Managing Inconsistent Specifications: Reasoning, Analysis, and Action

ANTHONY HUNTER
University College London
and
BASHAR NUSEIBEH
Imperial College

In previous work, we advocated continued development of specifications in the presence of inconsistency. To support this, we used classical logic to represent partial specifications and to identify inconsistencies between them. We now present an adaptation of classical logic, which we term quasi-classical (QC) logic, that allows continued reasoning in the presence of inconsistency. The adaptation is a weakening of classical logic that prohibits all trivial derivations, but still allows all resolvants of the assumptions to be derived. Furthermore, the connectives behave in a classical manner. We then present a development called labeled QC logic that records and tracks assumptions used in reasoning. This facilitates a logical analysis of inconsistent information. We discuss the application of labeled QC logic in the analysis of multiperspective specifications. Such specifications are developed by multiple participants who hold overlapping, often inconsistent, views of the systems they are developing.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering; D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; D.2.2 [**Software Engineering**]: Tools and Techniques—*computer-aided software engineering (CASE)*; D.2.4 [**Software Engineering**]: Program Verification—*validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*error handling and recovery*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*restructuring*; D.2.10 [**Software Engineering**]: Design—*representation*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*proof theory*

General Terms: Design, Theory, Languages, Verification

Additional Key Words and Phrases: Managing inconsistency, paraconsistent logics, requirements specification, viewpoints

The work was partially funded by the UK EPSRC as part of the Voila (GR J15483) and Mise (GR/L 55964) projects, the CEC as part of the Basic Research Actions Promoter II and Drums II, and the British Council.
This article is a revised and extended version of our earlier work which appeared in Proceedings of the 3rd International Symposium on Requirements Engineering (1997), pages 78–86; Authors' addresses: A. Hunter, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK; email: A.Hunter@cs.ucl.ac.uk; B. Nuseibeh, Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2BZ, UK; email: ban@doc.ic.ac.uk.

1. INTRODUCTION

In a previous paper [Finkelstein et al. 1994] we discuss the need for, and outline an approach to, acting in the context-dependent way in response to inconsistency. We proposed a framework in which inconsistencies can be detected logically (using classical logic), and in which the information surrounding each inconsistency can be used to focus continued development.

While many software engineering formalisms can be translated into classical logic, classical logic does not allow useful reasoning in the presence of inconsistency: the proof rules of classical logic allow any formula of the language to be inferred. Hence, classical logic does not provide a means for continued deduction in the presence of inconsistency. Moreover, attempts to formalize the notion of inconsistency, and approaches for handling it, have also been generally unsuccessful, since they take the view that inconsistency is "undesirable" and "unusable" [Gabbay and Hunter 1991; 1993].

In this article, we present a formal yet pragmatic approach that supports continued action (including reasoning) in the presence of inconsistency, and facilitates the recording and tracking of (inconsistent) information during reasoning. The technical contributions are in two main parts.

Reasoning. First, we address the notion of "tolerating inconsistency" by proposing an adaptation of classical logic, which we term quasi-classical (QC) logic, that allows useful reasoning in the presence of inconsistency. The adaptation is a weakening of classical logic that prohibits all trivial derivations but still allows all resolvants of assumptions to be derived. We illustrate the use of the logic in this setting through some examples of multiperspective development in which inconsistencies arise between different developers who hold different (inconsistent) views of a joint problem they are addressing. Using this approach, however, problems of analyzing and acting on inconsistent specifications remain: it is difficult to isolate the exact source of the inconsistency and to decide on appropriate changes, if any. This work is covered in Section 3 (with background material covered in Section 2).

Analysis. We then address the need to analyze inconsistent specifications in the above setting. In particular, we propose the use of *labeled* QC logic that records and tracks information used in reasoning. We illustrate how the amended (labeled) proof rules of QC logic can be used to track inconsistent information by propagating these labels (and their associated information) during reasoning. We further demonstrate how specifications in labeled QC logic can be analyzed in a variety of ways in order to gain a better understanding of the likely sources of inconsistencies that arise. Using this approach we can provide a "logical analysis" of inconsistent information. We can identify the likely sources of the problem, and use this to suggest appropriate actions. This "auditing" is essential if we are to facilitate further development in the presence of inconsistency. This work is covered in Section 4.

For completeness and to illustrate our overall approach to inconsistency management, we discuss the need, and outline an approach, to acting in the presence of inconsistency (Section 5). We suggest the use of metalevel rules of the form "inconsistency implies action," but emphasize that such inconsistency-handling actions need not necessarily remove the inconsistency. Rather, action in this setting may include ameliorating, but not necessarily resolving, the inconsistency. This work is speculative and is presented as an agenda for future work.

Section 6 presents an example application that illustrates the different facets of our contribution. The application is taken from a case study of the Computer-Aided Despatching system for the London Ambulance Service [Finkelstein and Dowell 1996]. It is followed in Section 7 by an outline of the scope of, and our contributions to, automated support for inconsistency management as described in the article.

We preface and conclude with a discussion on the context of the work, namely, multiperspective software development within the "ViewPoints framework" [Finkelstein et al. 1992]. ViewPoints provide an organizational framework within which multiple development participants hold multiple views on a problem or solution domain. Inconsistencies within and between different ViewPoints typically arise, and the work described in this article provides a means of managing such inconsistencies. Moreover, ViewPoints provide a means for separation of concerns and thus can reduce software development complexity. Therefore, we argue and demonstrate that ViewPoints provide an approach to organizing large monolithic specifications, making them more amenable to the kind of reasoning and analysis we propose in this article.

Our approach is motivated by the need to deal with inconsistent specifications in a way that allows us to analyze the likely sources of the inconsistencies, allows us to continue reasoning in a rational fashion in the presence of inconsistency, and provides a basis for acting on inconsistencies. We believe that our work contributes to a better understanding of complex software development processes in which an "ideal" state of consistency maintenance is neither practical nor even desirable, and we conclude the article with a critical review of related work in the field.

2. BACKGROUND: SPECIFICATIONS, INCONSISTENCY, AND LOGIC

This section provides an overview of the background and motivation for our approach. We begin by introducing the kinds of software development information we are dealing with, the suitability of logic for representing such information, and the weaknesses of classical logic for reasoning in the presence of inconsistency.

2.1 Information for Developing Specifications

Information that is manipulated in an evolving specification can be partitioned, and is often represented, in different ways. Since we assume that multiple development participants might be involved in a development process, this information is typically also distributed among these different participants. Development information includes the following:

(1) *Specification information* about the actual system ("product") being developed, which in previous work we have captured as a collection of partial specifications denoted by loosely coupled, locally managed, distributable objects called "ViewPoints" [Finkelstein et al. 1992].

(2) *Method information* about the process of development and the representation schemes used to express partial specifications. This information also includes integrity constraints between representation schemes which we have captured as "inter-ViewPoint rules" [Nuseibeh et al. 1994]. These rules describe relationships between representation schemes, and thus the relationships between partial specifications expressed using those representation schemes. They are normally prescribed as part of the method definition, e.g., "every decomposed process in a dataflow diagram must have a corresponding parent process."

(3) *Domain information* which is incrementally captured in an evolving specification process [Easterbrook and Nuseibeh 1995; 1996]. This information might pertain to the problem domain in which the system will be installed (and which is usually captured and represented separately). For example, it might include ontologies for a particular problem domain such as telecommunications.

All three kinds of information described above (and represented as facts, rules, graphs, etc.) can be translated into logical formulae [Finkelstein et al. 1994]. In this article, however, we will focus on translation and inconsistency detection in specification information only. Additional assumptions can also be used to facilitate this inconsistency detection process. For example, the Closed World Assumption (CWA) can be used to make explicit negative information by adding the negation of certain facts, if those facts are not deducible in the specification.

We have adopted a logic-based definition of inconsistency because of the very precise and unambiguous way in which it can be defined (and subsequently detected). An inconsistency in logic results from the simultaneous assertion of a fact α and its negation, $\neg\alpha$. Using this definition, translating software engineering specifications into logic facilitates the detection of inconsistencies and allows us to concentrate on reasoning about these inconsistencies. Furthermore, we believe that problems of translation are outweighed by the improvement in inconsistency management.

For many software engineering formalisms, classical logic can be used to capture specifications. However, we do not assume that there is a unique form for representing any given specification. Rather, there are usually obvious ways of presenting any specification as a set of logical formulae.

2.2 Reasoning in the Presence of Inconsistency

We now focus on the problem of reasoning with information that might be inconsistent. By this, we mean the ability to continue development of a specification irrespective of any inconsistency in that specification, and irrespective of any inconsistency between that specification and some other related specification. We begin by briefly motivating the use of logic for performing such reasoning.

2.2.1 *Classical Logic is Appealing for Specifications.* Classical logic is very appealing for reasoning with specifications. A variety of notations for representing specifications can be translated into classical logic, including ER diagrams, dataflow diagrams, inheritance hierarchies, and much of the Z language. Furthermore, classical reasoning is intuitive and natural. The natural deduction rules and truth tables are very easy to understand. For example, if α is true and β is true, then $\alpha \wedge \beta$ is true. Similarly, if $\neg\alpha \vee \beta$ is true and α is true, then β is true.

The appeal of classical logic, however, extends beyond the naturalness of representation and reasoning. It has some very important and useful properties which mean that it is well understood and well behaved and that it is amenable to automated reasoning. First, there are a variety of proof theories and semantics—each with their own advantages—and these proof theories are all sound and complete. Second, the logic is decidable for the propositional case. This means that if we wish to know whether a particular formula holds in a specification, we can find this out in a finite number of steps. Third, the logic is semidecidable for the first-order case. While this is not as good as being decidable, it means that if a formula does hold in a first-order specification, then we can find this out in a finite number of steps. Furthermore, for both the propositional and first-order cases, it also means that there is a model of a consistent set of formulae.

In addition, there has been much progress in developing technology for classical reasoning [Clarke et al. 1996]. This includes automated reasoning systems for deriving inferences from sets of formulae (see for example L Wos et al. [1984], Fitting [1990], McCune [1990], Bibel [1993], Schumann [1993; 1994]), and model-building systems for giving models of consistent sets of formulae (see for example Costa et al. [1990], Atkinson and Cunningham [1991], Caferra and Zabel [1993], Bourley et al. [1994], Caferra and Peltier [1995], and Slaney [1996]).

2.2.2 *Problems of Reasoning with Inconsistency.* In practical reasoning, it is common to have "too much" information about some situation. In other words, it is common for there to be classically inconsistent information in a practical reasoning specification (e.g., multiple contradictory requirements about a system). The diversity of logics proposed for aspects of practical reasoning indicates the complexity of this form of reasoning. However, central to this is the need to reason with inconsistent information without the logic being trivialized. Classical logic is trivialized because, by the definition of the logic, any inference follows from inconsistent information *(ex falso quodlibet)* as illustrated by the following example.

