Managing Inconsistent Specifications: Reasoning, Analysis and Action*

Anthony Hunter and Bashar Nuseibeh

Department of Computing
Imperial College
London SW7 2BZ, UK
{abh, ban}@doc.ic.ac.uk

October 13, 1995

Abstract

In previous work, we have advocated continued development of specifications in the presence of inconsistency. To support this, we have 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 labelled QC logic that records and tracks assumptions used in reasoning. This facilitates a logical analysis of inconsistent information. We discuss the application of labelled QC logic in the analysis of multi-perspective specifications. Such specifications are developed by multiple participants who hold overlapping, often inconsistent, views of the systems they are developing. Finally, we discuss further the notion of acting in the presence of inconsistency, and examine the use of meta-level inconsistency handling rules to support such action. The feasibility of automated support for this kind of inconsistency handling is also discussed, and related work in the area is critically reviewed.

1 Introduction

In a previous paper [FGH+94] we advocated the need to tolerate inconsistencies in software development, and more importantly to be able to act in a context-

*Department of Computing Technical Report Number 95/15

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 an inference. Hence, it does not provide a means for continued deduction in the presence of inconsistency. Moreover, attempts to formalise 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" [GH91, GH93].

In this paper, we present a formal approach that supports continued action (including reasoning) in the presence of inconsistency, and facilitates the recording and tracking of (inconsistent) information during reasoning. The paper is organised as follows.

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 multi-perspective 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 analysing 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 sections 4 and 5.

Analysis We then address the need to analyse inconsistent specifications in the above setting. In particular, we propose the use of *labelled* QC logic that records and tracks information used in reasoning. We illustrate how the amended (labelled) 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 labelled QC logic can be analysed 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 6.

Action Finally, we discuss the need, and an approach, to acting in the pres-

ence of inconsistency. We suggest the use of meta-level rules of the form "inconsistency implies action", but empasise that such inconsistency handling actions need not necesarily remove the inconsistency. Rather, action in this setting may include ameliorating, but not necessarily resolving, the inconsistency. This work is covered in section 7.

We preface and conclude the paper with a discussion on the context of the work, namely, multi-perspective software development within the "ViewPoints framework". ViewPoints provide an organisational 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 paper provides a means of managing such inconsistencies.

Our approach is necessary because we need to deal with inconsistent specifications in a way that allows us to analyse the likely sources of the inconsistencies, allow 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 paper with a critical review of related work in the field.

2 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:

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" [FKN+92].

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" [NKF94]. These rules describe relationships between representation schemes, and thus the relationships between partial specifications expressed using those representation schemes.

Domain information which is incrementally captured in an evolving specification process [EN95a, EN95b]. This information might pertain to the

problem domain in which the system will be installed (and which is usually captured and represented separately). Alternatively, it might pertain to the solution domain (i.e., the product being specified), in which case it will be part of the specification information.

All three kinds of information described above (and represented as facts, rules, assumptions, graphs, etc.) can be translated into logical formulae, and inconsistences can be detected logically. 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 present in the specification.

3 Capturing development information in logic

Different kinds of development information can be translated into logical formulae which can then be handled in a uniform manner [FGH+94]. 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. Consider, for example, an entity-relationship description of a bank (which is based on a set of nodes, denoting entities, and binary arcs, denoting relations between entities). We can easily capture such a representation scheme as illustrated by the following formulae:

$$
\begin{array}{l}
holds(Bank, Account) \\
has(Customer, Account) \\
\vdots \\
inheritance(Cashier\text{-}transaction, Transaction)
\end{array}
$$

The symbols $Bank$, $Account$, $Customer$, $Account$, $Cashier\text{-}transaction$, and $Transaction$ are objects in the logic, the $holds$, has, and $inheritance$ symbols are relations in the logic. In addition, we assume formulae that capture the special nature of certain relations such as the following:

$$\forall X, Y, Z,\ inheritance(X, Y) \wedge inheritance(Y, Z) \rightarrow inheritance(X, Z)$$

Similarly, we can describe, various constraints on our formalisms, such as the relationships between representation schemes or partial specifications. For example, consider the following simple rule for ensuring consistency of use of the syntax for multiplicity of association between two classes in an object model [RBP+91]. Assume the relations *has-exactly-one* and *has-exactly-two* between pairs of classes (for example, *has-exactly-one(Son,Mother)*). For these, we have the following rule:

$$\forall X, Y,\ has\text{-}exactly\text{-}one(X,\ Y) \leftrightarrow \neg has\text{-}exactly\text{-}two(X,\ Y)$$

Constraints can be useful as either specification information or domain information. They can incorporate useful domain knowledge identified by a developer, or they can be the result of some agreement between developers. Furthermore, they can be added without their utility being immediately obvious.

