ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Gerard O'Regan and Sharon Flynn (Eds)

1st Irish Workshop on Formal Methods

Proceedings of the 1st Irish Workshop on Formal Methods, Dublin 3-4
July 1997

Paper:

Nonmonotonic Reasoning and Formal Software Development

Edel Sherratt

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Nonmonotonic Reasoning and Formal Software Development

Edel Sherratt

Department of Computer Science, University of Wales Aberystwyth
Aberystwyth, Ceredigion, Wales

Abstract

Formal nonmonotonic reasoning provides a promising basis for meeting the demands inherent in the incomplete, evolving and inconsistent requirements that characterise large-scale industrial software systems. This paper explores formal reasoning in requirements analysis, design and implementation. It indicates how the notion of provability in maximal consistency logics might be used to extend current logical foundations for existing formal notations for specifying requirements. A number of definitions are then developed, using notions from level default theories to specify preference between consistent subsets of possibly inconsistent requirements, (partial) satisfaction of requirements by programs and by designs, the best subset of the requirements satisfied by a given design, implementation of a design and design improvement. These definitions provide a foundation for some very interesting directions for formal methods research.

Keywords: Formal methods, software engineering, nonmonotonic reasoning, maximal consistency logics

1 Introduction

Formal methods facilitate reasoning in software engineering. They do this by providing formal systems in which problems can be modelled using clear, succinct notations, and inferences can be drawn using appropriate deductive apparatus[12].

If a set of requirements on a software system is expressed as a formal axiom set, it is, in principle, possible to prove that a given implementation meets those requirements. Better still, constructive methods like refinement calculus[9, 21] support simultaneous construction of a program and its proof. This is excellent for proving that a particular piece of software meets a given set of requirements, provided the requirements are consistent, and provided new requirements do not invalidate the proof. These conditions can usually be met for functional requirements on a software system, and for certain nonfunctional constraints, such as timing constraints.

However, in general, requirements are inconsistent, incomplete and volatile, and well engineered software represents an acceptable compromise between conflicting demands for elegance, portability, robustness, low cost, high execution speed, low use of space, rapid time to market and amenability to changes in the user environment and in the target environment in which the software operates. Because they are inherently inconsistent, these requirements cannot be expressed using notations whose meaning is only defined for consistent axiom sets. Because they are at least as important as functionality in determining end-user satisfaction, they must be taken into account in any practical software development method.

Now reasoning where information is incomplete, inconsistent or subject to change has been investigated by the artificial intelligence community since the 1960s[14], and that has resulted in nonmonotonic logics[22, 10, 3, 11, 1, 17]. These logics are called nonmonotonic because, unlike classical logics, adding new axioms to an existing theory may force the withdrawal of previous conclusions.

Existing work relating nonmonotonic formalisms with software engineering includes analysis of the need for nonmonotonicity in requirements analysis[23], a mathematical investigation aimed at establishing declarative logic programming as a viable software engineering paradigm, with emphasis on the impact of modifications to requirements specifications[16], and theoretical results relating domain theory with default logic[5, 6].

This paper explores how maximal consistency logics[10] can enable existing formal notations to support reasoning in a general, conventional software development process. The areas addressed are specification of requirements, identification of a satisfaction relation between formal specifications of requirements and formal implementations, and establishment of that relationship by improving an initial design with respect to the requirements. The exploration

results in a collection of definitions that form a basis for extending the semantics of existing formal notations to support formal software construction and improvement where requirements are inconsistent, incomplete or volatile. This complements the work of [16], which emphasises the declarative paradigm, and extends the work on software requirements presented in [23].

2 Formal Specification of Requirements

Conventionally, a specification of requirements is viewed both as a contract to produce a given software artifact, and as a standard against which the artifact can be evaluated. Depending on the software development process enacted, the specification of requirements can be the result of a single requirements analysis and specification phase, or it can be developed incrementally, with corresponding releases of software increments. This last approach is favoured where there is a significant element of risk or uncertainty in a software project [2], or where time to market is of overriding importance [4].

A formal requirements specification characterises statements about a proposed software system using a formal language, which has an associated reasoning or inference mechanism [24]. Methods like CR&F [18] facilitate derivation of formal specifications of requirements from less formal, structured, models.

Formal requirements specifications can be analysed for consistency [24]. This, together with the clarity of thought required when writing formal statements of requirements, supports the identification of areas where the understanding of requirements is lacking. Moreover, a formal statement of requirements provides an excellent basis for stabilising requirements [7] and for developing a contract to be fulfilled by a software system.

