The usual assumption is that the fault lies with the software developer who has forgotten to include some requirement in the specification, or has only partially specified the functions and operations. However, what if the user is not aware of all the requirements? And, what about requirements that change over time? It can be argued that these latter cases are not really the responsibility of the software developer. After all, one can only work with what one is given. Although this is a compelling argument, it is a facile one, given that software

requirements usually do change over the lifetime of the product. Should we not, therefore, define a specification language/methodology, or construct a specification, with that in mind? Perhaps, it would be beneficial to consider the definition of *complete* in this extended sense, as Knuth does with his classification of errors, even though it would mean *completeness* becomes a difficult property to achieve.

There are other concerns, as well. Any useful formal language will have inherent limitations with regards to decidability and completeness. A specification written in such a language will naturally be affected by whatever limitations the language possesses. This implies that we may never be able to eliminate all sources of incompleteness. Moreover, it has been suggested (*e.g.* [73]) that perhaps a certain amount of incompleteness is desirable since it permits some flexibility in developing the product — both for avoiding implementation bias and anticipating upcoming changes. Consequently, the goal of completeness in specifications may not even be a reasonable one. Thus, what is needed is not necessarily a method or tool that can establish absolute completeness, but rather one that can tolerate a certain amount of incompleteness.

1.4 Problem Statement

The general or initial goal of this thesis is to examine the various notions of incompleteness in specifications, to provide a useful and concrete definition of it, and then to propose and define a method, preferably formal, for handling or tolerating it. For now, "tolerating" means that the method allows us to reason about the system, even when it is incompletely specified. Once the meaning of (in)completeness has been fixed, however, both this definition and the goal will be refined accordingly. Since specifications undergo constant modification, the method should also be capable of readily accepting change.

The only other requirement concerns the systems to be specified. To make the study as general as possible, at least initially, very few restrictions are placed on the systems. However, in general, the emphasis is on reactive systems: systems that exhibit an on-going behaviour, often while interacting with their environments. In

particular, the focus is on (embedded) real time systems and concurrent systems whose actions have durations (*i.e.* not constrained by atomicity).

Reactive systems pose special problems [60]. They cannot be described simply in terms of functional relationships, from initial to final states. Approaches that just describe the state-changing effects of actions or operations, such as *VDM* [35], are therefore inadequate because they cannot capture a notion of behaviour. And of those that can, most rely on atomic actions — for example, the *situation calculus* [51] in the field of artificial intelligence, *CCS* [51] and *CSP* [33] in concurrency theory, or the majority of temporal logics [60] used for specification. While atomicity simplifies the problem of dealing with concurrency, it effectively reduces it to a non-deterministic, sequential interleaving of concurrent actions. On the other hand, if the specification formalism allowed for the expression of time intervals or actions with durations, then a very natural description of concurrency could be achieved because it would permit the specification of overlapping actions. Unfortunately, in such an approach there also exists the potential for interference between conflicting actions — one action doing one thing, while another does the exact opposite. Concurrency, as we shall see, complicates the description of systems, and restricts the range of formalisms we can use.

1.4.1 Hinting at the Proposed Solution

This thesis proposes a formal method for (partially) tolerating incompleteness in specifications. The "partially" qualification means that only certain types of incompleteness are tolerated. To my knowledge, both the specification language and the approach taken to reach this solution are novel. The basic tack taken is as follows.

After examining the various notions of (in)completeness in specifications, I introduce a pragmatic definition of incompleteness: it is a classification based on the potential sources of incompleteness in specifications. From this, I select one class that presents an interesting challenge: although it is a common source of incompleteness, eliminating it is neither easy nor desirable. A connection is then drawn between this class and a group of related problems involved in reasoning about time and action in artificial intelligence (AI): the qualification, frame, and ramification problem. Since

the techniques in AI that deal with these problems usually involve non-monotonic logics, the proposed solution, a formal specification language called KAT, is also based on one. Interestingly enough, these logics were initially introduced to model certain aspects of common-sense reasoning: specifically, the ability to make decisions and plans even when lacking relevant information.

1.5 The Rest of the Thesis

This thesis is directed towards the software engineer. Since it involves areas of study that might not be familiar to some, an attempt has been made to make it as self-contained as possible. As such, there may be parts, such as the logical preliminaries in Chapter 2, that might be very familiar to the reader, and hence may be omitted. Furthermore, this thesis contains a proposal: it offers for the reader's consideration a new idea. How it will be received is as yet unknown, but enough information is given to motivate and justify the various choices made along the way and to bolster the claims of the solution. The detailed description also gives the reader a better opportunity to judge and perhaps criticize the arguments made. As well, given the tack taken and the similarities between certain concepts in different fields, some repetition and paraphrasing of ideas and arguments was required; in general, however, the views are from slightly different perspectives. Finally, the thesis, almost by necessity, places a strong emphasis on the semantics of the language. This is clearly a departure from most specification language definitions, which are often just user's manuals, focusing almost entirely on the syntax and examples of use. However, given the importance of being precise and unambiguous in a specification, this thesis should perhaps be regarded as a model for such definitions.

The rest of the thesis is arranged as follows:

Since many of the ideas presented here depend on logic (of a non-standard nature), the next chapter, Chapter 2, provides some logical preliminaries. It briefly covers standard monotonic logics, including propositional, first-order predicate and modal. It also sets the notation for the rest of the thesis.

Chapter 3 first looks at the task of specification in more detail. Next, it formally examines the issue of (in)completeness from a number of perspectives, clearing up some of the ambiguities. It then presents a classification of incompleteness based on its potential sources. The general conclusion is that attaining completeness is unreasonable, the best tack being to invent a method/language that tolerates a certain amount of incompleteness. The focus of this method is one of the classes: partial specification.

Chapter 4, referring to the classification, proposes a general solution. It notes the connection between the chosen class and the problem of dealing with incomplete information in AI. Since the solution will require a non-monotonic formalism, various non-monotonic logics are examined. Most are found wanting in one way or another, however, Shoham's non-monotonic approach *semantical non-monotonicity* shows the most promise.

Chapter 5 describes Shoham's ideas and logics. In particular, the chapter begins with a description of semantical non-monotonicity and then describes the logics of temporal knowledge (TK) and chronological ignorance (CI). It also defines the notion of causal theories, which will form the basis of specifications under the new specification language. The chapter ends with a critique of the logics, pointing out what must be changed to make them more appropriate for the formal basis of the language.

Chapter 6 has two parts. The first extends Shoham's logic of TK to a first order, many-sorted case, $kFOTK$. It then redefines CI with respect to $kFOTK$. The second, presents the specification language itself, KAT. The syntax and form of the language is given, together with a new definition of causal theories, called p causal theories. A number of pragmatic issues are also discussed, including how it can be used to build specifications, how it deals with their modifications and how it tolerate their incompleteness. Finally, the various features of the language are demonstrated through the specification of the ubiquitous Plain Old Telephone System (POTS).

Chapter 7 concludes the thesis, briefly summarizing the key issues and suggesting some further areas of study.

Chapter 2

Logic Background

Much of the discussion in later chapters presupposes some familiarity with standard (monotonic) logics, including propositional, first-order predicate, and modal. Given the widespread use of these logics in computer science, especially the first two, it is probably safe to assume that the reader is already familiar with their fundamentals. Nevertheless, it might be wise to briefly cover them, if only to establish a consistent notation and introduce some relevant terms. For more detailed presentations on logics, consult most texts on the subject--see, for example, [79]. Although introductory in nature, I recommend [17] and [64] because the logics are discussed in the context of formal methods in artificial intelligence.

The next section informally introduces some key concepts. The final two sections briefly present the formal details of classical and modal logics, respectively.

2.1 Logics

Modern classical logics are formal languages. They were originally intended to formally capture the mental activity involved in logical (deductive) reasoning, particularly as it applied to mathematics--see Barwise's introductory chapter in [79] for some examples of this. Although the domains of interest in this thesis are quite different from mathematics, the reasons for using a logic are fundamentally the same: (1) with logic, one can very precisely express statements or assertions of the domains;

and (2) one can determine the consequences of a set of statements. Precision is a key characteristic of logics: it distinguishes them from natural (and other informal) languages. Both the interpretation of the symbols in the language and the definition of the inference rules are given with great precision. This permits us to not only formally derive proofs of statements within the language, but to also rigorously prove properties about the language itself. Thus, one can determine, for example, whether a logic is *sound* or *complete*.

There are a number of ways to define a logic, depending on what one wishes to do with the language [61, 79]. For instance, in order to determine the properties (or *meta-theorems*) of a language, it would be best to use as small a set of logical symbols and inference rules as possible, such as a *Hilbert system*. On the other hand, if the main concern is expressing facts and proving theorems within the language, then a language with more "syntactic sugar" and inference rules should be used, such as *Gentzen* or *tableau systems*. Furthermore, even within a particular approach there are minor variations in the way the logic is presented; see, for example, the variations in the way first-order predicate calculus is defined in [17], [61] and [79]. However, regardless of the way a logic is defined, a proper definition has three components:

syntax: With any language, one must be able to distinguish proper sentences from improper ones. The syntactic rules provide just such a mechanism. They indicate which symbols can be used, and which sequences of symbols are well formed. By itself, the syntactic component is not very interesting, but it plays a significant role in how the other components are defined and used.

semantics: The semantic component is perhaps the most important part of a logic; it gives meaning to the well-formed sentences in the language. This is accomplished through a set of semantic rules which systematically attach a meaning to each syntactic entity in the language. For the sake of precision, the meaning is usually given in terms of a mathematical structure, sometimes called an *interpretation* or a *frame*. Although it can take many forms, this structure is often simply a collection of objects, together with a set of relationships over them.

Logics are concerned with the truth of statements, and truth is related to the semantics. Two types of truths can be distinguished: a weak notion of truth, where a statement is true in some particular interpretation; and a strong notion, where it is true in all possible interpretations. Logicians consider statements of the latter type, called *valid* ones more important than those of the former type because they are universally applicable. In fact, the *validity problem* for a logic, which is the task of determining whether or not an arbitrary statement is a valid one, is one of the central issues in logic.

Once the semantics are well established and understood, one can virtually ignore them, relying solely on the syntax to write meaningful expressions.

inference system: This component is sometimes ignored in specification languages, but if one is concerned about the consequences of what one has written, then an inference mechanism is a necessity. In general, an inference system is composed of a set of inference rules and a set of axioms. The rules are "logical" ones, in the sense that they are derived from basic human intuitions regarding what constitutes a proper deductive inference. For example, if given the pair of statements, `if phone is picked up, then phone is off-hook` and `phone is picked up`, it seems quite natural to deduce that `phone is off-hook`. One constraint, however, is that the rules must preserve validity: any statement derived from a set of valid statements must also be valid. Otherwise, the inferences could not be trusted. This concern gives rise to another, a fundamental question in the study of logics: is it possible to define a suitable collection of axioms and rules that can be used to generate all the valid statements of a logic?

It should be emphasized, however, that these rules are entirely formal in nature: they relate syntactical entities to one another, not semantical ones. In order to use them, one only has to match the forms or patterns of the statements. In the example above, the rule, called *modus ponens*, can be reduced to "*from* `if p then q` *and* `p`, *infer* `q`." As long as two sentences matching the first two can be found, the third can always be inferred, regardless of what `p` and `q` represent. Chapter 1, however, shows that the range of human reasoning is much wider

than this, and most cannot be captured by such simple syntactic rules. It is also possible to find the consequences through the semantics, but it is much easier through an inference mechanism.

Another way of looking at inferences is in the context of proofs. An inferred statement is a *provable* one. The sequence of statements and rules used to prove a statement is considered its *proof*. Thus the inference system is sometimes called the *proof theory* of the logic. Although it is not always possible, the inference system may be used to decide whether or not an arbitrary statement follows from some set of statements.

Since the semantics are used to determine which statements are valid, and the inference system is used to derive valid statements, there is clearly a very important connection between these components. Ideally, anything that is provable within the proof theory of a logic should be valid according to its semantics; and inversely, anything that is valid should be provable. The former property is called *soundness*, the latter *completeness*. Any logic that permitted one to draw conclusions that were not valid would not be very dependable, and consequently not useful. Hence, a logic must be sound. But a logic does not necessarily have to be complete to be useful.

And now for some formal details and definitions of logics. The descriptions are based on the treatments in [17, 61, 79, 31].

2.2 Classical Logics

2.2.1 Propositional Logic

Propositional logic (*PL*) is concerned with formulating arguments involving propositions. Propositions are declarative sentences that can either be true or false, such as `phone is off-hook`.

Syntax

The alphabet of classical propositional logic (in a Hilbert system style) consists of the following symbols:

- a non empty, countable set P of *proposition constants*: $p, q, r, \ldots$;
- the *logical connectives*: $\rightarrow$ (*implication*) and $\neg$ (*negation*);
- a pair of punctuation marks: '(' and ')'.

The connectives and punctuation marks are called *logical symbols*; the proposition constants are determined by the domain of interest or application and are considered *non-logical symbols*.

Definition 1 The set of *well-formed formulae* (*wffs*), or simply *formulae*, over the alphabet is the smallest set satisfying the following rules.

1. every proposition constant is a *wff*.
2. if ϕ and ψ are *wffs*, then so are $(\neg\phi)$ and $(\phi \rightarrow \psi)$. $\diamond$

The set of all possible formulae is called the *language* of PL, and is denoted by $\mathcal{L}_{PL}$. We say "if p, then q" for $(p \rightarrow q)$, and "not p" for $(\neg p)$.

The syntax seems fairly limited, but it is sufficient to express any propositional statement. The logic, however, can be made easier to use by extending the syntax with a few common connectives:

- $\vee$ (*disjunction*): $(p \vee q) \stackrel{\text{def}}{=} (\neg p \rightarrow q)$
- $\wedge$ (*conjunction*): $(p \wedge q) \stackrel{\text{def}}{=} \neg(\neg p \vee \neg q)$
- $\equiv$ (*equivalence*): $(p \equiv q) \stackrel{\text{def}}{=} ((p \rightarrow q) \wedge (q \rightarrow p))$

We say "p or q" for $(p \vee q)$, "p and q " for $(p \wedge q)$, and "p iff q" for $(p \equiv q)$, where *iff* stands for "if and only if." When it will not cause any confusion, the use of the punctuation marks is relaxed, and the following precedence of operators will be observed: $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\equiv$.

Semantics

The semantics for PL is very simple, since all that is required is to assign a truth value to all the propositions. Thus, only two objects are required.

Definition 2 An *interpretation* for the propositional language $\mathcal{L}_{PL}$ is a function m $P \longrightarrow \{\mathbf{t}, \mathbf{f}\}$.

Obviously, $\mathbf{t}$ and $\mathbf{f}$ represent the truth values **true** and **false**, respectively.

Next, a meaning is attached to each syntactic entity in the language based on the interpretation.

Definition 3 Let $A \in \mathcal{L}_{PL}$, and m be an interpretation for $\mathcal{L}_{PL}$. The value of A under m, expressed as $V^m(A)$, is defined as follows:

1. $V^m(p) = m(p)$, where p is any proposition constant.
2. $V^m(\neg A) = \mathbf{f}$ if $V^m(A) = \mathbf{t}$ and
   $\mathbf{t}$ if $V^m(A) = \mathbf{f}$;
3. $V^m(A \rightarrow B) = \mathbf{f}$ if $V^m(A) = \mathbf{t}$ and $V^m(B) = \mathbf{f}$
   $\mathbf{t}$ otherwise.

The meanings for the other connectives can likewise be defined.

If $V^m(A) = \mathbf{t}$, we say that A is *true in m*, that A is *satisfiable* and that m is a *model* of A. Furthermore, A is a *valid* formula, commonly called a *tautology* if $V^m(A) = \mathbf{t}$ for all interpretations m. The *validity problem* for propositional logic is decidable. This means that there exists an algorithm that can determine whether or not an arbitrary propositional formula is valid.

Theories and Entailment

A collection of formulae from some language, often intended to describe some domain world or application, is a *theory*. Obviously, *propositional theories* are collections of propositional formulae. These formulae are sometimes called the *axioms* or *premises*

of the theory. An interpretation m is a *model of the theory* T if all the premises of T are true in m. Theories may have none, one or many models.

A theory T *entails* a formula A, or, alternatively, a formula A is a *semantical consequence* of a theory T, written $T \models A$, iff A is true in every model of T. Clearly, if T is empty, resulting in $\models A$, then A is a tautology. The symbol $\models$, called the *entailment relation*, relates a theory to a formula. It enjoys a number of properties, but the most relevant to this thesis is *monotonicity*:

Definition 4 If $T \subseteq T'$, then $\{A \mid T \models A\} \subseteq \{A \mid T' \models A\}$. ◇

In other words, if any formula A logically follows from a theory T, then it still follows from a theory T' which has been extended from T with the addition of new axioms. New axioms cannot invalidate old consequences of a theory.

Inference System

An inference (or *deduction* or *proof*) system for propositional logic is composed of three elements: the language $\mathcal{L}_{PL}$, a set of logical axioms (tautologies) S, and a set of inference rules R. The rules map one or more formulae from the language into another. The *modus ponens* rule mentioned above, for example, maps $A \rightarrow B$ and A into B. This is usually written as

$$\frac{A \rightarrow B,\ A}{B}.$$

B is considered a *consequence* of $A \rightarrow B$ and B by virtue of the application of the *modus ponens* rule. A rule is *sound* or *truth-preserving* iff the conclusion of the rule is true in m whenever the hypotheses of the rule are true in m.

Let $IS = \langle \mathcal{L}_{PL}, S, R \rangle$ be an inference system and T a theory over the language. A *proof* of a formula A from a theory T is a (finite) sequence of formulae $A_1, \ldots, A_n$ from $\mathcal{L}_{PL}$, such that A_n is A and each A_i, $1 \leq i < n$, is either a formula from $S \cup T$ or a consequence of some preceding formulae in the sequence by virtue of the application of a rule in R. We say A is *provable* from T in IS, written $T \vdash_{IS} A$, iff there exists such a proof. We also say that A is a *syntactical consequence* or a *theorem* of T in IS. If T is empty, then $\vdash_{IS} A$ is a theorem in the logic; in this case,

a theorem of propositional logic. The symbol $\vdash$ is called the *provability relation*, and like its semantic counterpart, the entailment relation, it also enjoys the monotonicity property:

Definition 5 If $T \subseteq T'$, then $\{A |\ T \vdash A\} \subseteq \{A |\ T' \vdash A\}$.

A theory T is said to be *consistent* iff it is not possible for both $T \vdash A$ and $T \vdash \neg A$. An inconsistent theory is quite pathological since it would permit the derivation of every formula in the language.

If the axioms are tautologies and the inference rules are sound, then all the theorems of a logic are also tautologies. Such an inference system is called a *sound* one. A *complete* inference system is one with which it is possible to prove all tautologies. There are a number of sound and complete inference systems for propositional logic. In a Hilbert system, they have a single rule, *modus ponens*, but the set of axioms varies. The axioms are given as *schemata* that describe the form of the axioms, thus referring to an infinite set of formulae with the same pattern. For example, one common axiom is $A \rightarrow (B \rightarrow A)$, where A and B can be any formulae in $\mathcal{L}_{PL}$.

Propositional logic enjoys a number of important properties. It is sound, complete and *decidable*. The latter one stems from the question of *decidability*, the syntactical counterpart to the validity problem, and means that there exists an algorithm that can determine for any A whether or not $\vdash A$; the former two indicate that the semantics and inference system of propositional logic coincide quite nicely:

1. *Soundness*: If $T \vdash A$, then $T \models A$;

2. *Completeness*: If $T \models A$, then $T \vdash A$.

2.2.2 First-Order Predicate Logic

Propositional logic is not very expressive. It lacks the vocabulary to enable us to make statements about individuals, their properties and their relationships to each other; it lacks the vocabulary to permit us to make generalizations over sets of individuals. For example, in order to say that `all phones are off-hook` in propositional logic, we would have to write an awfully large number of statements of the form

phone1 is off-hook. phone2 is off-hook. and so on. With first-order predicate logic (*FOPL*), the expression can be simplified to a single statement by making use of a suitable predicate, in this case Phone_off-hook(x), whose truth-value is dependent on how the variable x, representing phones, is instantiated. First-order predicate logic is sufficiently expressive for most applications, but, as we shall see, the increased expressiveness comes with a price.

Most of the definitions and concepts introduced in the previous section apply here as well, but the changes to the syntax and semantics are significant enough to require definition.

Syntax

The alphabet of first-order logic consists of the following elements:

- a set of V of *individual variables*: $x, y, z \ldots$;
- a pair of logical connectives: $\rightarrow$ and $\neg$;
- one *quantifier*: $\forall$ (*universal quantification*);
- a non empty, countable set $\mathbf{P}$ of *predicate symbols*, where each $P \in \mathbf{P}$ has a fixed arity n;
- a countable set $\mathbf{F}$ of *function symbols*, where each $f \in \mathbf{F}$ has a fixed arity n;
- a pair of punctuation marks: '(' and ')'.

The predicate and function symbols are the non-logical symbols of the language. The arity associated with the predicates and functions represents the number of *arguments* they accept. Unary predicates are often used to define the properties of individuals; those of higher arity, the relationships among them. An example of the former is Phone_off-hook(x), and of the latter, Phones_connected(x,y), which might be used to refer to those pairs of phones which are connected to each other. 0-arity predicates are simply proposition constants. 0-arity functions are (individual) *constants*. There are a couple of common variations to the alphabet: (1) the set of individual constants

is distinguished (and not lumped under the function symbols), and (2) an equality "=" symbol, which is a binary predicate, is included in the alphabet.

The *terms* of the language are defined as follows:

- all individual variables and individual constants are terms,

- if $\alpha_1, \ldots, \alpha_n$, $n \geq 1$, are terms, and $f \in \mathbf{F}$ is an n-ary function symbol, then $f(\alpha_1, \ldots, \alpha_n)$ is a term.

Definition 6 The set of well-formed formulae over the alphabet is the smallest set satisfying the following conditions.

1. all proposition constants are wffs.

2. if $\alpha_1, \ldots, \alpha_n$, $n \geq 1$, are terms, and $P \in \mathbf{P}$ is an n-ary predicate symbol, then $P(\alpha_1, \ldots, \alpha_n)$ is a wff.

3. if ϕ and ψ are wffs, and x is a variable, then so are $(\neg\phi)$, $(\phi \rightarrow \psi)$, and $(\forall x(\phi))$

The set of all formulae is called the language of *FOPL*, and is denoted by $\mathcal{L}_{FOPL}$. We say, "for all **x**, **A** is true" for $(\forall x(A))$. In the expression $(\forall x(A))$, the variable x is *universally quantified* over A, and A is in the *scope* of $\forall x$. All occurrences of x in the formula A are considered *bound* by $\forall x$. Any variable in A that is neither quantified nor bound is considered *free* (of $\forall x$). A term that contains no free variables is called a ***ground*** one; a formula that contains no free variables is called a *sentence*.

Aside from the other connectives of propositional logic ($\vee$, $\wedge$, and $\leftrightarrow$), one more symbol can be added to the alphabet:

- $\exists$ (*existential quantifier*): $(\exists x(A)) \stackrel{\text{def}}{=} \neg(\forall x(\neg(A)))$

Note that the quantifiers are duals, so that $(\forall x(A)) \stackrel{\text{def}}{=} \neg(\exists x(\neg(A)))$. We say "for some **x**, **A** is true" for $(\exists x(A))$.

To reduce the number of parentheses, the following convention will be observed. The quantifiers and their variables will be combined whenever possible, and the quantification and scope will be separated by a '$\cdot$' symbol. For example, instead of writing $(\forall x(\forall y(\exists z(A))))$ we write $\forall xy\exists z \cdot A$.

Finally, if the language includes the equality symbol, then a set of *equality axioms*, which define its usual properties, are implicitly part of the set of axioms of the logic.

Semantics

The language of *FOPL* has elements that refer to individuals and their relationships. Therefore, a structure for the logic must be capable of representing such things.

Definition 7 An *interpretation* or *frame* for a first-order language $\mathcal{L}_{FOPL}$ is a pair $\mathcal{M} = \langle D, m\rangle$, where:

- D is a non-empty set, called the *domain* or *universe* of $\mathcal{M}$; and
- m is a function that assigns to each
  1. n-ary predicate symbol $P \in \mathbf{P}$, an n-ary relation over D; and
  2. n-ary function symbol $f \in \mathbf{F}$, an n-ary function from D^n into D. $\diamond$

Since there are only two possible 0-ary relations, 0-ary predicate symbols (proposition constants) are assigned one of two possible values, which echoes nicely the interpretations of $\mathcal{L}_{PL}$. Likewise, 0-ary function symbols (individual constants) are identified with individuals from the domain D, since 0-ary functions have $D^0 \longrightarrow D$ signatures. Thus, m assigns to each proposition constant p an element from $\{\mathbf{t}, \mathbf{f}\}$, and to each individual constant c an element from D.

An *assignment* a in $\mathcal{M}$ is a function from the set of variables to the domain D. In other words, it assigns to each variable x an individual $a(x) \in D$. This is sometimes called the *meaning* of x. The assignments are then used to give, via a *valuation function* $V_a^{\mathcal{M}}$, a value (or meaning) from the domain D for each term in the language.

Definition 8 Let $\mathcal{M}$ be a first-order frame for $\mathcal{L}_{FOPL}$, and $V_a^{\mathcal{M}}$ be the valuation function for the terms in the language under a frame $\mathcal{M}$ and an assignment a. The *value* of a term α, $V_a^{\mathcal{M}}(\alpha)$, is an element of D and is defined as follows:

1. $V_a^{\mathcal{M}}(x) = a(x)$, where x is any individual variable,

2. $V_a^{\mathcal{M}}(c) = m(c)$, where c is any individual constant;

3. $V_a^{\mathcal{M}}(f(\alpha_1, \ldots, \alpha_n)) = m(f)(V_a^{\mathcal{M}}(\alpha_1), \ldots, V_a^{\mathcal{M}}(\alpha_n))$.

Definition 9 A frame $\mathcal{M}$ and a variable assignment a *satisfy* a formula ϕ, written $\mathcal{M} \models \phi[a]$, under the following conditions:

1. $\mathcal{M} \models p[a]$ iff $m(p) = \mathbf{t}$, where p is a proposition constant,

2. $\mathcal{M} \models P(\alpha_1, \ldots, \alpha_n)[a]$ iff $\langle V_a^{\mathcal{M}}(\alpha_1), \ldots, V_a^{\mathcal{M}}(\alpha_n) \rangle \in m(P)$,

3. $\mathcal{M} \models (\neg\phi)[a]$ iff $\mathcal{M} \not\models \phi[a]$;

4. $\mathcal{M} \models (\phi \rightarrow \psi)[a]$ iff either $\mathcal{M} \not\models \phi[a]$ or else $\mathcal{M} \models \psi[a]$,

5. $\mathcal{M} \models (\forall x(\phi))[a]$ iff $\mathcal{M} \models \phi[a']$, for all assignments a' that agree with a everywhere, except possibly on x.

The symbol $\models$ is called the *satisfaction relation*; it relates a frame to a formula. Since the symbol is overloaded, it will be distinguished from the entailment relation when the context is not clear. The truth or falsity of $\mathcal{M} \models A[a]$ depends only on the assignments $a(x)$ given to the free variables x in A. Obviously, since sentences do not have any free variables, their truth values are independent of a.

A formula A is *satisfiable* iff $\mathcal{M} \models A[a]$ for some frame $\mathcal{M}$ and some assignment a; otherwise, it is *unsatisfiable*. A is *true in a frame* $\mathcal{M}$ (and $\mathcal{M}$ is a *model* of A) iff $\mathcal{M} \models A[a]$ for all assignments a. A is *valid* iff it is true for every frame $\mathcal{M}$. A is also valid if $\neg A$ is unsatisfiable.

Inference System and Properties

A sound and complete inference system can be defined for *FOPL*. The system is an extension of that for *PL*: it requires a couple of more axioms and one more inference rule, the *rule of generalization*: $A/(\forall x(A))$.

The only significant difference between *FOPL* and *PL*, with regards to their properties, is that *FOPL* is no longer decidable. There does not exist a mechanical procedure that can determine whether or not an arbitrary first-order formula is valid, nor whether or not the formula is provable. In general, there does not exist an algorithm that can determine whether or not an arbitrary formula is entailed from some theory. This negative result is the price paid for increased expressiveness. But it is not completely hopeless, for the logic is actually *semi-decidable*: a procedure can be defined that can determine that a formula A is valid, provided it is, but may not terminate with a result if it is not. The entailment and provability relations, however, are monotonic.

2.3 Modal Logics

The classical logics above permit us to express and reason about the facts that are true in *the* world. They do not distinguish between those facts that just happen to be true and those that cannot be otherwise. The former are called *contingent* or *possible* truths, the latter, *necessary* truths. This is a significant distinction because implicit in it is the notion of change. Some things are always true (or false); others, however, change over time, or from world to world. Although it is possible to capture this notion with classical logics, the resulting theory would be quite awkward to use. For example, if using *FOPL*, one would have to add worlds (or times) to the set of individual constants, add an extra argument to each predicate to refer to these worlds, and establish a set of predicates to relate the worlds to each other. Modal logic was developed to make this kind of reasoning easier and more concise.

The transition from a classical logic to a modal one is not very difficult. After all, *PL* and *FOPL* already refer (implicitly) to a single world. All that is needed is some way of referring (implicitly) to many worlds.

Only the propositional case is described below. The extension of *FOPL* to a modal first order logic proceeds in a similar fashion.

2.3.1 Modal Propositional Logic

Syntax

The alphabet of modal propositional logic *MPL* is that of *PL* extended with a couple of *modal operators* to identify the necessary and possible propositions.

- a pair of unary modal operators: $\Box$ (*necessity*) and $\Diamond$ (*possibility*).

Sometimes the modal operators are labeled as 'L' and 'M', respectively

Definition 10 The set of *wffs* over the alphabet is the smallest set such that.

1. every *wff* of $\mathcal{L}_{PL}$ is a *wff*;
2. if ϕ is a *wff*, then so are $(\Box\phi)$, and $(\Diamond\phi)$.

The set of all possible formulae is the language of *MPL* and is labeled $\mathcal{L}_{MPL}$. We traditionally say "box p" or "p is necessary" for $(\Box p)$, and "diamond p" or "p is possible" for $(\Diamond p)$. However, the modal operators can be given a variety of intuitive names depending on the application. For example, in a basic temporal logic, one says "always p" and "sometimes p," respectively. Like the quantifiers of *FOPL*, the modal operators are duals:

- $(\Box p) \stackrel{\text{def}}{=} \neg(\Diamond(\neg p))$; and
- $(\Diamond p) \stackrel{\text{def}}{=} \neg(\Box(\neg p))$.

As for precedence in a formula, they bind as tightly as $\neg$.

Semantics

The syntax does not make reference to the possible worlds over which the formulae are expected to hold. That aspect is "hidden" in the semantics. Kripke [40] proposed a *possible-worlds semantics*, now often called *Kripke semantics*, for modal logics. It is based on the observation that necessary truths are true in all possible worlds, while contingent truths are true in only some of those possible worlds.

Definition 11 A *Kripke frame* or *Kripke interpretation* for the language $\mathcal{L}_{MPL}$ is a triple $\mathcal{M} = \langle W, R, m \rangle$, where

- W is a non empty set of *worlds*, sometimes called the *universe*;
- R is binary relation over W, called the *accessibility relation*, and
- m is a function $P \times W \longrightarrow \{\mathbf{t}, \mathbf{f}\}$, where P is the set of proposition constants.

◇

The worlds can be a variety of abstract objects, such as time points. The relation indicates how the worlds are connected to each other. A tuple $\langle w_1, w_2 \rangle \in R$ means that w_2 is *accessible* or *reachable* from w_1. All worlds that are accessible from a world w are considered its *possible worlds*. Since the truth value of a proposition is dependent on what world it is in, the meaning function m requires the additional world argument.