Example 2.1. From a specification α, $\neg\alpha$, $\alpha \rightarrow \beta$, δ, reasonable inferences might include α, $\neg\alpha$, $\alpha \rightarrow \beta$, and δ by reflexivity; β by modus ponens; $\alpha \wedge \beta$ by $\wedge$ introduction; $\neg\beta \rightarrow \neg\alpha$, and so on. In contrast, trivial inferences might include γ and $\gamma \wedge \neg\delta$. □

For classical logic, trivialization renders the specification useless, and therefore classical logic is obviously unsatisfactory for handling inconsistent information. A possible solution is to weaken classical logic by dropping some of the inferencing capability (*reductio ad absurdum*), such as for the C_ω paraconsistent logic [da Costa 1974]. However, this kind of weakening of the proof rules means that the connectives in the language do not behave in a classical fashion [Besnard 1991]. For example, disjunctive syllogism does not hold, $((\alpha \vee \beta) \wedge \neg\beta) \rightarrow \alpha$, whereas modus ponens does hold, as illustrated by the following example.

Example 2.2. Let Specification-1 be $\{\alpha \vee \beta, \neg\beta\}$ and Specification-2 be $\{\neg\beta, \rightarrow \alpha, \neg\beta\}$; then α does not follow from Specification-1, but it does from Specification-2, according to C_ω. □

There are many similar examples that could be confusing and counterintuitive for users of such a practical reasoning system. An alternative compromise is quasi-classical (QC) logic [Besnard and Hunter 1995]. In the following section we present a development of QC logic called labeled QC logic that is oriented to reasoning in the context of inconsistent specifications.

3. QUASI-CLASSICAL LOGIC

We begin this section with an informal presentation of QC logic, then present an overview of its semantics, and conclude with a formal definition of the labeled QC logic and its proof theory. We only consider the propositional case here.

3.1 Introduction and Example

The proof theory of QC logic is based on reasoning with formulae that are in conjunctive normal form (CNF). These are formulae of the following form:

$$\alpha_1 \wedge .. \wedge \alpha_n$$

where each α_i is of the form

$$\beta_1 \vee .. \vee \beta_m$$

and each β_i is a literal.

The proof theory of QC logic provides the power to derive a CNF of any formula, together with the power of resolution:

$$\frac{\alpha \vee \beta \quad \neg\alpha \vee \gamma}{\gamma \vee \beta}$$

Only as a last step in any derivation is disjunction introduction allowed. This means that any resolvant of a set of formulae can be derived, but no trivial formulae can be derived. This proof theory is presented as a set of natural deduction rules such as

$$\frac{\neg(\alpha \vee \beta)}{\neg\alpha \wedge \neg\beta}.$$

All the QC natural deduction rules hold in classical logic, but the logic is weaker than classical logic in the way it is *used.* QC logic is used by providing any set of classical formulae as assumptions, and any classical formula as a query. The query follows from the assumptions if and only if there is a derivation of a CNF of the query from the assumptions using the QC natural deduction rules. For example, returning to Specification-1 and Specification-2 in Example 2.2, α follows from both sets using the QC logic.

This kind of reasoning is potentially useful for a range of activities in the management of inconsistency. These include diagnosing the source of the inconsistency (Section 4), supporting negotiation between participants in software development, continuing development without immediately resolving the inconsistency, and deciding on actions to handle the inconsistency (Section 5).

3.2 Semantics of QC Logic

Before we present the language and proof theory of QC logic (the "how"), we briefly present the essential ideas and intuitions behind the logic (the "why"). We restrict this coverage to propositional clauses. For a complete account of the semantics and relevant correctness results, the reader is referred to Hunter [1996]. We begin our overview with the following definition of a model.

Definition 3.1. Let S be some set. Let O be a set of objects defined as follows, where $+\alpha$ is a positive object, and $-\alpha$ is a negative object.

$$O = \{+\alpha | \alpha \in S\} \cup \{-\alpha | \alpha \in S\}$$

We call any $X \in \wp(O)$ a model, where $\wp$ is the power-set function. □

Informally, we can consider the following meaning for positive and negative objects being *in* or *out* of some model X:

$+\alpha \in X$ means α is "satisfiable" in the model

$-\alpha \in X$ means $\neg\alpha$ is "satisfiable" in the model

$+\alpha \notin X$ means α is not "satisfiable" in the model

$-\alpha \notin X$ means $\neg\alpha$ is not "satisfiable" in the model

Since we can allow both an atom and its complement to be satisfied in the same model, we have decoupled, at the level of the model, the link between a formula and its complement. In contrast, in a classical model, if a model satisfies a literal, then it is forced not to satisfy the complement of the literal.

Definition 3.2. Let $\alpha_1 \vee .. \vee \alpha_n$ be a clause; then Literals$(\alpha_1 \vee \ldots \vee \alpha_n)$ is the set of literals $\{\alpha_1, \ldots, \alpha_n\}$ that are in the clause. □

Definition 3.3. Let $\alpha_1 \vee \ldots \vee \alpha_n$ be a clause, and let α_i be a literal, such that $\alpha_i \in \mathit{Literals}(\alpha_1 \vee \ldots \vee \alpha_n)$. $\mathit{Focus}(\alpha_1 \vee \ldots \vee \alpha_n, \alpha_i)$ is a clause $\beta_1 \vee \ldots \vee \beta_{n-1}$, where $\{\beta_1, \ldots, \beta_{n-1}\}$ is Literals$(\alpha_1 \vee \ldots \vee \alpha_n) - \{\alpha_i\}$. □

Definition 3.4. Let $\mathcal{L}$ be the language of classical logic, and let $\models_s$ be a satisfiability relation, called strong satisfaction, such that $\models_s \subseteq \wp(\mathrm{O}) \times \mathcal{L}$. For $X \in \wp(O)$, we define $\models_s$ as follows, where $\vee$ is commutative and associative, α is an atom, and $\alpha_1, \ldots, \alpha_n$ are literals in $\mathcal{L}$.

$$X \models_s \alpha \text{ if } +\alpha \in X$$

$$X \models_s \neg\alpha \text{ if } -\alpha \in X$$

$$X \models_s \alpha_1 \vee \ldots \vee \alpha_n \qquad \text{iff}[X \models_s \alpha_1 \text{ or } \ldots \text{ or } X \models_s \alpha_n]$$

$$\text{and } \forall i\ [X \models_s \sim \alpha_i \text{ implies } X \models_s \mathit{Focus}(\alpha_1 \vee \ldots \vee \alpha_n, \alpha_i)]$$

where $\sim\alpha_i$ is the complement of some literal α_i in Literals$(\alpha_1 \vee \ldots \vee \alpha_n)$, and for any literal α, every model satisfies Focus(α, α). □

The first two parts of this definition covers literals. The third part covers disjunction. This definition for disjunction is more restricted than the classical definition. In addition to at least one disjunct being satisfied, there is also a notion of "focusing" incorporated into the definition. Essentially, for each disjunct α_i in the formula $\alpha_1 \vee \ldots \vee \alpha_n$, if the model satisfies the complement (denoted by $\sim\alpha$) of that disjunct, then the model must also satisfy the focused formula $\mathit{Focus}(\alpha_1 \vee \ldots \vee \alpha_n, \alpha_i)$, where the focused formula is just the original formula without the disjunct α_i.

The reason we need this definition for disjunction that is more restricted than classical disjunction is that we have decoupled the link between a formula and its negation in the model. Therefore, in order to provide a meaning for resolution, we need to put the link between each disjunct, and its complement, into the definition for disjunction. As a result, to ensure a clause is satisfied, we need to ensure, that if necessary, every more focused clause is also satisfied.

However, proofs in this paraconsistent reasoning are two-stage affairs. The first is decompositional, forming resolvants from clauses using resolution. The second is compositional, forming clauses from the assumptions and resolvants, using disjunction introduction. As a result of incorporating disjunction introduction, we also need to extend the semantics.

Definition 3.5. Let $\models_w$ be a satisfiability relation, called weak satisfaction, such that $\models_w \subseteq \wp(O) \times \mathscr{L}$. For $X \in \wp(O)$, we define $\models_w$ as follows, where $\bigvee$ is commutative and associative, α is an atom, and $\alpha_1, \ldots, \alpha_n$ are literals in $\mathscr{L}$.

$$X \models_w \alpha \text{ if } +\alpha \in X$$

$$X \models_w \neg\alpha \text{ if } -\alpha \in X$$

$$X \models_w \alpha_1 \bigvee \ldots \bigvee \alpha_n \text{ iff}[X \models_w \alpha_1 \text{ or } \ldots \text{ or } X \models_w \alpha_n] \qquad \square$$

Both strong and weak satisfaction are straightforwardly extended to the full classical propositional language. In the following definition, we can see that QC entailment is of the same form as classical entailment, except we use strong satisfaction for the assumptions and weak satisfaction for the inference.

Definition 3.6. Let $\models_Q$ be an entailment relation such that $\models_Q \subseteq \wp(\mathscr{L}) \times \mathscr{L}$, and is defined as follows:

$$\{\alpha_1, \ldots, \alpha_n\} \models_Q \beta \text{ iff } \forall X(X \models_s \alpha_1 \text{ and} \ldots \text{and } X \models_s \alpha_n \text{ implies } X \models_w \beta) \qquad \square$$

We can consider the strong satisfaction relation as capturing the decomposition of the set of assumptions. Models are only acceptable to the strong satisfaction relation if they support focusing. In contrast, we can consider weak satisfaction relation as capturing the composition of formulae from resolvants, allowing disjuncts to be introduced.

Example 3.1. Let $\Delta = \{\alpha\}$, and let $X1 = \{+\alpha\}$ and $X2 = \{+\alpha, -\alpha\}$. Now $X1 \models_s \alpha$, and $X2 \models_s \alpha$, whereas $X1 \models_s \alpha \bigvee \beta$, and $X2 \not\models_s \alpha \bigvee \beta$. However, $X1 \models_w \alpha \bigvee \beta$, and $X2 \models_w \alpha \bigvee \beta$, and indeed $\Delta \models_Q \alpha \bigvee \beta$. $\square$

Example 3.2. Let $\Delta = \{\alpha \bigvee \beta, \neg\alpha\}$. For all models X, if $X \models_s \alpha \bigvee \beta$, and $X \models_s \neg\alpha$, then $X \models_s \beta$. Hence, $\Delta \models_Q \alpha \bigvee \beta$, $\Delta \models_Q \neg\alpha$, and $\Delta \models_Q \beta$. $\square$

One ramification of this definition for QC proof theory and semantics is that, in general, classical tautologies do not hold.