However, while good practice reduces incompleteness, inconsistency and instability in requirements specifications, is neither feasible nor desirable to eliminate these characteristics altogether [23, 24]. For example, a specification could easily be dominated and obscured by axioms defining environmental characteristics to be preserved when a new software system is installed, or axioms defining the social context [13] in which requirements are situated.

Moreover, it is often reasonable to construct part of a software system before the requirements for the system as a whole have been finalised. In fact, delaying development until requirements have been finalised may well result in the ousting of a good software system by an inferior system which became available sooner. Of course, in order to contain the risk of this strategy, it is important to formulate a suitable contract for incremental development.

As well as being incomplete, requirements must change to meet changing environmental conditions. This has been recognised for many years [19], and has resulted in the growth of a thriving software maintenance industry. Good engineering practice entails stabilisation of requirements for each release, with changes being held over until the next release. However, this is not always practicable as requirements can be invalidated between releases by conditions beyond the control of the engineers responsible for developing or maintaining a software system.

To meet these demands, any formal system for specifying software requirements must enable description of possibly inconsistent requirements, and formulation of proofs based on consistent subsets of those requirements. It is highly desirable that this should be done using existing notations – for example, Z, VDM, OBJ – which implies that the deductive apparatus provided by a formal system should include classical first order inference. Specifically, a formal system is required that supports

- specification of possibly inconsistent requirements,
- identification of consistent subsets of those requirements,
- rational selection between subsets of requirements,
- construction of proofs based on consistent subsets of requirements.

Maximal consistency logics partly meet these needs by allowing premises to be expressed using classical logical formulations, by defining maximal consistent subsets of premises, and by defining provability from possibly inconsistent premises in terms of provability from maximal consistent subsets of those premises. The following definitions,

from[10], are used here to enable formal specification of inconsistent requirements.

Definition 1 Let T be a (possibly inconsistent) set of premises. $SUB(T)$ is a maximal consistent subset of T iff

1. $SUB(T) \subseteq T$,
2. $SUB(T)$ is consistent, and
3. If $SUB(T) \subset SUB'(T) \subseteq T$, then $SUB'(T)$ is not consistent

Definition 2 Let T be a (possibly inconsistent) set of premises, $MaxSub(T)$ the set of maximal consistent subsets of T , and $\sqsubset$ a strict partial order on $MaxSub(T)$. The formula p is strongly (weakly) provable from $(T, \sqsubset)$ iff p is provable from every (at least one) $\sqsubset$ -minimal element of $MaxSub(T)$.

This allows us to use existing formal notations to specify possibly inconsistent requirements and assumptions, and then to define maximal consistent subsets of these to be used in further formal software development. For example, consider the following requirements for a hotlist of useful web url's.

A hotlist of url's accessible using suitable keywords is to be developed.

The hotlist will allow a user to insert, retrieve and delete url's as well as to access the resources specified by url's.

The insert operation will be optimised for speed of execution.

The retrieve operation will be optimised for speed of execution.

The list, constructed by means of the insert operation, will be optimised for use of space.

An existing formal notation – for example, Z or OBJ – could be used to create a theory containing axioms that defined

1. the types **hotlist**, **url**, **keyword**,
2. and the operations *insert*, *retrieve*, *delete*, and *apply_url*.

The same notation could also be used to write axioms like

A1 *speed_optimal(insert)*

A2 *speed_optimal(retrieve)*

A3 *space_optimal(insert)*

and to define the meanings of the predicates *speed_optimal* and *space_optimal*.

Now suppose it were found that any acceptable definition of these predicates meant that following theorems were derivable from consistent subsets of the requirements¹

Th1 *speed_optimal(insert) $\Rightarrow$ $\neg$ speed_optimal(retrieve)*

Th2 *speed_optimal(insert) $\Rightarrow$ $\neg$ space_optimal(insert)*

Th3 *speed_optimal(retrieve) $\Rightarrow$ $\neg$ space_optimal(insert)*

This is not an unlikely assumption; every time a B-tree representation is chosen over a singly linked list, insertion speed is compromised in order to improve retrieval speed.