Definition 12 A frame $\mathcal{M}$ and a world $w \in W$ *satisfy* a formula ϕ, written $\mathcal{M}, w \models \phi$, under the following conditions:

1. $\mathcal{M}, w \models p$ iff $m(p, w) = \mathbf{t}$, where p is any proposition constant;
2. $\mathcal{M}, w \models (\neg\phi)$ iff $\mathcal{M}, w \not\models (\phi)$;
3. $\mathcal{M}, w \models (\phi \rightarrow \psi)$ iff either $\mathcal{M}, w \not\models \phi$ or else $\mathcal{M}, w \models \psi$;
4. $\mathcal{M}, w \models (\Box\phi)$ iff $\mathcal{M}, w' \models \phi$, for all $w' \in W$ such that $\langle w, w' \rangle \in R$;
5. $\mathcal{M}, w \models (\Diamond\phi)$ iff $\mathcal{M}, w' \models \phi$, for some $w' \in W$ such that $\langle w, w' \rangle \in R$. ◇

Observe that the truth of a formula $\Box A$ (and $\Diamond A$) in some world is dependent on the truth value of A in other worlds. This is how a formula makes (implicit) reference to what holds in other worlds. Observe further that the interpretation of such formulae is implicitly affected by the nature of the accessibility relation. A number of restrictions can be imposed on the relation to enable us to match the intuitive interpretation we give to the worlds and modal operators. For example, if the universe is taken to be

a linear time line and the worlds time points along it related by the < precedence relation, then the relation on the worlds is transitive and antisymmetric. Different combinations of these properties give rise to different *modal systems*. The three most common are: T (reflexive), S4 (reflexive and transitive), and S5 (reflexive, transitive, and symmetric). The modal systems associated with these accessibility relations are labeled S_T, S_{S4}, and S_{S5}, respectively. Associated with each is a set of axioms that capture their properties. For example, transitivity implies the axiom $\Box p \rightarrow \Box\Box p$.

The usual definitions regarding satisfaction, truth and validity apply here as well. In particular, a formula A is true in a frame $\mathcal{M}$ (and $\mathcal{M}$ is a model of A) iff $\mathcal{M}, w \models A$ for all $w \in W$. A formula A is valid iff it is true in all frames $\mathcal{M}$. Sometimes the frame is identified with the accessibility relation, for instance, a *T-frame* is a frame $\mathcal{M}$ whose accessibility relation is reflexive. Similarly, one can say that a formula A is *T-valid* if it is true for all T-frames.

A complete and sound inference system can be defined for *MPL*. The set of axioms is dependent on the particular modal system employed. The logic is monotonic, sound, complete, and decidable.

2.3.2 Applications

It has already been intimated that modal logics are used in temporal reasoning. Clearly, modal logics are ideally suited for this application. In fact, there exist a wide variety of temporal logics. Part of the variation arises from the choice of underlying (temporal) structure: among some of the possible choices for the world are time points or states; and for the accessibility relation, totally ordered (linear time) or partially ordered (branching time). Another source of variation is in the choice of modal operators, with many logics employing more than the usual pair. A good general account of temporal logics is given by Rescher and Urquhart in [66]. When used for the specification of software systems, the temporal logic formulae are usually interpreted over (sets of linear or branching) computations or system behaviours (sequences of actions or states). Pnueli covers the field in [60]. See also articles in [84].

Modal logics have also been used as logics of knowledge and belief. Here, the

purpose is to reason about knowledge. What is knowledge and how is it different from belief? What does an agent know? Does it know what it knows? What are the effects of transmitting that knowledge to other agents? Can they achieve common knowledge? What knowledge is required to perform an action? These are just some of the concerns regarding the nature of knowledge and the behaviour of the agents that act on it. The reason for choosing a modal logic for this task is a very familiar one: an agent is said to *know* something if that something is true in all the worlds that it considers possible. There is some dispute over the exact nature of knowledge, but a possible-worlds semantics, first proposed by Hintikka [32], is able to capture that variation — again, by choosing the appropriate modal system. Typically, the $\Box p$ notation is replaced by a Kp one. If one wishes to model the behaviour of many reasoning agents, then a knowledge operator K_i is needed for each agent i. One can then make statements about whether agent x knows what agent y knows ($K_x K_y p$), and so on. Some overviews and surveys of these logics are given in [24, 25] and [18].

Chapter 3

(In)Completeness in Specifications

In the introductory chapter some dissatisfaction was expressed with the manner in which completeness, as it pertains to software specifications, has been commonly addressed. Certainly, the vagueness and misconceptions surrounding the term, along with the casual manner with which it is used, have rendered the primary goal of this thesis — that of defining a method for tolerating incompleteness — problematic at best. It is not exactly clear what precisely is supposed to be tolerated. It is important, therefore, to provide a useful definition of incompleteness, one that readily affords possible solutions. One possible approach is to first identify and classify the sources of incompleteness. Hopefully, from this classification it might become possible to determine which sources can be readily eliminated, which ones can be minimized, and which ones are inevitable and unavoidable. Then, a number of sources, from among those in the former two categories, can be singled out and appropriate solutions proposed for dealing with them. *This chapter is devoted to examining (in)completeness* from a number of perspectives, and provides such a definition.

The rest of the chapter is organized as follows. The next section looks at the nature of specification building and the role played by specifications in the software engineering process. From the discussion an informal definition of complete specifications is given. The second section examines the issue more rigorously, giving not just formal definitions of completeness, but weakened notions of it as well. Finally, the third section presents the classification of incompleteness and categorizes the sources

as indicated above.

3.1 Building a Specification

The importance of the specification phase of the software development process cannot be over-emphasized. Let us briefly examine the purpose of this phase and the important roles that the specification plays.

There are a number of ways to view the software development process. The overall process begins with the conception of an application, taken from some application domain, and terminates with a program. One useful and simple view of this process splits it into two sub-processes: The first sub-process deals with describing the application to be built. Traditionally, it has been the responsibility of the client to arrive at an adequate description, usually in the form of a set of (informal) requirements. What constitutes an adequate description, however, remains a fairly vague and debatable issue. For now, intuition suggests that it should at least be taken to mean that the requirements must accurately capture all the behaviours (or properties) of the application and that they be in a form that is suitable for communicating that information in a clear and concise manner. In any case, once the requirements are delivered, the second sub-process begins, that of constructing the program or implementation. At that point, it becomes the developer's duty to produce a correct implementation of the requirements, meaning that the implementation exhibits all the behaviours (or properties) that the client has requested for the product. We sometimes say that such an implementation *satisfies* the client's requirements, thus identifying the primary relation in the software construction process. Note that the definition of correctness[1] does not restrict a correct implementation from exhibiting additional behaviours beyond those expressed in the requirements, consequently there may be many implementations that can satisfy a particular set of requirements. Leaving aside for a moment what is meant by "all the behaviours," this sub-process may be regarded as a transformation between two very different objects: While the

[1] Correctness, like completeness, is another emotionally-charged term. Furthermore, it, too, is probably an unattainable property. See [73] for a detailed discussion on the meaning of correctness.

set of requirements is (usually) written in an informal (natural) language, the implementation is written in a formal (programming) language. Additionally, while the requirements only need to state what behaviours (or properties) are expected of the product, the program must focus on how to produce that behaviour with the given programming language. Thus, the program will contain considerable implementation detail which, though necessary for its execution, obfuscates what is being described. The difficulty of this process should be immediately apparent: how does one compare these two objects? To wit, given that there is no way to formally compare these two objects, how can we be sure that the transformation has been a successful one, that the implementation satisfies the requirements? Furthermore, without such a verificational guide, how can we effectively bring about this transformation?

There is a further complication. Consider again the satisfaction relation and the view of the software development process described above. Clearly, the requirements are of paramount importance in the process: everything hinges on them. If they do not adequately describe the application, then the program will probably not satisfy the client, even if it does satisfy the requirements. Unfortunately, the initial set of requirements provided by the client are rarely adequate. They often contain errors, ambiguities, inconsistencies, or may even lack some requirements. Some of the inadequacies may stem from using an informal language; others, from the client's unfamiliarity with some aspects of the application domain or the potential mutability of the client's requirements; still others, from the very nature of the task of trying to describe an entity that either exists in the client's mind or is taken from some complex real world domain, such as a telecommunications network. Assuming that an adequate description of the application can be discovered, and using the expression *intended requirements* to refer to it, then obviously we would prefer to have the implementation based on the intended requirements, not the initial ones. Thus, to bridge the gap between the conception of an application and its implementation, not only must there be a transition from an informal entity to a formal one, but the developer, with the help of the client, must also uncover the intended requirements. It should be mentioned, however, that it may be very difficult, if not impossible, to define the intended requirements for all but the most trivial of applications, especially if they

are written in an informal language. In essence, the intended requirements represent the ideal case.

Before suggesting a better approach and giving a definition of completeness, one final point must be made: the transition from informal to formal cannot be a smooth, gradual one. Indeed, either something is formal or it is not, keeping in mind that all semi formal structures, such as data-flow diagrams, are strictly not formal. At some point along the way, therefore, a formal entity will have to be introduced. Often, that entity (the first formal entity in the process) is the final entity in the process, the source code itself. But that is too late in the process, too late to uncover the errors, ambiguities, and inadequacies in the requirements. True, the source code can be executed, permitting the customer to directly examine the product, but any mistakes found in it will be very difficult and costly to correct. It would be far more efficacious in the long run if the first formal entity in the process appeared much earlier in the process, specifically at the level of the intended requirements. If anything, it would at least reduce or eliminate ambiguity in the process. But more importantly, given the importance of attaining an adequate application description and the pivotal role that it plays in the software development process, a formal description (of the requirements) would solve some of the problems mentioned so far. Let us now re-examine these issues from a slightly different perspective.

3.1.1 Software Specification is Theory Formation

To summarize, we need a bridge between the application and the implementation that eases the task of software development, in general, and the task of verifying program satisfaction, in particular. It is still assumed that the client is responsible for at least delivering to the developer some initial requirements. One possible solution is to build a formal *theory* of the application, where a theory (as usual) is simply taken to be a set of formal expressions. Although not a perfect one, there is an analogy here with the paradigm of theory formation in the natural sciences [11, 73]. Without delving too deeply into the nature of scientific study and what constitutes a proper theory (*cf.* [41, 61, 83]), theories in the sciences are succinct abstractions of observed phenomena. The symbols in a theory represent those factors or aspects of the real world that are

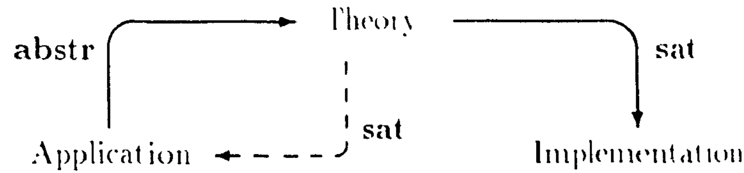

Figure 3.1: Theories in the software development process

deemed relevant. The theories are then verified by testing the real world for all the consequences deduced from them (after having suitably translated the formal symbols contained in them into real world ones). For example, one consequence of Einstein's theory of gravitation is that light must be attracted by heavy bodies, such as stars which was later shown to be the case. If any consequence of the theory cannot be verified in this fashion, then the theory is inadequate and must, hence, either be refined or abandoned in favour of a new one.

Similarly, in software development, a theory should be an abstraction of the application. It should express in a concise and formal way what the set of (intended) requirements attempts to expresses informally. However, unlike a theory in the sciences, this theory plays a dual role: in addition to being a description of the application, it is also a prescription for the final product. In other words, it also determines what the implementation will be like. The implementation, in this view, is considered a model of the theory, not in the logical sense, but rather as a concrete realization or instance of the theory. And although they are both formal theories (after all, the implementation is also just a set of formal expressions), the theory is much smaller than the implementation, since it lacks implementation-specific details. Thus, it falls somewhere in-between the application and the implementation, resulting again in a pair of relationships (the solid arcs in Figure 3.1) hinged at the theory: first, the theory is created by abstracting from the application (T **abstr** A); and then the implementation is derived from the theory, via the usual software construction process, such that it satisfies the theory (I **sat** T). Let us look at each in more detail, beginning with the right-hand side of Figure 3.1.

Since the implementation and theory are formal in nature, their relationship can in principle, be formally established (verified). Re-visiting the satisfaction relation, we

can say that an implementation satisfies a theory, $I \mathbf{sat} T$, if and only if I expresses all the behaviour of T. As in the sciences, we can make use of the notion of a consequence set here. Specifically, we note that the set of behaviours (or properties) expressed by a set of formal expressions can be captured by its *consequence closure*, which is the set of expressions derived from the theory and closed under the inference rules of the language. Therefore, though it might not be technically very easy to accomplish, given the difficulties associated with comparing formal theories written in different languages, it should be possible to determine whether or not $I \mathbf{sat} T$ by determining whether or not the consequence closure of T is contained in that of I. Observe that here, too, the consequence closure of I (and hence it's set of behaviours) can be a proper superset of T.

The other relationship, however, between application and theory, is much harder to establish. This is the relationship that most resembles the real-world-to-theory one in the sciences. As is the case with the real world, the application domains are usually informal or at least, the language and rules of these domains are still so inscrutable to us that, for all intents and purposes, they can be considered informal. There are exceptions, however; if the application domain is very well known and largely formal, then it might be possible to begin with a well-defined set of requirements, perhaps even formal ones. In any case, in the abstraction process the application domain can be regarded as a collection of entities, together with relations and functions over them. Furthermore, the client must decide what aspects (and consequently, what level of detail) of the domain to include in the description. For a telecommunications network, for example, one can concentrate on a high level view by looking at just the services that are to be provided for its users, or one can focus on lower level details, such as the switching networks. The question now is: how can we be sure that the abstraction process has been successful?

We can attempt to apply the same technique on the left-hand side of Figure 3.1 (the dashed arc) as we did on the right, except that now the informal nature of the application makes it very difficult to deduce its consequences. The analyst must rely on the client, or whoever conceived the application (or its initial requirements), and on their grasp of the application domain. Therefore, after having defined a

theory based on the application (or initial requirements), the analyst must interact extensively with the client in order to determine whether or not A**sat**T. Naturally, if the result is negative, then the theory must be refined and the whole satisfaction test repeated. Clearly, under such conditions — in which testing is very dependent on good human communication — one can never really be certain that the theory is absolutely correct. Instead, as in the sciences, if a point is reached where the testing no longer reveals any more differences between the two, then the best that can be claimed is that one has a "high confidence" in the theory. The theory can then be considered a suitable abstraction of the requirements. Since this is the hardest task in the software development process, the greatest effort and time in the process should be spent here. Once a good theory has been built, most of the ambiguities and inconsistencies in the requirements will have probably been removed, consequently making the rest of process easier to execute. Moreover, once we have confidence in both relationships (T**abstr**A and I**sat**T), we can say that the implementation is "equivalent" to the application — equivalent is given in quotes because in this case it is not a relationship that can be properly defined, let alone formally established. In this way, the theory "binds" the requirements to the implementation [73]. It should be obvious by now that the theory is the specification, or perhaps forms the basis for the specification, and that theory formation should be the primary purpose of the specification phase. It should also be evident that if a good theory is discovered for an application, then the specification represents an adequate description of the application.

Given some of the above observations, we are now in a position to provide an informal definition of a complete specification. To do so, we can make use of the observation that a good specification should be equivalent to the client's intended requirements. Equivalence here does not necessarily mean that the specification is simply a formal translation, expression by expression, of the intended requirements. Indeed, they may have radically different forms; what is important is that they express the same set of behaviours (or properties). Thus:

Definition 13 A specification is considered to be complete if and only if its consequence closure is equivalent to that of the set of the client's intended requirements. That is, the specification should contain the same behaviours (or properties) as the

set of requirements, no more, no less. ◇

In simple terms, this definition states that a specification is complete if it contains all the requirements that the client wants for the application. Of course, generating the consequence closure of the intended requirements and the notion of equivalence in the definition are somewhat problematic. In practice, since people are not that systematic in writing down everything they know about a subject and tend to leave gaps, it is taken for granted that completeness is achieved when all the straightforward inferences follow from the axioms written down as part of the theory. Ultimately, it seems that this notion of completeness will remain weak, unless we eventually learn how to link, in a systematic fashion, the real world entities modeled by the specification with expressions in the consequence closure of the theory.

3.2 Some Formal Considerations

The problem of completeness has been well examined in the context of logics. For example, it can be proven (*cf.* [61, 79]) whether or not a logic is decidable or complete. All the logics described in Chapter 2 are complete. Higher order logics, however, such as second-order predicate logic, are not complete. Furthermore, the notion of what constitutes a complete axiom set is a familiar one to logicians. It is a set of independent formulae (*i.e.* none derivable from the others) that is sufficient to generate, with the inference rules of the language, all the valid formulae in the language—if $\models A$ then $\vdash A$. These definitions, while useful to logicians, are not directly applicable to specifications. Logicians are not particularly concerned with the non-logical symbols, nor with their specific interpretations. Rather, they abstract away from such details (as in model theory), looking for general properties.

There are a number of ways to formally define the notion of completeness as it pertains to specifications. One way is to look at just the specification or theory itself; another is to define it with respect to other entities, such as implementations or requirements, as in Definition 13 above. Both are examined below. Unless otherwise noted, the discussion below refers to first-order predicate logic, but the notions may be extended to other formal systems.

3.2.1 Complete Theories

We need a definition that applies to arbitrary theories, keeping in mind that these theories use only some of the symbols in the language. In particular, the constant, predicate and function symbols vary from application to application. The treatment below is based primarily on that in [73].

A *complete theory* is a set T of independent formulae that is powerful enough to determine whether or not $T \vdash A$, for any formulae A in the language. Observe that this definition is slightly different from the one above (for a complete axiom set). Essentially, we want the theory to leave nothing (about the application) unsaid, identifying not just the positive facts, but the negative ones as well. Since it can be used to generate, with the inference rules of the language, every true formulae, the theory T is sometimes called a *complete axiomatization* (of the application).

Recall that with a logic that is both complete and decidable, it is possible to determine whether or not an arbitrary formula is deducible from a theory. In practice, however, the formal systems used in specification are neither complete nor decidable. Consequently, there will be true formulae that are not deducible from T; in other words, non-deducible true formulae are indistinguishable from false ones. From this definition, therefore, two possible sources of incompleteness can be identified: (1) the theory T may be too weak to determine whether or not an arbitrary formula follows from it– it lacks some information about the application or domain, and (2), the language may have inherent limitations– it is not even possible to develop a complete theory.

In general, an incomplete theory can be extended by adding independent formulae to it, reducing the incompleteness due to the first source. It should be noted, however that these extensions cannot go on indefinitely. A point will be reached where the only independent formulae left are those that are contradictions to the formulae already in the theory (or, rather, in its consequence closure). Adding these to the theory makes it inconsistent, and although this also makes it complete (since an inconsistent theory will generate every possible formula in the language), this sort of completeness is useless. Therefore, taking this into account, a complete theory is also one to which an independent formula could not be added without rendering it inconsistent.

Alternatively, a consistent theory is complete if and only if every larger theory is inconsistent.

Now, a specification of an application only uses symbols that relate to it. A specification of a phone system, for example, will have predicate symbols for the various phone states (`On_hook(x)`, `Ringing(x)`, `Connected(x,y)`, *etc.*), but no symbols for the orbital positions of the planets, nor any for the vital signs in a medical monitoring system, nor, in general, any symbols from any other application or domain. Technically, therefore, any formulae that refer to the planetary system or a monitoring system are independent of those for the phone system, and if they are not included in the theory, then it is, by definition, incomplete. This is clearly not what we intended. The definition does not take into account the heterogeneous nature of application symbols; and, unwittingly, the mismatch in symbols has introduced another source of incompleteness. One way to avoid this difficulty is to only compare theories that use the same set of symbols. This is done by type-categorizing the constants in the language, which, in turn, type-categorizes the variables, function and predicate symbols. Each formula can then be associated with the set of types that occur in it, and a set of formulae by the union of their respective type sets.

Definition 14 Let T be a theory characterized by types $\tau_1, \ldots, \tau_n$. Then, T is complete if and only if it can determine whether or not $T \vdash A$, for any formulae A in the language that is characterized by types $\tau_{A1}, \ldots, \tau_{Ak}$, where $\{\tau_{A1}, \ldots, \tau_{Ak}\} \subseteq \{\tau_1, \ldots, \tau_n\}$. $\diamond$

In a similar fashion, a consistent theory T characterized by types $\tau_1, \ldots, \tau_n$ is complete if and only if every larger theory characterized by the same types is inconsistent.

An interesting corollary to this definition is that when a fixed interpretation (a model $\mathcal{M}$) is given to the theory (and the language contains the *law of the excluded middle* axiom, $p \vee \neg p$), then the truth value can be determined for any sentence in the language. Under this condition, the above definitions imply that the theory (not necessarily an axiomatization) is complete.

3.2.2 Complete Specifications With Respect to Implementations

There are a number of ways to formulate the definition of a complete specification as it relates to its potential implementations. The one given below, taken from [11], expresses the notion using just set theory and logic.

The formal language used for specification is sometimes called a *formal* or *linguistic system*. Let Φ denote a formal system, defined as a pair $\langle \mathcal{L}, Cn \rangle$, where $\mathcal{L}$ is the language generated by some syntax **syn**($\mathcal{L}$), and $Cn : \mathcal{P}(\mathcal{L}) \longrightarrow \mathcal{P}(\mathcal{L})$ (where $\mathcal{P}$ is the powerset) is the *consequence closure operator* for the system. The consequence closure operator is simply the set of inference rules for the language. For example, with *PL* it is simply *modus ponens*; for *FOPL*, *modus ponens* and the rule of generalization; for the algebraic specification system [22], it might include rules of equational inference and structural induction. The operator enjoys the following closure properties:

1. $\forall x \subset \mathcal{L} \cdot x \subseteq Cn(x)$, (each theory is contained in its own closure);

2. $\forall x, y \subseteq \mathcal{L} \cdot x \subseteq y \rightarrow Cn(x) \subseteq Cn(y)$, (monotonicity),

3. $\forall x \subseteq \mathcal{L} \cdot Cn(x) = Cn(Cn(x))$, (closure is maximal).

The consequence closure of a theory T, $Cn(T)$, represents all the behaviours that are observable in the formal system. Note that changing the consequence closure operator, while keeping the same language, results in different sets of observable behaviours.

A *specification* S in formal system $\Phi = \langle \mathcal{L}, Cn \rangle$ is simply a set of formulae in the language; hence, $S \subset \mathcal{L}$. S is *consistent* iff $Cn(S) \subset \mathcal{L}$. This definition is sufficient to assure consistency because an inconsistent theory would contain every possible formula in its closure. A specification S' in Φ' *implements* a specification S in Φ (written S' **impl** S), where $\mathcal{L} \subseteq \mathcal{L}'$, iff

$$Cn(S) \subset Cn(S').$$

In other words, S' may be more detailed than S, perhaps containing implementation details, but it must contain all the consequences (behaviours) of S.

A *complete specification* is one that *fully determines* all its implementations. That is, the behaviour of each object in such a specification is so completely defined that no implementation can be constructed in which the object has more or less behaviour (as observed from the formal system of the specification). Formally,

Definition 15 A specification S is complete iff

$$\forall S' \cdot S' \ \mathbf{imp}\ S \rightarrow Cn'(S'/\ \mathbf{syn}(S)) = Cn(S),$$

where $(S'/\ \mathbf{syn}(S))$ means S' restricted to the syntax of S. ◇

As expected, the consequence closure of any implementation (suitably translated into the specification's language) must be equivalent to that of the specification.

A variation of this definition is given by Würges in [74], in which the application is viewed as an abstract machine (or black box)—it is made up of a set of states, a set of operations over those states, and a set of value functions (to describe the states). Briefly, a specification of an abstract machine is complete if and only if every possible program (sequence of operations) is well-defined, where well-defined means that a unique predicate transformer can be derived for the program from the specification. Since it is not possible to enumerate, let alone specify, each possible sequence of operations, Würges provides a set of rules for composing complex well-defined programs from simpler ones. The details of the specification method are not relevant, but essentially, to construct a specification, the analyst must first identify these basic sequences and then define a predicate transformer for each.

3.2.3 Comparison of the Notions of Completeness in Specifications

The definitions given so far attest to the variation in the meaning of the word completeness as it applies to specifications. Each of the three definitions (13, 14, and 15) focuses on a different intuitive notion regarding this concept.

Definitions 14 and 15, for example, are similar in that they capture, from different perspectives, the notion that a complete specification should have no loose ends, that there should be a value for everything. However, they do not adequately capture the

important consideration that a specification is about something. A specification is not simply a collection of unrelated statements; all the statements are about a specific application. Even if all the statements use symbols drawn from some finite set of types, there is no guarantee that they are describing the same particular system. Only Definition 13 relates completeness to the client's set of intended requirements. Unfortunately, for reasons given in Section 3.1, it is also the hardest one to verify. Nevertheless, this view of completeness is a very common one. Definition 15 also captures the prescriptive nature of a specification, one that fully determines its implementations. Note as well that the pair of relative definitions (13 and 15) appear to anchor the specification between the set of intended requirements on one side and the set of potential implementations on the other. Hence from this union, one can say that a complete specification must, for each behaviour stated in the requirements, fully determine its potential implementations.

Any of the three is appropriate, depending on what one perceives as important; for instance, someone with a strong logic background might prefer Definition 14. However, it seems that the concept is best captured by all three, and for the purposes on this thesis, all three will be regarded as part of the definition of a complete specification. This makes it a rather strong concept, but if the intention is to identify all possible sources of incompleteness, regardless of their immediate usefulness, then it is reasonable to base them on the strongest notion possible.

3.2.4 Weaker Notions of Completeness

Complete specifications or axiomatizations are nearly impossible to achieve. All of the definitions given above represent ideal situations. Unfortunately and inevitably the task of determining whether or not a theory is complete remains an undecidable one: it is simply not possible to define an algorithmic procedure that conclusively verifies this property. Acknowledging this problem Wing [74] offered the following four suggestions as possible solutions:

1. abandon the notion of completeness in specifications,

2. weaken the notion in some way;

3. weaken some other requirement, such as the form of the sentences permitted in the specification, or restrict the set of admissible specifications, perhaps by restricting the application domain;

4. establish a set of informal guidelines for showing completeness, such as identifying whether there are enough statements of a particular type; these guidelines may also prove useful to the developer as an aid to constructing the specification.

Although many developers end up (by default) adopting the first option, it is the least appealing since without some clear notion of completeness, it becomes very difficult to determine how closely the specification describes the application. Of the rest, Würges employs the fourth option, providing rules for simplifying the task of establishing completeness of predicate-transformer specifications. The technique, however, is an informal one, requiring one to prove completeness via force of (rigorous) argument.

Guttag and Horning [22] used a combination of the second, third and fourth options in an attempt to define a weak notion of completeness for their type algebras. Specifically, the notion, called *sufficient completeness*, is a weak version of complete axiomatization (of which Definition 14 is an example). Briefly, an algebraic specification is used to specify abstract data types. It can be regarded as a many-sorted first-order logic theory, complete with type, constant and function definitions. Each function must have at least one argument of the type being defined, the so-called type of interest *TOI*. Its axiom set, however, can only be composed of universally quantified equational axioms. Additionally, the left hand side of each axiom is restricted to a single term with at most a single level of nesting of functions. Basically, these conditions restrict the form (or "shape") that the algebraic specifications may take; thus, only a subset of possible specifications can even be considered. The set of functions can be divided into two types: those that return values of the TOI generate new elements of TOI, so they are called *generators*; and, those that return other values reveal the properties or behaviours of the TOI, so they are called *behaviours*. Other designations are possible, such as functions that modify the TOI (*modifiers*), but for the purposes of this discussion, these will suffice. An axiomatization of the TOI is

considered sufficiently complete if all the properties of the TOI are derivable from the axiom set. In other words, for every possible set of arguments for each *behaviour* function, a value can be found. Note the weakening in the definition of sufficient completeness: it does not require derivability of every possible formulae constructible within the theory; rather, it requires derivability for just the *behaviours*. Unfortunately, even with the restrictions and weakening of the notion of completeness, it is still an undecidable problem. As a solution, Guttag and Horning had to establish a *sufficient*, but not necessary, set of conditions for achieving sufficient completeness. If the conditions are met, then the axiomatization is considered sufficiently complete, otherwise, its completeness cannot really be ascertained. These conditions or rules recommend, among other things, that the functions be made total (for example, by introducing a distinguished *error* element as a return value for functions invoked with elements not in their domains), and that for each *generator* g and *behaviour* b in the theory, an axiom of the form $b(g(x, y'), z') = u$ must appear in the axiom set, where x is a TOI, y' and z' are tuples (possibly empty) of arguments, and u is a valid expression.

Turski and Maibaum [73] have also examined this issue. They believe that completeness is not only unrealistic, but unnecessary as well. A specification, they argue, should not define everything so completely; it should allow the developer some flexibility during software construction and provide some leeway for changes in requirements particularly if not all the requirements are known ahead of time. As a substitute they introduce another concept called *permissiveness*. The word was partly chosen because they felt that it is less emotionally-charged than completeness. Indeed, saying that a specification is permissive does not elicit the same reaction that saying it is incomplete would, even though they basically mean the same thing. To their credit, they chose a term that does not have an "either or" connotation — as in, *either* something is complete *or* it is not. Instead, permissiveness, which is not formally defined in the text, is envisioned as ranging between sufficient completeness on one side (most permissive) and the tightest limit imposed by the inherent limitations of the language on the other (least permissive). Since this thesis is partly concerned with the absolute notions of completeness, the term *permissive* will not be adopted here.

There is considerable merit in Turski and Maibaum's perspective on the issue. A certain amount of permissiveness may in fact make the developer's job easier, but how much permissiveness? At what point does it begin to adversely affect the process? At what point does it make it difficult to adequately verify or reason about the application? Is sufficient completeness a reasonable lower limit? Is a fixed limit suitable for all applications? How much permissiveness is tolerable for the specifications of critical applications, where any deviant behaviour is potentially disastrous? These are very important questions. Unfortunately, these issues are rarely addressed—not even by Turski and Maibaum—except perhaps in a superficial manner, hence no satisfactory answers are available for them. It is also possible that the concern over a reasonable limit is the wrong tack to take. Perhaps, it should be left fluid, with the main focus being directed towards finding methods for tolerating some permissiveness.

3.3 Types of Incompleteness

While the definitions given so far establish the notion of completeness in specifications, they are not directly useful from a pragmatic point of view. They basically imply that an incomplete specification is one that is lacking some information. A far more useful account is one that points out the ways in which a specification can be made incomplete, the intention being to isolate the activities or motivations involved. Once done, these sources of incompleteness might lead to appropriate solutions. Hence, the following classification of incompleteness types is based on the potential sources of incompleteness in a specification. Some of the types are based on direct violations of the definitions given above; others, on interpretations of problems (sometimes unrelated to the issue of completeness in specifications or simply ignored by software engineers) mentioned in different sources (texts and articles).

3.3.1 The Classification

Most software engineering texts point to two types of incompleteness (terms taken from [17]):

▷ 1. **External Incompleteness**: The specification does not adequately describe the application. This arises when the specifier has neglected to specify a desired property or behaviour. This is a pragmatic concern: some of the client's intended requirements have not been included in the specification. The obvious potential danger here, aside from not meeting the client's needs, is that if these omissions are not caught early in the process, they become difficult and costly to correct.

▷ 2. **Internal Incompleteness**: This occurs whenever the document has undefined entities (terms, operations, functions, *etc.*) in the specification. A typical example is defining the behaviour of an operation in terms of sub operations or functions whose behaviours have not been specified. Often, this type of incompleteness can be discovered either by inspection or by mechanical means for example, using symbolic execution for algebraic specifications

Notes: These types of incompleteness can be largely attributed to negligence or incompetence on the part of the specifier. Thus, these sources are rather pathological and should be eliminated if at all possible. To do so, a greater emphasis must be placed on the specification phase: extensive culling and analysis of requirements must be performed, together with the incorporation of better verification and validation procedures and tools.

Other potential sources -often ignored or not considered are the following:

▷ 3. **Weak Theory**: Quite simply, the theory is lacking independent statements about the application (positive or negative), rendering it too weak to determine the theoremhood of arbitrary statements in the language. It can also be due to a mismatch of symbols, although a strictly typed language should be able to take care of this.

▷ 4. **Permissive Specification**: The specification (of some component) has been deliberately left underspecified. This may be cleared up later (see class 5), or may be left that way, resulting in an implicit non determinism It is sometime

necessary to underspecify the description of actions in a concurrent system (see class 9)

Notes: These are general classes of incompleteness. They can probably be lumped in with some of the other classes. Many of them, for example, result in weak theories. However, since we are interested in specific reasons or motivations, they deserve their own classes.