Example 3.3. Let $\Delta = \emptyset$. Hence $X = \emptyset$ is a model that strongly satisfies all the formulae in Δ, but $\Delta \models_Q \alpha \bigvee \neg\alpha$ does not hold. $\square$

If we consider formulae representing arguments, then $\alpha \bigvee \neg\alpha$ ("excluded middle") means we have an argument for α or an argument for $\neg\alpha$. But this should not hold for every proposition α. Since for example, there are many situations for which one would have neither an argument for or against. In general, however, the failure of excluded middle, and other tautologies, is not a problem in many software engineering applications (as illustrated in our example in Section 6).

3.3 Language and Proof Theory of Labeled QC Logic

We now provide a formal definition of labeled QC logic. We use a labeled language to allow us to uniquely identify each item of development information. We propagate the labels by labeling consequences with the union of the labels of the premises. This means we can identify the ramifications of each item in the reasoning, since each inference will be labeled. Labels can be used to differentiate different types of development information, and in particular they can indicate the sources of information. Labeling is a development of QC logic. It does not affect the proof theory or semantics. It only tracks assumptions used in a proof. We demonstrate the utility of the labels in Section 4.

3.3.1 *Language of Labeled QC Logic.* For this article, we assume a classical first-order language without function symbols and existential quantifiers. In other words, we assume a universally quantified formula is just an abbreviation for a conjunction of the formulae that can be formed by systematically instantiating the variables of the quantified formula with the constants in the language of the specification. Essentially, this means we have restricted the language to the propositional case. This gives certain computational advantages. In particular, it renders consistency checking decidable for this language.

Definition 3.7. Let $\mathcal{P}$ be a set of predicate symbols, $\mathcal{V}$ be a set of variable symbols, and $\mathcal{C}$ a set of constant symbols. Let $\mathcal{A}$ be a set of atoms, where $\mathcal{A} = \{p(q_1, \ldots, q_n) | p \in \mathcal{P}$ and $q_1, \ldots, q_n \in \mathcal{V} \cup \mathcal{C}\}$. We call $p(q_1, \ldots, q_n)$ a ground atom if and only if $q_1, \ldots, q_n$ are all constant symbols; otherwise we call it unground. □

Example 3.4. Let telephone-number be a predicate symbol and X, Y be variable symbols; and John, Peter, 7295, and 1342 be constant symbols. From these symbols, examples of atoms include

telephone-number(X, Y) is an unground atom
telephone-number(X, 1342) is an unground atom
telephone-number(John, 1342) is a ground atom. □

Definition 3.8. Let $\mathcal{F}$ be the set of classical formulae formed from a set of atoms $\mathcal{A}$, and the $\wedge$, $\vee$, $\rightarrow$, and $\neg$ connectives. We abbreviate $\alpha \wedge \neg\alpha$ by $\perp$, which we read as "inconsistency." We call a formula ground if and only if it is made from only ground atoms; otherwise we call it unground. □

Definition 3.9. Let $\mathcal{G} \subset \mathcal{L}$ be the set of formulae formed from $\mathcal{F}$, where if $\alpha \in \mathcal{F}$, and $X_1, \ldots, X_n$ are the free variables of α, then $\forall X_1, \ldots, \forall X_n \alpha \in \mathcal{G}$. □

Hence the set $\mathcal{G}$ contains only universally quantified formulae, where the quantifiers are outermost, and ground formulae. This restriction aids our exposition.

The following two definitions are used to explain the proof theory concisely.

Definition 3.10. For $\alpha_1 \wedge .. \wedge \alpha_n \in \mathcal{L}$, $\alpha_1 \wedge .. \wedge \alpha_n$ is in conjunctive normal form (CNF) if and only if each of $\alpha_1, \ldots, \alpha_n$ is a clause. □

Definition 3.11. For $\alpha_1 \wedge .. \wedge \alpha_n \in \mathcal{L}$, and $\beta \in \mathcal{L}$, $\alpha_1 \wedge .. \wedge \alpha_n$ is a CNF of β if and only if $\alpha_1 \wedge .. \wedge \alpha_n \vdash \beta$ and $\beta \vdash \alpha_1 \wedge .. \wedge \alpha_n$, and $\alpha_1 \wedge .. \wedge \alpha_n$ is in CNF, where $\vdash$ is the classical consequence relation. □

In general, a formula can appear complex: there may be many levels of nesting of the connectives. Representing a formula in CNF removes this problem, since there are at most two levels of nesting. This then allows a straightforward application of resolution.

For any $\alpha \in \mathcal{L}$, a CNF of α can be produced by the application of distributivity, negation elimination, and de Morgan laws. Clearly this holds for the grounded formulae. It also holds for the quantified formulae, since we have restricted the language to universally quantified formulae where the quantifiers are outermost.

Definition 3.12. Let $\vdash_X$ be some consequence relation for some logic X, defined by some proof rules. Then, the logic X is trivializable if and only if for all α, β in the language of X, $\{\alpha, \neg\alpha\} \vdash_X \beta$. □

Note that classical logic is trivializable according to this definition. We now provide a definition for a trivial formula. For this, we require the following function *Atoms*(Δ) which gives the set of atoms used in the set of formulae in Δ.

Definition 3.13. Let $\Delta \in \wp(\mathcal{L})$. Let Ground($\Delta$) be the set of all ground formulae that can be formed by grounding the formula in Δ with the constants that appear in Δ. Let Atoms(Δ) be the set of ground atoms used in the subformulae of Ground(Δ). □

Example 3.5. Let Δ be $\{(\alpha \vee \beta) \wedge \neg\gamma, \delta \rightarrow \neg\neg\gamma\}$. Then $Atoms(\Delta) = \{\alpha, \beta, \gamma, \delta\}$. □

Definition 3.14. A clause $\alpha \in \mathcal{L}$ is trivial with respect to Δ if and only if $Atoms(\Delta) \cap Atoms(\{\alpha\}) = \emptyset$. □

Example 3.6. Let Δ be $\{\alpha, \neg\alpha\}$, and let β be an inference from Δ; then β is a trivial inference. □

So for any database Δ, and any inference α, if α has no ground atom in common with Δ, then α is a trivial inference.

We now consider the labeled form of the language.

Definition 3.15. Let $\mathcal{S}$ be some set of atomic symbols such as an alphabet. If $i \subseteq \mathcal{S}$, and $\alpha \in \mathcal{G}$, then $i : \alpha$ is a labeled formula. Let $\mathcal{M}$ be the set of labeled formulae. □

All development knowledge that we are analyzing is translated into labeled formulae, where each formula has a unique label.

Example 3.7. Let $\mathcal{S} = \{a, b, c, \ldots\}$. Then examples of labeled formulae include

$$\{a\} : \text{telephone-number(John, 1342)}$$

$$\{b\} : \text{telephone-number(John, 1342)} \vee \text{telephone-number(Peter, 7295)} \quad \square$$

There are many strategies that we could adopt for labeling development information (see Section 4.2). Options include combinations of the source of the item and that time the item was inserted. For this, some mapping from labels to their associated meaning needs to be recorded. For instance, different developers could use different disjoint subsets of the labels.

3.3.2 *Proof Theory for Labeled QC Logic.* The proof theory of QC logic provides the power to derive each clause of any CNF of any formula, together with the power of resolution. As a "last step" in any derivation, disjunction introduction is also allowed. This means that any resolvant of a set of formulae can be derived, but no trivial formulae can be derived. This proof theory is presented as a set of natural deduction rules. All the QC natural deduction rules hold in classical logic, but some classical deduction rules, such as *ex falso quodlibet*, do not hold in QC logic. For a more complete coverage of QC logic the reader is referred to Besnard and Hunter [1995] and Hunter [1996; 1998].

We obtain labeled QC logic by using only labeled formulae as assumptions, and by amending the natural deduction rules to propagate the labels. The label of the consequent of a rule is the union of the labels of the premises of the rule.

Definition 3.16. Assume that $\wedge$ is a commutative and associative operator, and $\vee$ is a commutative and associative operator. In these proof rules, disjuncts can be void—for example, for negation elimination, if we have $\neg\neg\alpha$ as a premise, so that the disjunct β is void, then we can obtain α as a consequent.

$$\frac{i : \alpha \wedge \beta}{i : \alpha} \quad \text{[Conjunct elimination]} \qquad \frac{i : \alpha \vee \alpha \vee \beta}{i : \alpha \vee \beta} \quad \text{[Disjunct contraction]}$$

$$\frac{i : \alpha \vee \beta}{i : \neg\neg\alpha \vee \beta} \quad \text{[Negation introduction]} \qquad \frac{i : \neg\neg\alpha \vee \beta}{i : \alpha \vee \beta} \quad \text{[Negation elimination]}$$

$$\frac{i : \alpha \vee (\beta \rightarrow \gamma)}{i : \alpha \vee \neg\beta \vee \gamma} \frac{i : \alpha \vee \neg(\beta \rightarrow \gamma)}{i : \alpha \vee (\beta \wedge \neg\gamma)} \quad \text{[Arrow elimination]}$$

$$\frac{i : \forall x \alpha}{i : \beta} \quad \text{[Universal instantiation, where } \beta \text{ is obtained from } \alpha \text{ by replacing every occurrence of } x \text{ by the same constant]}$$

$$\frac{i : \alpha \vee \beta \qquad j : \neg\alpha \vee \gamma}{i \cup j : \beta \vee \gamma} \quad \text{[Resolution]}$$

$$\frac{i : \alpha \vee (\beta \wedge \gamma)}{i : (\alpha \vee \beta) \wedge (\alpha \vee \gamma)} \quad \frac{i : (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)}{i : \alpha \wedge (\beta \vee \gamma)} \quad \text{[Distribution]}$$

$$\frac{i : \neg(\alpha \wedge \beta) \vee \gamma}{i : \neg(\alpha \vee \neg\beta) \vee \gamma} \quad \frac{i : \neg(\alpha \vee \beta) \vee \gamma}{i : (\neg\alpha \wedge \neg\beta) \vee \gamma} \quad \text{[de Morgan laws]}$$

$$\frac{i : \alpha}{i : \alpha \vee \beta} \quad \text{[Disjunct introduction—but only as a last step in a proof]}$$

Labeled QC logic is used by providing any set of labeled formulae (i.e., any $\Delta \subseteq \mathcal{M}$) as assumptions, and any labeled formula (i.e., any $\alpha \in \mathcal{M}$) as a query. In this, all development information is a set of assumptions.