Now this means that theory defining the requirements is inconsistent. Maximal consistent subsets of the requirements include the definitions of the types **hotlist**, **url**, **keyword**, the operations *insert*, *retrieve*, *delete*, and *apply url*, and the predicates *speed_optimal* and *space_optimal*, together with one of

- *speed_optimal(insert)*

¹For simplicity, *speed_optimal* and *space_optimal* are treated here as predicates which are either true or false in any interpretation. In practice, they would be specified as having measurable values, which could be traded off.

- $speed_optimal(retrieve), space_optimal(insert)$

However, although the definition of maximal consistency makes possible identification of subsets of the requirements that can, in principle, be met, it does not provide formal basis for choosing between alternative maximal consistent subsets of the requirements. This need can be met by level default theories, which partition possibly inconsistent premises into separate levels[10], with premises at any given level being regarded as being more reliable or more important than those at the next level. This idea is captured by the following definition[10].

Definition 3 Let $(T, \prec)$ be a (possibly inconsistent) theory with the partial ordering $\prec$ on its premises. Let $\prec_t$ be any total ordering that contains $\prec$. S is the preferred subtheory of T generated by $\prec_t$ iff

1. S is a maximal consistent subset of T , and
2. $p \in T$ and $\{p\} \cup \{q \in S \mid q \prec_t p\}$ is consistent implies $p \in S$

This provides an intuitively appealing approach to managing inconsistent requirements. If $\prec$ represents the priority associated with a given requirement, we can then prefer maximal consistent subsets of the highest priority requirements.

In the hotlist example, if retrieval speed is given a higher priority than insertion speed then maximal consistent subsets of the requirements containing

$speed_optimal(retrieve), space_optimal(insert)$

will be preferred.

However, since requirements are frequently incomplete, and since their relative importance may be unknown at any given point in software development, it will not always be feasible to construct preferred theories at the start of the software development process. For practical purposes, we need to be able to identify adequate theories, which then serve as specifications of requirements until better information becomes available.

Definition 4 Let $(T, \prec)$ be a (possibly inconsistent) theory with the partial ordering $\prec$ on its premises. Let $R1$ and $R2$ be subsets of T , and let $\prec_t$ be any total ordering containing $\prec$. Then $R2$ is preferred over $R1$ in $(T, \prec_t)$ if any of the following conditions hold:

1. $R2$ is consistent and $R1$ is not,
2. $R1$ and $R2$ are both consistent, and $R1 \subset R2$
3. $R1$ and $R2$ are both consistent, neither is a subset of the other, and if $p \in T$, $\{p\} \cup \{q \in R1 \mid q \prec_t p\}$ is consistent, $\{p\} \cup \{q \in R2 \mid q \prec_t p\}$ is consistent and $p \in R1$, then $p \in R2$

Thus, more nearly maximal consistent subsets of higher priority requirements are preferred. This enables reasonable choices to be made between different subsets of the requirements, even when the preferred theory is not readily identifiable.

3 Formal Implementation

A formal implementation includes a language with an operational semantics. For example, terms and term rewriting systems constitute formal implementations of functional languages, while state transforming mechanisms constitute formal implementations of imperative languages. A formal implementation serves as a basis for reasoning about some real implementation, and so must be valid with respect to the real implementation. That is, the real implementation must model the formal implementation.

As well as a language with an operational semantics, a formal implementation may also provide a quantitative description of the speed of execution of reduction steps or state transformations, or expression evaluations. This is essential if formal proof of real-time behaviour is required. In the case of state transforming mechanisms, a definition of execution speed could refer to the number of assignments or guard evaluations performed by the program. For term rewriting systems, speed of execution is defined by[8] as the minimum number of steps necessary to reduce an expression to a normal form. A similar definition of improvement in usage of space might refer to the maximum or mean size of any expression in the sequence of reductions.

The purpose of formal software development is to produce a program, expressed in a formal implementation language, whose operational interpretation models some formally specified requirements. This concept of operational correctness is extended in the following definition to take account of possibly inconsistent requirements whose meaning is defined in terms of a level default theory.

Definition 5 Let $(T, \prec)$ be a possibly inconsistent set of requirements prioritised by $\prec$. Let $\prec_t$ be a total ordering that contains $\prec$ and let R be the preferred subtheory of $(T, \prec)$ generated by $\prec_t$. Let M be a program with operational interpretation $\mathcal{I}(M)$. Then $M \prec_t$ -satisfies $(T, \prec)$ iff $\mathcal{I}(M) \models R$, read “ $\mathcal{I}(M)$ models R ”.