▷ 5. **Changing Requirements**: Requirements are not static; they often change throughout the lifetime of the software product. This could be the result, for example, of the client realizing, after the initial requirements were defined, that the software should now have more (less or different) capabilities. Or, the client may wish to deliberately leave some aspect of the software open (not fixed) pending more study. These changes could be requested not only during the development of the product, but even after it has been delivered.

Note that these changes do not have to be monotonic in nature. In other words, the addition of new requirements may make some previous ones no longer acceptable or even possible.

▷ 6. **Incomplete Domain Knowledge**: Often, information about the application domain or the application itself is found to be lacking. There are a number of possible reasons:

1. the client (or developer) is not very familiar with the domain or application. The specification should be written by someone who has extensive expertise in the domain.

2. many domains are too complex to be handled *in toto*. Thus, the theory is built around one particular view of the domain. Different views of the domain may produce different theories, but selecting the appropriate view for a particular application is not a straightforward task. It is also possible that more than one view may be required.

3. there were elements of the domain that were truly unknown at the time the requirements or specification were made. Perhaps, new (recently discovered) information has superseded the old, or it has provided new insights into the domain. This is sometimes the case, for example, with software that depends on various scientific theories.

Notes: The above pair of sources of incompleteness are similar in two respects. One is that they have the potential of making a specification that was at one time considered "complete," no longer that; the other, that they provide good arguments for demanding that specification languages permit easy modifiability of specifications. It can also be argued, however, that they should not even be considered as sources, since they are beyond the control or responsibility of the developer (or client).

One can also consider software that is capable of overcoming the latter form of specification incompleteness as *robust*, in the sense that it has the ability to perform satisfactorily under "abnormal" conditions. For example, a robot that is designed to work in a "blocks world," yet has little difficulty maneuvering about in a completely different environment, say a "real world," is robust. Note that the tasks that cause abnormality arise from the environment, which is usually external to specification; were they to be specified, the abnormal situations would become part of specification, reducing the incompleteness of the specification.

▷ 7. **Inadequate Specification Language**: In general a specification language is not capable of expressing all the properties or behaviours of all systems. The language may simply be too weak, or it may not have the necessary constructs for specifying a particular feature of a particular system. For example, in the former case, a language such as propositional calculus is incapable of easily handling situations involving a variety of individuals, their properties and interrelationships; in the latter case, a language such as VDM [11, 35] is incapable of explicitly specifying concurrency. The language is also inadequate if it does not permit easy modifiability of the specifications.

▷ 8. **Undecidability and Incompleteness in Language**: Any useful formal language, one that is strong enough to express what is desired, will suffer from

these two inherent limitations. Although these have serious implications with regards to theory building, in practice their effects may not be very significant [64].

Notes: The above pair of incompleteness are related to the language that is chosen to write the specification. While we have some control over the former type, the latter one indicates that we may be forced to accept a certain amount of incompleteness in the specification.

▷ 9. **Partial Specification:** This very common type arises when a particular component of the system is not fully specified in the sense of Definition 15: it admits to more than one behaviour. This class can best be explained by breaking it down into three varieties. The terms are taken from the field of artificial intelligence; the reason for this will be made clear in the next chapter (Sub section 1.1.2).

1. The *qualification problem*: This refers to the situation where the pre-conditions for an action or operation have only been partially specified. This can arise either because some pre-conditions have not been considered (perhaps, they may even be infinite), or the developer has neglected to specify exception conditions. The latter case is sometimes called a *partial function*: if the pre-condition holds then the operation can be executed; if not, then the result is undecided.

2. The *frame problem*: Actions have a limited effect: a small number of objects are affected, the rest are not. The frame problem is the problem of having to specify all the things that have remained unchanged by the execution of an action. Usually, this is done with a *frame axiom*, a statement that indicates that a particular object has not been changed by the action. Since, in general, the number of objects unaffected by an action far exceeds the ones that are, the specifier often does not bother to include the frame axioms in a specification, relying instead on some implicit assumption that anything not mentioned has remained unchanged. Unfortunately, formally speaking, anything not stated explicitly cannot be relied upon.

3. The *ramification problem*. This occurs when one neglects to specify the changes (in state) that might occur to some objects as a consequence of a particular object being modified by an action. For example, let the complex object be a `folder` that contains a number of `file` sub-objects. If this `folder` is destroyed, then (intuitively) all of its `files` must also be destroyed, but if this is not specified explicitly, the status of the `file` objects becomes uncertain. Since the effects of an action are usually expressed in a post-condition, this problem indicates an inadequate post condition.

Notes: At first glance it might appear that this source of incompleteness can be attributed to laziness or negligence on the part of the specifier, and that it could be eliminated by simply being careful, systematic and vigilant during the writing of the specification: to wit, make sure all the pre-conditions have been included; do not define partial functions; include all frame axioms; insure that the post conditions include all possible side-effects. In short, do not assume anything will hold implicitly, state everything explicitly. Some specification languages provide the means with which to do some of this. For example, consider the following trivial VDM specification

Example 1 A Trivial Specification:

o State:: *Obj*:: X: *Int*
Y: *Int*
Z: *Int*;

OP1
ext wr o: *Obj*
pre $X(o) > 0$
post $X(o)' = Y(o) + Z(o)$

The following program fragments *implement* this specification

```
o if (x >= 0) then x = y + z;

o if (x >= 0) then x = y + z; else print("error ...");
```

```
o if (x >= 0) then x = y + z; y = z;
```

All of these fragments display the required behaviour, thus satisfying the specification, but the incompleteness in the specification results in many different (but not equivalent) behaviours in the specification. It suffers from both a partially defined pre-condition and the frame problem. The "corrected" version given below will select only the first implementation above (since the **errs** clause here means that no state change occurs whatsoever if the pre-condition fails):

o State:: *Obj*:: X: *Int*
Y: *Int*
Z: *Int*;

OP2
ext wr o: *Obj*
pre $X(o) > 0$
errs BADCOND: $X(o) \leq 0$
post $X(o)' = Y(o) + Z(o) \wedge Y(o)' = Y(o) \wedge Z(o)' = Z(o)$

Alternatively, the state could have been modified to make X of type *Nat*, thus eliminating the **pre**-condition altogether. This has the effect of turning a partial function into a total one. □

The real problem, however, with these sources of incompleteness is a pragmatic one. Quite simply, it would be a very tedious task to include all the necessary frame axioms, qualifications and ramifications for each action, particularly if the specification is large and complex. In fact, if an exhaustive attempt is made, the specification will have far more of this "background" material than of the "foreground" type, thus drowning out the relevant descriptive information and making the document very difficult to read. Furthermore, they exacerbate the problem of changing the specification. Changes to the foreground will often require significant changes to the background, with the necessary changes usually propagating in an unpredictable manner. Add a new state component, for example, and every action description is affected.

Since these problems complicate the task of specification, one might be tempted to ignore them. One might also be tempted to justify this by stating that humans are quite capable of (instinctively) dealing with, say, the frame problem. Why specify something that can easily be inferred by humans? The implementor will simply understand that anything not mentioned explicitly has remained unchanged. Unfortunately, this is not a dependable assumption because different people will interpret (underspecified) statements in different ways. Anything left unsaid, therefore, becomes a potential source for unwanted behaviour in the implementation. It might then be argued, following the permissive argument, that that is not such a bad thing, that leaving the specification of an operation or action a little underdetermined is beneficial. We may wish to be purposefully vague about the complete behaviour of an action, especially if its exact nature is not yet known — in a sense, introducing an implicit nondeterminism in the specification. This is the tack taken by Khosla and Maibaum in [37], even though they recognize that their specification formalism suffers from the frame problem. They claim that their approach is not affected by the problem to the same extent as approaches like VDM because it does not employ an explicit notion of state nor action descriptions that are explicit transformation of states. This may indeed mitigate the effects of the frame problem, but it does not eliminate it. In any case, it is not enough to say that the specification is permissive and leave at that. Since the traditional formal systems (such as those based on the logics in Chapter 2) cannot handle permissiveness, formally manipulating and reasoning about such specifications becomes quite difficult (unless one resorts to meta-theoretical constructs). The capabilities of these formalisms can only be exploited if each behaviour is completely and explicitly specified. This is a great concern to anyone working with formal methods—see for example [35], in which Jones employs a logic of partial functions [4], essentially a three-valued logic (true, false and "non value"), in an attempt to deal with just the problem of partial functions in VDM.

There are elements to this problem, however, that go even beyond the pragmatic concerns. One is that it might be impossible to include all the background information because it might be infinite. A more serious concern, however, and one that affects this thesis, involves the complications surrounding the specification of concurrent

systems. In the VDM example above, adding the frame axioms to the **post**-condition implies not just that Y and Z are unchanged by the operation, but also that *nothing else* can change them during its execution. This operation and any other that affects Y, for example, have been rendered automatically and unreasonably incompatible. In essence, it has forced an unwanted sequentialization of operations that might have otherwise executed concurrently. As a side note, the **ext** clause in VDM serves as an implicit global frame axiom for everything not mentioned there. Obviously, if we are interested in just sequential systems, this problem does not crop up; for concurrent systems, however, we are left with a dilemma: frame axioms as stated above are simply wrong, and yet they are needed to completely describe the action. Unfortunately, we cannot fix it by simply including a blanket statement of the form "Everything not mentioned explicitly is unchanged by an action, except where noted otherwise" to deal with the frame problem - or more generally, of the form "All the relevant facts have been explicitly mentioned" - because such statements cannot be expressed within traditional logics. In fact, they involve notions that have proven to be rather difficult to capture formally (see Chapter 4).

In summary, on the one hand, we need everything explicitly and completely stated in order to be able to properly reason about, and capture the behaviour of, the system, on the other, the task is a very difficult, if not undesirable, one (especially for a concurrent system). One solution is to restrict the domain significantly, which would perhaps keep the amount of information at manageable levels. Another is to define a language, methodology or tool that could *tolerate* partial specification, and thus not oblige the developer/specifier to deal with it. Such an approach would allow the specifier to state only the relevant facts, thus implicitly assuming that the description of the system is an incomplete one: and yet, it would still permit one to reason about and effectively describe the system. It would, in essence, "fill-in" any missing information whenever needed. Of course, this cannot be achieved for any type of information (where would it get it?); rather, only for information that exhibits predictable patterns.

3.3.2 Discussion

We are now in a position to further categorize the sources in the classification according to categories mentioned at the start of this chapter. That is, categorize them into the following three categories: sources that can be eliminated, sources that can be minimized, and sources that are inevitable and unavoidable.

I -*sources to eliminate* In this category are those types that are due to negligence or incompetence on the part of the developer (and, sometimes, the client). Types 1, 2 and perhaps type 6.1 and 7 (assuming that an appropriate language can be found, otherwise 7 belongs in the next category) can be lumped here.

II -*sources to minimize* In this category are those types that are still too difficult to be handled by our present capabilities (and knowledge). Types 3, 4, 5, 6.2, 6.3, and 9 should be included here. Of these sources, the ones due to incomplete domain knowledge and changing requirements are the most difficult to address, and perhaps they really belong in the next category. After all, how can one know in advance what one will need or know in the future (other than through some form of prophesy)? At best, it may be possible, through the appropriate selection of specification language or techniques, to build into the specification the ability to permit change easily. But how this may be achieved is as yet problematic.

III -*unavoidable sources* In this category, clearly, we find only type 8.

The first and third categories either have relatively easy solutions or none at all, respectively, thus they will be ignored. Of the classes in the second category, three general strains of incomplete information can be identified (making it an alternative categorization):

A -*genuinely incomplete information* The missing information is either truly unknown or very difficult to ascertain, such as changing requirements or uncertain (and perhaps infinite) pre-conditions;

B -*deliberately incomplete information* The information has been deliberately left out (permissiveness); and

C *implicitly incomplete information* It involves incomplete representation of (supposedly) complete information, such as the frame problem – at least, that is the case for sequential systems with restricted (closed-world) domains.

All of these general problems deserve attention. However, since the partial specification class isolates specific forms, is fairly ubiquitous, overburdens the specifier, and involves (to a certain extent) a couple of the above strains of incomplete information, it has been chosen as the focal point for the proposed method. Besides, as Chapter 4 will shortly show, the qualification and frame problems, at least, exhibit patterns that can be exploited.

Finally, it should be stated that the classification of sources of incompleteness may itself not be be complete (!), and that there is possibly some overlap of sources. Nevertheless, the classification and the associated categorizations, though crude, have proven to be quite useful, helping to identify a particular class of incompleteness to pursue.

Chapter 4

Incomplete Information and Non-Monotonicity

With the clarification of the notion of incompleteness in specifications and the associated classification of potential sources of incompleteness presented in the previous chapter, the goal of this thesis can now be refined and an appropriate solution proposed. Regarding the notion of incompleteness, a number of salient points were made: (1) basically, an incomplete specification is lacking the information necessary to achieve a complete axiomatization, a complete description of the application, or a complete determination of the potential implementations; (2) it is nearly impossible to arrive at a complete specification for non-trivial applications; and (3) a certain amount of permissiveness (how much?) is desirable in some cases since completeness may actually hinder the implementor's task or make it difficult to deal with changing requirements. From the classification, one particular source of incompleteness, partial specification, was singled out as an imminently worthwhile target, not only because it presents an interesting challenge, but also because it addresses a fundamental problem: namely, how can we capture the notion that "all the relevant facts have been explicitly mentioned?" Indeed, any formalism capable of capturing this notion would probably be capable of tolerating a variety of incomplete information, perhaps even that arising from changing requirements (*i.e.* genuinely incomplete information). It would also ease the specifier's task since it would no longer be necessary to explicitly

include all the 'background' axioms. Thus, the goal is now to define a specification language/methodology that

- tolerates partial specification (in particular, the qualification and frame problems),
- permits fairly easy modification of the specifications, and
- can be used to express real-time and/or concurrent systems (involving non-atomic actions)

What should be the basis for this language? One possible tack is to look at how similar problems are handled in other disciplines. As it happens, incomplete information is a concern in the field of artificial intelligence (AI). Specifically, it appears in the area of commonsense reasoning. Since many of the tasks that fall under that area often exhibit non-monotonic behaviour, it is not surprising that non-monotonic approaches have usually been advocated as potential solutions. This chapter examines in some detail this AI connection and the approaches involved, exploring their capabilities and weaknesses.

The rest of the chapter is arranged as follows: The next section describes the AI tasks that deal with incomplete information, including how the frame, qualification and ramification problems enter the picture. The second and final section reviews a few common non-monotonic approaches and their weaknesses, including their difficulties with the famous *Yale Shooting Problem*. From the known solutions to the latter, it identifies a suitable non-monotonic formal basis for the specification language.

4.1 Commonsense Reasoning and Incomplete Information

Nothing astonishes men so much as common-sense and plain dealing.
Ralph Waldo Emerson, *Essays*, xii. Art.

> Common sense is not a simple thing. Instead, it is an immense society of hard-earned practical ideas — of multitudes of life-learned rules and exceptions, dispositions and tendencies, balances and checks.
>
> Marvin Minsky [55, pg. 22]

Common sense is decidedly enigmatic. How can something that seems so simple, so obvious, and so natural to humans — something even children can do — be so incredibly difficult to capture, either formally or computationally? Such is the difficulty that neither goal is even remotely close to being achieved (except perhaps in the realms of science fiction). In contrast, tasks which appear very difficult to most people, such as solving mathematical formulas, playing chess against grand masters, or those involving a deep expertise in a particular field (for example, medical diagnosis), have been readily translated into programs, many by the mid 1960's [55, 58]. This dichotomy certainly appears perplexing, until one considers how much information each requires. Although the so called "expert" tasks may often require a considerable amount of esoteric information, their domains are strictly circumscribed — *i.e.* they involve a lot of knowledge about a narrow field. The facts in such domains are usually well known and enumerable. Common sense, on the other hand, consists of an astonishing body of knowledge in a wide range of domains, from the very general — such as quantity, space and time — to the very specialized, much of it interrelated in complex ways. This knowledge has yet to be compiled, in fact, it is not even clear what breadth and depth of knowledge is involved. Furthermore, this *commonsense knowledge* is combined with *commonsense reasoning*, a powerful array of methods for making sensible, rational inferences from this knowledge, including logical deduction (what the logics in Chapter 2 capture), abduction (inferring a plausible explanation for some situation), and induction (inferring a plausible general rule to explain the occurrence of a number of particular instances) [13]. The reason that it appears so deceptively simple is that people begin to acquire this intricate information and reasoning capability at a very young age, so by the time they reach adolescence most of it is already ingrained. It is also possible that some of it, especially those aspects that require sensory and motor skills, is already "hard wired" in the brain, a product of eons of evolution.

Since common sense is regarded a fundamental component of intelligence, there has been, almost from the inception of the field of AI, attempts to somehow model it. John McCarthy in 1959 was probably the first to propose this goal, but it was not until Patrick Hayes' exhortation in "The Naive Physics Manifesto" in 1978 (later revised in [29]) that a concerted effort was made in this regard. Hayes urged everyone in the field to begin formalizing, preferably in a first-order logic or an extension of one, the commonsense knowledge involved in a number of domains — his contribution, for example, was on liquids [30]. Since then, however, there has probably been a greater emphasis placed on the reasoning side of common sense, most notably on non monotonic reasoning (see next Sub-section). Nevertheless, the flurry of activity over the past decade on formalizing both commonsense knowledge and reasoning has perhaps, more than anything else, convinced everyone in the field how daunting and ambitious a task it really is. It is beyond the scope of this thesis to delve too deeply into the nature and nuance of human common sense, nor into the variety of approaches that have been advocated to model it. A provocative, though informal, glimpse of the nature of common sense can be found in [55]. As for the approaches, a good general overview and survey is found in [13], which uses many examples to convey the notions; [76] contains a collection of articles covering just commonsense knowledge, while [17] gives an extensive survey of the various formal treatments of non monotonic reasoning, a harsh critique of Hayes' position is given by McDermott in [53], which is, in turn, followed by numerous rebuttals from researchers in the field [77], see also [80], [81], and proceedings of the *AAAI* and *IJCAI* conferences.

What does common sense have to do with the goal of this thesis? Well, one characteristic of human commonsense reasoning is its remarkable ability to deal with incomplete information. Indeed, people rarely find themselves having all the necessary or relevant information in everyday situations, and yet they rarely have difficulty reasoning under such conditions. To take a simple example, consider the physical world — it is so immense, complex and varying that it is clearly not possible to keep a complete representation of it in our heads. Realistically, we can only positively claim to have knowledge of the things in this world that we can directly perceive with our senses — The computer in front of me on the desk, the books and papers

strewn around me on the table and floor, the Alex Colville "Couple on Beach" print hanging on the wall to the left, and so on, are some of the things that I am certain about, that I *know* about, because I can see them. The existence of everything else, however, I must infer from, amongst other things, my (partial) knowledge of the world around me and a basic understanding of the nature of physical objects, including the rules of "lawful change" of such objects. One such useful rule is the law of inertia. (It is rather fortunate that it applies to our world, otherwise the world would be so chaotic that making inferences about it would be probably impossible.) I am quite certain, for example, that there is a glass of water on the counter in the kitchen, even though I left it there over an hour ago and can no longer see it from where I am now sitting. I have a high degree of confidence in this inferred 'fact' because of the law of inertia, together with the facts that I left the glass (stationary) on a level surface and there is no one else around who could move it. I am also certain of the existence of Concordia's main downtown campus building because I have been there many times and it is even less likely to be moved than the glass. Still, neither inference is guaranteed to be correct. A mischievous friend might silently pilfer my glass, an earthquake might topple the building without my knowing it. Thus, inferences based on incomplete information are tentative, and consequently may have to be retracted on the basis of new evidence. This is where the non-monotonicity appears, revealing a key aspect of commonsense reasoning: it proceeds through a string of plausible or rational assertions or inferences, but since 'plausible' and 'rational' do not imply certainty, this string can be undermined by new information. In contrast, deductive reasoning deals only with absolute certainties. It yields inferences that are necessarily true if the premises are true; they cannot be assailed by new information.

This ability is much more powerful, complex and subtle than is demonstrated by this simple example. For example, we have the innate ability to appreciate the difference between the persistence of the glass-of-water-on-counter fact and that of the existence of the building. Since glasses can easily be moved, we do not expect the former fact to last very long. We have a 'feel' for approximately how long something should persist and how reliably we could depend on it in our reasoning. Thus we can qualify the uncertainty or incompleteness. There are many other subtleties to it

but most are not relevant to this thesis; the next pair of sub-sections discuss the ones that are. In any case, what is important here is that incomplete information does not prevent people from making plans, predictions and decisions, or undertaking actions. It does not paralyze them, even in the event that these turn out to be incorrect, inappropriate or inapplicable in the light of new information: they simply adapt to the (unexpected) changes in their information base. They are not bogged down by it as is the proverbial robot confronted with a new environment.

Note, however, that while reasoning with incomplete information has its drawbacks, it does not imply that the it would be easier to reason with complete information (assuming one were somehow able to acquire it). Experimental mobile robots, for example, that depend on having a (suitably) complete internal representation of their environments spend most of their operating time sensing and building two or three dimensional models of their worlds [10, 58]. One of the most successful of these robots was SRI's Shakey. Shakey's environment was specially engineered to assist its vision and image processing capabilities. The rooms were clean, well defined (dark baseboards were used to distinguish walls from floors), well lighted, and bare except for large, uniformly coloured blocks and wedges. Yet, even in this contrived environment, Shakey did little more than lurch about slowly, taking an hour of computing time (to build a world model and decide on a plan of action based on it) before each movement. Moreover, all the processing and detailed information provided Shakey with only a very limited ability to deal with unexpected events. Rodney Brooks [9, 10], giving other examples as well, is especially critical of the reliance on a complete internal world model, citing it as the main reason for the poor performance of these machines. After all, there are computational limits to take into account. But more importantly, if we wish to build systems that exhibit some "intelligence," systems that will react quickly and effectively to external stimuli, systems that will be more than just toys playing in and reasoning about very restricted domains, then they will have to be equipped with the ability to handle incomplete information.

Let us now examine the fundamental nature of non-monotonic reasoning and how non monotonicity creeps into reasoning about change and time.

4.1.1 Non-Monotonic Reasoning

Consider the following example: Usually, Fred meets Emma on Friday mornings at 10:00 to discuss the progress of his work. These are scheduled meetings and, based on past experience, Fred expects Emma will be in her office at the appointed time. However, there are potentially very many reasons why Emma would not be in her office on Friday this week, from the mundane, such as coming down with the flu, to the unlikely, such as a bomb destroying the building, to the incredible, such as being abducted by space aliens. And yet, despite all these possibilities, Fred still assumes that Emma will be in her office at 10:00 this Friday. And Fred arrives at this belief without running through all these possibilities in his mind (most of which he has no information about, anyway). He does not attempt (at least not consciously) to determine the truth-value of an expression like this:

> If she does not have the flu, and if she is not away on a business trip, and if the building has not been destroyed by a bomb, and if ..., then she is in her office on Friday at 10:00.

In fact, if his decision (on whether or not to go to her office) depended on an evaluation of this rule, then he would indeed be paralyzed since it is not computable; not simply because he may not have enough information to evaluate some of the propositions, but also because the "..." in the expression represents a potentially infinite list of propositions. So, in assuming that Emma will be there, Fred is actually "jumping to that conclusion" on the basis of just a few 'facts.' Most of the time this provisional conclusion is the right one; but occasionally, it turns out be wrong. This is the price that must be paid for the benefits of such a powerful inference mechanism. For example, let us now suppose that on his way to the meeting, Fred runs into Emma's secretary who informs him that Emma would not be in that day because she has the flu. The new information has invalidated Fred's original conclusion. It has changed his set of beliefs *non-monotonically* because the new information is incompatible with the conclusion. Formally, this type of human reasoning violates the monotonicity principle. That is, if $T \subseteq T'$, then it is not necessarily the case that $\{A \mid T \vdash A\} \subseteq \{A \mid T' \vdash A\}$. Alternatively, if $A \models B$ (A entails B) then it is not the case that

$A \land C \not\models B$. The AI community has dubbed this type of reasoning *non-monotonic reasoning* or *defeasible reasoning*; specifically, it is the drawing of conclusions which may be invalidated by new information. Obviously, the monotonic logics described in Chapter 2 are not capable of capturing an inference of this sort. Their inference systems, centered around trusty *modus ponens*, restrict them to handling deductive reasoning.

While many aspects of human commonsense reasoning (perhaps most) are quite different from deduction, there is something about Fred's reasoning that has a quasi-deductive feel. This has given the researchers in the field the hope that it might be formalizable. Returning to the example, why did Fred reach the conclusion that Emma would be in her office at the appointed time? He really only had knowledge of the following facts: that (1) (because of their regular meetings) "Emma is usually in her office at 10:00 on Friday," and (2) "It is 10:00 on Friday." At first glance it may appear reasonable that he arrived at conclusion (C) "Emma is in her office" using an inference rule of the form: given (1) and (2), infer (C). Unfortunately, this rule is insufficient: it does not allow Fred the freedom to reject (C) when given new information. If the rule is accepted as a valid one, how will he reconcile (C) with the news that she is at home with the flu? Actually, Fred's reasoning tacitly involves an ignorance clause. To wit, the reasoning proceeds as follows: given (1) and (2), *in the absence of evidence to the contrary*, infer (C). Thus, commonsense conclusions are based on both the presence and the absence of information. This leads to the following definition.

Definition 16 In general, a *non-monotonic rule* has the following form:

Given A, and the absence of evidence B, infer C. ◇

A common variant of this rule makes no mention of B at all, stating that "given A, in the absence of evidence to the contrary, infer C," or "given A, if it is consistent (with the theory) to believe C, then assume C." These should not be mistaken as syntactic or inferential rules for some language; they are merely definitions. And despite their seemingly deductive feel, they are nonetheless non-deductive rules. How to represent these rules in a language and how to distinguish between known and assumed facts

are two key issues that the language must address. A number of approaches have been advanced in this regard, some of which are described below

There is one other element to Fred's reasoning that must be highlighted Statement (1) uses the word 'usually,' which implies that this represents a highly probable case, a so-called *default* case. That is why Fred can usually count on it Obviously, the higher the probability of its occurrence, the higher the confidence one can have in any conclusions derived from it.

4.1.2 Reasoning About Change

Reasoning about change is a fundamental concern in most (real world or AI) domains It manifests itself in any discussion involving notions like time, state, action, and causation. By change we usually mean a change in state, or a change in the state of an entity, where a state is taken to be a 'snapshot' in time of the world (or entity) Instinctively, we associate change with the passage of time, with notions of past, present, and future, with before and after The light was off a short time ago, but now it is on, therefore there has been a change. In a static world in which nothing changed, the concept of time itself would be meaningless: without change, how would we even measure time [68]? We also associate change with action because actions are usually the cause of change. The light was off, but I flipped the switch, and now it is on. This, in turn, points to causation as another means through which change can be described. In any case, regardless of which approach we choose to describe change, we run into three classical problems: the *qualification, frame* and *ramification* problems (time [13, 68], action [17, 21], causation [70]). Hence the reason for the nomenclature given to the types of partial specification in Sub section 3.3 1 Interestingly enough the qualification and frame problems were responsible for stimulating much of the early research into non-monotonic inference [47].

What is the nature of these problems? Where does the non monotonicity enter the picture. Let us have another look at them from slightly different perspectives

1. **qualification problem:** In order to determine or predict the outcome or

success of an action, it is important to first determine whether all of its pre-conditions have been satisfied. Obviously, we can only have confidence in the prediction if we are certain that all possible pre-conditions for the action have been considered. However, as we have already seen, this is rarely achievable in commonsense reasoning because the number of pre-conditions can be infinite. Put simply, it is very difficult, in general, to specify all the pre-conditions for an action. This problem clearly echoes the general one presented in the previous section. That is, it can be expressed in terms of the following non-monotonic rule: "given that the pre-conditions hold, and in the absence of evidence to the contrary, assume that the action succeeds." Another way of viewing the qualification problem is as a trade-off between the accuracy of the prediction versus the efficiency of computing that prediction [21, 68, 70]. Very often, most of the pre-conditions are either negligible because they are almost always true, or not evaluable because of insufficient information. There may also be a large class of conditions that are unknown. By ignoring these types of conditions it becomes possible to arrive at a prediction quickly. In fact, even if we *did* have enough information to evaluate all the possible pre-conditions, it would still be too costly computationally to do so. For example, before using a car, we do not normally check the car battery (even though we could), or anything else under the hood. We assume it is working because batteries can last a fairly long time before failing. The drawback is, as expected, that our predictions will occasionally turn out to be wrong.

2. **frame problem**: McCarthy and Hayes [51] were the first to recognize this problem, which is the difficulty of specifying all those things that are not affected (persist) when an action is performed or time passes. If we knew all the actions and objects in the domain, it would be possible to provide all the necessary frame axioms (and consequently avoid resorting to a non-monotonic formalism). In the simplest approach, this would require a separate axiom for almost every combination of action and object, each basically stating that object o has the same value after the execution of action a as it did before. There are several problems with this approach: (1) since most actions affect only a small set of

objects, the number of frame axioms will be enormous — if there are n actions and m objects, then there will be on the order of nm frame axioms; (2) the large number of frame axioms makes the task of determining which objects are unaffected computationally costly, perhaps intractable [21]; (3) adding new actions and objects requires significant changes to the theory; and (4) concurrent actions cannot be specified (see Sub-section 3.3.1). Davis [13] gives a number of alternative formulations, one of which permits concurrent actions, but while they require fewer frame axioms, the axioms are more complex, additional constraining axioms are needed, and the proofs harder. Perhaps, therefore, the problem is best tackled from a non-monotonic perspective: "assume that all things in a state persist, except for those which have been explicitly changed."

Shoham [68] generalizes the frame problem by including an implicit notion of increasing uncertainty with time. The *extended prediction problem* is the difficulty of making predictions over an extended period of time into the future. It is related to McDermott's notion of *persistence* [52]. For example, using the glass of water example again, can we predict whether it will still be on the counter 10 seconds later? 10 hours later? how about 10 months later? Now, what about Concordia's main downtown campus building? Clearly, there is a qualitative difference between these persistences. The length of time into the future is important and it varies from fact to fact. In general, however, predictions over a short period of time into the future are more reliable than those over longer ones. Thus, here too, there is a trade-off between the reliability of the prediction and its computational cost: an unreliable yet very efficient long term prediction versus a costly long-term one made up of very many reliable short term ones. This view, however, is probably more appropriate to the prediction task in AI. Finally, the reader is also directed to [78] for some alternative (and broader) views on the frame problem, from philosophical and cognitive perspectives.

3. **ramification problem**: This is the problem of specifying all the consequences of an action. The difficulty arises because some consequences may imply others which in turn imply others, and so on. If I drive my car from point A to point

B, then not only am I and the car there, but so is the engine, the steering wheel, the battery, *etc.* Again, it is unreasonable to explicitly mention all these consequences: many axioms would be required and it would be computationally costly to reason about them. There is a non-deductive element here as well, but unlike the other two problems, expressing this notion in the form of a non-monotonic rule is problematic.

Most formalizations of reasoning about change in AI have dealt primarily with the qualification and frame problems (with the latter one getting the bulk of the attention). The ramification problem, on the other hand, has been largely ignored, [21] being a notable exception. As an initial proposal, this thesis will follow suit, concentrating only on the first two problems.

These three problems take part in a larger problem, the *temporal projection problem* (*cf.* [26]), which involves the following: given an initial description of the world, a set of action descriptions, and the occurrence of some events (action instances), determine what facts will hold after the events have occurred. An interesting parallel can be drawn between this problem and specifications. The three components can be viewed roughly as the boundary conditions (including invariants), operation descriptions, and behaviour of the system, respectively. The first two give a (static) description of the system, while the last a prescription of the system's dynamic behaviour. Most specification formalisms, such as VDM, typically allow one to specify only the former. Khosla and Maibaum [37], however, argue persuasively that such specifications are inadequate. The fallacious assumption in such languages is that system behaviour can be inferred from static information that basically defines change implicitly (through `pre-` and `post-`conditions in operation descriptions). Reasoning about system behaviour is consequently difficult and quite limited. Once suitably recast, therefore, the temporal projection problem should serve as a useful focus for the specification language in this thesis: any AI formalism that handles it properly must be considered a candidate for the basis of the language.