For a query that is a ground formula, it follows from the assumptions with some label i (denoted $\Delta \vdash_Q i : \alpha$) if and only if there is a derivation of each conjunct of a CNF of the query, labeled i, from the assumptions using the labeled QC natural deduction rules, remembering that disjunction introduction is only allowed as a last step in a derivation. For a query that is universally quantified, form a ground formula by instantiating it with constants that do not appear in the assumptions, and then treat the query as above. □

This proof theory allows any resolvant of a set of formulae to be derived, but no trivial formulae can be derived. We give an example of labeled QC reasoning for ground formulae below and an example with quantified formulae in Section 6.

Example 3.8. Suppose we have the following development information, where a is a constraint (e.g., an inter-ViewPoint rule); b is some domain knowledge; and c and d denote fragments of partial specification information.

$$\{a\} : \alpha \vee \neg\beta$$

$$\{b\} : \gamma \rightarrow \beta$$

$$\{c\} : \neg\alpha$$

Clearly this development information is inconsistent. However, using the QC proof rules, we can obtain a number of nontrivial inferences. Consider the following proof:

Step 1 *is* $\{b\} : \neg\gamma \vee \beta$	*from* $\{b\} : \gamma \rightarrow \beta$ *using arrow elimination*
Step 2 *is* $\{b, d\} : \beta$	*from* $\{d\} : \gamma$ *and Step* 1 *using resolution*
Step 3 *is* $\{a, b, d\} : \alpha$	*from* $\{a\} : \alpha \vee \neg\beta$ *and Step* 2 *using resolution*
Step 4 *is* $\{a, b, c, d\} : \perp$	*from* $\{c\} : \neg\alpha$ *and Step* 3

In Step 2, we obtain the formula β that is labeled with $\{b, d\}$. This formula is nontrivial and may be useful for furthering the evolution of the specification. We show in Section 4.1 how we use the label to qualify this inference. In qualifying an inference, we indicate the relationship between the data used for the inference and the inconsistent data in the development information: the closer the relationship, the more we qualify the inference.

In Step 4, we obtain an inconsistency labeled $\{a, b, c, d\}$. This label tells us which parts of the development information have given the inconsistency. This label can be used for identifying the likely sources of an inconsistency, as discussed in Section 4.2. $\square$

QC logic is weaker than classical logic in a number of ways. This is the price to pay for avoiding trivialization. Two relevant classical features lost are that (1) classical tautologies do not necessarily hold and (2) lemmas cannot be used in subsequent QC proofs. We explain these losses below. However, we believe that in the context of requirements engineering, these losses are not problematic.

The first loss is that from any empty set of assumptions, no inferences follow. For example, $\alpha \vee \neg\alpha$ does not follow from the database using the QC proof theory. The reason for this can be seen most clearly from the semantics. In a model there are arguments for and against propositions. But a model does not necessarily have an argument for or against every proposition. It can specify neither. Hence $\alpha \vee \neg\alpha$ does not hold in every model. This reasoning holds for any classical tautology. Of course, once the set of assumptions is nonempty, then the set of models is constrained, and then some classical tautologies do hold.

The second loss results from the restriction on the use of disjunction introduction. If this restriction was lifted, trivialization would result. Consider $\{\alpha, \neg\alpha\}$. Suppose we used disjunction introduction to obtain $\alpha \vee \beta$, and then used resolution; we could obtain β. In this way, we have trivialization. Hence, we cannot, in general, have proofs used as lemmas in further proofs. Reasoning with QC logic is a one-step process. If we want to derive an inference, we need to constrain a complete proof.

There have been a number of other proposals for reasoning with classical formulae that are inconsistent. Key logics include C_w [da Costa 1974] and four-valued logic [Belnap 1977]. QC logic differs in having a richer proof theory, but at the price of losing tautologies and prohibiting proofs to be used as lemmas in further proofs. For a comparison of some paraconsistent logics see Hunter [1998].

4. LOGICAL ANALYSIS OF INCONSISTENT SPECIFICATIONS

So far we have shown how development information can be represented as logical formulae and how we can undertake nontrivial reasoning with such formulae, even if they are mutually inconsistent. In this section we turn to logical analysis of inconsistent specifications. In particular, we consider qualifying inferences from inconsistent specifications, and identifying likely sources for an inconsistency. These are just two ways of providing a logical analysis of inconsistent specifications.

4.1 Qualifying Inferences from Inconsistent Information

When considering inconsistent information, we have more confidence in some inferences over others. For example, we may have more confidence in an inference α from a consistent subset of the database if we cannot also derive $\neg\alpha$ from another consistent subset of the database.

Definition 4.1. Let Δ be a set of labeled formulae. We form the following sets of sets of formulae:

$$CON(\Delta) = \{\Gamma \subseteq \Delta | \Gamma \nvdash_Q i : \bot\}$$

$$INC(\Delta) = \{\Gamma \subseteq \Delta | \Gamma \vdash_Q i : \bot\} \qquad \square$$

Essentially, $CON(\Delta)$ is the set of consistent subsets of Δ, and $INC(\Delta)$ is the set of inconsistent subsets of Δ.

In the following definition, $MI(\Delta)$ is a set of sets of minimally inconsistent formulae. A set of formulae is minimally inconsistent if every proper subset is consistent. Similarly, $MC(\Delta)$ is a set of sets of maximally consistent formulae. A set of formulae is maximally consistent, if the set is consistent and adding any further formulae to the set from Δ causes the set to be inconsistent.

Definition 4.2. Let Δ be a set of labeled formulae. We form the following sets of sets of labeled formulae:

$$MI(\Delta) = \{\Gamma | \Gamma \in INC(\Delta) \text{ and } \forall \Phi \in INC(\Delta)\ \Phi \not\subset \Gamma\}$$

$$MC(\Delta) = \{\Gamma | \Gamma \in CON(\Delta) \text{ and } \forall \Phi \in CON(\Delta)\ \Gamma \not\subset \Phi\}$$

$$FREE(\Delta) = \bigcap MC(\Delta) \qquad \square$$

We can consider a maximally consistent subset of a database as capturing a "plausible" or "coherent" view on the database. For this reason, the set $MC(\Delta)$ is important in many of the definitions presented in the next section. Furthermore, we consider $FREE(\Delta)$, which is equal to $\cap MC(\Delta)$, as capturing all the "uncontroversial" information in Δ. In contrast, we consider the set $\cup MI(\Delta)$ as capturing all the "problematic" data Δ. Note, $\cap MC(\Delta)$ is equal to $\Delta - \cup MI(\Delta)$. So reasoning with $FREE(\Delta)$ is equivalent to revising the database by removing all the "problematic" data. This means we have a choice. We can either reason with the data directly using $FREE(\Delta)$, or we can revise the data by removing the formulae corresponding to $\cup MI(\Delta)$.

We now use these concepts to define three qualifications for an inference from inconsistent information. The approach is a derivative of argumentative logics [Elvang-Goransson and Hunter 1995].

Definition 4.3. Let Δ be a set of labeled formulae, and let $\Delta \vdash_Q i : \alpha$ hold. We form the following qualifications for inferences:

$$\alpha \text{ is an existential inference if } \exists \Phi \in MC(\Delta)\ \exists j \text{ such that } \Phi \vdash_Q j : \alpha$$

$$\alpha \text{ is an universal inference if}$$
$$\forall \Phi \in MC(\Delta)\ \exists j \text{ such that } \Phi \vdash_Q j : \alpha$$

$$\alpha \text{ is a free inference if } \exists j \text{ such that } FREE(\Delta) \vdash_Q j : \alpha$$ □

So a formula is an existential inference if it is an inference from a consistent subset of the database. A formula is a universal inference if it is an inference from each maximally consistent subset of the database, whereas a formula is a free inference if it is an inference from the intersection of the maximally consistent subsets of the database.

If α is a free inference, it is also a universal inference. Similarly, if α is a universal inference, it is also an existential inference. Clearly, if α is only an existential inference, then we are far less confident in it than if it was a universal inference. If it is a free inference, then it is not associated with any inconsistent information.

Example 4.1. Consider the following assumptions:

$$\{a\} : \alpha \wedge \beta$$

$$\{b\} : \neg\alpha \wedge \beta$$

$$\{c\} : \gamma$$

This gives two maximally consistent subsets:

Set 1	**Set 2**
$\{a\} : \alpha \wedge \beta$	$\{b\} : \neg\alpha \wedge \beta$
$\{c\} : \gamma$	$\{c\} : \gamma$

From this, α and $\neg\alpha$ are only existential inferences, whereas γ is a free inference, and β is universal inference. □

These kinds of qualification are useful when reasoning with inconsistent information because they provide a clear and unambiguous relationship between the inferences and problematic data. They could also be useful in facilitating development in the presence of inconsistency, since we would feel happier about relying on the less qualified inferences. Furthermore, they provide a useful vocabulary for participants in the development process to discuss the inconsistent information.

Handling inconsistent information using maximally consistent subsets has been considered in a number of approaches including logics (for a review see Elvang-Goransson and Hunter [1995]), combining knowledge bases (for example Baral et al. [1992]), and truth maintenance systems (for example Doyle [1979] and Kleer [1986]); we go beyond this by adopting universal and free inferencing. Furthermore, by adopting labeling, we integrate our inconsistency management with QC reasoning and with identifying likely sources of inconsistency.

4.2 Identifying Likely Sources of an Inconsistency

When we identify an inconsistency in our development information, we want to analyze that inconsistency before we decide on a course of action (for example, further reasoning, removing inconsistency, etc). Using labeled QC reasoning, we obtain the labels of the assumptions used to derive an inconsistency. We use the term "source" to denote the subset of the assumptions that we believe to be incorrect.

Example 4.2. Suppose we have the specification,

$$\{a\} : \alpha$$

$$\{b\} : \neg\alpha \vee \neg\beta$$

$$\{c\} : \beta$$

and suppose $\{a\} : \alpha$ and $\{b\} : \neg\alpha \vee \neg\beta$ have been a stable and well-accepted part of the specification for some time, and by contrast $\{c\} : \beta$ is just a new and tentative piece of specification. Then for the inconsistency $\{a, b, c\} : \perp$, we could regard $\{c\} : \beta$ as the source of the inconsistency. □

Identifying the source facilitates appropriate actions to be invoked. We discuss this further below and discuss "acting on inconsistency" in the next section.