If $M \prec_t$ -satisfies $(T, \prec)$ for every total ordering $\prec_t$ containing $\prec$, then we say M strongly satisfies $(T, \prec)$; if $M \prec_t$ -satisfies $(T, \prec)$ for some $\prec_t$, then M weakly satisfies $(T, \prec)$.

In practice, a program that only satisfies some of its requirements might be acceptable. This is particularly true of programs for which time to market is critical[4]. Partial satisfaction of requirements is defined as follows.

Definition 6 Let $(T, \prec)$ be a possibly inconsistent set of requirements prioritised by $\prec$. A program M with operational interpretation $\mathcal{I}(M)$, partly satisfies $(T, \prec)$ iff

1. $\mathcal{I}(M) \models R$ for some consistent, nonempty $R \subseteq T$, and
2. if R' is a consistent subset of T , and $R \subseteq R'$, and $p \in (R' - R)$, $M \not\models \neg p$

where $(R' - R)$ stands for set difference.

4 Formal Design and Refinement

The task of creating programs that satisfy their requirements is conventionally achieved by an incremental process. Incremental software development processes — for example, stepwise refinement[19], spiral development[2] and some forms of rapid application development[4] — are characterised by repeated improvement of an initial design to achieve better and better approximations to a program that satisfies its requirements.

The language used to express initial and improved designs varies between methods, as does the meaning of improvements. For example, with stepwise refinement[19], designs are expressed using a mixture of implementation language constructs, natural language, and stylised pseudo-code. The initial design usually includes few, if any, operationally defined constructs, and improvement means replacing non-operational constructs by constructs of the formal implementation language. In the case of rapid application development, the initial design is actually an operational program that satisfies a (typically small) subset of the requirements. This initial program should be readily extensible; for example, it might be an empty pull-down menu. Each improvement results in an program that extends subset of satisfied requirements; for example, by implementing some menu options.

Formal methods can be used in the context of incremental processes to guide the development process, and to enable creation of proofs that programs satisfy their requirements. For example, classical refinement[21, 9] enables formal incremental derivation of a program and its proof from an initial relation between input and output states or values.

If a formal method is to guide an incremental development process, it must enable proof of design improvement with regard to formally specified requirements. If the method is to support simultaneous construction of programs together with proofs that requirements are satisfied, it must be possible to decompose the proof that a program satisfies given requirements into proofs about the relationships between the requirements, the initial design, any improved designs, and the program.

Definitions are developed in the following sections to cover the concepts of partial satisfaction of requirements by a design, of preference between designs, of design improvement and of the relationship between a design and a program that implements it. These all assume that the meaning of the notation used to specify requirements is defined in terms of maximal consistency logics, and so cater for inconsistent or changing requirements.

4.1 Designs, Requirements and Programs

A software design breaks a large software system into manageable parts. The granularity of the breakdown varies with the scope of the system being designed and with the design methods used. For the present purposes, it is assumed that a design breaks a proposed software system into abstract data type specifications or into object classes, depending on the

design method used. That is, a design defines subsystems that can be improved independently without compromising overall system improvement.

A design comprises one or more modules (that is, abstract data types or object classes), and allocates each functional requirement to a module. Modules might be expressed as object classes, if an object-oriented design approach were used, or they might be expressed as abstract data types. Different kinds of module are interface modules, which communicate directly with the environment in which the software being designed will operate, functional models, which provide the required functionality, and internal modules, which are used by each other, by the functional modules and by the interface modules.

Modules are characterised by their interfaces — message protocols in the case of object oriented designs, or signatures in the case of algebraic designs — and by the observable behaviour of these interfaces. Modules are related by containment, and by the communication which passes between them. This communication, whether it consists of function or procedure calls or of messages, defines the degree of cohesion and coupling within the software system.

In a typical development, some modules represent software subsystems for which implementations are already available, while others represent subsystems to be developed. The scale of modules can vary enormously. In particular, existing subsystems can be very large indeed[13], and a design may capture only parts of their interfaces.

Designs often contain simplifying assumptions about the requirements on a proposed software system. That is, they temporarily drop some requirements in order to focus attention on others. For example, it is often convenient to limit an initial design to modules that are defined meet the functional requirements on the system being developed. It is also common for designs to include partial functions, whose results are undefined for input outside their domains, and which therefore lack robustness, or functions that accept infinitely large arguments or deliver infinitely large results.