4 Reasoning in the presence of inconsistency

Here we 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.

4.1 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 Z specifications, ER diagrams, dataflow diagrams, and inheritance hierarchies. 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 semi-decidable

for the first-order case. Whilst this is not as good as being decidable, it means that if a formula does hold in a 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 we can construct a model of a consistent set of formulae.

In addition, there has been much progress in developing technology for classical reasoning. This includes automated reasoning systems for deriving inferences from sets of formulae [Fit90], and model building systems for giving models of consistent sets of formulae [CCB90, AC91].

4.2 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 trivialised. Classical logic is trivialised because, by the definition of the logic, any inference follows from inconsistent information *(ex falso quodlibet)* as illustrated by the following example.

Example 4.1 *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 and introduction;* $\neg\beta \rightarrow \neg\alpha$ *and so on. In contrast, trivial inferences might include* γ *and* $\gamma \wedge \neg\delta$*.*

For classical logic, trivialisation 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 [dC74]. However, this kind of weakening of the proof rules means that the connectives in the language do not behave in a classical fashion [Bes91]. 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 4.2 *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 follow from Specification-2.*

There are many similar examples that could be confusing and counter-intuitive for users of such a practical reasoning system. An alternative compromise is quasi-classical (QC) logic [BH95]. In the following section we present a development of QC logic called labelled QC logic that is oriented to reasoning in the context of inconsistent specifications.

5 Quasi-classical logic

We begin this section with an informal presentation of QC logic, and conclude with a formal definition of the labelled QC logic.

5.1 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:

$$\frac{\neg\alpha \;\; \alpha \vee \beta}{\beta}$$

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 4.2, α follows from both sets using the QC logic.

To illustrate the use of QC in a multi-perspective development setting, consider the following example.

Example 5.1 *Suppose, we have two partial specifications VP1 and VP2. In VP1, there is the association:*

$$\mathit{has\text{-}exactly\text{-}one}(\mathit{Cashier}, \mathit{Terminal})$$

and in VP2 there is the association:

$$has\text{-}exactly\text{-}two(Cashier, Terminal)$$

Recall, the constraint:

$$\forall X, Y, has\text{-}exactly\text{-}one(X, Y) \leftrightarrow \neg has\text{-}exactly\text{-}two(X, Y)$$

This "inter-ViewPoint rule" together with VP1 and VP2 is inconsistent. However, the definitions of the relations has-exactly-one and has-exactly-two also imply the following constraints.

$$\forall X, Y, has\text{-}exactly\text{-}one(X, Y) \rightarrow has\text{-}one\text{-}or\text{-}more(X, Y)$$

$$\forall X, Y, has\text{-}exactly\text{-}two(X, Y) \rightarrow has\text{-}one\text{-}or\text{-}more(X, Y)$$

These further constraints could be either additional useful domain knowledge, discovered perhaps during development, or the result of agreements between developers. Hence for both VP1 and VP2, and despite the inconsistency between them, we can derive the potentially useful non-trivial inference:

$$has\text{-}one\text{-}or\text{-}more(Cashier, Terminal)$$

Constraints therefore allow us to identify problems in the specifciation information such as one partial specification stating has-exactly-one(Cashier, Terminal) and the other partial specification stating has-exactly-two(Cashier, Terminal). They also allow us to identify interesting ramifications such as has-one-or-more(Cashier, Terminal).

Reasoning such as in the above example is potentially useful for a range of activities in the management of inconsistency. These include diagnosing the source of the inconsistency (section 6), for supporting negotiation between participants in software development, for continued development without immediately resolving the inconsistency, and for deciding on actions to handle the inconsistency (section 7).

5.2 Formal definition

In this section we provide a formal definition of labelled QC logic. We use a labelled language to allow us to uniquely identify each item of development information. We propagate the labels by labelling 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 labelled. Labels can be used to differentiate different types of development information and in particular they can indicate the sources of information. We demonstrate the utility of the labels in section 6.

5.2.1 Language of labelled QC logic

We assume a classical first-order language, though we restrict quantifiers to just ranging over constants. In other words we do not have function symbols. This gives certain computational advantages. In particular, it renders consistency checking decidable.