Definition 4.4. Let Δ be a set of labeled formulae, and let i be some label of some inference from Δ. The set of assumptions from Δ corresponding to the label is defined as follows:

$$Formulae(\Delta, i) = \{j : \alpha \in \Delta | j \subseteq i\} \qquad \square$$

Definition 4.5. For an inconsistency $i : \perp$, $Formulae(\Delta, j)$ is a possible source of the inconsistency if $j \subseteq i$ and $Formulae(\Delta, i - j) \in CON(\Delta)$. □

Essentially, $Formulae(\Delta, j)$ is a possible source of the inconsistency if it corresponds to a subset of the assumptions used to obtain the inconsistency, and the remainder of the assumptions are consistent.

There may be a large number of possible sources of an inconsistency, and a number of options for addressing this, such as working only with the smallest sources, or working with only the sources that have the least effect on the number of inferences from the specification. However, if we are to act on inconsistency, then we really need to identify the "likely" sources of inconsistency. To address this, we assume that for any development information, there is some ordering over that information, where the ordering captures the likelihood of the information being erroneous. So, if i is higher in the ordering than j, then $i : \alpha$ is less likely to be erroneous than $j : \beta$. We assume this ordering is transitive, though not necessarily linear.

If we assume there is an ordering over assumptions, then a more likely source is the smallest source that contains less preferred assumptions.

Example 4.3. Returning to Example 4.2, we can use explicit ordering to formalize this reasoning. For the specification above, $\{a\} : \alpha$ and $\{b\} : \neg\alpha \vee \neg\beta$ are both ordered higher than $\{c\} : \beta$. Hence β is a likely source. □

Obviously, this approach does not guarantee that a likely source can be uniquely determined from a set of possible sources.

Example 4.4. Consider the following data:

$$\{a\} : \gamma$$

$$\{b\} : \beta$$

$$\{c\} : \alpha$$

$$\{d\} : \beta \rightarrow \neg\alpha$$

where a, b, c, and d is a linear ordering such that a is most preferred and d is least preferred. Here $\{\{b\} : \beta, \{d\} : \beta \rightarrow \neg\alpha\}$ and $\{\{c\} : \alpha\}$ are likely sources of the inconsistency. □

Assuming an ordering over development information is reasonable in software engineering. First, different kinds of information have different likelihoods of being incorrect. For example, method rules are unlikely to be incorrect, whereas some tentative specification information is quite possibly incorrect. Second, if a specification method is used interactively, a user can be asked to order pieces of specification according to likelihood of correctness.

There are a number of ways that this approach can be developed. First, there are further intuitive ways of deriving orderings over formulae and sets of formulae. These include ordering sets of formulae according to their relative degree of contradiction [Gabbay and Hunter 1998]. Second, there are a number of analyses of ways of handling ordered formulae and sets of ordered formulae. These include the use of specificity [Poole 1985], ordered theory presentations [Ryan 1992], preferred subtheories [Brewka 1989], explicit preferences [Prakken 1993], and prioritized syntax-based entailment [Benferhat et al. 1993].

5. TOWARD ACTING ON INCONSISTENCY

The logical analysis of inconsistency described in this article is part of a wider framework for handling inconsistency. The analysis described in the previous section can be used to generate a "report" that identifies inconsistencies and provides a "diagnosis" of these inconsistencies. Using this, we should be able to say something about the possible actions that can be performed based on the nature of the inconsistencies identified. In what follows, we outline an approach we intend to pursue in order to make further use of the reasoning and analysis results of the previous sections.

Effective action in the presence of inconsistencies requires gathering a wider appreciation of the nature and context of these inconsistencies. While further work is still needed to examine the kinds of inconsistency-handling actions that are appropriate in different situations, we make some preliminary observations. If we accept that acting in the presence of inconsistency requires external (probably human) input, we can adopt a metalevel approach to prescribe inconsistency-handling rules of the form

Inconsistency implies Action.

One approach is to deploy an action-based temporal logic that allows us to specify the past context and source of an inconsistency in order to prescribe future actions to handle the inconsistency [Gabbay and Hunter 1993; Finkelstein et al. 1994]. Thus for example the rule

$$
\begin{aligned}
&[\mathit{data}(\Delta_1) \wedge \mathit{data}(\Delta_2)\\
&\wedge\ \mathit{union}(\Delta_1, \Delta_2) \vdash \perp\\
&\wedge\ \mathit{inconsistency\text{-}source}(\mathit{union}(\Delta_1, \Delta_2), S)\\
&\wedge\ \mathit{likely\text{-}spelling\text{-}problem}(S)\\
&\wedge\ \neg \mathit{LAST}^1\ \mathit{likely\text{-}spelling\text{-}problem}(S)\\
&\wedge\ \neg \mathit{LAST}^2\ \mathit{likely\text{-}spelling\text{-}problem}(S)]\\
&\qquad\qquad \rightarrow \mathit{NEXT}\ \mathit{tell\text{-}user}(\text{“}\mathit{is\ there\ a\ spell\ problem?}\text{”}, S)
\end{aligned}
$$

specifies that the user should be prompted to check the inconsistent data for spelling mistakes. This rule uses the temporal operators LAST^n and NEXT^n to refer to n time units in the past or future, respectively. So for example, if our time units are minutes, then LAST^5 *likely-spelling-problem* could mean 5 minutes ago the spell checker was run. Also note that

—$\mathit{data}(\Delta_1)$ and $\mathit{data}(\Delta_2)$ hold if the formulae in the databases Δ_1 and Δ_2, respectively, are translations into classical logic of some development information.

—*union*$(\Delta_1, \Delta_2) \vdash \perp$ holds if the union of the databases Δ_1 and Δ_2 classically implies inconsistency.

—*inconsistency-source*(*union*(Δ_1, Δ_2), S) holds if S is a minimally inconsistent subset of the union of Δ_1 and Δ_2, as defined in Section 6.1. A likely source, S, of the inconsistency may be identified, for example, as described in Section 6.2.

—*likely-spelling-problem*(S) holds if the cause of the inconsistency is likely to result from typographical errors in S. Since we are using a temporal language at the metalevel, we can also include conditions in our rule that we have not checked this problem at a previous point in time. In this way, the past history can affect future actions.

—*tell-user*(*"is there a spell problem?"*, S) is an action that may be triggered if the inconsistency is identified as above. In this case, the action is a message displayed to the user, together with the likely source of the inconsistency.

Identifying the appropriate inconsistency-handling action in a rule such as the one described above remains a difficult, but important, challenge. It depends on the kinds of inconsistency that can be detected and the degree of inconsistency tolerance that can be supported. Different developers and researchers have adopted different strategies to acting in the presence of inconsistency. We have identified at least four kinds of such actions [Nuseibeh 1996]:

(1) *Ignoring* the inconsistency completely and continuing development regardless: This may be appropriate in certain circumstances where the inconsistency is isolated and the software engineer believes that it does not prevent further development from taking place. This is analogous to "commenting out" inconsistent parts of the specification (perhaps for later inspection).

(2) *Circumventing* the inconsistent parts of the specification being developed and continuing development: This may be appropriate in order to avoid inconsistent portions of the specification and/or to delay resolution of the inconsistency.

(3) *Removing* the inconsistency altogether by correcting any mistakes or resolving conflicts. This depends on a clear identification of the inconsistency and assumes that the actions required to fix it are known. Restoring consistency completely can be difficult to achieve, and is quite often impossible to automate completely without human intervention.

(4) *Ameliorating* inconsistent situations by performing actions that "improve" these situations and increase the possibility of future resolution. This is an attractive approach in situations where complete and immediate resolution is not possible (perhaps because further information is required from another development participant), but where some steps can be taken to "fix" part or some of the inconsistent information. This

approach requires techniques for analysis and reasoning in the presence of inconsistency, such as those described in this article.

Deciding which kind of action is appropriate or feasible to perform in the presence of inconsistency is the first step toward specifying effective inconsistency-handling rules.

Bearing in mind the above work, we are also interested in supporting multiperspective software development, that is, the development of systems in which multiple development participants hold different views on the systems they are developing. Since we have adopted a decentralized approach in which multiple ViewPoints represent different participants and the views that they hold, we have also examined decentralized process modeling, enactment, and support in this context. For example, we have attempted to analyze "work records" of individual ViewPoints, in order to identify some key "situations" to which we know how to react (e.g., situations in which dataflow diagrams have been incorrectly decomposed, with the reactions including guidance on which parent-child processes to check). We have implemented a prototype to support this which uses regular expressions to specify particular situations and rules to associate actions with these situations [Leonhardt et al. 1995]. Further work is needed to incorporate the kinds of logical analysis we have described in this article within this ViewPoints prototype. Nevertheless the formal nature of our analysis makes it amenable to automated support.

6. APPLICATIONS

In order to validate our work, we have completed a series of examples that illustrate the reasoning, analysis, and action described in this article. The next step is to apply our techniques to a large case study to examine how they scale up in an industrial setting. However, we do not intend to demonstrate such scalability by simply applying our approach to very large specifications. Our techniques are intended for use in conjunction with other traditional development approaches. So, for example, we do not expect the wholesale translation of large, monolithic specifications into QC logic on which we can then perform the kind of reasoning we have described. Rather, our aim is to reduce the complexity of our reasoning and analysis by restricting them to smaller partial specifications, and to demonstrate its potential usefulness when used with a host of other tools from the requirements engineer's toolbox. Our earlier work [Nuseibeh et al. 1994] deliberately focused on the partitioning of specifications into manageable units (ViewPoints) to which we now apply the techniques described. Inconsistency management in this setting, then, addresses inconsistencies within and between selected ViewPoints. We have considerable academic and industrial experience in using the ViewPoints framework and its support environment (The Viewer), and therefore expect our logic-based inconsistency management techniques to complement and support such a multiperspective development approach.

In the following, we present excerpts of an example application: eliciting and specifying the requirements of the London Ambulance Service (LAS). This case study was the focus of, and common example used by, delegates at the Eighth International Workshop on Software Specification and Design (IWSSD-8) [Finkelstein and Dowell 1996]. The intention is to illustrate the techniques we have described in this article in the context of a "real-world" example. The exposition is somewhat artificial in order to illustrate the issues and contributions presented in the article.

6.1 Requirements Document

Consider the requirements for a computer-aided ambulance despatching system. A reasonable requirements engineering method involves interviews with the staff involved in order to elicit the system's requirements. In our example, stakeholder analysis yielded, among others, the following "client authorities" who laid down the procedures deployed by the LAS. The requirements document for the LAS contains information such as the following:

—**Stakeholder 1: LAS Incident Room Controller**
  (1) A medical emergency is either the result of an illness or accident.
  (2) On receipt of a phone call reporting a medical emergency, the nearest available ambulance should be despatched to the scene.
  (3) On receipt of a phone call, if the incident is judged not to be a medical emergency, then the call should be transferred to another emergency service (e.g., police or fire brigade).