A design may also include information which goes beyond that contained in the requirements specification. For example, by defining modules and operations, a design determines the structure of the proposed software system, and may present a specific strategy, expressed in terms of operations or state transformations, for meeting the functional requirements on the proposed software system.

A design and a formal specification of requirements each constitute a theory about a software system to be developed. These two theories share a common subset of theorems, but, in general, the two theories will each have theorems not included in the other. For example, an initial design which focuses on the functional requirements on a new system will have as theorems the axioms specifying those functional requirements, and theorems about the structure of the system to be developed, but will not necessarily have as theorems many of the nonfunctional requirements on the new system.

For example, a design for the hotlist might include formal definitions of

- modules for the user interface and for the functions,
- functions `new_hotlist`, `add_url`, `retrieve_url`, `delete_url`, which allows construction of infinitely large hotlists,
- a definition for hotlist entries, where infinitely long names are allowed, and entries are always assumed to be valid
- a definition for the user interface, relating a button with each hotlist function,
- a type for buttons, giving each button a position, geometry and label,

but might not address the requirements for optimising retrieval or insertion speed, or use of space.

If a proof that a program satisfies its requirements is to be based on the argument that it implements a design which satisfies those requirements, then appropriate relationships between programs, designs and requirements must be defined.

Partial satisfaction of requirements by a design, and the best subset of the requirements satisfied by a given design are defined as follows.

Definition 7 Let $(T, \prec)$ be a possibly inconsistent statement of requirements, prioritised by $\prec$. Then a design D partly satisfies $(T, \prec)$ iff

1. $D \vdash R$, for some consistent, nonempty $R \subseteq T$, and
2. if R' is a consistent subset of T , and $R \subseteq R'$, and $p \in (R' - R)$, then $D \not\vdash \neg p$

where $D \vdash R$ stands for “the sentences R are first-order provable from the sentences D ”.

Definition 8 Let $(T, \prec)$ be a possibly inconsistent statement of requirements, prioritised by $\prec$. Let $\prec_t$ be a total ordering containing $\prec$. Suppose D is a design that partly satisfies $(T, \prec)$. R is the best subset of $(T, \prec_t)$ satisfied by D iff

1. $D \vdash R$
2. if $D \vdash R'$, then R is preferred over R' in $(T, \prec_t)$

An implementation relationship between designs and programs is also needed. The definition of an implementation relationship has the following form.

Definition 9 Let $(T, \prec)$ be a possibly inconsistent statement of requirements, prioritised by $\prec$. Let $\prec_t$ be a total ordering containing $\prec$. Let M be a program with operational interpretation $\mathcal{I}(M)$. Suppose D is a design that partly satisfies $(T, \prec)$, and suppose R is the best subset of $(T, \prec_t)$ satisfied by D . Then M implements D iff

1. $M \prec_t - \text{satisfies}(R, \prec)$
2. D and M are related by a well defined structural equivalence associating modules and interfaces specified by D with equivalent constructs in M .

To complete this definition, the structural equivalence between the implementation and design notations must be specified; for example, it might be defined as homomorphism. However, its exact form depends on the structuring mechanisms of the formal design and implementation languages, and is beyond the scope of the current exploration.

4.2 Formal Refinement

Starting with an initial design, formal refinement repeatedly improves the design until the requirements are satisfied, or until an acceptable subset of the requirements is satisfied. Where refinement is carried out in the context of a traditional software process – for example, a waterfall – only the final design is implemented as a program. If an incremental process is used, intermediate designs are also implemented.

In classical stepwise refinement (e.g.[19, 8]), the intermediate designs form a sequence in which each design is related by a refinement relationship with its predecessor. Thus, if $\mathcal{R}$ is the refinement relationship, D_0 is the initial design, and $D_1 \dots D_{n-1}$ the sequence of intermediate designs, culminating in the implementation S_n , then $S_0 \mathcal{R} \dots \mathcal{R} S_n$. The refinement relation is defined so as to preserve correctness, and, sometimes, so that later designs improve earlier designs. Of course, different parts of a design specification can also be refined in parallel, giving a cascade- or tree-like rather than a sequential structure to the pattern of refinement. In this case, the refinement relationship must hold between a node in the tree of refinements, and the specification formed by combining its descendants according to the structure of the parent.

Refinement can be carried out by hand, and this is the approach demonstrated, for example, in classical presentations of refinement calculus[9, 21]. Refinement can also be performed automatically, by application of formal transformations.