4.2 Non-Monotonic Approaches

With missing information, monotonic logics cannot be used to reason about the system; on the other hand, if all the missing information is added to the theory of the system (assuming that that is even possible), then the task of reasoning is rendered computationally costly or intractable. Monotonic logics are therefore clearly unsuitable to the task. This section briefly describes the three most common formal non-monotonic approaches. The presentation is not a comprehensive one; rather, the primary intention is to give a flavour of the types of solutions that have been proposed for one of the questions raised above: how to represent non-monotonic rules. Each of the following approaches achieves this in a different way. Additionally, each subsection will highlight a different key aspect of non-monotonic reasoning that is relevant to this thesis: under default logic, the nature and difficulties surrounding non-monotonic inference; under autoepistemic logic, its epistemic connections, and under circumscription, the notion of minimal sets. There are other approaches, including some computational ones, but these three have garnered the most research over the last decade and have spawned numerous variants. See [47] for more detailed descriptions of these approaches and references to others.

4.2.1 Default Logic

Default logic, introduced by Reiter [65], almost literally adopts the non monotonic rule. A *default theory* is a pair $\langle W, D \rangle$, where W is a set of (first order) sentences representing what is known to be true, and D is a set of *default rules* of the form

$$\frac{\alpha : \beta}{\gamma},$$

where α, β and γ are formulae in the language. This default rule, or simply *default* is informally interpreted as "if α is known and it is consistent to believe β then infer γ." In other words, if α (the *prerequisite*) is taken to be true, and we have no reason to doubt β (the *justification*), then we can accept γ (the *the consequent*). Free variables in the formulae of the defaults are considered to be universally quantified such defaults are called *open defaults* and may be viewed as schemas. There may

be more than one justification in a default, but frequently it is the same as the consequent $\alpha : \beta / \beta$, with meaning "given α, without evidence to the contrary, infer β." These default rules, it must be stressed, are not part of the object language; rather, they are meta-theoretical constructs added to the language to achieve plausible reasoning. As an illustration of a default theory, the Emma-Fred example above can be written as follows:

$$D = \{\texttt{friday_at_10:00} : \texttt{Emma_in_office} \,/\, \texttt{Emma_in_office}\}$$
$$W = \{\texttt{friday_at_10:00}\}.$$

Non-monotonicity complicates the semantical and inferential components of a logic. Reiter, in fact, did not provide a semantics for the logic (although one can be given for certain classes of theories [15]); he did, however, give some general procedures for determining the inferences from a theory. Basically, the idea is to find all those sentences that are consistent with the theory. Such a set, called an *extension* of the theory, would include inferences from both the monotonic and non-monotonic sides of the logic. Unfortunately, finding an extension is not a simple task because the justifications in a theory's default rules can lead to circular reasoning; that is, given a rule δ, $\alpha : \beta \,/\, \gamma$, whether one has reason to doubt β often depends on whether δ itself can be applied. Furthermore, it is possible that accepting one default renders others inapplicable, consequently a theory may have more than one extension (or even none at all). The circularity suggests a fixed-point construction for the extensions: Let $T = \langle W, D \rangle$ be a default theory and let E be a set of sentences in the language. (The definition is restricted to closed theories (*i.e.* no free variables in the sentences and defaults), but this distinction is not critical to this discussion.) In a simplified form, then, E is an extension of theory T if

$$E = \bigcup_{i=0}^{\infty} E_i,$$

where

$$E_0 = W$$
$$E_{i+1} = Cn(E_i) \cup \{\gamma \mid (\alpha : \beta \,/\, \gamma) \in D, \text{ and } \neg\beta \notin E\}$$

Note the circularity in the definition: in particular, $\neg\beta \notin E$ in the definition for E_{i+1}. This means that a default inference can only be applied if $\neg\beta$ cannot be inferred, but this means that E has to already be known. Simply put, to find an extension E one has to practically begin with a set E, use it to determine which defaults are applicable, derive all possible inferences and then compare this set with E. If they are the same, then E is an extension. Obviously, this definition does not lead to a satisfying definition of inference. In addition to multiple (and usually incompatible) extensions the most glaring disadvantage is that the task of determining whether or not an arbitrary sentence follows from or is consistent with a theory is undecidable. Reiter suggests that default reasoning is really one of selecting one of the extensions and then reasoning the usual way within it until new information invalidates it. Unfortunately, even this is problematic since there does not exist a procedure for generating the extensions (or even determining their number) in the first place. (There are other formulations of default logic and its inferential component, as well as partial solutions to the problems raised here, see [15, 17, 65].) We end this section with a pair of examples.

Example 2 *Tweety Bird.* The typical example used in the literature involves Tweety the bird. Most birds can fly. Can Tweety fly? Without information to the contrary, we infer that Tweety can fly:

$$D = \{\texttt{Bird(x)} : \texttt{Can_fly(x)} \;/\; \texttt{Can_fly(x)}\}$$
$$W = \{\texttt{Bird(tweety)}\}.$$

This theory has a unique extension, which is the desired one.

$$E = Cn(\{\texttt{Bird(tweety)}, \texttt{Can_fly(tweety)}\})$$

If we find out, however, that Tweety is actually a penguin, then we must add to W the sentences Penguin(tweety) and $\forall x \cdot \texttt{Penguin(x)} \rightarrow (\texttt{Bird(x)} \wedge \neg\texttt{Can_fly(x)})$. Now the new extension includes ¬Can_fly(tweety), as expected. □

Example 3 *Nixon Paradox.* This example demonstrates multiple extensions. Quakers tend to be pacifists, while republicans tend not to be. Nixon is both a quaker and

a republican

D {Quaker(x) . Pacifist(x) / Pacifist(x).
Republican(x) . ¬Pacifist(x) / ¬Pacifist(x)}

W {Quaker(nixon).Republican(nixon)}.

Without more information this default theory has two possible extensions, both containing the sentences Quaker(nixon) and Republican(nixon): one, however, also includes Pacifist(nixon) while the other ¬Pacifist(nixon). □

4.2.2 Autoepistemic Logic

Unlike default logic, autoepistemic logic, introduced by Moore [57], incorporates the non monotonic rules within the language. It is a modal non-monotonic logic whose primary modality represents the notion of belief. Specifically, it employs a modal operator 'L' interpreted as "it is believed," and its complement 'M' (= ¬L¬, as usual), interpreted as "it is consistent to believe." A typical (autoepistemic) non-monotonic rule has the following form:

$$\forall x \cdot (\mathtt{Bird(x)} \wedge \mathtt{MCan_fly(x)}) \rightarrow \mathtt{Can_fly(x)}.$$

with interpretation "if **x** is a bird, and it is consistent to believe that **x** can fly (alternatively, it is not believed that **x** cannot fly), then **x** can fly." In other words, the only birds that cannot fly are the ones that are believed not to; every other bird, however, can be inferred to fly. This then is the way that autoepistemic logic handles the rule "in the absence of evidence to the contrary, infer that a bird can fly."

As the name would suggest, the logic is intended to model an agent reflecting upon its own beliefs. Moore contends that autoepistemic reasoning is quite different from default reasoning. As an example, note the difference in the following reasonings about Tweety.

default Since most birds can fly, I predict that Tweety can fly.

autoepistemic If Tweety could not fly I would know it, therefore, Tweety can fly.

The former argument relies on a fact that has a high probability, hence the conclusion is a fairly safe, though defeasible, one. the latter one, however, implies that I have complete knowledge (or believe I have complete knowledge) of all the birds that cannot fly, and since Tweety is not in that set, the conclusion is a logically valid one. The implicit assumption (justifiable or otherwise) of completeness of information within a particular set of beliefs (a *context*) makes the conclusion not defeasible. The rule, therefore, is merely an incomplete representation of supposedly complete information. Nevertheless, autoepistemic reasoning is non monotonic because it still deals with the absence of information. Once the rule is embedded in a different context (a different agent's or even the same agent's at a different time), previous inferences may no longer be derivable.

However, the distinction — and many others, for example [17] defines six varieties of non-monotonic reasoning — is not significant here, except perhaps from a philosophical point of view. Both logics draw conclusions from both the presence and absence of information. And autoepistemic logic can handle the Tweety bird problem, which is largely considered to be a case of default reasoning. Shoham [68] goes so far as to argue that there is no distinction whatsoever. Instead, Shoham suggests a more useful one: we should distinguish between the meaning of the sentences and the extra-logical reason for adopting that meaning. The meaning can usually be viewed epistemically — for example, "most birds fly" means "if I do not *know* that a particular bird can fly, then I should infer that it can." The reasons for adopting such a meaning are computational efficiency and economy of memory space. To wit, the meaning suggests that only birds that cannot fly need to be mentioned explicitly; the rest can be inferred. Then, whenever one needs to check if a particular bird can fly or not, one simply checks the list of exceptions.

One of the advantages of incorporating the rules within the language is that they can be given a formal semantics. Moore looked only at the propositional case — like many logics dealing with knowledge and belief, quantifying into the scope of an epistemic operator is problematic (*cf.* [56]) — which, aside from the modalities, has the usual interpretation. The question is: what meaning should be given to L (and M) to capture the above notions? From the discussion, we note that the interpretation of L*p*

(and Mp) is dependent on the agent's set of beliefs. On this basis, an *autoepistemic interpretation* for the logic is a pair $\langle m, S \rangle$, where m is the interpretation function assigning truth values to the propositions and S is the set of beliefs. Given such an interpretation, Lp is true if and only if p is in S, and Mp is true if and only if $\neg p$ is not in S. These definitions clearly capture the intuitive notions.

Like default logic, autoepistemic logic involves some circular reasoning. What an agent can infer depends on the set of beliefs, and the set of beliefs is determined by what can be inferred. Thus, Moore also gives a fixed-point construction to define the extensions, called *stable sets*, of a theory. A belief set S is stable if and only if it satisfies the following conditions.

1. $S = Cn(S)$,

2. if $p \in S$, then L$p \in S$, and

3. if $p \notin S$, then $\neg$L$p \in S$.

Such a set represents the set of beliefs of an *ideally* rational agent. The first condition states that the agent can handle ordinary deductive reasoning; the other two indicate that it is aware of what it believes and disbelieves. Like default theories, autoepistemic theories may have one, many or even no stable sets.

For other formulations of autoepistemic logic, including a first-order case and possible worlds semantics, and an examination of its properties, see [39, 47, 57].

4.2.3 Circumscription

Circumscription, introduced by McCarthy [49], is based on the notion that the only objects that satisfy a particular predicate are the ones that can be shown to do so; every other object, therefore, can be inferred to not satisfy it. This echoes, in spirit, the implicit assumption of completeness in autoepistemic logic resulting from the belief that one knows all of the objects that have some property. Like autoepistemic logic, circumscription incorporates the non-monotonic rules within the language — in this case, a classical (first-order or second-order) predicate logic; unlike it, however, the inferences proceed in the usual syntactic fashion of classical logic — no

need to resort to a fixed-point construction. *Prima facie*, this appears to be a major advantage. Unfortunately, as one might warily expect with non-monotonicity, it comes with a price: first-order non-monotonic inference in circumscription reduces to *second-order* monotonic inference, and second order logic does not have a complete deduction system. Nevertheless, it is certainly preferable to reason monotonically than through a fixed-point construction. If an automatic higher-order theorem prover were ever developed, it would make circumscription a very attractive formalism for commonsense reasoning (although the actual usefulness of such a prover is debatable [53]).

Since McCarthy's original formulation (called predicate circumscription), there have been about ten different versions of circumscription, including formula [50], second-order [43], and pointwise circumscription [45]. The following description (based partly on that given in [47]) is a general one, the notions applicable to most.

Recall that the denotation[1] of an n-ary predicate is the set of n-tuples over the universe of objects that satisfy it. Circumscribing a predicate p in a theory T has the effect of minimizing its denotation such that it could not be made smaller without contradicting T. To represent non-monotonic rules a special predicate Ab, for *abnormal*, is employed in the rule [50]. For example, the statement "in the absence of evidence to the contrary, assume that a bird can fly" can be expressed as

$$\forall \mathtt{x} \cdot (\mathtt{Bird(x)} \land \neg \mathtt{Ab(x)}) \rightarrow \mathtt{Can_fly(x)}.$$

Literally, this means that if **x** is a bird and it is not an abnormal one, then it can fly. Now, let T_1 be the theory containing this rule and Bird(tweety). Since this theory does not have any abnormal objects, the minimal possible denotation of Ab is clearly the empty set. Thus, by circumscribing Ab in T_1, we should derive

$$\forall \mathtt{x} \cdot \neg \mathtt{Ab(x)},$$

meaning that there are no abnormal objects. From this and the theory it is easy to derive Can_fly(tweety). Note that, without resorting to circumscription, it is not possible to deduce this from the theory using just the usual first order inference

[1] This is sometimes also called the *extension*, but to avoid confusion this term will not be used.

mechanism. Now, suppose that Clyde is a non-flying bird, let theory T_2 be the conjunction of T_1 and

$$\texttt{Bird(clyde)} \wedge \neg\texttt{Can_fly(clyde)} \wedge \texttt{clyde} \neq \texttt{tweety}.$$

Note that the inequality expressed in the last conjunct is required in circumscription in order to differentiate the objects in the domain. clyde is clearly abnormal, consequently he is the only member of the minimal possible denotation of Ab. From this observation, we expect that from the circumscription of Ab in T_2 we should be able derive

$$\forall \texttt{x} \;\; \texttt{x} \neq \texttt{clyde} \rightarrow \neg\texttt{Ab(x)}.$$

and then

$$\forall \texttt{x} \cdot (\texttt{Bird(x)} \wedge \texttt{x} \neq \texttt{clyde}) \rightarrow \texttt{Can_fly(x)}.$$

How can this minimization be captured? How can such derivations be achieved syntactically? Most circumscription formalisms achieve it through a second-order *circumscription axiom* that is assumed to be implicitly included with the theory. This axiom differs from formalism to formalism. Perhaps the commonest version is that of second-order circumscription [13]. It permits one to circumscribe predicates while allowing others to vary. (McCarthy's original formulation permitted only the circumscription of predicates, and was consequently generally too weak to handle the non monotonic rules.) Given a tuple of distinct predicate constants $\overline{P}$, a tuple of predicate constants $\overline{Q}$ disjoint with $\overline{P}$, and a theory $T(\overline{P},\overline{Q})$, the *second-order circumscription of $\overline{P}$ in $T(\overline{P},\overline{Q})$ with variable $\overline{Q}$*, written CIRC(T; $\overline{P}$; $\overline{Q}$) is the sentence

$$T(\overline{P},\overline{Q}) \wedge$$

$$\forall \Phi \Psi \;\; [[T(\bar{\Phi},\bar{\Psi}) \wedge \bigwedge_{i=1}^{n} [\forall \overline{x} \cdot \Phi_i(\overline{x}) \rightarrow P_i(\overline{x})]] \rightarrow \bigwedge_{i=1}^{n} [\forall \overline{x} \cdot P_i(\overline{x}) \rightarrow \Phi_i(\overline{x})]]$$

The first conjunct is the theory; the second is the circumscription axiom. In the antecedent of the axiom, $T(\bar{\Phi},\bar{\Psi})$ guarantees that the denotations of $\overline{P}$ are minimized

with respect to T. The axiom basically states that if Φ_i is any predicate that satisfies T and is at least as strong as P_i, then P_i is exactly as strong as Φ_i, for all circumscribed predicates. The effect is similar to that of autoepistemic logic: if a predicate p cannot be shown to follow from theory T, then from $CIRC(T, p)$ we should be able to derive $\neg p$.

To illustrate, taking theory T_1 above and circumscribing Ab while allowing Can_fly to vary, we get

$$
\begin{aligned}
CIRC(T{:}\texttt{Ab}{:}\texttt{Can_fly}) = {} & T_1 \\
& \wedge \forall \Phi \Psi \cdot [[\texttt{Bird(tweety)} \wedge [\forall \texttt{x} \cdot (\texttt{Bird(x)} \wedge \neg \Phi\texttt{(x)}) \rightarrow \Psi\texttt{(x)}] \\
& \wedge [\forall \texttt{x} \cdot \Phi\texttt{(x)} \rightarrow \texttt{Ab(x)}]] \longrightarrow [\forall \texttt{x} \cdot \texttt{Ab(x)} \rightarrow \Phi\texttt{(x)}]]
\end{aligned}
$$

Next, we have to make the appropriate substitutions for Φ and Ψ such that all the objects are normal (substitute Φ with false) and all the fliers are birds (substitute Ψ with Bird(x)). Making these substitutions and simplifying gives us the expected result, $\forall$x $\cdot \neg$Ab(x). This, in turn, can be used to derive Can_fly(tweety). Observe that if another sentence were to be added to the theory, the circumscription axiom would change as well. New derivations from this changed axiom may invalidate old ones—adding the above clyde sentence, for instance, would invalidate the $\neg$Ab(x) one—hence the axiom is the source of the non-monotonicity.

Circumscription, as is partly evident in even this trivial example, is not easy to use. There are two problems: (1) although it is not the case here, deciding which predicates to circumscribe and which ones to vary is not always clear—different choices will result in different circumscription axioms (and hence, different derivations), and (2) once a circumscription axiom has been defined, carrying out the derivations through an appropriate set of substitutions, is highly problematic. Even if a theorem prover were available, it would only be of use with regards to the second problem, not the first. In essence, the formalism requires a considerable amount of input from us. If we have an idea of what conclusions we would like to see, then it might be possible to arrive at the appropriate choices to avoid both problems. But this can only be effectively accomplished for small, familiar theories, which obviously limits its usefulness.

4.2.4 Objections to Non-Monotonic Formalisms

Many have remarked that non-monotonic logics are not logics at all. Certainly, if one defines a logic as a language based around deduction, then none of them are logics. Whether they are 'logics' or not, however, is a trifling matter. A much more important concern is whether they can be used to effectively deal with incomplete information, in general, and the qualification or frame problems, in particular. Of course, this issue must also be addressed in the context of software specification.

Despite the differences in how they achieve non-monotonicity, the three logics above, as well as any other formalism to date, suffer from the same problems, a number of which have already been addressed. McDermott [53], in assessing the suitability of using deductive and near-deductive formalisms to model commonsense reasoning in AI, summed up the main problems of non-monotonic formalisms as "You can't find out" and "You don't want to know." The first one means that it is usually very difficult to determine, without considerable effort, the consequences of a theory. Quite simply, the logics in the general cases are hopelessly intractable. With fixed-point constructions, it is not possible to tell what is and is not inferable until everything has been inferred. With circumscription, one has to practically know the conclusions in advance (which clearly defeats the whole purpose). See also [67], which shows that general circumscription is inordinately uncomputable. The second problem means that the inferences are sometimes so weak that they are of little practical use, making the effort spent deriving them doubly wasteful. This commonly manifests itself in multiple extensions. Furthermore, while some of the extensions are reasonable, most are counter-intuitive and would probably be rejected by an ideal, 'rational' agent. Default and autoepistemic logics obviously suffer from this, yet so does circumscription, although in a different form. In circumscription, one can usually take the disjunction of all the possible derivations from conflicting minimizations. For instance, minimizing the abnormalities in the Nixon theory (with rules $\forall x \cdot (\texttt{Quaker}(x) \wedge \neg\texttt{Ab1}(x)) \rightarrow \texttt{Pacifist}(x)$ and $\forall x \cdot (\texttt{Republican}(x) \wedge \neg\texttt{Ab2}(x)) \rightarrow \neg\texttt{Pacifist}(x)$) would make it possible to derive $\texttt{Pacifist}(\texttt{nixon}) \vee \neg\texttt{Pacifist}(\texttt{nixon})$, which is true but hardly illuminating. It has been suggested that fixed point derivations could be treated in a similar fashion by

taking the intersection of all the extensions (*cf.* [17]). Supposedly, this intersection would represent the core set of beliefs, but since there is no general effective procedure for generating the extensions, the suggestion is moot, at best. McDermott's caustic critique of non-monotonic formalisms concludes (pg. 157) with,

> The original goal, of a simple, general extension of classical logic that would grind out "obviously correct" conclusions, has eluded us.

It should be mentioned, however, that McDermott does not extend the criticism to computational approaches of non-monotonicity since many AI systems already incorporate some form of non-monotonicity. For example, with a database containing the current facts, one can practically ignore the frame problem. After each action, one merely updates the database with the effects of the action, adding new facts and deleting others; the rest remain unchanged, hence they 'persist.' This is just one of those cases where it is easier to implement a notion than formally specify it. In condemning the practice of using just logics (without programs) to model common sense, the article has added more fuel to the perpetual proceduralist vs. logicist conflict in AI.

While McDermott's ultimate assessment of the utility of current non-monotonic formalisms is perhaps too pessimistic, the observations are fairly acute. Even many of the respondents who challenged the critique agreed that the problems exist, although they disputed their severity [77]. McCarthy and Lifschitz [77, pp. 196-197], for example, blame the difficulties on the relative infancy (approximately a decade old at the time) of the field of formalized non-monotonic reasoning. They also point out that there are certain restricted classes of some non-monotonic languages that are tractable. They do not mention any specifically, but some examples can be found in [72], where the complexity of three membership problems are examined for a variety of default theory classes. To give an example of how much these theories must be restricted to make them tractable, here is one: the class of propositional default theories made up of (W) single-literal clauses (where a literal is a propositional atom) and (D) prerequisite-free default rules in which the consequence is identical to the justification. But is such a serious loss in expressivity worth the gain in computability?

Two points must be considered. One is that the dispute here is over the usefulness of non monotonic formalisms in the context of AI; for a specification language, the concerns are a bit different. We are not interested in building a machine that can reason correctly and efficiently, rather, we want a language that is expressive enough to allow us to describe at least the systems we are interested in. The other is that even first order predicate logic is intractable and logical deduction computationally expensive, yet people have not been reluctant to use them (or languages based on them) to specify and reason about a system. Worse, many specification languages do not even have an established inference mechanism. So, although it would be nice to be able to determine computationally the consequences of a theory, expressivity is more important. Still, non monotonicity complicates proofs sufficiently to warrant restricting the class of admissible theories. Fortunately, we need just enough expressive power to handle the qualification and frame problems (or the temporal projection problem)

Another common response is that the problem is not with the languages themselves, but rather with the way they are used, or with the user who has misunderstood their capabilities, strengths and weaknesses (*e.g.* [77, deKleer (pp. 174 175), Hayes (pp. 179 185), Moore (pp. 198 201) and Poole (pp. 205 206)]). This argument is a valid one, regardless of whether the logic is monotonic or not. No formalism is (as of yet) a panacea for representing ideas. One must be aware of what can and cannot be expressed with the language; one must have a clear idea of the semantics; one must appreciate the limitations of the inference mechanism. Furthermore, using a logic does not mean that one will automatically avoid writing incorrect or suspect statements. Thus, if the derivations one gets from a theory are unexpected or appear peculiar, it could mean that either the language is at fault, the theory is incorrect, or one has misconstrued the language's notion of inference. With this in mind, what then is the nature of non-monotonic inference in the logics above? Why do they often result in multiple or weak extensions? The simple answer is that it is a natural result of reasoning with incomplete information. Different extensions arise from the different ways that the missing information could be filled in (as was shown above for conflicting non monotonic rules). And if there is a lot of missing information, then

it is not that surprising that the theories generate fairly weak extensions. Expecting more from them is unrealistic. Just because the logics are meant to capture a certain aspect of commonsense reasoning does not mean that they are imbued with it. An interesting possibility arises, however, from the perspective of specification languages: the multiple extensions can be viewed as representing the implicit non-determinacy of underspecified (permissive) specifications. This potential connection should perhaps be further explored.

Unfortunately, there is a deeper concern here, one that goes beyond this simple explanation. It was mentioned above that some of the extensions are counter-intuitive and truly unwanted. Could this point to some inherent weakness or inadequacy in the languages themselves? The following famous problem illustrates this concern.

4.2.5 The Yale Shooting Problem

In trying to test the usefulness of non-monotonic formalisms, Hanks and McDermott [26] conceived of a scenario that isolated one particular problem but was otherwise simple enough to permit easy intuitive answers. The scenario, which has since been dubbed the *Yale Shooting Problem* (YSP), is a particular instance of the temporal projection problem (see Sub-section 1.1.2) in which just the frame problem is involved. Thus, the implications of this problem are especially relevant to this thesis.

The problem is typically expressed in the *situation calculus* [51], a language intended for the formalization of reasoning about action. A variant of first order predicate logic, it permits us to specify what *facts* hold in particular *situations* (states) and the changes in states caused by *events*. The former is achieved through a special predicate `Holds(f,s)`, where `f` is a fact and `s` a situation; the latter, through a special function `result(e,s)`, which maps an event `e` and situation `s` into another situation. For example, `Holds(gun_loaded, result(load_gun,s))` means that in the situation after a `load_gun` event occurs in situation `s`, the gun is loaded. Note that each state change is caused by a single action, hence the language forces a strict sequencing of (atomic) actions.

To express the assertion that the occurrence of an event `e` in situation `s` has no

effect on fact f (*i.e.* a simple frame axiom), one writes

$$\texttt{Holds(f,s)} \rightarrow \texttt{Holds(f,result(e,s))}.$$

As has already been explained above, we would need many such axioms to deal with the frame problem within a monotonic language (and situation calculus is no exception). Thus, we resort to a non-monotonic formalism, say, to circumscription and abnormality predicates:

$$\forall \texttt{f,e,s} \cdot \texttt{Holds(f,s)} \wedge \neg\texttt{Ab(f,e,s)} \rightarrow \texttt{Holds(f,result(e,s))},$$

where Ab(f,e,s) means that fact f is abnormal with respect to event e in situation s. This 'global' frame, or more appropriately, *persistence* axiom basically states that if a fact is not abnormal with respect to the event (*i.e.* is not affected by the event), then it must persist across the occurrences of that event (in all situations). With such an approach, we are only required to specify all the 'abnormalities' or exceptions to this rule—in other words, instead of specifying all the things that are not changed by an action, we specify just the things that are. Note the significant difference between this view and the one implied by the simple frame axiom. This is exactly the type of approach that would be suitable for the specification language, but will it work?

The YSP involves the loading of a gun, followed by a short period of waiting before the trigger is pulled, the problem is to determine what facts hold in the various situations and, in particular, whether or not the gun will make a loud noise[2]. Obviously, the persistence of the loaded gun fact is central to the problem. If it persists beyond the waiting period, the gun will make a noise, thus shattering the quiet; otherwise, the quiet will be unaffected. The theory contains the following axioms:

1. $\texttt{Holds(loaded,}S_0\texttt{)} \wedge \texttt{Holds(quiet,}S_0\texttt{)}$

2. $\forall \texttt{s}\ \texttt{Holds(loaded,s)} \rightarrow \texttt{Holds(noise,result(shoot,s))} \wedge \texttt{Ab(quiet,shoot,s)}$

3. $\forall \texttt{f,e,s} \cdot \texttt{Holds(f,s)} \wedge \neg\texttt{Ab(f,e,s)} \rightarrow \texttt{Holds(f,result(e,s))}.$

[2]Actually, in [26] the problem also involved an individual (initially alive). The problem was to predict whether or not the individual would be killed by the shooting, but this violent aspect is really unnecessary.

The problem has been simplified for this discussion by assuming that the gun is already loaded in the initial situation (axiom 1). Shooting a loaded gun makes a loud noise; furthermore, since shooting a loaded gun affects the quiet, it is clearly abnormal with respect to the latter (axiom 2). Now, consider the following sequence of situations:

- S_0
- S_1 = result(wait,S_0)
- S_2 = result(shoot,S_1).

We know that loaded and quiet hold in S_0. What facts hold in the other situations? To solve this problem, we circumscribe the theory over Ab while allowing Holds to vary. Since we have no reason to assume that the wait event has any effect on loaded and quiet, we assume ¬Ab(loaded,wait,S_0) and ¬Ab(quiet,wait,S_0). From these and axiom 3, we infer Holds(loaded,S_1) and Holds(quiet,S_1); thus both facts persist through the waiting period. Then, from the former fact and axiom 2, we infer Ab(quiet,shoot,S_1), meaning that quiet will not persist to situation S_2, and Holds(noise,S_2). These are the results that we were intuitively expecting (they are summarized in Table 4.1, interpretation A). Unfortunately, it is not the only possible interpretation that will satisfy the theory. Nothing prevents us from initially assuming Ab(loaded,wait,S_0). It might not make sense to us, but if we proceed from this assumption, we will not be able to derive Holds(loaded,S_1), and consequently there will be no noise in S_2 (see interpretation B in Table 4.1). Since both interpretations involve a single abnormality (Ab(quiet,shoot,S_1) in A and Ab(loaded,wait,S_0) in B), both are minimal and neither is preferable over the other. Therefore the best that we can deduce is that Holds(noise,S_2) ∨ Holds(quiet,S_2).

Why is the second interpretation unacceptable? Well, we were hoping to capture the notion that all of the events that can affect a fact have been explicitly specified and that the fact should persist until it is clipped by one of them. The theory does not state anywhere that a wait can affect a loaded fact; therefore, there should not be any "mysterious" unloadings during a wait. The quiet fact, on the other hand

	S_0	S_1	S_2
A	loaded quiet	loaded quiet	loaded noise
B	quiet loaded	quiet	quiet

Table 4.1. Two interpretations of the Yale Shooting Problem

is known to be affected by a `shoot` event, and its clipping in the first interpretation is entirely appropriate.

What is the reason for this unexpected extension? Hayes [77, pp. 179–185], for instance, blames the axiomatization and not the logic, suggesting that one should add ¬`Ab(loaded,wait,s)` to the theory. This will indeed work, but this is just a disguised frame axiom — we should not be required to state the obvious, just the abnormal. Besides, it hinges on the sequentiality of the situation calculus; in a formalism capable of expressing concurrency, it would simply be false. While a `wait` cannot affect a `loaded` fact, another action occurring during the `wait` might do so. Other suggestions are addressed, and equally dismissed, in [26] and [77, McDermott (pp. 223–227)]. Is the language itself, then, at the root of the problem? Hanks and McDermott [26] show that the problem is not peculiar to circumscription, since it also happens with default logic and McDermott's non-monotonic modal logic *NML*. Nor is the situation calculus to blame, since they got the same results with a simplified version of McDermott's temporal logic [52]. It appears, therefore, that this is an inherent weakness in the three logics covered above, and perhaps of non-monotonic inference in general. This is a very pessimistic result, which essentially disqualifies these logics as potential candidates for the formal basis of the specification language.

Solutions to the YSP

Negative results can at times be as useful as positive ones, but the situation is not entirely gloomy. Since the discovery of the YSP, many solutions have been proposed for it (*e.g.* [3, 27, 41, 45, 59, 68]). Before commenting on these, however, one important observation must be made about the two interpretations above. Another way to arrive at the second one is to begin at situation S_2 and reason *backwards* in time.

To wit, we begin by first postulating Holds(quiet,S_2) and then figuring out how such a fact could have come about. This is sometimes called *backward projection* and, along with temporal (or forward) projection, is a necessary component of any theory of explanation — the task of explaining what went wrong when an unexpected outcome occurs [59]. Thus, these logics are perhaps giving us more than we need. Observe further that the abnormality in the first one occurs *later* than in the second (Ab(quiet,shoot,S_1) versus Ab(loaded,wait,S_0), respectively). This gives us a clue on how to select an interpretation: not the ones that just have a minimal number of abnormalities, but rather those in which the abnormalities occur as late as possible. Hanks and McDermott [26] assert that this minimality or preference criterion, coined *chronological minimality* by Shoham [68], is the correct one for the temporal projection problem. The three logics above could not handle the YSP because they are simply incapable of representing this criterion. It should be noted, however, that chronological mininization is not the only criterion or solution to the temporal projection problem.