Definition 5.1 *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 $\mathcal{A} = \{p(q_1, ..., q_n) \mid p \in \mathcal{P}$ and $q_1, ..., q_n \in \mathcal{V} \cup \mathcal{C}\}$. We call $p(q_1, ..., q_n)$ a ground atom iff $q_1, ..., q_n$ are all constant symbols, otherwise we call it unground.*

Example 5.2 *Let has-exactly-one be a predicate symbol; X, Y be variable symbols; and Cashier, Terminal be constant symbols. Then has-exactly-one(X, Y) is an unground atom, has-exactly-one$(X,$Terminal$)$ is an unground atom, and has-exactly-one$($Cashier$,$Terminal$)$ is a ground atom.*

Definition 5.2 *Let $\mathcal{F}$ be the set of classical propositional formulae formed from a set of atoms $\mathcal{A}$, and the $\wedge, \vee, \rightarrow$ and $\neg$ connectives. We abbreviate the formula $\alpha \wedge \neg\alpha$ by the formula $\perp$, which we read as "inconsistency". We call a formula grounded iff it is made from only ground atoms, otherwise we call it unground.*

Definition 5.3 *Let $\mathcal{L}$ be the set of formulae formed from $\mathcal{F}$, where if $\alpha \in \mathcal{F}$, and $x_1, .., x_n$ are the free variables in α, then $\forall x_1, ..., \forall x_n \alpha \in \mathcal{L}$.*

Hence the set $\mathcal{L}$ contains only universally quantified formulae, where the quantifiers are outermost, and ground formulae. This restriction aids our exposition. We provide a full classical language in a later paper.

Definition 5.4 *Let $\vdash_x$ be some consequence relation for some logic X, defined by some proof rules. Then, the logic X is trivialisable if and only if for all α, β in the language of X, $\{\alpha, \neg\alpha\} \vdash_x \beta$.*

Note that classical logic is trivialisable according to this definition. The following two definitions are used to explain the proof theory concisely in the next section.

Definition 5.5 *For each atom $\alpha \in \mathcal{L}$, α is a literal and $\neg\alpha$ is a literal. For $\alpha_1 \vee .. \vee \alpha_n \in \mathcal{L}$, $\alpha_1 \vee .. \vee \alpha_n$ is a clause iff each of $\alpha_1, .., \alpha_n$ is a literal. For $\alpha_1 \wedge .. \wedge \alpha_n \in \mathcal{L}$, $\alpha_1 \wedge .. \wedge \alpha_n$ is in a conjunctive normal form (CNF) iff each of $\alpha_1, .., \alpha_n$ is a clause.*

Definition 5.6 *For $\alpha_1 \wedge .. \wedge \alpha_n \in \mathcal{L}$, and $\beta \in \mathcal{L}$, $\alpha_1 \wedge .. \wedge \alpha_n$ is in CNF of β iff $\alpha_1 \wedge .. \wedge \alpha_n \vdash \beta$ and $\beta \vdash \alpha_1 \wedge .. \wedge \alpha_n$, and $\alpha_1 \wedge .. \wedge \alpha_n$ is in a CNF.*

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 is 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 5.7 *Let $\mathcal{S}$ be some set of atomic symbols such as an alphabet. If $i \subseteq \mathcal{S}$, and $\alpha \in \mathcal{L}$, then $i : \alpha$ is a labelled formula. Let $\mathcal{M}$ be the set of labelled formulae. All development knowledge that we are analysing is translated into labelled formulae, where each formula has a unique label.*

Example 5.3 *Let $\mathcal{S} = \{a, b, c, ...\}$. Then examples of labelled formulae include,*

$$a : holds(bank, account)$$

$$b\text{: } performs(cashier, transaction) \vee performs(ATM, transaction)$$

There are many strategies that we could adopt for labelling development information. Options include combinations of the source of the item, and 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.

5.2.2 Proof theory for labelled QC logic

The proof theory of QC logic provides the power to derive a 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.

We obtain labelled QC logic by using only labelled 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 5.8 *Assume that $\wedge$ is a commutative and associative operator, and $\vee$ is a commutative and associative operator.*

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

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

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

$$\frac{i : \beta \rightarrow \gamma}{i : \neg\beta \vee \gamma} \qquad \frac{i : \neg(\beta \rightarrow \gamma)}{i : \beta \wedge \neg\gamma}$$

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

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

$$\frac{i : \alpha \quad j : \neg\alpha \vee \beta}{i \cup j : \beta} \qquad \frac{i : \alpha \vee \beta \quad j : \neg\alpha}{i \cup j : \beta}$$

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

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

$$\frac{i : \neg(\alpha \wedge \beta)}{i : \neg\alpha \vee \neg\beta} \qquad \frac{i : \neg(\alpha \vee \beta)}{i : \neg\alpha \wedge \neg\beta}$$

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

Labelled QC logic is used by providing any set of labelled formulae (ie. any $\Delta \subseteq \mathcal{M}$*) as assumptions, and any classical formula (ie. any* $\alpha \in \mathcal{L}$*) 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 a CNF of the query, labelled i, from the assumptions using the labelled 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 labelled QC reasoning for ground formulae below and an example with quanitfied formulae in section 9.