When refinement is carried out with a view to obtaining an executable implementation, some part the current specification is selected which is not already expressed in terms of the implementation language, and this part is replaced by a suitable implementation fragment. The replacement may consist of calls on newly introduced operations, which may entail adding new modules to the design, or it may consist of calls or implementation language fragments which can be interpreted according to the operational semantics of the implementation language without further refinement. In essence, this kind of refinement entails making a transition from an implicit, often axiomatic definition of behaviour

towards an explicit, operationally defined implementation of that behaviour, and the refined specification improves its predecessor insofar as it is more nearly expressed in terms of the formal implementation language.

Refinement is enhanced by the concept of formal improvement[8]. The improvement theorem presented in[8] shows that if an expression e is formally improved by e' , where e and e' are operationally equivalent, then replacing e by e' in any program results in a correct, improved program. Formal improvement is defined as reduction of the minimum number of steps required to reduce an expression to normal form according to a given operational semantics. This definition is then used as a basis for establishing results about the correctness and usefulness of automatic transformations at the level of individual operations.

However, a more general concept of formal improvement is needed if improvement is to be used as a basis for deciding when and how and at what level to refine a software system. For example, if retrievals were more frequent than insertions in the hotlist, then improvement might mean increasing the speed of retrievals at the expense of insertions, by, say, representing the hotlist as a balanced tree rather than a list.

This can be achieved by treating nonfunctional requirements as goals, and then selecting satisficing² (sub)goals to be achieved by refinement steps[15, 20].

Formal design improvement is defined in terms of preference for the best subsets of the requirements satisfied by the different designs.

Definition 10 *Let $(T, \prec)$ be a (possibly inconsistent) statement of requirements ordered by $\prec$. Let $\prec_t$ be any total ordering containing $\prec$. Suppose $D1$ and $D2$ are designs that partly satisfy $(T, \prec)$. Let $R1$ be the best subtheory of $(T, \prec_t)$ satisfied by $D1$, and $R2$ be the best subtheory of $(T, \prec_t)$ satisfied by $D2$. Then $D2$ improves $D1$ iff $R2$ is preferred over $R1$ in $(T, \prec_t)$.*

5 Conclusions and Directions

The demands placed on logical systems to support formal reasoning in requirements analysis, design and implementation have been explored, and some initial definitions of important concepts have been developed. This represents a small but significant step towards the application of formal software development methods where incompleteness and inconsistency are usual, and where software quality, cost and time to market are at least as important as functionality.

A very great deal remains to be done. For example, properties implied by the definitions presented above need to be proved; in particular, it is essential to show that arguments relating requirements, designs and implementations are logically coherent. It is also necessary to define constructive approaches to approximating preferred subtheories of possibly inconsistent requirements.

As well as this, the implications of extending the logical foundations of existing formal notations from first-order to maximal consistency logics should be explored.

Application of these extended formalisms in the context of existing lifecycle models should also be demonstrated. Of particular interest are the different approaches to establishing relationships like satisfaction and improvement. For example, conventional top-down development improves a design by making it more nearly executable, while rapid application development may reduce functionality in order to increase speed of development[4], retaining the option of improving early releases by adding functionality.

Parallels between reasoning in software engineering and the problems for which nonmonotonic logics were originally developed by the artificial intelligence community should also be exploited.

To provide automatic support for reasoning about software, adaptation of truth maintenance systems to software development is of particular interest. Nonmonotonic logics are notoriously intractable; the maximal consistency logics assumed here are not even semi-decidable[10]. Truth maintenance systems are used to keep track of inferences made, and enable revision of arguments in the light of new information. This is relevant to maintenance and revision of software designs to meet changing environments.

The more general application of prioritised logics[3] to requirements definition and software refinement is also of great interest. Of particular concern are the implications of multiple prioritisations, and the application of methods for taking decisions under uncertainty where requirements are multiply prioritised. This is needed to argue about the likelihood that design decisions will result in programs that satisfy given requirements, as opposed to the relative importance of the requirements themselves.

In all, there is scope for much fruitful research in this area, which may well provide the key to achieving commercial demand for formal methods.

²The term "satisficing" is used by Chung and Nixon in[15] to indicate that a (sub)goal may contribute to the satisfaction of a goal without fully satisfying that goal.

6 Acknowledgement

This work was partially supported by the Commission of the European Communities ACTS programme, project AC227 (SCREEN).