However, most of the solutions proposed for the YSP are inappropriate for the goal of this thesis. Some employ suitably enhanced or modified versions of circumscription, such as Lifschitz's pointwise circumscription [45] (which also uses chronological minimization) or Baker's solution [3], but these suffer from the same difficulties as any other general circumscription logic. In particular, one must already know what derivations to expect from a theory before being able to carry them out [29]. At best, this permits us to verify our original intuitions, but it provides us with no new information. Others, such as Haugh's [27] and Lifschitz's [44] causal minimization approaches, are based on extensions or variations of the situation calculus. These solutions are quite dependent on this formalism and would not work with, say, any temporal representation formalism that permitted one to specify concurrent action. Besides, they also use circumscription to minimize the causes, thus inheriting its drawbacks as well. Still others, such as Morgenstern and Stein's [59] approach based on motivated actions, are intended for larger problems such as explanation. This does not necessarily detract from it, since it at least appears to avoid the problems of the other approaches listed here. However, since this thesis proposes a rather original

approach to the problem of dealing with incomplete specifications, it is probably prudent to proceed with an approach that just handles the temporal projection problem. Shoham's approach [68], though it has been criticized for it (*cf* [3, 59]), does just that. It is, perhaps, the most promising approach for a number of reasons: (1) it handles a more general YSP that includes the qualification problem; (2) not only does it provide the means with which to establish the preference criteria, it also provides an algorithm to generate the facts true in all the preferred interpretations for a certain class of propositional theories; (3) it does not depend on a sequential temporal representation language; and (4) it accomplishes this through a very elegant formalism that is quite different from the ones encountered above, and is general enough to have a potentially wide applicability.

Shoham's approach achieves non-monotonicity through the semantics of the language. The general approach, called *semantical non-monotonicity* or *model preference*, places a partial ordering relation, determined by some preference criteria, on all the interpretations that satisfy a theory. From this, one selects the most preferred. Obviously, different preference criteria will select different preferred models. In fact, it is possible within this general framework to give preference criteria that capture, wholly or in part, the non-monotonicity of other formalisms. For example, note how in circumscription we prefer those models that are minimal with respect to a (circumscribed) predicate's denotation. The details of model preference and the logics that Shoham has proposed for solving the temporal projection problem are given in the following Chapter.

Chapter 5

Semantical Non-Monotonicity

The previous chapter gave the motivation for using a non-monotonic formalism to deal with the problem of partial specifications—by showing the connection between incomplete information and non-monotonic reasoning, and how these are involved in the qualification, frame and ramification problems. It further gave motivation for choosing Shoham's non-monotonic approach to serve as the formal basis for the specification language by examining a number of non-monotonic formalism and explaining why the rest were inadequate or otherwise inappropriate given the goal of the thesis. The question now is, how can Shoham's pioneering approach be transformed or incorporated into an appropriate specification language? Before this question can be answered, however, the fundamentals of semantical non-monotonicity and the details of Shoham's logics must be given. There are two components to the approach: the model preference criteria and the logic for representing temporal information. The former has already been already established: chronological minimality. Shoham actually uses a slight variation of this criterion. The latter, however, must match our requirements for the specification language, which includes easy modifiability and the ability to express both on-atomic actions and time. Fortunately, time plays a significant role in Shoham's logics.

The rest of the chapter is arranged as follows: The next section presents the general framework for achieving non-monotonicity through the semantics of a language. The second extensively covers Shoham's notions and logics. The final section points out

some of the shortcomings of the logics.

5.1 Semantical Non-Monotonicity

As the name implies, semantical non-monotonicity, or the preferential model approach to non-monotonicity, is entirely model-theoretic. It makes no specific reference to the object language involved, nor to its inferential mechanism. It is based on the following idea [68]. Recall that the meaning of a theory in classical or modal logic is the set of interpretations that satisfy it, or its set of models. In monotonic logic, A entails B ($A \models B$) if B is true in all models of A, and since all models of $A \wedge C$ are also models of A, then $A \wedge C \models B$. The key here is that all the models are involved. If we restrict the set of models of a theory in any way, *i.e. prefer* some subset of it, then it no longer becomes possible to assure this monotonic property. To wit, B may be true in all preferred models of A, but $A \wedge C$ may no longer have the same preferred models as A, so B cannot be entailed from $A \wedge C$.

The technique is a very general one, permitting the definition of a variety of non-monotonic logics. One can vary both the logic and the preference criteria. The only requirement for the logic is that it should have the usual model theoretic denotational semantics. For the purposes of the discussion, it will be assumed that the logics involved are either classical or modal logics, propositional or first order. As well the term 'interpretation' will refer to either classical interpretations or modal Kripke frames. Let $\mathcal{L}$ be a logic, $\sqsubset$ be a strict partial order on the interpretations of $\mathcal{L}$, and let $\mathcal{M} \sqsubset \mathcal{M}'$ mean that the interpretation $\mathcal{M}'$ is *preferred over* $\mathcal{M}$. A *preference logic* $\mathcal{L}_\sqsubset$ is one made up of a language $\mathcal{L}$ and a preference relation $\sqsubset$. Obviously, the syntax of $\mathcal{L}_\sqsubset$ is the same as that of $\mathcal{L}$, while the semantics are as follows:

Definition 17 An interpretation $\mathcal{M}$ *preferentially satisfies* a formula A, $\mathcal{M} \models_\sqsubset A$, if $\mathcal{M} \models A$ and there is no other interpretation $\mathcal{M}'$, such that $\mathcal{M} \sqsubset \mathcal{M}'$ and $\mathcal{M}' \models A$. As usual, if $\mathcal{M} \models_\sqsubset A$, we say that $\mathcal{M}$ is a *preferred model* of A with respect to $\sqsubset$.

Shoham then goes on to define a variety of other notions, such as preferential validity, but most of them are not relevant to this thesis, except perhaps the notion of preferential entailment:

Definition 18 *A preferentially entails B*, $A \models_{\sqsubset} B$, if for any $\mathcal{M}$, if $\mathcal{M} \models_{\sqsubset} A$ then $\mathcal{M} \models B$ ◇

Let us look at an example that demonstrates how to define other non-monotonic approaches using this framework; in this case, simple circumscription (taken from [68]).

Example 4 Recall that circumscribing a predicate $P(x)$ has the effect of minimizing the instances of x which satisfy P. The following preference criterion captures this notion. $\mathcal{M} \sqsubset \mathcal{M}'$ if

1. $\mathcal{M}$ and $\mathcal{M}'$ agree on the interpretation of function symbols and all predicate symbols other than P,
2. for all x, if $\mathcal{M}' \models P(x)$ then also $\mathcal{M} \models P(x)$, and
3. there exists a y such that $\mathcal{M} \models P(y)$ but $\mathcal{M}' \not\models P(y)$.

In other words, the two interpretations agree on everything except on the instances that satisfy P. The rejected one, $\mathcal{M}$, satisfies an additional instance. □

One can see, however, that it is possible to have more than one (different) preferred interpretation, which was the case with the YSP. To select the right one for the YSP, we must choose the chronologically smaller one. In order to define this, we need a way to express a notion like *later*, hence the predicates or propositions must be associated with a temporal component, such as situations or time. To simplify the definition, all the predicates take a single temporal argument.

Definition 19 Let S be a set of predicates (that we wish to minimize). $\mathcal{M}'$ is *chronologically smaller in S* than $\mathcal{M}$, $\mathcal{M} \sqsubset_S \mathcal{M}'$, if there exists a time t_0 such that

1. for all $p \in S$ and $t \leq t_0$, if $\mathcal{M}' \models p(t)$ then also $\mathcal{M} \models p(t)$, and
2. there exists a $p \in S$ such that $\mathcal{M} \models p(t_0)$ but $\mathcal{M}' \not\models p(t_0)$. ◇

In other words, as we move forward in time, if we encounter a fact p at time t_0 that holds in one interpretation but not in the other, then the latter interpretation is preferred. It is easy to see that if the set S contains the abnormalities, these will delayed as much as possible. The only hitch is the need to define this set. It changes from application to application, and deciding what to put in it is not always clear. Given that this set greatly affects the selection of preferred models, removing the uncertainty surrounding its definition would be very beneficial. In particular, if it were possible to automatically define this set, then perhaps it would lead to a method of general applicability. Shoham proposes one way to achieve this.

5.2 Shoham's Logics

In [68], Shoham attempts to deal with the difficulties of reasoning about time and change, in general, and with the qualification and extended prediction problems, in particular. A preference logic, the *logic of chronological ignorance*, is given as a solution to both problems, although to solve the latter one, Shoham had to also introduce the concept of *potential histories*. Recall, however, that the extended prediction problem (see Sub-section 4.1.2) is a broader notion than the frame problem, and consequently its solution is more than we require. This inadequacy, as well as a few others, is addressed in more detail in the final section of this chapter. Despite these shortcomings, Shoham's solution still forms an integral part of the specification language. Therefore, the description given below is a fairly extensive one; however, to keep it a reasonable length, only the solution to the qualification problem is discussed.

Shoham actually defines four logics: a pair of logics of time intervals (propositional and first-order STL), a logic of temporal knowledge, and the logic of chronological ignorance. The first pair are variants of McDermott's temporal logic [52], the second augments propositional STL with a modal knowledge operator; and, the third is a non-monotonic version of the second. The reasons for employing a knowledge operator are twofold: (1) Shoham believes that there is a strong epistemic connection to non-monotonic reasoning and causation (see also [70]); and (2) it provides the means with which to tackle the problem of deciding what propositions or predicates to minimize

Only the latter two logics will be covered below, as well as the notion of *causal theories*, a class of non-monotonic theories that has a couple of nice properties.

5.2.1 The Logic of Temporal Knowledge

In order to reason about time and change, we need the means with which to represent information of the form fact f holds at situation or time t. State- or change-based approaches were rejected because they usually involve atomic actions and instantaneous effects, and prohibit the description of overlapping actions [71]. Instead, a time-based formalism was chosen, one in which the (inexorable) passage of time is the only fundamental notion of change. Although the underlying temporal ontology does not have to be fixed, it is assumed to be discrete and linear. On top of this structure, the logics allow the representation of intervals, with the intervals defined by time points. All of the logics employ a special construct to associate a primitive proposition with a pair of time points, $\mathtt{TRUE}(t_1, t_2, p)$, meaning that p is true over the interval $\langle t_1, t_2 \rangle$. TRUE is neither a modality, nor a truth predicate; rather, it is a *reifying context* [2]. The implications of this reification will also be discussed in the final section.

The logic of temporal knowledge (TK) is a propositional logic of knowledge of temporal information. Like other logics of knowledge (see, for example, [24, 25, 48]), it is used to describe and reason about what is *known* about the world. In this case, however, the things that are known have an explicit temporal aspect to them.

Syntax

Given

- P: a set of primitive propositions,
- V_t: a set of temporal variables,
- C_t: a set of temporal constants: the set $\{\ldots, -1, 0, 1, 2, \ldots\}$, and
- U_t: $V_t \cup C_t$.

the set of well-formed formulas is defined by the following formation rules

1. if $u_1, u_2 \in U_t$, then $u_1 = u_2$ and $u_1 \preceq u_2$ are wffs;
2. if $u_1, u_2 \in U_t$ and $p \in P$, then $\mathsf{TRUE}(u_1, u_2, p)$ is a wff;
3. if ϕ_1 and ϕ_2 are wffs, then so are $\phi_1 \wedge \phi_2$, $\neg\phi_1$, and $\mathbf{K}\phi_1$;
4. if ϕ is a wff and $v \in V_t$, then so is $\forall v\ \phi$.

Note that the notation used here is slightly different from that in [68]. Instead of the usual knowledge operator **K**, Shoham uses the modal operator $\Box$. $\mathbf{K}\phi$ was chosen to avoid any confusion with $\Box\phi$, which is often used in temporal logic (specification) formalisms to mean "always ϕ."

The other common propositional operators ($\vee$, $\rightarrow$, $\leftrightarrow$, and $\exists$) have their usual definitions. As for the dual to **K**, it is labeled **L**, where $\mathbf{L} = \neg\mathbf{K}\neg$. Futhermore, the following syntactical conventions will be observed: $\mathbf{K}(u_1, u_2, p)$ will be preferred to $\mathbf{K}\mathsf{TRUE}(u_1, u_2, p)$; $\mathbf{K}(u_1, u_2, \neg p)$ preferred to $\mathbf{K}\neg\mathsf{TRUE}(u_1, u_2, p)$; and, $\mathbf{K}(u, p)$ preferred to $\mathbf{K}(u, u, p)$.

The general intuitive meaning for expressions of the form $\mathsf{TRUE}(u_1, u_2, p)$ is that p holds over the interval between u_1 and u_2, where it is assumed, without loss of generality, that $u_1 \leq u_2$. Shoham, however, does not place any restrictions on the nature of p. Whether p is, say, a "property" or an "event" type, depends on how the truth of the proposition over its interval is related to its truth over other intervals. A proposition is considered a property type, for example, if whenever it holds over an interval, it also holds over all its sub-intervals; that is, it exhibits *homogeneity*:

$$\forall x, y \cdot u_1 \leq x \leq y \leq u_2 \wedge \mathsf{TRUE}(u_1, u_2, p) \rightarrow \mathsf{TRUE}(x, y, p)$$

An event, however, is a proposition that does not hold over all its sub intervals. Many more types can likewise be defined. Shoham argues that there are two advantages of leaving the proposition categorization separate from the logic proper: (1) distinctions do not have to be made when they are not needed; and (2) if needed, depending on the application, then they can be tailored appropriately, often achieving distinctions much finer than the typical event or property ones.

An expression of the form $\mathbf{K}(u_1, u_2, p)$ can be read as "it is known that p holds over the interval between u_1 and u_2;" similarly, $\mathbf{L}(u_1, u_2, p)$ can be read as "it is not known that p does not hold between ...." Alternatively, the latter can be considered as stating that p is assumed to hold over the interval—that is, p is *usually* expected to hold and we do not have *explicit* evidence that it does not. As was shown with autoepistemic logic, epistemic operators can be used to express non-monotonic rules:

$$\forall \texttt{t} \cdot \mathbf{K}(\texttt{t,loaded}) \wedge \mathbf{L}(\texttt{t,}\neg\texttt{emptied}) \rightarrow \mathbf{K}(\texttt{t+1,loaded}),$$

which reads "if we know that the (gun) was loaded at time t and we believe that it was not emptied, then we know that it is still loaded at time t+1." Hence, it is an implication from knowledge and ignorance of the past, to knowledge of the future, echoing the general non-monotonic rule. How this rule is captured through the semantics is described below in Sub-section 5.2.2. Compare, as well, this frame axiom to the ones in Sub-section 4.2.5.

Semantics

Modal logics are usually given a Kripke possible-worlds interpretation, and TK is no exception. Moreover, like any other logic of knowledge, to say that a fact is known means that the fact must be true in all the worlds that are considered accessible from the local one. Two things must therefore be identified: a set of worlds and an accessibility relation over the worlds. In TK, the universe is composed of a set of infinite time lines, where each time line is considered a "world," representing one possible course or run of the universe. Furthermore, each world is given the same "copy" of time—in other words, there is an identity of time across worlds. Time is simply represented by the set of integers, $\mathcal{N}$, together with the ordering relation $\leq$. The accessibility relation is determined by what we wish to express with the knowledge operator. There are many possibilities [24, 25, 48], most reflecting agents that have a limited ability to reason about what they know and do not know. For TK, however, Shoham decided upon the S_5 system over the worlds, meaning that the relation is reflexive, transitive, and symmetric. The set of possible worlds, therefore,

form an equivalence class, in which all worlds are considered accessible from each other. Thus, it is not necessary to make explicit use of an accessibility relation.

Let $\mathcal{M} = \langle W, m \rangle$ be a Kripke frame or interpretation, where W is the set of all possible worlds, and m is a meaning function, $m : P \longrightarrow 2^{W \times \mathcal{N} \times \mathcal{N}}$. In essence, each proposition $p \in P$ is associated with a set of 3-tuples $\langle w, t_1, t_2 \rangle$, indicating all the worlds and the intervals on those worlds, over which it holds. Since the only variables are temporal ones, the *variable assignment* function is just $a : V_t \longrightarrow \mathcal{N}$. Finally, the *valuation* function is defined as $V_a^{\mathcal{M}} : T_t \longrightarrow \mathcal{N}$, where if $u \in V_t$ then $V_a^{\mathcal{M}}(u) = a(u)$, and if $u \in C_t$ then $V_a^{\mathcal{M}}(u) = u$.

A Kripke interpretation $\mathcal{M} = \langle W, m \rangle$ and a world $w \in W$ satisfy a formula ϕ under variable assignment a (written $\mathcal{M}, w \models f[a]$) under the following conditions.

- $\mathcal{M}, w \models u_1 = u_2[a]$ iff $V_a^{\mathcal{M}}(u_1) = V_a^{\mathcal{M}}(u_2)$
- $\mathcal{M}, w \models u_1 \leq u_2[a]$ iff $V_a^{\mathcal{M}}(u_1) \leq V_a^{\mathcal{M}}(u_2)$
- $\mathcal{M}, w \models [u_1, u_2]p[a]$ iff $\langle w, V(u_1), V(u_2) \rangle \in M(p)$
- $\mathcal{M}, w \models (\phi_1 \wedge \phi_2)[a]$ iff $\mathcal{M}, w \models \phi_1[a]$ and $\mathcal{M}, w \models \phi_2[a]$
- $\mathcal{M}, w \models \neg\phi[a]$ iff $\mathcal{M}, w \not\models \phi[a]$
- $\mathcal{M}, w \models (\forall v\ \phi)[a]$ iff $\mathcal{M}, w \models \phi[a']$ for all a' that differ from a at most on v
- $\mathcal{M}, w \models \mathbf{K}\phi[a]$ iff $\mathcal{M}, w' \models \phi[a]$ for all $w' \in W$

Note that the notation used here is different from that in [68]; instead, it follows the format presented in Chapter 2.

A Kripke interpretation and world are a *model* for a formula ϕ (written $\mathcal{M}, w \models \phi$) if $\mathcal{M}, w \models \phi[a]$ for some a. Also, a wff is *satisfiable* if it has a model, and *valid* if it is true for all models. Furthermore, all the usual S_5 axioms and inference rules apply. As well, the fixed representation of time across worlds permits the inclusion of the *Barcan formula*: $\mathbf{K}\forall v\ \phi \equiv \forall v\ \mathbf{K}\phi$. (See also [31]).

There is no proof theory for the language, as the consequences of a set of sentences in the language will be carried out entirely through the model theory.

5.2.2 Chronological Ignorance

The logic TK is monotonic; it is made non-monotonic by imposing a preference ordering on its models. The question that must now be answered is: What preference criteria will allow us to deal with the temporal projection problem? As well, how can we capture the notion that "all the relevant facts have already been mentioned explicitly?" Chronological minimization has already been postulated as the appropriate candidate. However, there still remains the question of what facts should be minimized. Referring to the loaded gun example again, in the logic of time intervals (propositional STL), the rule about firing the gun has the following form:

$$
\begin{aligned}
&\forall \mathtt{t} \cdot \mathtt{TRUE(t,loaded)} \wedge \mathtt{TRUE(t,shoot)} \\
&\quad \wedge\ \mathtt{TRUE(t,\neg broken_trigger)} \wedge \mathtt{TRUE(t,\neg vacuum)} \wedge \Phi \\
&\qquad \rightarrow \mathtt{TRUE(t+1,noise)}.
\end{aligned}
$$

where Φ is a conjunction of more things that could prevent the gun from firing. In order to predict that the gun will fire, we should therefore minimize all such facts. Specifically, in Definition 19, we define the set $S = \{\mathtt{broken_trigger},\ \mathtt{vacuum}, \ldots\}$. By going this route, however, we slipped into a circumscriptive mode of thinking. Deciding what to minimize is application dependent and requires having an idea before-hand of the types of predictions we would like to make. Besides, it appears that the sort of things that are being minimized are just those types that we wish to avoid considering altogether; they are not the "relevant" ones.

To avoid application dependence, the minimized set should be an easily distinguishable, general class of propositions. Unfortunately, it is not easy to identify such a class in the (non-modal) logics of time intervals. We cannot even minimize all the propositions at once because of the law of the excluded middle—minimizing p would hence have the effect of maximizing $\neg p$. TK, however, being a modal logic of knowledge, is capable of distinguishing between propositions that are known (*i.e.* true in all possible worlds) and those that just happen to be true. And known propositions are not subject to the law of the excluded middle: for any formula ϕ, it is *not* the case that either ϕ or $\neg\phi$ is known. It turns out that by minimizing knowledge (or

alternatively, maximizing ignorance), we end up with a logic, the logic of chronological ignorance (CI), that has the necessary properties. Intuitively, the facts that we would like to retain are just those known facts — the "relevant" ones — that are explicitly mentioned in the theories. These facts, by definition, should appear in all models. Hence, all models that have more knowledge than this minimum indicate that additional (unwanted) knowledge has somehow creeped into the theory. Such models must be rejected.

To define chronological ignorance it will be necessary to identify the *latest time point* of each sentence of a theory.

Definition 20 A *base formula* is one that does not contain the modal knowledge operator. The *latest time point (ltp)* of a base formula is the (chronologically) latest time point mentioned in it. It is determined by the following rules:

1. The *ltp* of TRUE(t_1, t_2, p) is t_2, assuming $t_1 \leq t_2$.
2. The *ltp* of $\phi_1 \wedge \phi_2$ is the latest between the *ltp* of ϕ_1 or the *ltp* of ϕ_2.
3. The *ltp* of $\neg\phi$ is the *ltp* of ϕ.
4. The *ltp* of $\forall v\ \phi$ is the earliest among the *ltp*'s of all the ϕ' that result by substituting constant values for all occurrences of v in ϕ. If there is no such point, then the *ltp* is $-\infty$.

The last rule may seem counter-intuitive. However, keep in mind that the procedure for uncovering the facts that hold over time starts at the earliest point and moves forward in time. Thus, in a universally quantified formula, the earliest *ltp* of all possible ϕ' entailed from it must be identified.

The *logic of chronological ignorance*, $TK_{\sqsubset_{ci}}$, is a preference logic based on the logic TK with the following preference criterion placed on its models.

Definition 21 A Kripke model M' is *chronologically more ignorant* than another model M (written $M \sqsubset_{ci} M'$) if there exists a time t_0 such that

1. for any base sentence ϕ whose $ltp \leq t_0$, if $M' \models \mathbf{K}\phi$ then also $M \models \mathbf{K}\phi$, and

2. there exists some base sentence ϕ whose ltp is t_0 such that $M \models \mathbf{K}\phi$ but $M' \not\models \mathbf{K}\phi$ $\diamond$

Definition 22 A model M is said to be a *chronologically maximally ignorant (cmi)* model of ϕ, $M \models_{\sqsubset_{ci}} \phi$, if $M \models \phi$ and there is no other M' such that $M' \models \phi$ and $M \sqsubset_{ci} M'$. $\diamond$

In other words, to determine which models are preferred, all the models are compared by checking all their known propositions. The comparison begins with the earliest known proposition and moves forward in time. If, at any point in time, some model has a known proposition that the others do not, then it is considered "less ignorant" and consequently rejected. This weeding out process continues until there are no more known propositions left to be checked. At that point, the remaining ones are *cmi* ones.

5.2.3 Causal Theories

Unfortunately, as is the case with other non-monotonic logics, determining the consequences of general CI theories is an intractable problem. Theories may have more than one class of *cmi* models, or even none. To circumvent this problem, the class of theories must be restricted in some way. Ideally, we would like a theory to have only one class of *cmi* models, all of which have exactly the same set of known base sentences. Shoham in [68] identifies two such classes, and proves that they have this property. The class of causal theories, defined below, is sufficient for the qualification problem; the class of inertial theories (which will not be described here), for both the qualification and extended prediction problems.

Definition 23 A *causal theory* Ψ is a theory in CI, in which all sentences have the form

$$\Phi \wedge \Theta \rightarrow \mathbf{K}\phi$$

where

1. ϕ is an atomic base sentence $\mathtt{TRUE}(t_1, t_2, p)$, where p can be positive or negative;

2. Φ is a conjunction of sentences $\mathbf{K}\phi_i$, where ϕ_i is a (positive or negative) atomic base sentence with an *ltp* of t_i such that $t_i < t_1$;

3. Θ is a conjunction of sentences $\mathbf{L}\phi_j$, where ϕ_j is a (positive or negative) atomic base sentence with an *ltp* of t_j such that $t_j < t_1$;

4. Either Φ or Θ may be empty. If Φ is empty, the sentence is a *boundary condition*, otherwise, the sentence is a *causal rule*.

5. There exists a time point t_0, such that for all boundary conditions $\Theta_i \to \mathbf{K}\phi_i$ in Ψ, where the *ltp* of ϕ_i is t_i, then $t_0 \leq t_i$;

6. There do not exist two sentences in Ψ such that one contains $\mathbf{L}(t_1, t_2, p)$ on its left hand side (l.h.s.) and the other includes $\mathbf{L}(t_1, t_2, \neg p)$ on its l.h.s., for all p, t_1, and t_2;

7. If $\Phi_1 \wedge \Theta_1 \to \mathbf{K}(t_1, t_2, p)$ and $\Phi_2 \wedge \Theta_2 \to \mathbf{K}(t_1, t_2, \neg p)$ are two sentences in Ψ, then $\Phi_1 \wedge \Theta_1 \wedge \Phi_2 \wedge \Theta_2$ is inconsistent

Boundary conditions are simple enough, they are used to introduce known facts. Causal rules, on the other hand, require some explanation. They can be considered as representing causation [68, 70]. For example, in a statement (s_0) such as

$$\mathbf{K}\phi_1 \wedge \mathbf{L}\phi_2 \to \mathbf{K}\phi_3$$

ϕ_1 is the cause, ϕ_2 the enabler, and ϕ_3 the effect. The cause is responsible for producing the effect, but it could not happen if the enabler turned out to be false. For example, pulling the trigger of a loaded gun causes it to fire, but only if the gun is in proper working order. Since the enabler is usually assumed to hold, it is sufficient to not know that it is false. The cause, on the other hand, must be known to have occurred in order to predict the effect. From this discussion, it should be clear that a causal rule cannot be evaluated by itself: it depends on the theory in which it is embedded. To illustrate this point, let T_1 be a theory consisting of s_0 and the boundary condition $\mathbf{K}\phi_1$. With this pair of statements, one can infer $\mathbf{K}\phi_3$. But, if we also add $\mathbf{K}\neg\phi_2$ to T_1, resulting in theory T_2, we can no longer infer $\mathbf{K}\phi_3$, hence revealing the

non monotonic nature of the logic as well. Moreover, the strict timing constraints placed on causes and effects, insures that the former precede the latter.

The fifth rule establishes a "starting point" from which to begin computing the known facts. The sixth is needed because conditions $\mathbf{L}\phi$ assign, by default, truth values to the theory ($\mathbf{L}\phi \rightarrow \phi$); consequently, if both ϕ and $\neg\phi$ were thus assigned, the theory would have multiple models. The seventh reflects the idea that consistent causes should not produce inconsistent effects. This is especially critical in concurrent systems, since it prevents two different actions, while executing simultaneously, from producing conflicting results.

Shoham proves that if Ψ is a causal theory, then it is consistent, has a *cmi* model, and that all *cmi* models of the theory have exactly the same known facts. Furthermore, an algorithm is given in [68] that computes all the known facts in a *cmi* model of a theory. Since these results can only be achieved if the formulae in the theory are sentences, universally quantified formulae or schemas must first be instantiated. The algorithm basically starts at the earliest mentioned point in the theory and moves forward in time, adding at time t_i all the new known facts in the theory: any boundary condition with an *ltp* of t_i, and any consequent of a causal rule, whose consequent has an *ltp* of t_i and whose $\mathbf{K}$ antecedents (with *ltp*s of $t < t_i$) are known while the negation of all its $\mathbf{L}$ antecedents are not known. This is repeated, simulation like, until all the time points mentioned in the theory have been considered.

Example 5 *The Yale Shooting Problem, (Revisited)* The shooting scenario may be represented by the following causal theory (where point based facts are used):

1. $\mathbf{K}$(1,loaded)

2. $\mathbf{K}$(5,shoot)

3. $\forall$t $\cdot$ $\mathbf{K}$(t,loaded) $\wedge$ $\mathbf{L}$(t,$\neg$shoot) $\wedge$ $\mathbf{L}$(t,$\neg$emptied)
   $\rightarrow$ $\mathbf{K}$(t+1,loaded)

4. $\forall$t $\cdot$ $\mathbf{K}$(t,loaded) $\wedge$ $\mathbf{K}$(t,shoot)
   $\wedge$ $\mathbf{L}$(t,air) $\wedge$ $\mathbf{L}$(t,$\neg$broken_trigger) $\wedge$ Φ_{others}
   $\rightarrow$ $\mathbf{K}$(t+1,noise)

The first two axioms are boundary conditions indicating that it is known that at time 1 the gun is loaded and at time 5 the trigger is pulled. The third axiom, a causal rule, describes the persistence of the loaded-gun condition. The fourth, also a causal rule, captures the expected behaviour of shooting a loaded gun. Applying the preference criterion, it is easy to derive that all *cm* models have exactly the same known facts: `TRUE(1,loaded)`, `TRUE(2,loaded)`, ..., `TRUE(5,loaded)`, `TRUE(5,shoot)`, and `TRUE(6,noise)`. □

Finally, one syntactic variation of causal rules will prove useful for the specification language. Causal theories are fairly modular, meaning that one can add new rules to a theory without having to modify existing ones. This was partly demonstrated above in the example involving theories T_1 and T_2. They can be further modularized by employing the technique (see Sub-section 1.1.2) of specifying a behaviour by stating its main rule together with a list of exceptions to the rule. Under this technique, axiom 4 above would be written as follows:

4a. $\mathbf{K}$(t,loaded) $\wedge$ $\mathbf{K}$(t,shoot) $\wedge$ $\mathbf{L}$(t,normcond17)
  $\rightarrow$ $\mathbf{K}$(t+1,noise)

4b. $\mathbf{K}$(t,¬air) $\rightarrow$ $\mathbf{K}$(t,¬normcond17)

4c. $\mathbf{K}$(t,broken_trigger) $\rightarrow$ $\mathbf{K}$(t,¬normcond17)

  ⋮

Then, whenever a new exception is found, instead of modifying axiom 4 (by expanding the Φ_{others} conjunct), a new exception axiom is added to the theory. Note, however, that the exception conditions refer to the same time point on both sides of the implication, hence they are violations of the causes-strictly-precede-effects constraint. This circular implication affects the task of computing the known facts, but the set of rule-exception axioms are equivalent to the original formulation. For computational purposes, however, the procedure should (at each time point) postpone evaluating the knowledge of abnormality conditions until the other `t`-sentences have been determined. Theories that contain such statements, called *normal*, still have all the properties of "pure" causal theories [68, 70].

5.3 Shortcomings of Shoham's Logics

There is much to recommend the logic of chronological ignorance for the formal basis of the specification language:

- it uses time and temporal intervals;
- it is non-monotonic, employs a chronological minimization criterion that does not require us to decide what to minimize, and handles the qualification problem;
- a class of *CI* theories has been identified that possesses a number of promising attributes: the theories consist of sentences that are suitable for a specification, they are modifiable, and their consequences can be algorithmically determined.

Nevertheless, the logic falls short in other respects. Keep in mind that it was designed to be used for temporal reasoning in AI, not for the specification of systems. Its primary inadequacies are the following: it is a propositional logic, it employs reification, it does not distinguish actions from other entities, and requires special constructs to deal with the extended-prediction problem.

propositional *TK* and *CI*, though interesting, are propositional logics and thus not powerful enough for most applications. Shoham in [68] does not give first-order versions of these logics, although a first-order version of *STL* is provided. In specifications, we will need the ability to talk about the properties and relationships of classes of objects and their members. Therefore, these must be extended to first-order cases. One potential complication from moving to a first-order case arises from quantifying into the scope of a modality [18, 56].

reification To reify something is to concretize it. Reified logics, such as Allen's [1], treat (propositional or first-order) formulae as terms inside special predicates like `Holds`. This is highly problematic since it requires one to accord the formulae ontological status in the semantics in other words, they must be denoted to objects[2, 18, 28]. In Allen's case, almost any expression that would normally

be considered a first-order formula can appear as a term. An expression like Holds(ı,(($\phi(x)$ $\wedge$ $\psi(x)$) $\rightarrow$ $\chi(x)$))[1], for example, implies that ϕ, ψ and χ are no longer predicates, rather they are now functions that map objects onto "fact" terms. Similarly, the logical connectives become functions, making Holds a predicate that takes a time interval ı and a "fact" as arguments. To what objects should we denote ϕ and $\wedge$? And if they are just functions, should we be able to nest them, as in $\phi(\phi(x))$? If on(a,b) asserts that object a is on object b, what does on(on(a,b),c) mean? The only advantage of reification is the ability to quantify over these "predicate" terms, cheaply attaining some of the power of second-order logics, but this must be done with great care. The meaning of the terms and the notion of scope must be precisely defined. Allen, in fact, does not even give a semantics to the language. Nor is it clear how this should be accomplished.