—**Stakeholder 2: Operations Manager**
  (1) On receipt of a phone call reporting an incident, the nearest available ambulance should be despatched to the scene.
  (2) On receipt of a phone call reporting an incident, if an ambulance is not the nearest available, then it should not be despatched to the scene.

—**Stakeholder 3: Logistics Manager**
  (1) If no ambulance operators (drivers/medics) are available for an ambulance, then the ambulance is not the nearest available ambulance.
  (2) If an ambulance is not the nearest available ambulance, then do not initiate a call for that ambulance.
  (3) If one year has passed since the maintenance work was last done on an ambulance, then perform a safety check on that ambulance.

Below we consider a preliminary version of a formal specification of these requirements.

6.2 Preliminary Specification

From the above requirements document, we can generate agent hierarchies, dataflow diagrams, action tabulations, object diagrams, and so on. Rather than using a representation scheme that some readers are not familiar with, we will instead just use QC logic (as presented in this article), directly, to represent this specification information. Note, *Person*, *Location*, and *Vehicle* are variable symbols (and not sort names).

—**Stakeholder 1: LAS Incident Room Controller**

$\{a\} : \forall Person, Location,$
$accident(Person, Location) \vee illness(Person, Location) \leftrightarrow medical\text{-}emergency(Person, Location)$

$\{b\} : \forall Person, Location, Vehicle,$
$call(Person, Location) \wedge medical\text{-}emergency(Person, Location) \wedge nearest\text{-}ambulance\text{-}available(Vehicle) \rightarrow despatch\text{-}ambulance(Vehicle, Location)$

$\{c\} : \forall Person, Location,$
$call(Person, Location) \wedge \neg medical\text{-}emergency(Person, Location) \rightarrow (transfer\text{-}to\text{-}police(Person, Location)$ $(\vee\ transfer\text{-}to\text{-}fire\text{-}service(Person, Location))$

—**Stakeholder 2: Operations Manager**

$\{d\} : \forall Person, Location, Vehicle,$
$call(Person, Location) \wedge nearest\text{-}ambulance\text{-}available(Vehicle) \rightarrow despatch\text{-}ambulance(Vehicle, Location)$

$\{e\} : \forall Person, Location, Vehicle,$
$call(Person, Location) \wedge \neg nearest\text{-}ambulance\text{-}available(Vehicle) \rightarrow \neg despatch\text{-}ambulance(Vehicle, Location)$

—**Stakeholder 3: Logistics Manager**

$\{f\} : \forall Vehicle\ \neg has\text{-}crew(Vehicle) \rightarrow \neg \text{nearest-ambulance-available}(Vehicle)$

$\{g\} : \forall Vehicle\ \neg nearest\text{-}ambulance\text{-}available(Vehicle) \rightarrow no\text{-}call\text{-}issued(Vehicle)$

$\{m\} : \forall Vehicle\ over\text{-}one\text{-}year\text{-}since\text{-}last\text{-}maintenance(Vehicle) \rightarrow initiate\text{-}ambulance\text{-}safety\text{-}check(Vehicle)$

We are now in a position to analyze the preliminary requirements specification using the inconsistency management techniques described in this article.

6.3 Inconsistency Management

For clarity of presentation, we work through our example in the same order as the techniques described in Sections 3, 4 and 5 were presented. We then briefly evaluate the techniques, highlighting strengths and weaknesses that have emerged from our case study.

6.3.1 *Reasoning.* If we want to check certain scenarios with regard to the preliminary specification, we must add further relevant facts (e.g., domain knowledge) to model each scenario. For example, consider the following facts:

$$\{h\} : accident(John, London\text{-}Road)$$
$$\{i\} : call(John, London\text{-}Road)$$
$$\{j\} : \neg has\text{-}crew(Ambulance1)$$

$$\{k\} : \neg illness(John, London\text{-}Road)$$
$$\{n\} : over\text{-}one\text{-}year\text{-}since\text{-}last\text{-}maintenance(Ambulance2)$$
$$\{p\} : nearest\text{-}ambulance\text{-}available(Ambulance1)$$

From this scenario and the preliminary specification, we can generate the following (inconsistent) inferences:

$$\{a, b, h, i, p\} : despatch\text{-}ambulance(Ambulance1, London\text{-}Road)$$

$$\{e, f, i, j\} : \neg despatch\text{-}ambulance(Ambulance1, London\text{-}Road)$$

We analyze this inconsistency in the next section (6.3.2).

Nevertheless, using QC logic, we can still continue reasoning with the above facts, together with the preliminary specification, to generate additional inferences such as the following:

$$\{f, g, j\} : no\text{-}call\text{-}issued(Ambulance1)$$

$$\{m, n\} : initiate\text{-}ambulance\text{-}safety\text{-}check(Ambulance2)$$

So even though the assumptions are inconsistent, we can generate other useful inferences. It is possible then, for example, to develop a definition for the no-call-issued procedure or the initiate-ambulance-safety-check procedure, without necessarily having to resolve the inconsistencies in the preliminary specification (although one may want to perform the analysis below before developing a definition for a procedure that may later have to be retracted).

6.3.2 *Analysis: Qualifying Inferences.* The following inferences are only existential inferences, and hence need to be treated with caution when the specification is revised or analyzed further.

$$\{a, b, h, i, p\} : despatch\text{-}ambulance(Ambulance1, London\text{-}Road)$$

$$\{e, f, i, j\} : \neg despatch\text{-}ambulance(Ambulance1, London\text{-}Road)$$

In contrast, the following inference is a universal inference, and hence is less likely to be retracted when the specification is revised.

$$\{f, g, j\} : no\text{-}call\text{-}issued(Ambulance1)$$

The following is a free inference from the specification, which is reassuring given the preliminary nature of the specification.

$$\{m, n\} : initiate\text{-}ambulance\text{-}safety\text{-}check(Ambulance2)$$

6.3.3 *Analysis: Identifying Sources of Inconsistency.* For the inconsistency identified above, there are two sets of labels, in particular, that refer to problematical data. These are the labels attached to the conflicting inferences generated above, $\{a, b, h, i, p\}$ and $\{e, f, i, j\}$. There are many possible sources of the inconsistency. However, if we assume the facts we added for the scenario, labeled from the set $\{h, i, j, k, n, p\}$, are not causing the problem, we order these *above* the set of labels, $\{a, b, c, d, e, f, m\}$, referring to the preliminary specification. Using this ordering, we obtain a smaller subset containing the likely sources of the inconsistency, namely $\{a, b, e, f\}$. These pieces of procedural information were elicited from all three stakeholders, who may now need to be consulted again in order to rectify this problem (although we may also have some ordering of information according to the particular participant from which it was elicited, e.g., "the boss is always right"!).

6.3.4 *Action.* Qualifying the kinds of inconsistency in our specification information, and identifying their possible sources, provides us with some insight (and possibly guidance) about the kinds of handling actions that can be taken in the presence of such inconsistency. So, for example, analysis of the inconsistency in issuing ambulance dispatch requests was probably caused by the conflicting rules laid down by the Incident Room Controller and Operations Managers. An action informing these two individuals of the problem (and even invoking a negotiation support tool) might be appropriate in this setting. More sophisticated actions could also include the invocation of a decision support system that chose between alternatives (perhaps based on the seniority of the individuals involved). Metalevel inconsistency-handling rules such as those very briefly described in Section 5 can be used to specify such actions:

$$
\begin{aligned}
&[\mathit{data}(\Delta_1) \wedge \mathit{data}(\Delta_2)\\
&\wedge \mathit{union}(\Delta_1, \Delta_2) \vdash \bot\\
&\wedge \mathit{inconsistency\text{-}source}(\mathit{union}(\Delta_1, \Delta_2), S)\\
&\wedge \mathit{likely\text{-}conflict\text{-}between\text{-}staff}(S)\\
&\wedge \mathit{LAST}^1\ \mathit{likely\text{-}conflict\text{-}between\text{-}staff}(S)]\\
&\qquad \rightarrow \mathit{NEXT}\ \mathit{invoke\text{-}a\text{-}decision\text{-}based\text{-}on\text{-}seniority}(S)
\end{aligned}
$$

This (simplified) rule states that if there is an inconsistency in the specification data, $\Delta_1 \cup \Delta_2$, and the likely source, S, of the inconsistency is a conflict between two development participants, and this problem has already been established (in the last time unit), then suggest a decision based on the seniority of the staff involved in the conflict (as mentioned at the end of the last section).

6.4 Evaluation

The application of the inconsistency management techniques to the above scenario supports the thesis that our logic-based approach provides simple mechanical tools for living with, even making use of, inconsistent specification information. The techniques allow continued classical reasoning in the presence of inconsistency and provide some analysis and tracking tools for managing such reasoning. If one accepts that such techniques should be used as part of a larger software engineering toolbox, then the kind of guidance they provide can be very useful to the specification developer.

It is clear, however, that deciding and modeling what kinds of action are most appropriate in the presence of different kinds of inconsistency are still difficult and open issues. Nevertheless, our analysis and reasoning techniques do offer potentially useful input to the decision-action process.

7. AUTOMATED REASONING AND TOOL SUPPORT

There are three broad areas of inconsistency management that are amenable to automated support. The first is in the area of inconsistency *detection*. Here, the emphasis is on the actual process of consistency checking in which the tools essentially detect if a rule has been broken. In a logic-based approach, this amounts to the detection of a logical inconsistency in which both a fact, α, and its negation, $\neg\alpha$, are found to hold simultaneously. Many commercial CASE tools implicitly adopt this strategy (e.g., Software through Pictures, Rational Rose, HOODNice), and simple prototypes can be constructed easily in logic-based languages such as Prolog. However, given the intractability of classical consistency checking for propositional logic, and the undecidability for first-order logic, consistency-checking systems should be developed to exploit user intervention.

The second area is that of inconsistency *classification*. Here, the emphasis is on identifying the kind of inconsistency that has been detected in, or between, partial specifications. Tool support for this kind of activity is more limited, because there is little work that attempts to classify inconsistencies. Some tools (for example CONMAN [Schwanke and Kaiser 1988, Section 10] search for one of a predefined number of kinds of inconsistency and try to match the detected inconsistency with one of these.

The third area is that of inconsistency *handling*, which offers varied and challenging scope for tool support. For example, negotiation-support tools can be used to facilitate removing inconsistencies and resolving conflicts. There are fewer tools available however that explicitly support reasoning and analysis in the manner we have described in this article. Nevertheless, tools that support automated reasoning and deduction are available (for example Fitting [1990]). Such tools can take assumptions, logic proof rules, and queries as input and make automatic deductions from this information. In order to make our approach feasible within the context of software development as a whole, we envisage reasoning and analysis tools to operate “in the background”—trying to make inferences and provide useful analyses of inconsistent specifications while development participants are engaged in other development activities. Some theorem provers [Lindsay 1988] provide inconsistency-handling support in this way by attempting to prove that a description (e.g., a specification) satisfies a set of properties or contains no contradictions.