References

- [1] Ramsay A. *Formal Methods in Artificial Intelligence*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.
- [2] Boehm B.W. A spiral model of software development and enhancement. *Software Engineering Notes*, pages 22–42, August 1986.
- [3] Cayrol C. On the relation between argumentation and non-monotonic coherence-based entailment. In Mellish C., editor, *Proc. IJCAI 95*, pages 1441–1563. International Joint Conference on Artificial Intelligence Inc. 1995, Morgan-Kaufmann, August 1995. Nonmonotonic Logics.
- [4] McConnell Steven C. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, 1996. ISBN:1-55615-900-5.
- [5] Rounds William C. and Zhang Guo-Qiang. Defaults in domain theory. In *10th International Conference in the Mathematical Foundations of Programming Semantics Manhattan Kansas*, April 1994.
- [6] Rounds William C. and Zhang Guo-Qiang. Domain theory meets default logic. *Journal of Logic and Computation*, circa 1995. also available at <http://www.cs.uga.edu/gqz/>.
- [7] Craigen D., Gerhart S., and Ralston T. An international survey of industrial applications of formal methods, volume 1: Purpose approach analysis and conclusions. Technical Report MD 20899, U.S. Department of Commerce Technology Administration NIST, Computer Systems Laboratory Gaithersburg., 1993.
- [8] Sands D. Proving the correctness of recursion-based automatic program transformation. In *Proc. TAPSOFT '95 Aarhus 22-26 May Lecture Notes in Computer Science No. 915*, 1995.
- [9] Dijkstra E.W. *A Discipline of Programming*. Prentice-Hall, 1976.
- [10] Brewka G., Dix J., and Konolige K. A tutorial on nonmonotonic reasoning. In Brewka G., Jantke K.P., and Schmitt P.H., editors, *Nonmonotonic and Inductive Logic Lecture Notes in Artificial Intelligence No. 659*. Springer-Verlag, 1991.
- [11] Rintanen J. On specificity in default logic. In Mellish C., editor, *Proc. IJCAI 95*, pages 1441–1563. International Joint Conference on Artificial Intelligence Inc. 1995, Morgan-Kaufmann, August 1995. Nonmonotonic Logics.
- [12] Woodcock J. and Loomes M. *Software Engineering Mathematics*. Pitman, 1988.
- [13] Goguen J.A. Formal methods and social context in software development. In *Proc. TAPSOFT '95 Aarhus 22-26 May Lecture Notes in Computer Science No. 915*. Springer-Verlag, May 1995.
- [14] McCarthy John. Programs with common sense. In M. Minsky, editor, *Semantic Information Processing*. The MIT Press, 1968.
- [15] Chung L. and Nixon B.A. Dealing with non-functional requirements: Three experimental studies of a process oriented approach. In *Proc. 17th International Conference on Software Engineering ICSE 17*. Association for Computing Machinery, April 1995.
- [16] Gelfond M. and Przymusinska H. Towards a theory of elaboration tolerance: Logic programming approach. *Journal on Software and Knowledge Engineering*, circa 1995. also available at <http://cs.utep.edu/gelfond/gelfond.html>.
- [17] Stefik M. *Introduction to Knowledge Systems*, chapter 6. Morgan Kaufmann Publishers Inc., 1995.

- [18] Marques M.M., Belsnes D., and Murphy M. Formalisation based on understanding. In *Proc. FORTE '93 Boston Massachusetts October 26-29*, October 1993.
- [19] Wirth N. *Systematic Programming: An Introduction*. Prentice-Hall, 1973.
- [20] Darimont R. and van Lamsweerde A. Formal refinement patterns of goal-driven requirements elaboration. *ACM Software Engineering Notes*, November 1996.
- [21] Back R.J.R. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
- [22] Benferhat S., Dubois D., and Prade H. How to infer from inconsistent beliefs without revising. In Mellish C., editor, *Proc. IJCAI 95*, pages 1441–1563. International Joint Conference on Artificial Intelligence Inc. 1995, Morgan-Kaufmann, August 1995. Nonmonotonic Logics.
- [23] Alagar V.S. and Kourkopoulos D. (in)completeness in specifications. *Information and Software Technology*, 36(6):331–342, 1994.
- [24] Turski W.M. and Maibaum T.S.E. *The Specification of Computer Programs*. Addison-Wesley, 1987.