Shoham, aware of these difficulties, chose to restrict what can appear inside a TRUE construct. For the propositional case of *STL*, only a single propositional constant is allowed; for the first-order version, only a single relation (TRUE(t_1,t_2,R(x_1,x_2, ...))). Moreover, nesting of TRUEs is forbidden. While this solves the problem, allowing Shoham to provide a clear semantics for all the logics, the restrictions have actually made the reification unnecessary [2, 18]. Not only does reification, in this case, not present a clear advantage, it may perhaps hinder further extensions of the logics. Therefore, *TK* (and concomitantly *CT*) must be unreified.

What form should the language take? Bacchus *et al.* [2] define a non-reified, standard two-sorted temporal logic *BTK* and show that it subsumes Shoham's first-order *STL*. They objected to the non-standard syntax and semantics of *STL*, and the resulting inapplicability of standard proof theory which is not a concern here. Their logic, however, is too general, permitting the predicates and functions in the logic to take any number of temporal arguments. Hence, it cannot easily be reconciled into the definitions of *CT* and causal theories. Galton

[1] Allen's notation is slightly different from this. Instead of logical symbols like $\wedge$, Allen uses **and**, **not**, and so on.

[18] proposes a very general method for unreifying logics, including complex ones like Allen's. The technique is based on the idea of distinguishing the types of objects from their instances (*tokens*). While reifying types is problematic and "philosophically dubious," reifying tokens is ontologically sound since they are individuals. The technique has merit, but it is perhaps more appropriate for more heavily reified logics. Haugh [28] offers the method of temporal arguments (MTA) as an alternative to reified logics. Like *BTK*, this approach attaches a temporal component to each predicate, but unlike it, MTA uses a non-standard semantics to highlight the temporal aspects of the logic. In some ways, therefore, it is closest in spirit to *STL*, but from an unreified perspective. Nevertheless, it is deficient in a significant way: no atemporal functions are permitted in the predicates.

The most reasonable approach for the unreification, therefore, is a combination of MTA and *BTK*, suitably adapted to match the requirements for *CI* and causal theories, of course.

actions Shoham, as explained above, chose to distinguish very few objects in *TK*. Although this resulted in a logic that is both elegant and parsimonious, it is neither conducive to, nor practical for, the task of specification. It was given very little "syntactic sugar." If *TK* (or *CI*) is used "as is" for specification, virtually everything would have to be defined from scratch, even the most basic concepts, including what constitutes an action. And this would have to be done for each new document. Such documents would not only be difficult and tedious to write, but difficult to read as well. We need to enhance the naturalness of expression of the logic without loosing too much of its expressive power.

Actions must be distinguished, not just because this is one of the requirements for the specification language, but also because of the significant role they play in any theory of causation. Indeed, one of Galton's criticisms of Shoham's account of causation is that it fails to make a distinction between actions and *fluents* (value-bearing facts) [19]. Although Shoham disputes this need, most formalisms that deal with change or causation rely on the intuition that all

changes in the values of fluents are caused by actions (*e.g.* [1, 41, 52, 59]). Even Shoham concedes in [69] that actions are needed, if only to capture the notion of agents influencing the course of history through the performance of actions. Besides, lumping them together as Shoham does hides their fundamental differences: actions are active elements, fluents passive; actions take a circumscribed time, fluents persist indefinitely; actions are defined by the whole interval of time over which they occur, fluents by any sub-interval as well.

One way to emphasize actions in a formalism is to make them into modalities see dynamic logics, for example, or Khosla and Maibaum's specification language [37]. This is not recommended here because of potential conflicts with the knowledge modality in Shoham's logics. A more common method for distinguishing actions (or any other entity) in a logic is through sorting. That is, by making it a *sort* or type in a many-sorted version of the logic. Such a logic sub-divides the universe of discourse into different kinds, and requires all the elements in the language to be of a certain type. An example of one is the situation calculus. Since one of the benefits of sorted logics is improved naturalness of expression, they are often used for specification languages (for example, [22, 37]). The restricted domains and strict typing also make it easier to check the specification for errors.

One final concern is how to distinguish between the type of an action and an instance of it. The latter entity will be called an *event*. Using events, rather than actions, allows us to specify that events of the same type *can occur concurrently*, and provides the means for the counting of events of a particular type [28]. Galton [18] recommends adding event token arguments to the predicates. For example, instead of `load_gun(x)`, the action of loading gun `x`, use `load_gun(x,e)`, where `e` refers to a specific gun loading. An alternative is to employ occurrence predicates that instantiate an action by associating it with its time of occurrence, `Occurs(t,load_gun(x))`. In order to avoid confusion however, the entities in the predicates must be properly identified as actions, hence this form is well-suited for a many-sorted logic. Thus, the latter approach will be used in the specification language.

the frame problem Shoham provides a solution for the extended prediction problem. Recall that the problem involves not just the persistence of facts but an *upper bound* on the expected time of persistence as well. To solve the problem, Shoham had to resort to the concept of a potential history, defined by formulae of the form:

$$\exists \mathtt{v} \cdot (\mathtt{t}_1 \leq \mathtt{v} \leq \mathtt{t}_2 \wedge \mathbf{K}(\mathtt{t}_1, \mathtt{v}, \mathtt{p\text{-}i})).$$

This formula asserts that the potential history $\mathtt{p\text{-}i}$ should last from $\mathtt{t}_1$ to $\mathtt{t}_2$, providing it is not clipped at some point $\mathtt{v}$. If it is clipped, then the potential history would have manifested only part of its history. The potential history can be viewed as the "mechanism" that causes or maintains the persistence of a fact. To extract or project the fact from it, a formula of the following form must be used:

$$(\exists \mathtt{v} \cdot (\mathtt{t}_1 \leq \mathtt{t}_2 \leq \mathtt{t}_3 \leq \mathtt{v}) \wedge \mathbf{K}(\mathtt{t}_1, \mathtt{v}, \mathtt{p\text{-}i}))) \rightarrow \mathbf{K}(\mathtt{t}_2, \mathtt{t}_3, \mathtt{p}).$$

Basically, this means that the fact $\mathtt{p}$ can be inferred to hold over any sub-interval of the manifested potential history $\mathtt{p\text{-}i}$.

Causal theories are extended to *inertial theories* by including sentences of these forms. Shoham is able to show that inertial theories enjoy the same properties of causal theories [68]. However, the solution is rather complex and cumbersome, making it difficult to express simple persistences, which is all that we really need. One must define and carefully instantiate not just facts, but potential histories as well.

The best solution would be one that stayed within the context of causal theories. Unfortunately, it is not easy to express a "global" frame axiom in *TK*. That can only be accomplished in a circumscription logic by means of a second-order formula. Thus, it will be necessary to employ a number of persistence axioms. The details are covered in Chapter 6.

In addition to these points, there is the matter of what form the specifications will take and what additional "syntactic sugar" will be required. The pragmatics must also be considered. These issues, among others, are answered in the next chapter.

Chapter 6

The Specification Language KAT

The development of the ideas presented in the previous three chapters culminates here in the definition of the specification language. A suitable formal basis for it was selected, the logic CI, but before it can be used as a specification language, its shortcomings must be addressed. From the discussion in Section 5.3, TK must first be transformed into a non-reified, many-sorted first-order logic ($kFOTK$). The transformation, however, must respect the semantics of first-order STL and TK; otherwise, adapting the notions of chronological ignorance and causal theories to $kFOTK$ becomes problematic.

With the formal basis finally established, the remaining shortcomings and sundry other pragmatic concerns must be tackled. These include, among other things, a solution to the frame problem and a suitable form for the specifications. The result is the KAT (knowledge of action and time) specification language. Specifications written in this language form a subset class of causal theories, hence enjoying their nice properties.

This chapter covers both formalisms, $kFOTK$ in the first section and KAT in the second. Since many of the notions have already been covered in the previous chapters and much of the formal machinery defined, especially Shoham's logics, the presentation in some sections below is quite terse. The chapter ends with an extended example, a plain old telephone system (POTS), demonstrating the use of KAT.

6.1 kFOTK: a Logic of Temporal Knowledge

In the spirit of *TK*, *kFOTK*, a k-sorted first-order logic of temporal knowledge, discerns only one sort: time. The temporal connection is further highlighted by insisting that all predicates take a couple of temporal arguments.

Before proceeding with the description of *kFOTK*, however, some aspects of first-order *STL* must be presented. Recall that *TK* was based on propositional *STL*; likewise, *kFOTK* must capture the fundamental nature of first-order *STL*. The first-order version of *STL* extends the propositional one with atemporal constants, variables and functions, quantifiers over the atemporal variables, and relations. The main syntactic difference is allowing relations within the TRUE constructs: if t_1 and t_2 are temporal terms, $a_1,\ldots,a_m$ atemporal ones, and R is an m-ary relation, then

$$\mathrm{TRUE}(t_1, t_2, R(a_1, \ldots, a_m))$$

is an atomic formula. Everything inside the TRUE construct, as in Shoham's other logics, is evaluated over the pair of time points. With the propositional logics this posed no difficulties since only propositional constants were permitted within them; with first-order *STL*, however, one must take into account the atemporal functions that appear in them as well. In an expression of the form $\mathrm{TRUE}(t_1,t_2,P(f(x),y))$, for example, the evaluation of $f(x)$ is also dependent of the pair of points t_1 and t_2. Thus, although the time points do not appear as direct arguments to P and f, the semantics must assure that they are indeed interpreted over the points.

We begin, as usual, with the syntax of *kFOTK*.

6.1.1 Syntax

Let k represent the number of distinct *basic types* or *sorts*, including time, and $\mathcal{S} = \{s_1, s_2, \ldots, s_{k-1}, s_t\}$ be the set of sorts, where s_t represents time. Sorts are used to provide a classification for variables, which in turn provides a classification for the predicate and function symbols.

The objects of the language are the following:

- V_t: a set of temporal variables, whose elements are of sort $s_t \in \mathcal{S}$ and are denoted by $t_1, t_2, \ldots$.

- V_i: $k-1$ sets of atemporal variables (which are disjoint from V_t and each other) Elements of V_i, $1 \leq i \leq k-1$, are of sort $s_i \in \mathcal{S}$ and are denoted by $v_{i1}, v_{i2}, \ldots$.

- **P**: a set of countable predicate symbols, where each $P \in \mathbf{P}$ has a fixed arity $m+2$ ($m \geq 0$) and is assigned a type $\langle s_t, s_t, s_{i1}, \ldots, s_{im} \rangle$, such that $s_t, s_{i1}, \ldots, s_{im} \in \mathcal{S}$. Since all predicates must have at least a pair of temporal arguments, proposition constants are simply predicates of arity 2.

- $\mathbf{F_t}$: a set of countable temporal function symbols, where each $f_t \in \mathbf{F_t}$ has a fixed arity $m \geq 0$, is assigned a type $\langle s_t, s_t^1, \ldots, s_t^m \rangle$, such that $s_t, s_t^i \in \mathcal{S}$, and returns values of type s_t. These are typically the various arithmetic operators. Temporal functions of single arity represent temporal constants.

- $\mathbf{F_v}$: a set of countable atemporal function symbols, where each $f_v \in \mathbf{F_v}$ has a fixed arity $m \geq 0$, is assigned a type $\langle s_i, s_{i1}, \ldots, s_{im} \rangle$, such that $s_i, s_{i1}, \ldots, s_{im} \in \mathcal{S}$, and returns values of type s_i. Atemporal functions of single arity represent atemporal constants.

The terms of the language are defined as follows:

- **temporal terms:** all members of V_t and all temporal constants are temporal terms; if $f_t \in \mathbf{F_t}$ is an m-ary function of type $\langle s_t, s_t^1, \ldots, s_t^m \rangle$ and $\alpha_1, \ldots, \alpha_m$ are temporal terms corresponding to (temporal) sort $s_t^i \in \mathcal{S}$, then $f_t(\alpha_1, \ldots, \alpha_m)$ is also a temporal term.

- **atemporal terms:** all members of V_i and all atemporal constants are atemporal terms; if $f_v \in \mathbf{F_v}$ is an m-ary function of type $\langle s_i, s_{i1}, \ldots, s_{im} \rangle$, and $\alpha_1, \ldots, \alpha_m$ are atemporal terms corresponding to sorts $s_i, s_{i1}, \ldots, s_{im} \in \mathcal{S}$, then $f_v(\alpha_1, \ldots, \alpha_m)$ is also an atemporal term.

Definition 24 The set of well-formed formulae for $kFOTK$ is defined by the following rules:

- **atomic formulae:**

  1. if α_1 and α_2 are temporal terms, then $\alpha_1 = \alpha_2$ and $\alpha_1 \leq \alpha_2$ are atomic formulas.

  2. if t_1 and t_2 are temporal terms, $P \in \mathbf{P}$ is an $m+2$-ary predicate, and $\alpha_1, \ldots, \alpha_m$ are atemporal terms, then $[t_1, t_2]P(\alpha_1, \ldots, \alpha_m)$ is a wff.

- **well formed formulae:**

  1. an atomic formula a wff.

  2. if ϕ_1 and ϕ_2 are wffs, then so are $\phi_1 \rightarrow \phi_2$, $\neg\phi_1$, and $\mathbf{K}\phi_1$.

  3. if x is a variable of sort s, and ϕ is a wff, then so is $\forall x : s\ \phi$. ◇

The usual intuitive meanings apply here, as do the usual definitions for the other logical connectives. A note about the somewhat unorthodox notation, $[t_1, t_2]p(x)$: the notation was selected to emphasize the point that every predicate must be associated with a time interval, but it should be remembered that this is merely a syntactic variation of $p(t_1, t_2, x)$. Furthermore, it must not be mistaken as a modality nor as a reified construct; in particular, not $\mathtt{TRUE}(t_1, t_2, p(x))$, where $p(x)$ is considered a relation, not a predicate, in first-order *STL*. Finally, the following pair of syntactic conventions will be observed: $\mathbf{K}[t_1, t_2]\neg P(x)$ is preferred to $\mathbf{K}\neg[t_1, t_2]P(x)$, and $\mathbf{K}[t]P(x)$ to $\mathbf{K}[t, t]P(x)$.

6.1.2 Semantics

As with the propositional case, the first-order case is given a Kripke possible-worlds semantics. The worlds are once again the infinite time lines and the accessibility relation is an equivalence relation between all time lines, making this an S_5 modal system. The additional types, however, complicate the situation.

An interpretation or frame is a tuple $\mathcal{M} = \langle D_1, \ldots, D_{k-1}, D_t, W, m\rangle$, where the D_i (including D_t, the set of time points) are the *subdomains* of the interpretation, W is the set of all possible worlds, and m is a meaning function. The subdomains are non-empty sets of values corresponding to the basic sorts $\{s_1, \ldots, s_{k-1}, s_t\}$, and their

union represents the domain of the interpretation. The time domain D_t will again be represented by $\mathcal{N}$ with $\geq$. The function m assigns

- to each pair $\langle P, W \rangle$, where P is of type $\langle s_t, s_t, s_{i1}, \ldots, s_{im} \rangle$, $m \geq 0$, a relation on $\mathcal{N} \times \mathcal{N} \times D_{i1} \times \ldots \times D_{im}$;

- to each temporal function symbol f_t of type $\langle s_t, s_t^1, \ldots, s_t^m \rangle$, $m \geq 0$, a function from $\mathcal{N}^1 \times \mathcal{N}^2 \ldots \times \mathcal{N}^m$ into $\mathcal{N}$; and

- to each triple $\langle \mathcal{N}, \mathcal{N}, f_v \rangle$, where f_v is of type $\langle s_i, s_{i1}, \ldots, s_{im} \rangle$, $m \geq 0$, a function from $D_{i1} \times D_{12} \ldots \times D_{im}$ into D_i

The last item shows that each function is associated with a pair of time points, even though they do not directly take temporal arguments.

The *variable assignment* function a now must handle not just the temporal variables, but the atemporal ones as well. Thus, if $v \in V_t$ then $a : V_t \longrightarrow \mathcal{N}$, and if $v \in V_i$, then $a : V_i \longrightarrow D_i$, for all i.

Since any terms that appear in a predicate (aside from the temporal ones) are dependent on the time interval, the valuation of arbitrary atemporal terms must take this into account. The *valuation* $V_a^{\mathcal{M}}$ of an arbitrary term α of sort s is defined as follows:

- **temporal terms**

  1. $V_a^{\mathcal{M}}(v) = a(v)$, if v is a variable;
  2. $V_a^{\mathcal{M}}(c) = a(c)$, if c is a constant; and
  3. $V_a^{\mathcal{M}}(f(\alpha_1, \ldots, \alpha_m)) = m(f)(V_a^{\mathcal{M}}(\alpha_1), \ldots, V_a^{\mathcal{M}}(\alpha_m))$, if $f \in \mathbf{F_t}$.

- **atemporal terms**

  1. $V_a^{\mathcal{M}}(t_1, t_2, v) = a(v)$, if v is a variable, for all $t_1, t_2 \in \mathcal{N}$;
  2. $V_a^{\mathcal{M}}(t_1, t_2, c) = a(c)$, if c is a constant, for all $t_1, t_2 \in \mathcal{N}$; and
  3. $V_a^{\mathcal{M}}(t_1, t_2, f(\alpha_1, \ldots, \alpha_m)) = m(t_1, t_2, f)(V_a^{\mathcal{M}}(\alpha_1), \ldots, V_a^{\mathcal{M}}(\alpha_m))$, if $f \in \mathbf{F_v}$, for all $t_1, t_2 \in \mathcal{N}$.

Definition 25 A Kripke frame $\mathcal{M} = \langle D_1, \ldots, D_{k-1}, D_t, W, M\rangle$ and a world $w \in W$ satisfy a formula ϕ under variable assignment a (written $\mathcal{M}, w \models \phi[a]$) under the following conditions. In the definition below, a_1 and a_2 represent arbitrary terms, while t_1 and t_2 represent only temporal terms.

- $\mathcal{M}, w \models t_1 = t_2[a]$ iff $V_a^{\mathcal{M}}(t_1) = V_a^{\mathcal{M}}(t_2)$
- $\mathcal{M}, w \models t_1 \leq t_2[a]$ iff $V_a^{\mathcal{M}}(t_1) \leq V_a^{\mathcal{M}}(t_2)$
- $\mathcal{M}, w \models [t_1, t_2]p(a_1, \ldots, a_m)[a]$ iff $\langle V_a^{\mathcal{M}}(t_1), V_a^{\mathcal{M}}(t_2), V_a^{\mathcal{M}}(a_1), \ldots, V_a^{\mathcal{M}}(a_m)\rangle$ $\in m(p, w)$
- $\mathcal{M}, w \models (\phi_1 \rightarrow \phi_2)[a]$ iff either $\mathcal{M}, w \not\models \phi_1[a]$ or $\mathcal{M}, w \models \phi_2[a]$
- $\mathcal{M}, w \models \neg\phi[a]$ iff $\mathcal{M}, w \not\models \phi[a]$
- $\mathcal{M}, w \models (\forall t\ \phi)[a]$ iff $\mathcal{M}, w \models \phi[a']$ for all a' that differ from a at most on t
- $\mathcal{M}, w \models (\forall x : s\ \phi)[a]$ iff $\mathcal{M}, w \models \phi[a']$ for all a' that differ from a at most on x, where x is of sort s
- $\mathcal{M}, w \models \mathbf{K}\phi[a]$ iff $\mathcal{M}, w' \models \phi[a]$ for all $w' \in W$ ◇

The usual definitions for model and satisfaction apply here as well. The S_5 structure and the identity of the domains across the worlds (same in every world), makes the Barcan formula valid for all domains: $\forall x : s\mathbf{K}\phi \equiv \mathbf{K}\forall x : s\phi$. Thus, the problem of quantifying into the scope of a modality is not an issue here (see [56] for an examination of the problem).

Comparison to First-Order STL

Although a direct comparison is not possible since *kFOTK* is a modal logic and *STL* is not, an argument can be made that *kFOTK* is a modal extension of it. In this respect, it matches the relationship between *TK* and propositional *STL*; hence, *kFOTK* may be viewed as a first-order non-reified extension of *TK*. The relationships between the various logics mentioned in this thesis may perhaps be a bit confusing. A graph of the

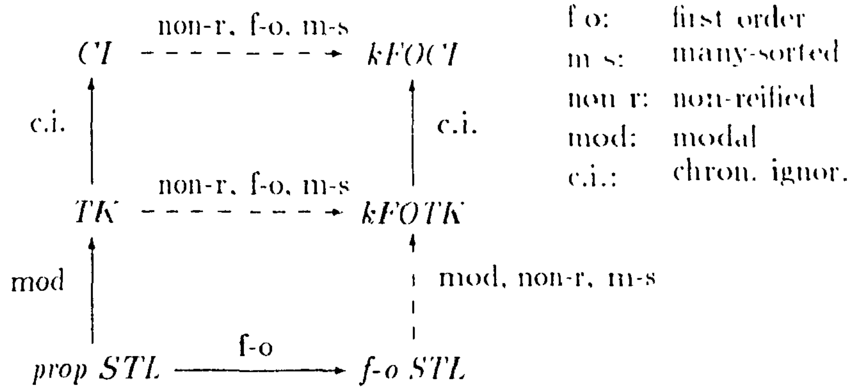

Figure 6.1: Relationships Between Logics

taxonomy may help: see Figure 6.1, where solid lines represent direct connections, and dashed lines inferred ones.

The syntactic transformation is relatively straightforward. Any base (non-modal) formulae in *kFOTK* can be translated into a first-order *STL* one: for example, from `[t₁,t₂]P(f(x),y)` to `TRUE(t₁,t₂,P(f(x),y))`. The sortal aspects can be handled through a rather cumbersome relativization: from `∀x:s· [t₁,t₂]P(x)` to `∀x· TRUE(t₁,t₂,Sort_s(x)) → TRUE(t₁,t₂,P(x))`, where `Sort_s(x)` asserts that `x` is of sort `s`. There are no other syntactic differences to consider (aside from the modalities, of course).

The semantic transformation is also simple. Both semantics preserve the notion that everything within a predicate (in *kFOTK*) or a `TRUE` construct (in *STL*) are interpreted with respect to a pair of time points. The difference between the two is that in the latter case, neither the relations nor the atemporal functions that appear in the constructs (directly) take temporal arguments, while in the former, only the functions do not. The interpretations of the functions are therefore quite similar. Both have the basic form $T \times T \times F \longrightarrow FN$, where T represents the time domain, F the set of atemporal m-ary functions in the language, and FN the set of functions from $D^m \longrightarrow D$, where D is the domain of atemporal objects. The domains in *kFOTK*, of course, are sorted. The case for the predicates vs. relations, however, is a bit different. In first-order *STL*, the meaning function for m-ary relations is like the one

for functions: $T \times T \times R \longrightarrow D^m$, where R is the set of relations in the language. The approach in *kFOTK* essentially bundles up all the time-relation instances and constructs a single predicate with two extra time points: $R \longrightarrow T \times T \times D^m$. Again, since *kFOTK* is a modal many-sorted logic, R is actually a predicate-world pair and D is many-sorted.

Thus, despite the syntactic and structural differences between the two, there is an underlying commonality of meaning between the two. This allows us to easily apply the notions of chronological ignorance and causal theories to *kFOTK*. We have, therefore, maintained the useful aspects of Shoham's logics, while correcting some of their flaws and providing a suitable formal base for the specification language *KA7*.

6.1.3 Chronological Ignorance

The definition for chronological ignorance for *kFOTK* follows closely that for *TK*. The only difference is in the definition for the *ltp* of a formula - compare with Definition 20:

Definition 26 A base formula is one that does not contain the modal knowledge operator. The latest time point *(ltp)* of a base formula is determined as follows:

1. The *ltp* of $[t_1, t_2]p$ is t_2, assuming $t_1 \leq t_2$.

2. The *ltp* of $\phi_1 \wedge \phi_2$ is the latest between the *ltp* of ϕ_1 or the *ltp* of ϕ_2.

3. The *ltp* of $\neg\phi$ is the *ltp* of ϕ.

4. The *ltp* of $\forall t\ \phi$, where t is of sort $s_t \in \mathcal{S}$, is the earliest among the *ltp*'s of all the ϕ' that result by substituting constant values for all occurrences of v in ϕ. If there is no such point, then the *ltp* is $-\infty$.

5. The *ltp* of $\forall x : s\ \phi$, where x is of sort $s \in \mathcal{S}$, is the *ltp* of ϕ. ◇

Since the definition of chronological ignorance (Definition 21) depends entirely on the knowledge operator, which is unchanged in *kFOTK*, the definition remains unchanged for *kFOTK*. Thus, by applying the criterion of chronological ignorance to *kFOTK*, we get its non-monotonic version: *kFOCI*. As with *CI*, we prefer models in which one knows as little as possible for as long as possible.

6.2 KAT: Knowledge of Action and Time

Three major issues remain to be resolved before a useful language for system specification is obtained: (1) the entities of the language, (2) an adequate solution to the frame problem, and (3) the form of the specifications. Furthermore, wherever possible the language must be given enough "syntactic sugar" to make it easier to read and write the specifications

6.2.1 Language Issues

Basic Entities

In addition to time and action, what other entities should be allowed to populate the language? One common entity used in the specification of system behaviour is the *agent*. Indeed, most concurrency theories require a notion of an autonomous agent (a typical example is CCS [51]). Agents are active elements; they perform actions; they affect their environments and interact with each other. They can be used to give a structure to the specification. Without them, the actions are unattached and, hence, implicitly performed by the "system." The result is a flat and unstructured description of the system, one that does not adequately model reality [37]. Thus, they will be added to the set of sorts of *KAT*. Since these three entities are quite sufficient for most needs, no others will be pre-defined. The user, however, is free to add more, free to tailor the language for particular applications; after all, the language is many sorted. One possible suggestion, taken from Allen's formalism [1], is a *process*, an entity that represents ongoing behaviour.

Definition 27 The syntactic entities of *KAT* are those of *kFOTK* augmented with the following:

- k sorts: of which three are pre-defined: s_t, s_a, and s_A. These three will sometimes be labeled as **time**, **act**, and **agt**, respectively;
- V_a and V_A: sets of variables corresponding to sorts **act** and **agt**, respectively.

- F_a and F_A: sets of m-ary atemporal functions that return values corresponding to sorts act and agt, respectively; and

- $\perp$ and ∞: a pair of constants of sort time, referring to the lowest and highest time points, respectively. ◇

Predicates

KAT has only one pre-defined predicate, an *occurrence predicate* that associates an agent to an action:

    Perf(A,a),

where A is a term of sort agt and a of sort act. It asserts that agent A performs action a. Others may be defined as needed. If we had chosen to allow agentless actions, for example, then a simple Occurs(a) predicate would have sufficed.

An *event* is an instantiation of an action. Note the distinctions in the following terms and expressions: lift(tel) is a generic action referring to the act of lifting the handset of any telephone; Perf(fred,lift(tel1)) is a generic phone-lifting activity performed by fred on telephone tel1; and [5]Perf(fred,(tel1)) is an instantiation of that activity: the event of fred lifting phone tel1 at time 5. Unlike property- or fact-type entities, events over an interval do not hold over any proper sub-interval. This meshes with our intuition that an event has a start and a finish; in-between those points, it is not an event, just an event-in-execution. The distinction is a very important one. Without it, we would have an infinite number of events, rendering such notions as event counting and event-ordering relations meaningless. In *KAT*, therefore, we disallow an agent from performing the same event over any sub-interval:

$$\forall\, t_1, t_2, t_3, t_4 \cdot (t_1 \leq t_2 \leq t_3 \leq t_4 \wedge t_1 \neq t_3 \wedge t_2 \neq t_4 \wedge [t_1, t_4]\mathrm{Perf}(A,a)) \rightarrow [t_2, t_3]\neg\mathrm{Perf}(A,a),$$

where A and a are ground terms. In Shoham's terminology [68], Perf would probably be considered a *gestalt* proposition-type.

The Frame Problem

It is not really possible to express within a first-order logic a statement of the form "all facts persist unless explicitly clipped," because it requires quantification over predicates (facts). We are, thus, forced to resort to a set of statements of the form "this fact persists unless explicitly clipped." Such statements, labeled a *persistence rules*, resembles frame axioms, but they are not: they are merely expressions of the principle of inertia. Compare a frame axiom

$$\mathbf{K}[\mathtt{t}]\mathtt{Off_hook(telx)} \land \mathbf{K}[\mathtt{t}]\mathtt{Perf(u,dial(telx,tely))}$$
$$\rightarrow \quad \mathbf{K}[\mathtt{t+1}]\mathtt{Off_hook(telx)},$$

which states that the act of dialing does not change the Off_hook state of the phone, to a persistence rule

$$\mathbf{K}[\mathtt{t}]\mathtt{Off_hook(telx)} \land \mathbf{L}[\mathtt{t}]\neg\mathtt{Perf(u,hangup(telx))}$$
$$\rightarrow \quad \mathbf{K}[\mathtt{t+1}]\mathtt{Off_hook(telx)},$$

which states that the unless we know the phone is hung up, its Off_hook state persists. We will clearly require many more frame axioms than persistence rules. Furthermore, as has already been explained in previous chapters, they are wholly inappropriate for descriptions of concurrent systems. A phone can be hung up while a dialing action happens. If we add the causal rule

$$\mathbf{K}[\mathtt{t}]\mathtt{Off_hook(telx)} \land \mathbf{K}[\mathtt{t}]\mathtt{Perf(u,hangup(telx))}$$
$$\rightarrow \quad \mathbf{K}[\mathtt{t+1}]\neg\mathtt{Off_hook(telx)},$$

to the theory containing the above frame axiom and boundary conditions

$\mathbf{K}[1]\mathtt{Off_hook(telx)}$,
$\mathbf{K}[1]\mathtt{Perf(u,hangup(telx))}$, and
$\mathbf{K}[1]\mathtt{Perf(u,dial(telx.tely))}$.

we arrive at a contradiction: the telephone is both on and off hook.

Since we do not know how long a fact will persist, the most conservative assertion we could make is the following: if it holds now, and we have no reason to believe

it has been clipped, then it should persist to the next time click. We cannot safely predict that it will hold beyond that, or over an extended future interval. Hence, with discrete and linear time, if the antecedents of a persistence rule hold at time t, the consequent should hold at time $t+1$. The actual temporal unit, however, is irrelevant nanoseconds, seconds, minutes, whatever—since with discrete time, the values of things can only be measured at the time clicks: in-between the clicks, nothing can be measured.

The other advantage to persistence rules is that they have the same form as causal rules. Hence, we stay within causal theories and can employ the rule-and-exceptions technique for specifying the persistences (compare with axioms 4a,b,c in Sub-section 5.2.3):

$$\mathbf{K}[\mathtt{t}]\mathtt{Off_hook(telx)} \wedge \mathbf{L}[\mathtt{t}]\mathtt{Normpers17} \rightarrow \mathbf{K}[\mathtt{t+1}]\mathtt{Off_hook(telx)}$$
$$\mathbf{K}[\mathtt{t}]\mathtt{Perf(u,hangup(telx))} \rightarrow \mathbf{K}[\mathtt{t}]\neg\mathtt{Normpers17}$$
$$\vdots$$

The persistence rule exception may alternatively be read as "the action is abnormal with respect to that fact," harking back to the abnormality predicates of circumscription. As with causal rule exceptions, the evaluation of these abnormality conditions must be delayed as much as possible at any particular point in time. Intuitively, this permits the actions to clip them at the right time.

The following example illustrates both the frame problem solution and how to determine the known facts of a theory.