Our intention is to provide the kind of tool support described above within our multiperspective software development environment, The Viewer [Nuseibeh and Finkelstein 1992]. We have already extended The Viewer to support decentralized process enactment, including consistency checking [Leonhardt et al. 1995]. This extension allows consistency checks between partial specifications (ViewPoints) to be invoked and applied in a controlled way, by specifying actions that can take place in certain, predefined, “situations” (as described in Section 5). For example, a consistent specification which has been subsequently edited by a developer is a kind of “situation” that we can discover (pattern match on its description as a regular expression in our particular implementation) and act upon accordingly (e.g., by invoking further consistency checking). We intend to supplement this with more sophisticated, Prolog-based, analysis and reasoning computational engines.

Finally, following on from work with Philips Research Labs on a Collaborative Requirements Engineering Support Tool (CREST) [Bearne 1996], we are currently developing a distributed, web-based implementation of the above using Java [Russo et al. 1998].

8. RELATED WORK

The overwhelming majority of work on consistency management has dealt with tools and techniques for maintaining consistency and avoiding inconsistency. Increasingly, however, researchers have begun to study the notion of consistency in software systems and have recognized the need to formalize this notion. For example, Hagensen and Kristensen have explicitly explored the consistency perspective in software development [Hagensen and Kristensen 1992]. The focus of their work is on the structures for representing information ("descriptions") and the relations between these structures. Consistency of descriptions is defined as relations between interpretations of descriptions. Consistency-handling techniques in software systems modeled in terms of descriptions, interpretations, and relations are also proposed.

We are not aware of any related work on explicitly analyzing inconsistent specifications in the manner we have described. However, a number of researchers have recognized the need to (1) tolerate inconsistency in software development and (2) provide ways of acting in the presence of inconsistency.

Schwanke and Kaiser have proposed an approach to "living with inconsistency" during development (implemented in the CONMAN programming environment) by identifying and tracking six different kinds of inconsistencies (without requiring them to be removed), reducing the cost of restoring type safety after a change (using a technique called "smarter recompilation"), and protecting programmers from inconsistent code (by supplying debugging and testing tools with inconsistency information) [Schwanke and Kaiser 1988]. The analysis of inconsistencies however is limited to identifying one of six predefined types of inconsistency in programming code.

Balzer has proposed the notion of "tolerating inconsistency" by relaxing consistency constraints during development [Balzer 1991]. The approach suggests that inconsistent data be marked by guards "pollution markers" that have two uses: (1) to identify the inconsistent data to code segments or human agents that may then help resolve the inconsistency and (2) to screen the inconsistent data from other segments that are sensitive to the inconsistencies. This approach, however, provides little analysis of the kinds of inconsistency present, preferring to focus on avoiding inconsistencies and leaving any analysis to external agents.

Narayanaswamy and Goldman proposed "lazy" consistency as the basis for cooperative software development [Narayanaswamy and Goldman 1992]. This approach favors software development architectures where impending or proposed changes as well as changes that have already occurred—are "announced." This allows the consistency requirements of a system to be "lazily" maintained as it evolves. The approach is a compromise between the optimistic view in which inconsistencies are assumed to occur infrequently and can thus be handled individually when they arise, and a pessimistic approach in which inconsistencies are prevented from ever occurring. A compromise approach is particularly realistic in a distributed development setting where conflicts or "collisions" of changes made by different developers may occur. Most of the analysis supported by this approach is preemptive in nature, that is, before actual inconsistencies are detected.

Zave and Jackson have proposed the construction of system specifications by composing many partial specifications, each written in a specialized language that is best suited for describing its intended area of concern [Zave and Jackson 1993]. They further propose the composition of partial specifications as a conjunction of their assertions in a form of classical logic. A set of partial specifications is then consistent if and only if the conjunction of their assertions is satisfiable. The approach is demonstrated using partial specifications written in Z and a variety of state-based notations. The approach identifies consistency checking as problematic, an issue we have partly addressed in this article. We have also taken consistency checking a step further by adapting classical logic in order to continue reasoning and analysis in the presence of inconsistency.

Work on programming languages which are supported by exception-handling mechanisms that deal with errors resulting from built-in operations (e.g., division by zero) is also relevant. Building on this work, Borgida proposed an approach to handling violations of assumptions in a database [Borgida 1985]. His approach provides for "blaming" violations on one or more database facts. In this way, either a program can be designed to detect and treat "unusual" facts, or a database can adjust its constraints to tolerate the violation in the data. Balzer's approach described above is based on Borgida's mechanisms.

Feather has recently also proposed an approach to modularized exception handling [Feather 1996] in which programs accessing a shared database of information impose their own assumptions on the database, and treat exceptions to those assumptions differently. The assumptions made by each program together with their respective exception handlers are used to provide each program with its own individual view of the database. Alternative—possibly inconsistent—views of the same information can therefore be used to support different users or developers of a software system. Each program's view is derived from the shared data in such a way as to satisfy all the program's assumptions. This is achieved by a combination of ignoring facts that hold in the shared data and "feigning" facts that do not hold.

Finally, a small body of work addresses inconsistencies that arise in software development processes themselves. For example, an inconsistency may occur between a software development process definition and the actual (enacted) process instance [Dowson 1993]. Such an inconsistency between “enactment state” and “performance state” is often avoided by blocking further development activities until some precondition is made to hold. Since this policy is overly restrictive, many developers attempt to fake conformance to the process definition (for example, by fooling a tool into thinking that a certain task has been performed in order to continue development). Cugola et al. [1995; 1996] have addressed exactly this problem in their temporal logic-based approach which is used to capture and tolerate some deviations from a process description during execution. Deviations are tolerated as long as they do not affect the correctness of the system (if they do, the incorrect data must be fixed, or the process model—or its active instance—must be changed). Otherwise, deviations are tolerated, recorded, and propagated—and “pollution analysis” (based on logical reasoning) is performed to identify possible sources of inconsistency.

The framework we have described in this article also provides more sophisticated options than truth maintenance (such as Doyle [1979] and Kleer [1986]) for managing inconsistency specifications. These include (1) paraconsistent reasoning with sets of inconsistent formulae, (2) labeling strategies to allow inconsistent formulae to be tracked, and likely sources identified using metalevel information, and (3) reasoning with universal and free inferences in addition to reasoning with existential inferences.

From the AI and logics communities there have been a number of other proposals that are of relevance, including fuzzy sets and nonmonotonic logics (for a review, see Krause and Clark [1993]). While they constitute important developments that could be incorporated into our framework, they are not directly oriented to the inconsistency management issues that we consider within this article. In the main they are focused on resolving inconsistency by finding the best possible inferences for any given set of information, whereas we really need to be able to analyze inconsistent information, consider options, and track information to find likely sources of inconsistency.

9. CONCLUSIONS

Our earlier work began by providing a framework for multiperspective software development in which multiple development participants, and the partial specifications they maintained, were represented by ViewPoints. The inconsistencies that inevitably arose between multiple overlapping ViewPoints led us to adopt an inconsistency-handling approach that was tolerant of such inconsistencies. This approach relied on identifying inconsistencies, the context in which they arose, and the actions that could be performed in their presence. We further recognized that such actions did not need to remove inconsistencies immediately, but rather allowed continued reasoning and development in their presence. Keeping track of deductions made during reasoning, and deciding what actions to perform in the presence of inconsistencies, identified the need to analyze inconsistencies in this context. This article addressed such inconsistency-handling activities in a formal setting.

The article explored the use of a logic-based approach to reasoning in the presence of inconsistency. We demonstrated how partial specifications might be translated into classical logic in order to detect inconsistencies between them. To overcome the trivialization of classical logic that results when an inconsistency is detected, we proposed the use of QC logic which allows continued development in the presence of inconsistency.

The use of logic provided us with a precise and unambiguous language in which to identify inconsistencies in evolving multiperspective specifications. It also provided us with the means to address issues of inconsistency management in a generic way that is independent of any particular software engineering method or formalism. Furthermore, QC logic has provided us with a formal foundation upon which we can build more sophisticated inconsistency-handling and reasoning mechanisms.

We also examined the use of labeled QC logic to "audit" reasoning results and to "diagnose" inconsistencies. The labels facilitated the identification of likely sources of inconsistencies. Such logical analysis also provided us with guidance about the actions one can perform in the presence of particular inconsistencies (for example, actions to resolve a conflict, delay resolution, ameliorate an inconsistent specification, etc.). Our immediate research agenda is to examine these inconsistency-handling actions further within our framework. One line of research we are investigating is the analysis of the consequences of taking different development actions to guide developers in choosing between alternatives [Nuseibeh and Russo 1998]. QC logic allows us to assess the impact of taking different inconsistency-handling actions that do not necessarily remove inconsistency from a specification.

The version of QC logic presented here is effectively propositional in expressibility. While this expressibility may be sufficient for some specifications, we will require a full first-order version of QC logic for more complex inconsistency-handling problems in requirements engineering. As can be seen in the case study, it is not possible to express certain kinds of useful specification without existential quantification, e.g., for the requirements statement "On receipt of a request for an ambulance from a neighboring ambulance station, if there is an ambulance available, then it should be lent to that ambulance station." This is the subject of current research.

We believe that our work provides the foundations for supporting a software specification process in which inconsistencies are analyzed to determine the course of action needed for further development. This recognizes the evolutionary nature of software development and provides a formal, yet flexible, mechanism for managing inconsistencies.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions and feedback of Alex Borgida, Gianpaolo Cugola, Elisabetta Di Nitto, Martin Feather, Anthony

Finkelstein, Dov Gabbay, Carlo Ghezzi, Michael Goedicke, Jeff Kramer, Jonathan Moffett, Alessandra Russo, and Axel van Lamsweerde.

REFERENCES

ATKINSON, W. AND CUNNINGHAM, J. 1991. Proving properties of safety-critical systems. *BCS/IEE Software Engineering Journal 6*, 2, 41–50.

BALZER, R. 1991. Tolerating inconsistency. In *Proceedings of 13th International Conference on Software Engineering (ICSE-13)* (1991), pp. 158–165. IEEE Computer Society Press.

BARAL, C., KRAUS, S., MINKER, J., AND SUBRAHMANIAN, V. 1992. Combining knowledge bases consisting of first order theories. *Computational Intelligence 8*, 45–71.