Example 6 A telephone is either on-hook or off-hook. When it is on-hook, then lifting the handset makes it off-hook; contrariwise, hanging up the phone returns it to the on-hook state. The following statements describe this scenario:

1 $\mathbf{K}[\mathtt{t}]\mathtt{Off_hook(x)} \wedge \mathbf{K}[\mathtt{t}]\mathtt{Perf(u,hang_up(x))}$
$\quad\rightarrow \mathbf{K}[\mathtt{t+1}]\neg\mathtt{Off_hook(x)}$

2 $\mathbf{K}[\mathtt{t}]\neg\mathtt{Off_hook(x)} \wedge \mathbf{K}[\mathtt{t}]\mathtt{Perf(u,lift(x))}$
$\quad\rightarrow \mathbf{K}[\mathtt{t+1}]\mathtt{Off_hook(x)}$

3 $\mathbf{K}[\mathtt{t}]\mathtt{Off_hook(x)} \wedge \mathbf{L}[\mathtt{t}]\mathtt{Norm_off} \rightarrow \mathbf{K}[\mathtt{t+1}]\mathtt{Off_hook(x)}$

	$t = 0$	$t = 1$	$t = 2$
A	¬Off_hook(x)	¬Off_hook(x)	Perf(u,lift(x))
B			¬Off_hook(x)
C			¬Norm_on

	$t = 3$ & $t = 4$	$t = 5$	$t = 6$
A	Off_hook(x)	Perf(u,hang_up(x))	¬Off_hook(x)
B		Off_hook(x)	
C		¬Norm_off	

Table 6.1: Known facts in every *cmi* model for phone example

3a **K**[t]Perf(u,hang_up(x)) → **K**[t]¬Norm_off

4 **K**[t]¬Off_hook(x) ∧ **L**[t]Norm_on → **K**[t+1]¬Off_hook(x)

4a **K**[t]Perf(u,lift(x)) → **K**[t]¬Norm_on.

Axioms 1 and 2 are causal rules describing the behaviour of the actions; axioms 3 and 3a are persistence rules for the off- and on-hook states; and axioms 4 and 4a are exception conditions for their respective persistences. Now, let us add the following boundary conditions to the theory:

5 **K**[0]¬Off_hook(x)

6 **K**[2]Perf(u,lift(x))

7 **K**[5]Perf(u,hang_up(x)).

Before $t = 0$, the usual tautologies are known. At $t = 0$, the only fact whose knowledge is imposed upon us is [0]¬Off_hook(x) from axiom 5 (see Table 6.1) Everything else is marked as unknown. We proceed to $t = 1$. The only fact necessitated by the theory is the consequent of axiom 4: since [0]¬Off_hook(x) is already a known fact, we add knowledge of [1]¬Off_hook(x). Again, everything else at $t = 1$ is unknown. We now proceed to $t = 2$. Off_hook(x) persists, as expected, but we now have a new fact imposed upon us from axiom 6, the performance of a lift action. This, in turn, triggers the abnormal condition [2]¬Norm_on in axiom 4a. Everything else at $t = 2$ is marked as unknown. When we move to $t = 3$, the first pair of known facts at time $t = 2$ (A and B in Table 6.1) cause [3]Off_hook(x) to be known (from

axiom 2). The ¬`Off_hook(x)` fact, however, does not persist because of the knowledge of the abnormality at $t = 2$. The rest of the facts listed in the table can likewise be derived. □

From this example, the persistence rules and their exceptions appear to correctly capture our intuitions.

Syntactic Sugar

A well-defined language with a solid formal foundation lends precision to the statements one writes, but that does not necessarily imply that writing and reading them is clear-cut. If anything, unless one is quite adept with logics, the strictness of the language appears to hinder and obfuscate, rather than aid comprehension. This applies even for well-known standard logics like first-order predicate calculus, let alone for a new and non-standard, perhaps counter-intuitive, one like *kFOTK*. This is one aspect of formal methods that has led many to decry their use.

Two basic obstacles stand in the path towards the general utility of *KAT*. The first is the language itself: like other logics, it has a rather limited syntax, requiring one to string together many symbols, often in intricate patterns, to express simple things. The second stems from the the novelty of the language: a frame of reference for it does not exist. Therefore, misconceptions of its capabilities or proper use are bound to occur. To mitigate the difficulties associated with first problem, *KAT* will be given some syntactic sugar. This may indirectly provide some clues regarding the second problem, but the latter is best characterized a pragmatic concern (see Subsection 6.2.4 below). For this proposal, only the following syntactic conventions are suggested:

- Atomic formulae over the same interval may be combined. An expression of the form $[t_1, t_2]\phi \wedge \mathbf{K}[t_1, t_2]\psi$ may be written as $[t_1, t_2](\phi \wedge \mathbf{K}\psi)$. Care must be taken to avoid nesting modal operators, since this is strictly forbidden.

- Causal rules that have the identical antecedents and whose consequents are over the same interval may be combined. The pair of expressions, $\Phi \rightarrow \mathbf{K}[t_1, t_2]\phi$ and $\Phi \rightarrow \mathbf{K}[t_1, t_2]\psi$, may be written as $\Phi \rightarrow \mathbf{K}[t_1, t_2](\phi \wedge \psi)$.

- Since exception conditions have the same general format, they can be given a special construct. There are two types, one each for causal and persistence rules, respectively:

  1. $\texttt{except}(p, \neg\texttt{cnorm}) \equiv \mathbf{K}[t]p \rightarrow \mathbf{K}[t]\neg\texttt{cnorm}$, and
  2. $\texttt{pexcept}(p, \neg\texttt{pnorm}) \equiv \mathbf{K}[t]p \rightarrow \mathbf{K}[t]\neg\texttt{pnorm}$.

  where, p, cnorm and pnorm are predicates.

Other possibilities include special constructs for persistence rules and causal rules. Persistence rules may be served by $\texttt{persist}(p, \texttt{pnorm}) \equiv \mathbf{K}[t]p \wedge \mathbf{L}\texttt{pnorm} \rightarrow \mathbf{K}[t+1]p$. General causal rules, however, require a complex structure, one that must highlight its various components—pre-conditions, actions, enablers and consequents. While such a construct, resembling perhaps a VDM operation description, would enhance readability, its structure and complexity would detract from the overall axiomatic framework.

6.2.2 P-Causal Theories

The theory in Example 6 is an example of *KAT* causal theories, herewith labeled *p-causal theories* to distinguish them from *CI* causal theories and to emphasize their persistence element. They consist of three basic statements: causal rules, persistence rules, and exception conditions. Causal rules here have a slightly different definition from those in causal theories. Since we wish to capture the active element in a cause-and-effect relationship, we will insist that each causal rule has at least one (known) action in its antecedent. Persistence rules, on the other hand, do not require an active element; indeed, it is the lack of such an element that enables the persistence of a fact. Clearly, these rules match rather nicely our intuitions regarding the nature of causation and persistence. Both, however, would be considered just causal rules in *CI* causal theories, since Shoham's logics do not distinguish proposition types.

The definition below formalizes many of the notions already discussed in the previous sections.

Definition 28 Let the term *active predicate* mean a predicate that takes action arguments, and *passive predicate* one that does not. Then, let a represent an active predicate, f a passive predicate, p a predicate of either type, and n a normal predicate. Further, let ϕ^a, ϕ^f, and ϕ^p represent atomic base sentence of the form $[t_3, t_4]q$, where q is one of a, f, or p, respectively, and $t_3 \leq t_4$. All the formulae in the theory are ground.

A *p-causal statement* is a sentence having one of the following forms:

causal rule	$\Phi_{\mathrm{pre}} \wedge \Phi_{\mathrm{act}} \wedge \Theta_{\mathrm{enab}} \rightarrow \mathbf{K}[t_1, t_2]p$
normal form:	$\Phi_{\mathrm{pre}} \wedge \Phi_{\mathrm{act}} \wedge \mathbf{L}[t_1]n_c \rightarrow \mathbf{K}[t_1, t_2]p$
persistence rule	$\mathbf{K}[t]f \wedge \Theta_{\mathrm{penab}} \rightarrow \mathbf{K}[t+1]f$
normal form:	$\mathbf{K}[t]f \wedge \mathbf{L}[t]n_p \rightarrow \mathbf{K}[t+1]f$
exception conditions	$\mathtt{except}(p, \neg n_c) \equiv \mathbf{K}[t]p \rightarrow \mathbf{K}[t]\neg n_c$, and
	$\mathtt{pexcept}(a, \neg n_p) \equiv \mathbf{K}[t]a \rightarrow \mathbf{K}[t]\neg n_p$

where

1. in causal rules: Φ_{pre} is a conjunction of pre-conditions $\mathbf{K}\phi_i^f$, where ϕ_i^f has an *ltp* of t_4 such that $t_4 < t_1$; Φ_{act} is a conjunction of active elements $\mathbf{K}\phi_i^a$, where ϕ_i^a has an *ltp* of t_4 such that $t_4 < t_1$; and Θ_{enab} is a conjunction of enablers $\mathbf{L}\phi_i^p$, where ϕ_i^p has an *ltp* of t_4 such that $t_4 < t_1$;

2. in persistence rules: Θ_{penab} is a conjunction of enablers $\mathbf{L}[t]\neg a_i$;

3. a *boundary condition* is a casual rule with empty Φ_{pre} and Φ_{act}.

A *p-causal theory* is a collection of p-causal statements, such that

1. There exists an initial time point t_0, such that for all boundary conditions $\Theta_i \rightarrow \mathbf{K}[t_i, t_j]p_i$, $t_0 \leq t_i$;

2. There do not exist two sentences in Ψ such that one contains $\mathbf{L}[t_1, t_2]p$ on its l.h.s. and the other includes $\mathbf{L}[t_1, t_2]\neg p$ on its l.h.s. (*soundness conditions*);

3. If $\Upsilon_1 \rightarrow \mathbf{K}[t_1, t_2]p$ and $\Upsilon_2 \rightarrow \mathbf{K}[t_1, t_2]\neg p$ are two sentences in Ψ, then $\Upsilon_1 \wedge \Upsilon_2$ is inconsistent.

A *normal p-causal theory* is one composed of causal and persistence rules, all in normal form, and exception conditions. It must satisfy the additional requirement that

- at any time t, knowledge of abnormalities is postponed until all other t-sentences have been determined.

P-causal theories enjoy the same properties as causal theories. Specifically, if Ψ is a p-causal theory, then it is consistent and has a unique class of *cmi* models. Example 6, which satisfies all of the requirements for a p-causal theory, clearly manifests these properties. To see that all p-causal theories enjoy these properties, first note that all the predicates are ground: this essentially reduces the theories to the propositional case (compare with Definition 23). All of the requirements match, except those concerning the form of the sentences. However, causal and persistence rules in p-causal theories are just special cases of causal rules in causal theories. As for the sentences in a normal p-causal theories, the additional requirement assures that any normal p-causal theory can be converted to an equivalent causal one [68, 70]. That is, a collection of rule-exception statements

$$\Phi \wedge \mathbf{L}[t_1, t_1]n \rightarrow \mathbf{K}[t_1, t_2]p$$
$$\mathbf{K}[t_{i1}, t_{i2}]q_i \rightarrow \mathbf{K}[t_1, t_1]\neg n, \text{ for } 1 \leq i \leq m$$

can be replaced by the single causal rule

$$\Phi \wedge \bigwedge_{i=1}^{m} \mathbf{L}[t_{i1}, t_{i2}]q_i \rightarrow \mathbf{K}[t_1, t_2]p.$$

Hence, p-causal theories are a subset of causal ones. This also means that it should be possible to define an algorithm for determining all the known facts that hold in the class of *cmi* models of a theory. Given that developing a proof theory for *KAT* is highly unlikely, which is the case with most non-monotonic formalisms, the benefits of this are clear: it provides an alternative way for determining some of the consequences of a theory.

Of course, when used in specifications, the majority of the statements will be schemas or universally quantified formulae, not sentences—and, strictly speaking, they should therefore no longer be considered p-causal theories, but such a distinction will not be made here. Nevertheless, the above results are useful. They suggest that by giving various boundary conditions and instantiating the formulae appropriately, it becomes possible to execute the specification. This aspect of the language will not be fully explored in this thesis, but it would surely enhance the utility of the language (see Sub-section 6.2.4 below for more details).

6.2.3 KAT Specifications

What form should the specifications take? Basically, a *KAT* specification contains some sorting information and a set of axioms. This suggests an algebraic-specification style of presentation, or more closely, one similar to that used by Khosla and Maibaum in [37]. *KAT* specifications, therefore, have the following form:

Specification *specification-name*

Language:

SORTS:	sort-list
CONSTS:	*constant-name* : sort
VARS:	*variable-name* : sort
PREDS:	*predicate-name* : ⟨ sort-list ⟩
ACTS:	*action-name* : ⟨ sort-list ⟩
FUNCTS:	*function-name* : ⟨ sort-list ⟩ $\longrightarrow$ sort

Axioms:

$$\text{normal p-causal theory} \begin{cases} \text{causal rules} \\ \text{causal rule exceptions} \\ \text{persistence rules} \\ \text{persistence rule exceptions} \end{cases}$$

End.

6.2.4 Pragmatic Issues

Acquiring a "feel" for *KAT* will doubtless take a significant amount of time and use. Certainly, its epistemic modalities and non-monotonicity will confound the effort. Nevertheless, from the discussion and examples above, *KAT* shows promise as a specification language. It seems to provide enough of the machinery required for the specification of systems: with it, we can define the objects that make up the system, and express its behaviours. But how should it be used? How do we take advantage of its many features? What is the proper way to apply the modalities?

So far, most of the language's features have been described from a technical, formal perspective. It might even be argued that there has been, perhaps, an inordinate emphasis placed on the semantics of the language, while the syntax and application of the language have been practically neglected. In defense of the tack taken, given that this is fairly new ground for a specification language, it is very important to be very clear about what everything means. Besides, this approach to specification language definition is a deliberate departure from the usual one which focuses only on the syntax, expecting the user to intuitively "understand" the meanings of the terms from the numerous examples given. Unfortunately, while formal definitions make everything clear, they do not always convey the overall scope and utility of the language. In this section, therefore, some practical and conceptual issues are addressed, and the key features of the language, together with its capabilities and known limitations, are compiled and re-examined from a pragmatic point of view.

(In)Complete Specifications

Are KAT specifications complete or not? Clearly, since it is no longer necessary to explicitly specify all the qualifications and frame axioms, the specifications cannot be considered complete. However, the formalism (through its semantics) takes this into account. It fills in the missing information; it "completes" the specification. The question is, does it fill in enough information? Since KAT only deals with partial specification and no other form of incompleteness, the answer is no. Does it therefore *partially* complete the specification? This is still an open question. For one thing, it only deals with two thirds of this class of incompleteness, since it does not tackle the ramification problem; for another, one would still be required to provide enough information to make it possible to fill in the rest. What is a sufficient amount of information? Can it be defined? And, can partial completeness, as it relates to KAT specifications, be formally defined and tested for? These are left for further study.

Linguistic Components

A proper specification language, one sufficient for the specification of a variety of systems, should be capable of expressing three types of information: static information, action descriptions and dynamic behaviour. The first two constitute a description of system behaviour, while the latter a prescription of it [37]. In most traditional approaches, such as VDM, only the first two can be specified; others, including the various process algebras such as CCS or CSP, only the third. KAT, however, can handle all three.

static information Static information represents the set of entities, properties, relationships and invariants of the system. Obviously, these are specified in the usual way via the first-order, many-sorted logic apparatus—everything in the *Language* section of a KAT specification —together with the boundary conditions. The rather strict form of the boundary conditions, however, limits the types of invariants that can be expressed. P-causal theories require fixed intervals and atomic formulae, but these rules may be relaxed a little. Specifically, the symbols $\perp$ and ∞ can be used to express universal invariants: $[\perp, \infty]p(x)$.

A bottom symbol, $\perp$, is used instead of $\neg\infty$, to avoid violating the requirement that there must exist a starting point (for computational purposes). Furthermore, by using some syntactic sugar, more complex statements can be made: `[⊥,∞]¬(off_hook(x) ∧ ringing(x))`, or even `Inv(¬(off_hook(x) ∧ ringing(x))`. However, quantifiers may not be used, and if executability is required, a suitable way to instantiate such statements must first be defined.

action description Action descriptions are specified through the causal rules. Ignoring for the moment the complications associated with the **K** and **L** modalities, *KAT* action descriptions resemble those of many other specification formalisms: to wit, they are given in terms of pre- and post-conditions. Unlike most of them, however, *KAT* does not depend on a notion (explicit or otherwise) of a before-state, an atomic action and an after-state. Pre conditions may hold over any (past) times; actions may have durations; more than one action may be involved; and the post-conditions need not take effect immediately after the execution of the action. Actions also require agents. This allows us to distinguish the actions taken by the user or environment from those of the system or even distinguish the actions taken by the sub components of a system.

Moreover, an action may be described by more than one causal rule, each having a different set of pre-conditions and different effects. This allows us to specify just the minimum relevant pre-conditions (context) for a particular effect of an action. For example, lifting the handset of a telephone has two possible (and different) effects: answering a call or initiating a call, depending on whether or not the phone is ringing, respectively. To describe this behaviour in VDM we would either have to put both effects within a single `lift` action (operation) description, or split them up into a pair of distinct actions, say `lift_init_call` and `lift_answer_call`. Neither solution is satisfactory: the former unnecessarily complicates the definition; the latter conceals the fact that they essentially involve the same action.

There is yet another difference between causal rules and most traditional approaches of describing actions: it is in the nature of their pre-condition. In

the traditional approaches, a pre-condition has a pair of meanings: it identifies the context within which the action may be performed, and it implicitly approves the execution of an action (provided the pre-condition holds). By lumping them together, however, this important distinction is lost [37]. The pre-condition should only establish the context: the execution of the action is strictly a system dynamics issue. Of course, if a language is not capable of expressing dynamic behaviour, then it cannot make this distinction: *KAT*, however, can (see next item). Hence, pre-conditions only serve to establish the context.

All these aspects allow for a more realistic (and natural) modeling of action behaviour than is possible with traditional approaches. They also permit us to model concurrent systems without having to resort to a non-deterministic interleaving of atomic actions (as in CCS and CSP). Atomicity, however, prevents interference between conflicting concurrent actions. In *KAT* theories, non-interference is safeguarded by the third condition in the definition of p-causal theories (Definition 28).

dynamic behaviour It is not possible with just static information and action descriptions based on pre- and post-conditions to implicitly derive system behaviour. Indeed, languages that only provide for the other two components cannot be used to specify reactive systems, which exhibit on-going behaviour. To describe system behaviour, the language must provide a way to associate actions to each other. This could be accomplished in a number of ways, such as by permitting the specification of sequences (or partial orders) of actions, or by permitting actions to cause (or oblige) other actions to execute. Apart from permitting multiple actions (in sequential and concurrent onfigurations) in the antecedent of a causal rule, *KAT* also allows actions in the consequent. As an example of the latter, dialing a phone number obliges the telephone exchange to attempt to make a connection (within a reasonable amount of time): $\mathbf{K}[t_1,t_2]\texttt{Perf(u,dials(x,y))} \rightarrow \mathbf{K}[t_2\texttt{+tmin},t_3]\texttt{Perf(ex,connect(x,y))}$. In addition to this, boundary conditions can be used to "force" the execution of

actions at specific times. Thus, to "trigger" a causal rule, it is not enough for the context to hold, the action must also be compelled (in one way or another) to execute. Finally, with a suitable interpreter for the language (see below), there exists the possibility of simulating the behaviour of a theory.

There are other specification languages that also allow the specification of all three types of information. Khosla and Maibaum's (K&M's) language [37], for example, is a many-sorted first-order logic with action modalities (for the action descriptions) and obligation/permission modalities (for prescribing system behaviour). And in certain ways, some of these languages are more expressive than *KAT*. Taking K&M's language as an example again, it is more expressive in three respects: (1) it does not restrict the form of the sentences; (2) behaviour-wise, it permits one to specify a sequence of obligated actions – one obligating another obligating yet another, and so on; and (3) it distinguishes two types of causality through the obligation and permission modalities.

KAT, however, has linguistic components that few or none of them have. Observe, for instance, that action descriptions as stated above do not completely describe an action: they do not indicate which things are not affected by it. Obviously, this is the role that persistence rules and their exception conditions play in *KAT* specifications. These, in turn, are part of a general mechanism for tolerating incomplete information. From a technical perspective, the main components of this mechanism are the epistemic modalities and the chronological ignorance preference criterion; from a practical perspective, however, it is the enabler in the rules. When used within a causal rule, the enabler handles the qualification problem; when used within a persistence rule, it handles the frame problem; and, when used within normal forms of both rules and combined with the exception conditions, it makes the modification of specifications easier.

The way *KAT* (and the enabler in causal rules) handles the qualification problem is analogous to that for *CI* theories (see Sub-section 5.2.3). Furthermore, the semantics of *kFOCI* cover the situations where a pre-condition (or enabler) does not hold for a causal rule: the rule is not triggered, and no effect produced. If a different effect is required, then one merely adds another causal rule for the action containing the negation of that pre-condition. As for the frame problem, the way it is handled in

KAT has already been explained above in Sub-section 6.2.1. Thus, *KAT* satisfies one of its primary requirements: tolerating incompleteness due to partial specification. Its modifiability is discussed in the next sub-section.

Modifiability and Theory Construction

Constructing specifications (theory building, Section 3.1) is a long, arduous process. Theories require constant modification, whether from the need to correct errors, to add or change requirements, or to simply find a more appropriate formulation. As explained in Sub-section 3.3.1, this constant flux is an unavoidable source of incompleteness. Furthermore, the changes will often be non-monotonic in nature: each change necessitating other changes in the theory. Therefore, all proposals for specification languages or methods should take these aspects of the software specification task into serious consideration. In general, the languages should minimize as much as possible the amount of revision required with each change to the theory.

To illustrate the basic approach, let us look at the qualification problem—the notions apply to the frame problem as well. Let us say we are trying to model a particular action χ. We ponder carefully on its expected behaviour. When we feel we know all of its pre-conditions, we confidently write down, in a monotonic logic in which the predicates take temporal arguments, an axiom like

$$\varphi_1(t) \wedge \cdots \wedge \varphi_m(t) \wedge \chi(t) \rightarrow \psi(t+1).$$

where the φ_i are the pre-conditions. Unfortunately, we discover shortly thereafter that some pre-conditions were missed. This makes the whole axiom obsolete and forces us to modify it. Later still, a new feature is added to the system, but we realize that it conflicts with some axioms already in the theory, including this one. Thus, more changes are necessitated. In a developing theory, such changes will be required again and again. Eventually, a certain stability in the theory might be reached, but a theory, it must be emphasized, can never be considered as fixed or correct or complete. It will always be subject to change.

In a *KAT* specification, on the other hand, we do not just write down the pre-conditions: we also must decide which ones are absolutely necessary for the action to

execute and which ones either merely lend support or are assumed to hold (at least for the time being). The former clearly become **K**-conditions, the latter **L** conditions. So, our first axiom now has the following form:

$$\mathbf{K}[t]\phi_1 \wedge \cdots \wedge \mathbf{K}[t]\phi_k \wedge \mathbf{L}[t]\phi_{k+1} \wedge \cdots \wedge \mathbf{L}[t]\phi_m \wedge \mathbf{K}[t]\chi \\ \rightarrow \mathbf{K}[t+1]\psi.$$

This axiom will also be subject to change initially, but usually the changes will be in the form of more **L**-conditions. Once the **K**-conditions become relatively stable, we may then collapse all the **L** ones into a single abnormality n, and add a set of exception conditions for them:

$$\Phi \wedge \mathbf{K}[t]\chi \wedge \mathbf{L}[t]n \rightarrow \mathbf{K}[t+1]\psi$$
$$\mathbf{K}[t]\phi_{k+1} \rightarrow \mathbf{K}[t]\neg n$$
$$\vdots$$
$$\mathbf{K}[t]\phi_m \rightarrow \mathbf{K}[t]\neg n$$

where Φ represents all the **K** pre-conditions. From then on, changes to the action description or changes elsewhere that affect this rule will only require the addition of more exception conditions, the axioms themselves being left intact. Of course, we can always begin with the normal form of the description, even if we not yet aware of any potential **L**-type conditions. This approach anticipates change, and promotes the idea of constructing the theory in stages, starting with a core or kernel of the system and slowly adding features to it. Note, however, that this approach does not guarantee that every change will be just a simple addition to the theory. Rather, it reduces the type of changes that necessitate modification of existing axioms.

A final note: Observe that any change to the original axioms in a theory of a monotonic language represents a non-monotonic change; in KAT, however, the non-monotonicity is "built-in," hence such non-monotonic changes can be achieved through simple monotonic modifications to the theory.

Executability

P-causal theories are in a form that is suitable for execution. The obvious benefits of executability are that it allows the developer to computationally test the specification

for inconsistencies and to verify that it matches the client's requirements. The key, of course, is to properly instantiate the axioms. It is beyond the scope of this thesis to explore the various ways of achieving this, but one suggestion is to instantiate every relevant axiom at each new time click that is reached in the simulation. For example, for a persistence rule of the form **K** [t]P(x) ∧ **L**[t]N → **K**[t+1]P(x), given an initial time $t = 0$ and current time click $t = 2$, the following axiom instances must be generated by the interpreter of the language:

K[0]P(a) ∧ **L**[0]N → **K**[1]P(a)
K[1]P(a) ∧ **L**[1]N → **K**[2]P(a).
K[2]P(a) ∧ **L**[2]N → **K**[3]P(a).

where a is a suitable instance of **x**. The interpreter should allow the user to instantiate the variables and to impose boundary conditions either as initial conditions for the execution or during the simulation, as it moves forward in time. An appropriately modified version of the algorithm given in [68, pp. 114-115] (for computing the *cmi* models of causal theories) can then be used to determine the new facts at each new time point. For output, it should display the current *cmi* facts, perhaps in a tabular form such as that in Table 6.1.

Another potential element of the specification that might be automated is the persistences. Note the form of the persistence rules and their exception conditions in Example 6, and compare them with the rest of the theory. Firstly, a (normal) persistence rule should be generated for each predicate of the theory (excluding the **Perf** and normal ones): more precisely, for each predicate $P_i(\overline{x})$, where $\overline{x}$ represents a list of arguments, a persistence rule of the form

$$\mathbf{K}[t]P_i(\overline{x}) \wedge \mathbf{L}[t]N_i \rightarrow \mathbf{K}[t+1]P_i(\overline{x})$$

must be generated. Then, for each causal rule in the theory that affects the predicate, $\Phi \wedge \mathbf{K}[t]\mathtt{Perf}(u, a(\overline{y})) \rightarrow \mathbf{K}[t+1]\neg P_i(\overline{x})$, an exception condition of the form

$$\mathtt{pexcept}(\mathtt{Perf}(u, a(\overline{y})), \neg N_i)$$

must be generated.

Objections to KAT

There are at least two objections to the *KAT* specification language. The first concerns the use of epistemic modalities. Galton [19], in particular, criticizes their use in causation and causal reasoning. After all, what do knowledge and belief have to do with causation? Surely, things cause other things whether or not we are aware of them. Causality, therefore, is best viewed as an objective phenomenon, one independent of what anyone believes. Furthermore, contrary to Shoham's claims [70], Galton argues that causal reasoning is not inherently non-monotonic; rather, non-monotonicity is an aspect of human reasoning in general. Although both arguments are persuasive, they are primarily philosophical concerns. The source of the non-monotonicity, for instance, is less important than whether the semantics delivers it. And there is nothing in the Kripke-style semantics of *kFOTK* that forces us to give the modalities strictly epistemic intuitive interpretations. Essentially, the modalities only serve to distinguish those facts that we are absolutely certain about from those that we are not. We then simply give the benefit of the doubt to the latter facts and assume they hold, unless we have explicit evidence to the contrary. Still, epistemic modalities are unusual for a specification language, and some may find them difficult to master. One way to conceal them is to resort to syntactic structures that highlight the various components of the causal and persistence rules without making explicit use of the modalities.

The other objection concerns the use of time. Turski and Maibaum in [73] claim that the use of explicit time constraints indicates that a proper analysis of the system was not carried out, and that the real or fundamental nature of its primitive elements and their relations have not been discovered. The use of absolute time intervals, they assert, results from a superficial observation of the phenomena in the system. For example, instead of specifying that the minimum permissible interval between airplane landings at airport X is, say, 240 seconds, the specifier should try to determine what underlying factors are responsible for such a figure. They further claim that for most real-time applications, a more concise and general description could be attained by studying the functional and relational dependencies between the events in the system

rather than by jotting down temporal constraints— that is, by looking at it as an *event-dependent* control system instead of a clock-based one. The ultimate drawback of employing time, especially in a framework that encourages the specification of systems via sets of rules and exceptions, is implementation bias. It is just too tempting to convert such formulae directly into program statements, the end result perhaps being a program with a complex and awkward control structure. While they are correct in asserting that most uses of explicit time are unwarranted, there are situations where it is unavoidable. How, for instance, do we impose temporal performance constraints, either for timely responses or for safety reasons, without it? Say the client absolutely requires a response in t microseconds, how can it be specified without recourse to explicit time? Unfortunately, it is not exactly clear how to reconcile these conflicting requirements. It certainly merits further study.

6.2.5 Example: Plain Old Telephone System

The examples presented thus far have given but a hint of *KAT*'s features and capabilities. In this section, therefore, a more substantial example will be presented and discussed, one that will better demonstrate these features. It is a *KAT* specification of a plain old telephone system (POTS), a very basic telephone service. It was chosen because

- it is universally familiar (at least from the user's point of view), thus the reader should be able to readily judge how well the language captures its behaviour.
- it is a concurrent system involving different (interacting) agents,
- it includes timing constraints,
- it is a real-world application in which there has been a considerable amount of research and development over the years, and it involves some of the largest real-time control systems ever invented,
- it is an application that is constantly being modified and updated with new features,

- it is often used as an example to demonstrate specification languages (see for example, [14, 16, 36, 37, 42]).

The basic system consists of a set of telephones connected to a telephone exchange. The exchange allows any pair of telephones to connect with each other that is, it permits the users of the telephones to communicate synchronously. The user's basic actions include lifting and hanging up the handset, and dialing a number. The exchange's duty is to respond to the users' requests (in a timely manner). It also provides the user with a number of signals or messages to indicate the status of the telephone, such as a dial tone, a busy tone, ringing its bell, *etc.*

To simplify the presentation, a point-based description is given and only the basic behaviours involved in making a connection are considered. Let us begin with the entities in the system (see Figure 6.2). Aside from the three basic sorts, the specification also requires a `tel` (telephone) sort. Only one exchange is identified, so it is made a constant. The other constants, the temporal ones, will be explained later on. The declaration of variables is straightforward, but note that it is possible to provide comments within a specification by enclosing them within the usual `/*` and `*/` brackets.

Aside from `Perf`, there are three types of predicates: (1) a set of properties or states of a particular telephone— the telephone may be `Hooked_up` or not, `Off_hook` or not, it may receive a `Dialtone` to indicate that a call may be made, a `Ringback` if the other telephone is ringing, a `Busytone` if it is not, or a `Hangup_msg` to admonish the user that the telephone has been left off-hook too long without it being used, and it may be `Ringing`; (2) a set of relations of pairs of telephones `Calling(x,y)` indicates that telephone `x` is in the process of calling `y`, and `Connected(x,y)` to indicate that the connection has been successful; and (3) a set of *normal condition* predicates—their meanings will be apparent from the discussion below on the axioms of the theory.

In this description, only three user actions are declared: `lift` represents the lifting of the handset, `hang_up` the replacing of the handset, and `dials(x,y)` the act of dialing `y`'s number from telephone `x`. To distinguish between the pair of telephones involved, the one initiating the call will be labeled the *calling telephone*, the other

the *callee telephone*; the users will likewise be distinguished, *caller* and *callee*, respectively. The exchange performs the following seven actions: `connect(x,y)` is the act of attempting to connect telephone `x` to y, `sgnl_timeout(x,t)` is an internal trigger or alarm that indicates that telephone `x` has exceeded a pre-set temporal limit `t`, and a set of send actions to deliver tones and messages to particular telephones—`snd_dialtn` sends a dial tone, `snd_ringbk` a ringback, `snd_ringbl` a ringing, `snd_busytn` a busy tone, and `snd_hangup` a hang-up message.

Figures 6.3 and 6.4 display the axioms of the POTS theory. We have already seen Axioms 1, 2, 18, 18a, 19, and 19a in Example 6, modeling the behaviours associated with going on- and off-hook. Axiom 0 is a boundary condition that fixes the initial state of each telephone: it is on-hook.

To make a call, a user begins by lifting the handset. Of course, the telephone must be on-hook and not ringing; if it is ringing, the lift action would produce a different effect (discussed below). The call initiation behaviour is captured by Axiom 3, which states that under normal conditions (`Normcallinit`), the lifting act obliges the exchange to send a dial tone to the telephone (when combined with Axiom 13). One possible abnormal situation is if the telephone is not yet hooked up to the network (Axiom 3a); another, not shown, is if the telephone is broken. Under such situations, the exchange would not receive the request to make the telephone ready for a call, and hence no dial tone would be forthcoming. Two further points must be made. The first concerns `MAXRESP`. This is the maximum prescribed time for the transmission of a stimulus or response from the telephone to the exchange; the same amount of time is postulated for the return trip. A more realistic description, however, might employ a temporal function that returns for each exchange/telephone pair a maximum value, perhaps even taking into account such factors as the distance between them. The second point concerns atomic actions. While the `lift` action is effectively atomic, the exchange's `snd_dialtn` one is probably not. An alternative formulation might include a function that returns the execution time for any action:

FUNCTS: $\quad$ `time_of: ⟨act⟩ → time`

$3'$ `let m = t+MAXRESP`

in { $\mathbf{K}$`[t]¬Off_hook(x)` $\cdots$

$\rightarrow$ $\mathbf{K}$`[m,m+time_of(snd_dialtn)]Perf(ex,snd_dialtn(x))` },

where the **let** construction is just more syntactic sugar (borrowed from VDM). The other actions performed by the exchange may be treated in a similar fashion.

Having received a dial tone, the user may then dial a number. Again, to simplify the presentation, the **dials** action is assumed to be atomic — not really an unrealistic assumption, given that many telephones have redial and pre-set number buttons. The dial action obliges the exchange to attempt a connection, provided of course that the situation is a normal one (Axiom 4). One possible abnormal situation arises when the user fails to perform a dial action within the pre-determined MAXDIAL time (Axiom 4a); when this occurs, a sgnl_timeout event is triggered, causing the exchange to terminate the telephone's dial tone (Axiom 5a) and send it instead a hang up message (Axioms 5b and 17). These axioms demonstrate how to specify any additional effects caused by abnormalities—aside from the usual clipping of causal rules. Another potential abnormal situation, also not shown in the theory, is dialing a bad or non-existent telephone number. The exchange's usual response in such a case is to send the calling telephone a bad-number message. MAXTRANS in Axiom 5b represents the maximum prescribed time to transmit (internal) signals between components of the exchange. Undoubtedly, it is, like MAXRESP, an oversimplification.

When it receives the connection request, the exchange cuts off the calling telephone's dial tone (Axiom 6). In a more realistic model, in which the dial action is composed of a sequence of send-digit actions, the dial tone would be cut off after the first digit had been sent; and a connection would not be attempted until the last digit had been received. In any case, there are two possible outcomes of a connection attempt. The first involves an available callee telephone — *i.e.* it is on-hook and not currently ringing. Under normal conditions, the connect(x,y) action establishes that x is calling y, and causes the callee telephone to ring, while notifying the calling telephone of this fact through the ringback tone (Axioms 7, 14 and 15). The caller

may hang-up the telephone even before receiving any response from the exchange: however, this results in an abnormal situation (Axiom 7a). The second outcome involves a callee telephone already in use: the exchange notifies the caller by sending it a busy tone (Axioms 8 and 16). The caller can kill the busy tone by hanging up the telephone (Axiom 9). (Rather, the hang-up causes a signal to be sent to the exchange, notifying it of said fact, but this additional complication was left out to simplify the presentation.) Note that the hang-up also entails (through Axioms 1, 18 and 18a) a cancellation of the `Off_hook` state, which has persisted until now. Thus, it was not necessary to explicitly include the $\neg$`Off_hook` effect in the post-condition of Axiom 9 or add an additional rule to indicate that this is a side-effect of Axiom 7a. In a sense, this can be viewed as a limited tolerance of the ramification problem, and can be used to advantage in writing the specifications. The only persistence rules shown in the theory are those for the on-hook, off-hook and dial tone (Axioms 20, 20a and 20b) telephone states; the rest can easily be generated by the method suggested in Sub-section 6.2.4.

Once a call is in progress, the caller may hang-up before waiting for the connection to complete or if no-one answers the callee telephone (Axiom 10). If, on the other hand, someone lifts the handset of the callee telephone, then the connection is established (Axiom 11). Finally, if the caller hangs-up, the connection is severed, and the callee telephone is sent a dial tone (Axiom 12). The callee may also hang-up in the middle of a connection, but since in some systems this only has the effect of suspending, but not terminating, the connection, a separate rule for it was not given.

Let us end the example by exploring the effects of adding a new feature to the basic POTS service. The feature is call-forwarding busy line (CFBL): if a telephone has the CFBL feature enabled and receives a call when it is busy, then the call will be forwarded to another telephone, one that the user has already defined. Obviously, this new feature will conflict with Axiom 8 concerning the behaviour of reaching a busy telephone. To incorporate this feature, the following elements and axioms must be added to the theory:

PREDS:	Cfbl_enbl:	⟨tel⟩:
	Cf_tone:	⟨tel⟩:
	Normcfbl:	⟨⟩:
ACTS:	snd_cftn:	⟨tel⟩;
FUNCTS:	forward:	⟨tel⟩ → tel;

8a except(Cfbl_enbl(y),¬Normbusy)

8b **K**[t](Off_hook(y) ∨ Ringing(y)) ∧ **K**[t]Cfbl_enbl(y)
∧**K**[t]Perf(ex,connect(x,y)) ∧ **L**[t]Normcfbl
→ **K**[t+MAXTRANS](Perf(ex,snd_cftn(x))
∧Perf(ex,connect(x,forward(y))))

8c **K**[t]Perf(ex,snd_cftn(x)) → **K**[t+MAXRESP]Cf_tone(x).

No axioms in the original theory need to be modified. Axiom 8a clips Axiom 8; Axioms 8b and 8c define the desired behaviour. In particular, if telephone y has CFBL enabled, then the exchange will attempt to connect x to the telephone indicated by the forward(y) function, while notifying the caller, via a Cf_tone, that the call is being forwarded. One possible abnormal condition for Axiom 8b is a forwarded call that results in a loop: the simplest is forward(y)=y, but since a forwarded call may be forwarded again, the system must check for larger loops as well. Other features, such as call-waiting, can similarly be added without modifying the original theory □

Specification *P-O Telephone System*

Language:

```
SORTS:      time, act, agt, tel;

CONSTS:     ex:  agt;
            MAXRESP,MAXTRANS,MAXDIAL: time;

VARS:       t:  time;
            u:  agt; /* telephone users */
            x,y:  tel;

PREDS:      Perf:           ⟨agt,act⟩;
            Hooked_up:      ⟨tel⟩;
            Off_hook:       ⟨tel⟩;
            Dialtone:       ⟨tel⟩;
            Ringing:        ⟨tel⟩;
            Ringback:       ⟨tel⟩;
            Busytone:       ⟨tel⟩;
            Hangup_msg:     ⟨tel⟩;
            Calling:        ⟨tel,tel⟩;
            Connected:      ⟨tel,tel⟩;
            Normcallinit:   ⟨⟩; /* normal call initiation condition */
            Normdial:       ⟨⟩; /* normal dialing condition */
            Normcall:       ⟨⟩; /* normal calling condition */
            Normresp:       ⟨⟩; /* normal phone answer condition */
            Normbusy:       ⟨⟩; /* normal busy phone condition */
            Norm_on:        ⟨⟩;
            Norm_off:       ⟨⟩;

ACTS:       lift:           ⟨tel⟩;
            hang_up:        ⟨tel⟩;
            dials:          ⟨tel,tel⟩;
            connect:        ⟨tel,tel⟩;
            sgnl_timeout:   ⟨tel,time⟩;
            snd_dialtn:     ⟨tel⟩;
            snd_ringbk:     ⟨tel⟩;
            snd_ringbl:     ⟨tel⟩;
            snd_busytn:     ⟨tel⟩;
            snd_hangup:     ⟨tel⟩;
```

Figure 6.2: POTS Specification: Language Declarations

Axioms:

```
0   K[⊥]¬Off_hook(x)

1   K[t]Off_hook(x) ∧ K[t]Perf(u,hang_up(x))
          → K[t+1]¬Off_hook(x)
2   K[t]¬Off_hook(x) ∧ K[t]Perf(u,lift(x))
          → K[t+1]Off_hook(x)

3   K[t](¬Off_hook(x) ∧ ¬Ringing(x)) ∧ K[t]Perf(u,lift(x))
         ∧L[t]Normcallinit
          → K[t+MAXRESP]Perf(ex,snd_dialtn(x))
3a  except(¬Hooked_up(x),¬Normcallinit)

4   K[t]Dialtone(x) ∧ K[t]Perf(u,dials(x,y))
         ∧L[t]Normdial
          → K[t+MAXRESP]Perf(ex,connect(x,y))
4a  except(Perf(ex,sgnl_timeout(x,MAXDIAL)),¬Normdial)
5a  K[t]Perf(ex,sgnl_timeout(x,MAXDIAL))
          → K[t+MAXRESP]¬Dialtone(x)
5b  K[t]Perf(ex,sgnl_timeout(x,MAXDIAL))
          → K[t+MAXTRANS]Perf(ex,snd_hangup(x))

6   K[t]Perf(ex,connect(x,y)) → K[t+MAXRESP]¬Dialtone(x)
7   K[t](¬Off_hook(y) ∧ ¬Ringing(y)) ∧ K[t]Perf(ex,connect(x,y))
         ∧L[t]Normcall
          → K[t+MAXTRANS](Calling(x,y)
              ∧Perf(ex,snd_ringbl(y)) ∧ Perf(ex,snd_ringbk(x)))
7a  except(Perf(u,hang_up(x)),¬Normcall)

8   K[t](Off_hook(y) ∨ Ringing(y)) ∧ K[t]Perf(ex,connect(x,y))
         ∧L[t]Normbusy
          → K[t+MAXTRANS]Perf(ex,snd_busytn(x))
9   K[t]Busytone(x) ∧ K[t]Perf(u,hang_up(x))
          → K[t+MAXRESP]¬Busytone(x)
```

Figure 6.3: POTS Specification: Axioms

```
10  K[t](Calling(x,y) ∧ Ringback(x)) ∧ K[t]Perf(u,hang_up(x))
        → K[t+MAXRESP](¬Calling(x,y)
            ∧ ¬Ringing(y) ∧ ¬Ringback(x))

11  K[t](Calling(x,y) ∧ Ringing(y)) ∧ K[t]Perf(u,lift(y))
       ∧L[t]Normresp
        → K[t+MAXRESP](Connected(x,y)
            ∧ ¬Calling(x,y) ∧ ¬Ringing(y)) ∧ ¬Ringback(x))

12  K[t](Connected(x,y) ∧ K[t]Perf(u,hang_up(x))
        → K[t+MAXRESP](¬Connected(x,y) ∧ Perf(ex,snd_dialtn(y)))

13  K[t]Perf(ex,snd_dialtn(x)) → K[t+MAXRESP]Dialtone(x)
14  K[t]Perf(ex,snd_ringbl(x)) → K[t+MAXRESP]Ringing(x)
15  K[t]Perf(ex,snd_ringbk(x)) → K[t+MAXRESP]Ringback(x)
16  K[t]Perf(ex,snd_busytn(x)) → K[t+MAXRESP]Busytone(x)
17  K[t]Perf(ex,snd_hangup(x)) → K[t+MAXRESP]Hangup_msg(x)

persistences
18  K[t]Off_hook(x) ∧ L[t]Norm_off → K[t+1]Off_hook(x)
18a K[t]Perf(u,hang_up(x)) → K[t]¬Norm_off

19  K[t]¬Off_hook(x) ∧ L[t]Norm_on → K[t+1]¬Off_hook(x)
19a K[t]Perf(u,lift(x)) → K[t]¬Norm_on.

20  K[t]Dialtone(x) ∧ L[t]Normdial → K[t+1]Dialtone(x)
20a K[t]Perf(ex,connect(x,y)) → K[t]¬Normdial.
20b K[t]Perf(ex,sgn_timeout(x,MAXDIAL)) → K[t]¬Normdial.

    ⋮
```

End.

Figure 6.1: POTS Specification: Axioms, continued

Chapter 7

Conclusions and Future Considerations

The fringed curtains of thine eye advance
And say what thou seest yond.
-**Shakespeare**, *The Tempest*, Act I, Scene II.

The main goal of this thesis has indeed been achieved: *KAT* is a specification language that is capable of tolerating certain forms of incompleteness in specifications. The positive result, however, was actually an unexpected one, since initially it did not seem possible to reconcile the counter-intuitive nature of non-monotonic logics with the rigorous requirements of a useful formal specification language. Though not perfect—does such a marvel even exist?—the method manifests some very nice properties. Particularly promising is the possibility of applying its underlying principles to other object languages. This, the final, chapter of the thesis takes stock of what has been accomplished. It briefly summarizes and evaluates the steps and key points in the development of the ideas, compiles the features and weaknesses of the method, and suggests areas of further study.

7.1 Summary of Thesis

Any serious research effort rarely achieves its goals in a straightforward manner (*cf* [83]). Only in retrospect can an intuitively direct path be discerned; during the

research, however, one flits from one promising avenue to another, often finding a cúl-de-sac at the other end. This thesis is no exception: the development of the ideas are presented in a fairly sensible sequence, but many of them were explored in a very different order. Nevertheless, the most direct, though illusory, path represents perhaps the best way to explain the ideas.

The original goal was essentially arrive at a useful definition of incompleteness in specification, and to propose a formal method for handling or tolerating it. The method would be used to specify real-time concurrent systems.

The research began with an examination of the notions, both formal and informal, of incompleteness in specifications. Some weakened notions were also considered. The obvious, though hardly illuminating, conclusions were that incomplete specifications basically lack information and that complete specifications are virtually impossible, if not inadvisable, to attain. The practical definition was in the form of a classification of incompleteness based on the potential sources of incompleteness. It served its purpose quite well, allowing us to identify those sources that can be eliminated, those that can be minimized, and those that are unavoidable. It also allowed us to select the class of partial specifications, which consists of three distinct problems—the qualification, frame and ramification problems—as the focus of the method. Partial specification was chosen for the following reasons: it is a fairly common source of incompleteness; it presents a non-trivial challenge, particularly for the specification of concurrent systems; and dealing with the qualification, frame and ramification problems poses too great a burden on the specifier. Specifying everything, as is required by traditional specification languages, is neither a practical nor a correct approach for dealing with this source of incompleteness. Instead, a method is needed that can tolerate it. To properly deal with partial specification, the language would have to capture the notion that "all the relevant facts have been explicitly mentioned."

Such a notion also happens to appear in the field of common-sense reasoning in AI. Moreover, the qualification, frame and ramification problems also crop up in a related field: in temporal reasoning and the temporal projection problem in AI. What they all have in common is the problem of having to deal with incomplete information. This observation then led to the idea of applying the techniques used in AI which deal with

incomplete information to the problem of partial specification. Three very common non-monotonic logics were studied – default, autoepistemic and circumscription – but none were deemed suitable for the formal basis of the language. Their two main drawbacks are difficulty in determining the consequences of a theory and weak or multiple extensions. Furthermore, they are inherently incapable of handling the temporal projection problem. There exist, however, other non-monotonic formalisms that can handle this problem, most of which employ a chronological minimality criterion on the models of the theory.

This led to a search of the various methods capable of capturing this criterion, and of these, Shoham's logic of chronological ignorance and the general approach of semantical non-monotonicity were chosen. Their main advantages over the rest include a general semantical framework, no dependency on a sequential (and atomic action) representation language, and a simple and efficient means of computing the consequences of a theory. Nevertheless, the logic has a number of shortcomings: it was not designed as a specification language, it is propositional, employs reification, does not deal with actions explicitly, nor handles the frame problem in a simple manner. All of these problems were addressed by the specification language *KAT* and its formal bases *kFOTK* and *kFOCI*.

7.2 Features of KAT

Despite its non-standard semantics and epistemic modalities, *KAT* is fairly intuitive both for expressing system behaviour and to use, as the examples have evinced. Its key features are the following:

- It has a well-defined syntax and semantics, and a method for determining what is semantically entailed by a theory:

  syntax the syntax is that of *kFOTK* – a first-order, many-sorted, non reified, modal logic of temporal knowledge – whose sentences are restricted to those prescribed by p-causal theories: it has three pre-defined sorts: time, action and agent:

semantics the semantics is that of *kFOCI*, a non-monotonic preference logic of *kFOTK* employing a chronological ignorance criterion; and

inference mechanism the inferences of a p-causal theory are derived by determining the theory's *cmi* models.

- Specifications in *KAT* are p-causal theories, which enjoy some nice properties: they are consistent and each has a unique class of preferred models.
- Normal p-causal theories, which are made up of normal form causal and persistence rules and their exceptions, are readily modifiable: most changes to the theory will be merely additions, while the original theory is left intact.
- Properly instantiated p-causal theories are potentially executable.
- It permits the specification of three types of information for the system: static information, action descriptions and dynamic behaviour.
- It can handle the qualification and frame problems.
- It can be used for the specification of reactive systems, including concurrent systems, without having to resort to atomic actions.

Its questionable aspects include the following:

- It is a non-standard formalism employing non-intuitive modalities, making it perhaps difficult to learn and use; given industry's reluctance to use even traditional formal methods, the chances of *KAT* being used are slim.
- The use of epistemic modalities in causal reasoning is perhaps philosophically dubious.
- The use of explicit time intervals, together with the general rule-and-exception framework, may not capture the fundamental nature of the system; implementation bias may result from this.
- Its effective use in the specification of real-time and/or concurrent systems has not really been established.

Given that *KAT*, a non-monotonic specification language, is unique, there do not exist precedents with which to compare it. One that remotely applies is WATSON [36]. WATSON is an automatic programming system that uses a weak 2-tense temporal logic for describing process behaviour and a closed-world assumption over a knowledge base. A simple non-monotonicity arises from the latter, but its developers view this as a nuisance, a by-product of the theorem prover's reasoning mechanism. It is not used, for example, to allow non-monotonic changes to the theory. As was stated in Sub-section 4.2.4, it is not difficult to attain non-monotonicity computationally, of which WATSON is another example. *KAT*, however, achieves non-monotonicity through the language itself (the semantics), and uses it for different reasons.

7.3 Future Considerations

There are a number of possible avenues to pursue. The first one deals with the language itself. How can it be made easier to use? A set of guidelines, for example, on how to use the modalities would be quite helpful. How much information is required to achieve partial completeness? Can such a notion even be defined? Can the ramification problem be addressed within the same framework? What other forms of incompleteness can be addressed? Are there other causal theories with nice properties? Can a more expressive language be defined? Is it possible to define at least a rudimentary proof theory for the language? From a more practical side, an interpreter for the language is needed. In this regard, one may try to adapt Shoham's algorithm [68], or perhaps use a logic programming language [63]. A more interesting option, however, would be a specification environment built around the language: in addition to providing an interpreter and editor, it could have a consistency checker.

Another avenue is to apply the notion of semantical non-monotonicity to other object languages. Addressing one of the drawbacks above, perhaps the choice would be for a logic that did not use time explicitly. Of course, this would require a different preference criterion than chronological ignorance. This, in turn, suggests exploring different criteria. What types are suitable for a specification language? And, would a specification formalism intended for reactive systems necessarily have a different

criterion from one intended for simple data processing systems? Further speculation might involve the types of languages that can be used. The approach, unfortunately, requires a model-based semantics. How much can this be relaxed? Can, for example, process algebras be used? It does not seem likely, unless (say) CCS trees can be viewed as models for a higher level specification language (*cf.* [31]). Still another possible option is to attempt to define a specification language based around some entirely different non-monotonic formalism. Possible candidates include any formalism capable of handling the temporal projection problem.

One final avenue concerns mapping the specifications in *KAT* to other languages: specifically, to concurrency modeling ones. Given the forms of the sentences allowed in *KAT*, and the way actions are defined in them, it seems that the best match would be with a partial order formalism [62]. Both can describe partial orders of actions, but not nondeterministic choice (except where it appears as a set of alternate behaviours). One possible benefit of such a mapping (and the one to CCS above) is that it might allow us to preview the potential effects that changing requirements (as seen through changes to *KAT* specifications) have on the implementations, descriptions written in these (operational) concurrency formalisms being considered lower level descriptions of the system.

Breaking new ground invariably unearths a wealth of new research possibilities: in such cases, one is usually prompted to declare (humbly) that further study is clearly warranted. Make it so...

Bibliography

[1] J.F. Allen, "Towards a General Theory of Action and Time," *Artificial Intelligence*, **23**, 1984, pp. 123-154.

[2] F. Bacchus, J. Tenenberg, and J.A. Koomen, "A Non-Reified Temporal Logic," *Artificial Intelligence*, **52**, 1991, pp. 87-108.

[3] A.B. Baker, "A Simple Solution to the Yale Shooting Problem," *Proc. Intl. Conf on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann Publishers, Inc., 1989, pp. 11-20.

[4] H. Barringer, J.H. Cheng, and C.B. Jones, "A Logic Covering Undefinedness in Program Proofs," *Acta Informatica*, **21**, 1984, pp. 251-269.

[5] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, **27** (1), January 1984, pp. 42-52.

[6] B.W. Boehm and P.N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Trans. on Software Engineering*, **4** (10), October 1988, pp. 1462-1477.

[7] F.P. Brooks, Jr, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley Publishing Co., 1975.

[8] F.P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, **20** (4), April 1987, pp. 10-19.

[9] R.A. Brooks, "Intelligence Without Representation," *Artificial Intelligence*, **47**, 1991, pp. 139-160.

[10] R.A. Brooks, "Intelligence Without Reason," AI Memo No. 1293, MIT AI Laboratory, Massachusetts Institute of Technology, 1991.

[11] B. Cohen, W.T. Harwood and M.I. Jackson, *The Specification of Complex Systems*, Addison-Wesley, 1986.

[12] J.L. Connell and L.B. Shafer, *Structured Rapid Prototyping: An Evolutionary Approach to Software Development*, Prentice-Hall, Inc., 1989.

[13] E. Davis, *Representations of Commonsense Knowledge*, Morgan Kaufmann Publishers, Inc., 1990.

[14] J. DeTreville, "Phoan: An Intelligent System for Distributed Control Synthesis," *SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software Development Environments*, ACM, 1984, pp. 96–103.

[15] D.W. Etherington, *Reasoning with Incomplete Information*, Pitman, 1988.

[16] M. Faci, L. Logrippo, and B. Stepien, "Formal Specification of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach," Report of the Protocols Research Group, Dept. of Computer Science, University of Ottawa (Canada).

[17] R. Fairley, *Software Engineering Concepts*, McGraw-Hill Inc., 1985.

[18] A. Galton, "Reified Temporal Theories and How to Unreify Them," in: *Proc. IJCAI*, 1991, pp. 1177–1182.

[19] A. Galton, "A Critique of Yoav Shoham's Theory of Causal Reasoning," in: *Proc. AAAI*, 1991, pp. 355–359.

[20] D.C. Gause and G.M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House Publishing Co., 1989.

[21] M.L. Ginsberg and D.E. Smith, "Reasoning about Action I: A Possible Worlds Approach," *Artificial Intelligence*, **35**, 1988, pp. 165–195.

[22] J.V. Guttag and J.J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, **10**, 1978 pp. 27–52.

[23] A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, September 1990, pp. 11–19.

[24] J.Y. Halpern, "Reasoning About Knowledge," in: *Theoretical Aspects of Reasoning About Knowledge*, Morgan Kaufmann Publ., 1986.

[25] J.Y. Halpern and Y. Moses, "A Guide to Modal Logics of Knowledge and Belief: Preliminary Draft," in: *Proc. IJCAI*, 1985, pp. 480–490.

[26] S. Hanks and D. McDermott, "Nonmonotonic Logic and Temporal Projection," *Artificial Intelligence*, **13**, 1980, pp. 379–412.

[27] B.A. Haugh, "Simple Cau[illegible]l Minimizations for Temporal Persistence and Projection," in: *Proc. AAAI*, 1987, pp. 218–223.

[28] B.A. Haugh, "Non-Standard Semantics for the Method of Temporal Arguments," in: *Proc. IJCAI*, 1987, pp. 449–455.

[29] P.J. Hayes, "The Second Naive Physics Manifesto," in: *Formal Theories of the Commonsense World*, Ablex Publishing Corp., 1985, pp. 1–36.

[30] P.J. Hayes, "Naive Physics I: Ontology For Liquids," in: *Formal Theories of the Commonsense World*, Ablex Publishing Corp., 1985, pp. 71–107.

[31] M. Hennessy and R. Milner, "Algebraic Laws for Nondeterminism and Concurrency," *Journal of the ACM*, **32** (1), 1985, pp. 137-161.

[32] J. Hintikka, "Semantics for Propositional Attitudes," in *Reference and Modality*, L. Linsky (ed.), Oxford University Press, 1971, 145–167.

[33] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall International (UK) Ltd., 1985.

[34] G.E. Hughes and M.J. Cresswell, *An Introduction to Modal Logic*, Methuen and Co. Ltd., 1968.

[35] C.B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall International Ltd., 1986.

[36] V.E. Kelly and U. Nonnenmann, "Inferring Formal Software Specifications From Episodic Descriptions," *Proc. AAAI*, 1987, pp. 127–132.

[37] S. Khosla and T.S.E. Maibaum, "The Prescription and Description of State Based Systems," in: *Temporal Logic in Specification*, LNCS 398, Springer-Verlag, 1989, pp. 243–294.

[38] D.E. Knuth, "The Errors of TeX," *Software-Practice and Experience*, **19** (7), July 1989, pp. 607–687.

[39] K. Konolige, "On the Relation between Default and Autoepistemic Logics," *Artificial Intelligence*, **35**, 1988, pp. 343–382.

[40] S.A. Kripke, "Semantical Considerations on Modal Logic," in *Reference and Modality*, L. Linsky (ed.), Oxford University Press, 1971, pp. 63–72.

[41] T.S. Kuhn, *The Structure of Scientific Revolutions*, (2ed.), The University of Chicago Press, 1970.

[42] A. Lee, "Formal Specification of a Basic Telephone Exchange," Technical Report No. 155, Key Centre for Software Technology, Dept. of Computer Science, University of Queensland (Australia), 1990.

[43] V. Lifschitz, "On the Satisfiability of Circumscription," *Artificial Intelligence*, **28**, 1986, pp. 17–27.

[44] V. Lifschitz, "Formal Theories of Action: Preliminary Report," in: *Proc. IJCAI*, 1987, pp. 966–972.

[45] V. Lifschitz, "Pointwise Circumscription," in: *Readings in Nonmonotonic Reasoning*, Morgan Kaufmann Publishers, 1988, pp. 179–193.

[46] S-T. Levi and A.K. Agrawala, *Real Time System Design*, McGraw Hill Publishing Co., 1990.

[47] W. Lukaszewicz, *Non-Monotonic Reasoning: Formalization of Commonsense Reasoning*, Ellis Horwood Ltd., 1990.

[48] G.L. McArthur, "Reasoning About Knowledge and Belief: A Survey," *Computational Intelligence*, **4**, 1988, pp. 223–243.

[49] J. McCarthy, "Circumscription – A Form of Non-Monotonic Reasoning," *Artificial Intelligence*, **13**, 1980, pp. 27–39.

[50] J. McCarthy, "Applications of Circumscription to Formalizing Commonsense Knowledge," *Artificial Intelligence*, **28**, 1986, pp. 89–116.

[51] J. McCarthy and P.J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in: *Machine Intelligence 4*, B. Meltzer, D. Michie (eds.), American Elsevier, 1969, pp. 463–502.

[52] D. McDermott, "A Temporal Logic for Reasoning About Processes and Plans," *Cognitive Science*, **6**, 1982, pp. 101–155.

[53] D. McDermott, "A Critique of Pure Reason," *Computational Intelligence*, **3**, 1987, pp. 151–160.

[54] R. Milner, *Concurrency and Communication*, Prentice-Hall International Ltd., 1989.

[55] M. Minsky, *The Society of Mind*, Simon & Schuster, Inc., 1988.

[56] R.C. Moore, "A Formal Theory of Knowledge and Action," in: *Formal Theories of the Commonsense World*, Ablex Publishing Corp., 1985, pp. 319–358

[57] R.C. Moore, "Semantical Considerations on Nonmonotonic Logic," *Artificial Intelligence*, **25**, 1985, pp. 75–94.

[58] H. Moravec, *Mind Children*, Harvard University Press, 1988.

[59] L. Morgenstern and L.A. Stein, "Why Things Go Wrong: A Formal Theory of Causal Reasoning," in: *Proc. AAAI*, 1988, pp. 518-523.

[60] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," in: *Current Trends in Concurrency*, LNCS 224, Springer-Verlag, 1986, pp. 510–584.

[61] K.R. Popper, *The Logic of Scientific Discovery*, 9ed., Hutchinson, 1977.

[62] V.R. Pratt, "Modelling Concurrency with Partial Orders," *Intl. Journal of Parallel Programming*, **15** (1), 1987, pp. 33-71.

[63] T.C. Przymusinski, "Non-monotonic Reasoning Versus Logic Programming: A New Perspective," in *The Foundations of Artificial Intelligence*, D. Partridge and Y. Wilks (eds.), Cambridge University Press, 1990, pp. 49–71.

[64] A. Ramsay, *Formal Methods in Artificial Intelligence*, Cambridge University Press, 1988.

[65] R. Reiter, "A Logic for Default Reasoning," *Artificial Intelligence*, **13**, 1980, pp. 81-132.

[66] N. Rescher and A. Urquhart, *Temporal Logic*, Springer-Verlag, 1971.

[67] J.S. Schlipf, "How Uncomputable is General Circumscription?" in: *IEEE Symp. on Logic in Computer Science*, 1986, pp. 92-95.

[68] Y. Shoham, *Reasoning About Change*, Massachusetts Institute of Technology, 1988.

[69] Y. Shoham, "Time for Action," in: *Proc. IJCAI*, 1989, pp. 954–959, 1173.

[70] Y. Shoham, "Nonmonotonic Reasoning and Causation," *Cognitive Science*, **14**, 1990, pp. 213-252.

[71] Y. Shoham and N. Goyal, "Temporal Reasoning in Artificial Intelligence," in: *Frontiers of Artificial Intelligence*, Morgan Kaufmann Publishers, 1988, pp. 419-438.

[72] J. Stillman, "It's Not My Default: The Complexity of Membership Problems in Restricted Propositional Default Logics," in: *Proc. AAAI*, 1990, pp. 571-578.

[73] W.M. Turski and T.S.E. Maibaum, *The Specification of Computer Programs*, Addison-Wesley, 1987.

[74] H. Würges, "A Specification Technique Based on Predicate Transformers," *Acta Informatica*, **15**, 1981, pp. 425-445.

[75] *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R.C. Waters (eds.), Morgan Kaufmann Publishers, Inc., 1986.

[76] *Formal Theories of the Commonsense World*, J.R. Hobbs and R.C. Moore (eds.), Ablex Publishing Corporation, 1985.

[77] "Taking Issue / Forum: A Critique of Pure Reason," H. Levesque (ed.), *Computational Intelligence*, **3**, 1987, pp. 149-237.

[78] *The Robot's Dilemma*, Z. Pylyshyn (ed.), Ablex Publishing Corp., 1987.

[79] *Handbook of Mathematical Logic*, J. Barwise (ed.), North-Holland Publishing Co., 1977.

[80] *Readings in Nonmonotonic Reasoning*, Morgan Kaufmann Publishers, 1988.

[81] *Proc. 2nd Int. Workshop on Non-Monotonic Reasoning*, LNAI 346, M. Reinfrank (ed.), Springer-Verlag, 1989.

[82] *New Paradigms for Software Development*, W.W. Agresti (ed.), IEEE Computer Society Press, 1986.

[83] *Introductory Readings in the Philosophy of Science*, E.D. Klemke *et al* (eds.), Prometheus Books, 1988.

[84] *Temporal Logic in Specification*, LNCS 398, B. Banieqbal *et al* (eds.), Springer-Verlag, 1989.