BEARNE, M. 1996. Hypermedia visualization for system requirements in a collaborative distributive environment. Technical Report TN3491, Philips Research Labs, Redhill, Surrey.

BELNAP, N. 1977. A useful four-valued logic. In G. Epstein Ed., *Modern Uses of Multiple-Valued Logic* (1977), pp. 8–37. Reidel.

BENFERHAT, S., DUBOIS, D., AND PRADE, H. 1993. Argumentative inference in uncertain and inconsistent knowledge bases. In *Proceedings of Uncertainty in Artificial Intelligence* (1993). Morgan Kaufmann.

BESNARD, P. 1991. Paraconsistent logic approach to knowledge representation. In M. de Glas M and D. G. D. Eds., *Proceedings of the First World Conference on Fundamentals of Artificial Intelligence* (1991), pp. 107–114. Angkor.

BESNARD, P. AND HUNTER, A. 1995. Quasi-classical logic: Non-trivializable classical reasoning from inconsistent information. In C. Froidevaux and J. Kohlas Eds., *Symbolic and Quantitative Approaches to Uncertainty*, Volume 946 of *Lecture Notes in Computer Science* (1995), pp. 44–51.

BIBEL, W. 1993. *Deduction: Automated logic.* Academic Press.

BORGIDA, A. 1985. Language features for flexible handling of exceptions in information systems. *Transactions on Database Systems 10*, 4, 565–603.

BOURLEY, C., CAFERRA, R., AND PELTIER, N. 1994. A method for building models automatically: Experiments with an extension of otter. In *Proceedings of the Twelfth Conference on Computer-Aided Deduction (CADE-12)*, Volume 814 of *Lecture Notes in Computer Science* (1994), pp. 72–86. Springer.

BREWKA, G. 1989. Preferred subtheories: An extended logical framework for default reasoning. In *Proceedings of the Eleventh International Conference on Artificial Intelligence* (1989), pp. 1043–1048.

CAFERRA, R. AND PELTIER, N. 1995. Model building and interactive theory discovery. In *Fourth Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, Volume 918 of *Lecture Notes in Computer Science* (1995), pp. 154–168. Springer.

CAFERRA, R. AND ZABEL, N. 1993. Building models by using tableaux extended by equational problems. *Journal of Logic and Computation 8*, 3–25.

CLARKE, E. AND WING, J., ET AL. 1996. Formal methods: State of the art and future directions. *ACM Computing Surveys 28*, 4, 626–646.

COSTA, M., CUNNINGHAM, R., AND BOOTH, J. 1990. Logical animation. In *Proceedings of the Twelth International Conference on Software Engineering* (Nice, 1990), pp. 144–149. IEEE Computer Society Press.

CUGOLA, G., NITTO, E. D., FUGGETTA, A., AND GHEZZI, C. 1996. A framework for formalizing inconsistencies and deviations in human-centered systems. *ACM Transactions on Software Engineering and Methodology 5*, 3, 191–230.

CUGOLA, G., NITTO, E. D., GHEZZI, C., AND MANTIONE, M. 1995. How to deal with deviations during process model enactment. In *Proceedings of 17th International Conference on Software Engineering (ICSE-17)* (Seattle, USA, 1995), pp. 265–273. ACM Press.

DA COSTA, N. C. 1974. On the theory of inconsistent formal systems. *Notre Dame Journal of Formal Logic 15*, 497–510.

DOWSON, M. 1993. Consistency maintenance in process sensitive environments. In *Proceedings of Workshop on Process Sensitive Environments Architectures* (Boulder, Colorado, USA, 1993). Rocky Mountain Institute of Software Engineering (RMISE).

DOYLE, J. 1979. A truth maintenance system. *Artificial Intelligence 12*, 231–272.

EASTERBROOK, S. AND NUSEIBEH, B. 1995. Managing inconsistencies in an evolving specification. In *Proceedings of 2nd International Symposium on Requirements Engineering (RE '95)* (1995), pp. 48–55. IEEE Computer Society Press.

EASTERBROOK, S. AND NUSEIBEH, B. 1996. Using viewpoints for inconsistency management. *BCS/IEE Software Engineering Journal 11*, 1, 31–43.

ELVANG-GORANSSON, M. AND HUNTER, A. 1995. Argumentative logics: Reasoning from classically inconsistent information. *Data and Knowledge Engineering Journal 16*, 125–145.

FEATHER, M. 1996. Modularized exception handling. In *Proceedings of International Workshop on Multiple Perspectives in Software Development (Viewpoints 96)* (1996), pp. 167–171. ACM Press.

FINKELSTEIN, A. AND DOWELL, J. 1996. A comedy of errors: The London Ambulance Service case study. In *Proceedings of 8th International Workshop on Software Specification and Design (IWSSD-8)* (1996), pp. 2–4. IEEE Computer Society Press.

FINKELSTEIN, A., GABBAY, D., HUNTER, A., KRAMER, J., AND NUSEIBEH, B. 1994. Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering 20*, 8, 569–578.

FINKELSTEIN, A., KRAMER, J., NUSEIBEH, B., FINKELSTEIN, L., AND GOEDICKE, M. 1992. Viewpoints: A framework for multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering (Special issue on Trends and Future Research Directions in Software Engineering Environments) 2*, 1, 31–57.

FITTING, M. 1990. *First-order Logic and Automated Theorem Proving.* Springer.

GABBAY, D. AND HUNTER, A. 1991. Making inconsistency respectable 1: A logical framework for inconsistency in reasoning, In *Fundamentals of Artificial Intelligence*, Volume 535 of *Lecture Notes in Computer Science* (1991), pp. 19–32. Springer.

GABBAY, D. AND HUNTER, A. 1993. Making inconsistency respectable 2: Meta-level handling of inconsistent data. In *Proceedings of the European Conference on Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ECSQARU '93)*, Volume 747 of *Lecture Notes in Computer Science* (1993), pp. 129–136. Springer.

GABBAY, D. AND HUNTER, A. 1998. Negation and contradiction. In *What is negation?* Kluwer.

HAGENSEN, T. M. AND KRISTENSEN, B. B. 1992. Consistency in software system development: Framework, model, techniques and tools. *Software Engineering Notes (Proceedings of ACM SIGSOFT Symposium on Software Development Environments) 17*, 5, 58–67.

HUNTER, A. 1996. Reasoning with contradictory information using quasi-classical logic. In *Technical report* (1996). Department of Computer Science, University College London. Available from www.cs.ucl.ac.uk/staff/a.hunter.

HUNTER, A. 1998. Paraconsistent logics. In *Handbook of Defeasible Reasoning and Uncertainty Management* (1998). Kluwer.

HUNTER, A. AND NUSEIBEH, B. 1997. Analysing inconsistent specifications. In *Proceedings of 3rd International Symposium on Requirements Engineering* (1997), pp. 78–86. IEEE Computer Society Press.

KLEER, J. D. 1986. An assumption-based TMS. *Artificial Intelligence 28*, 127–162.

KRAUSE, P. AND CLARK, D. 1993. *Representing Uncertain Knowledge.* Intellect.

L WOS, E. L., R OVERBEEK, AND BOYLE, J. 1984. *Automated Reasoning: Introduction and Applications.* Prentice Hall.

LEONHARDT, U., FINKELSTEIN, A., KRAMER, J., AND NUSEIBEH, B. 1995. Decentralised process enactment in a multi-perspective development environment. In *Proceedings of 17th International Conference on Software Engineering (ICSE-17)* (1995), pp. 255–264. IEEE Computer Society Press.

LINDSAY, P. A. 1988. A survey of mechanical support for formal reasoning. *BCS/IEE Software Engineering Journal (special issue on mechanical support for formal reasoning) 3*, 1.

MCCUNE, W. 1990. OTTER 2.0 user's guide. Technical Report ANL-90/9, Argonne National Laboratory, Argonne, Illinois.

NARAYANASWAMY, K. AND GOLDMAN, N. 1992. Lazy consistency: A basis for cooperative software development. In *Proceedings of International Conference on Computer-Supported Cooperative Work (CSCW '92)* (1992), pp. 257–264. ACM SIGCHI and SIGOIS.

NUSEIBEH, B. 1996. To be and not to be: On managing inconsistency in software development. In *Proceedings of 8th International Workshop on Software Specification and Design (IWSSD-8)* (1996), pp. 164–169. IEEE Computer Society Press.

NUSEIBEH, B. AND FINKELSTEIN, A. 1992. Viewpoints: A vehicle for method and tool integration. In *Proceedings of 5th International Workshop on Computer-Aided Software Engineering (CASE '92)* (Montreal, Canada, 1992), pp. 50–60. IEEE Computer Society Press.

NUSEIBEH, B., KRAMER, J., AND FINKELSTEIN, A. 1994. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering 20*, 10, 760–773.

NUSEIBEH, B. AND RUSSO, A. 1998. On the consequences of acting in the presence of inconsistency. In *Proceedings of the Ninth International Workshop on Software Specification and Design* (Ise-shima, Japan, 1998). IEEE Computer Society Press.

POOLE, D. 1985. A logical framework for default reasoning. *Artificial Intelligence 36*, 27–47.

PRAKKEN, H. 1993. An argument framework for default reasoning. In *Annals of mathematics and artificial intelligence*, Volume 9 (1993).

RUSSO, A., NUSEIBEH, B., AND KRAMER, J. 1998. Restructuring requirements specifications for inconsistency analysis: A case study. In *Proceedings of the Third International Conference on Requirements Engineering (ICRE98)* (Colorado Springs, USA, 1998). IEEE Computer Society Press.

RYAN, M. 1992. Representing defaults as sentences with reduced priority. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference* (1992). Morgan Kaufmann.

SCHUMANN, J. 1993. SCOTT: A model-guided theorem prover. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI '93)* (1993), pp. 109–114.

SCHUMANN, J. 1994. SETHEO V3.2: Recent developments. In *Proceedings of the Twelfth Conference on Computer-Aided Deduction (CADE-12)*, Volume 814 of *Lecture Notes in Computer Science* (1994). Springer.

SCHWANKE, R. W. AND KAISER, G. E. 1988. Living with inconsistency in large systems. In *Proceedings of the International Workshop on Software Version and Configuration Control* (1988), pp. 98–118. B G Teubner.

SLANEY, J. 1996. FINDER: Finite domain enumerator version 3 notes and guide. Technical report, Centre for Information Science Research, Australian National University.

ZAVE, P. AND JACKSON, M. 1993. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology 2*, 4, 379–411.

Received October 1995; revised March 1996, July and October 1997, and March 1998; accepted May 1998