Example 5.4 *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,*

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

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

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

$$\{d\} : \gamma$$

Clearly this development information is inconsistent. However, using the QC proof rules, we can obtain a number of non-trivial 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 $\{a, b, c, d\} : \bot$	*from* $\{c\} : \neg\alpha$ *and Step 3*

In Step 2, we obtain the formula β that is labelled with $\{b, d\}$. This formula is non-trivial, and maybe useful for furthering the evolution of the specification. We show in section 6.1, 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 labelled $\{a, b, c, d\}$. 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 6.2, below.

6 Logical analysis of inconsistent specifications

So far we have shown how development information can be represented as logical formaulae, and shown how we can undertake non-trivial 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.

6.1 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 can not also derive $\neg\alpha$ from another consistent susbset of the database.

Definition 6.1 *Let Δ be a set of labelled formulae capturing development information. We form the following sets of sets of formulae,*

$$CON(\Delta) = \{\Gamma \subseteq \Delta \mid \Gamma \not\vdash_Q i : \bot\}$$

$$INC(\Delta) = \{\Gamma \subseteq \Delta \mid \Gamma \vdash_Q i : \bot\}$$

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

Definition 6.2 *Let Δ be a set of labelled formulae capturing development information, then the function Labels is defined as follows,*

$$Labels(\Delta) = \{i \mid i : \alpha \in \Delta\}$$

In the following definition, $MI(\Delta)$ is a set of sets of labels, where each set of labels corresponds to a set 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 labels, where each set of labels corresponds to a set 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 6.3 *Let Δ be a set of labelled formulae capturing development information. We form the following sets of sets of labels,*

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

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

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

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 $\bigcap MC(\Delta)$, as capturing all the "uncontroversial" information in Δ. In contrast, we consider the set $\bigcup MI(\Delta)$ as capturing all the "problematic" data Δ. Note, $\bigcap MC(\Delta)$ is equal to $Labels(\Delta)$ - $\bigcup MI(\Delta)$. 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 $\bigcup MI(\Delta)$.

We now use these concepts to define three qualifications for an inference from inconsistent information. The approach is a derivative of argumentative logics [EGH95].

Definition 6.4 *Let Δ be a set of labelled formulae capturing development information, and let $\Delta \vdash_Q i : \alpha$ hold. We form the following qualifications for inferences,*

$$\alpha \textit{ is an existential inference if } \exists k \in MC(\Delta) \textit{ such that } i \subseteq k$$

$$\alpha \textit{ is an universal inference if } \forall k \in MC(\Delta)\ \exists j \textit{ such that } j \subseteq k \textit{ and } \Delta \vdash_Q j : \alpha \textit{ holds}$$

$$\alpha \textit{ is a free inference if } i \subseteq FREE(\Delta)$$

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 6.1 *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. This could 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.

Whilst there is an overlap for existential inferencing with the approach of truth maintenance systems (for example [Doy79, Kle86]), we go beyond this by adopting universal and free inferencing. Furthermore, by adopting labelling, we integrate our inconsistency management with QC reasoning and with identifying likely sources of inconsistency.

6.2 Identifying likely sources of an inconsistency

When we identify an inconsistency in our development information, we want to analyse that inconsistency before we decide on a course of action (for example, further reasoning, removing inconsistency, etc). Using labelled 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 6.2 *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 $\{a, b, c\} : \bot$, 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 6.5 *Let Δ be a set of labelled formulae capturing development information, 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,*

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

Definition 6.6 *For an inconsistency $i : \bot$, $Formulae(\Delta, j)$ 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 maybe 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 possible source that contains less preferred assumptions.

Example 6.3 *Returning to example 6.2, we can use explicit ordering to formalise 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 6.4 *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 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 [GH95]. Second, there are a number of analyses of ways of handling ordered formuale and sets of ordered formulae. These include the use of specificity [Poo85], ordered theory presentations [Rya92], and prioritized syntax-based entailment [BDP93].

7 Acting in the presence of inconsistency

The logical analysis of inconsistency described in this paper 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 "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.

Acting in the presence of inconsistencies may also require 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 have some preliminary results and observations. We have explored the use of a meta-level approach to prescribe inconsistency handling rules of the form:

INCONSISTENCY IMPLIES ACTION

Our approach deploys 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 [GH93, FGH+94]. Thus for example the rule:

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

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 $LAST^5$ *switched-machine-on* would mean 5 minutes ago the machine was switched on. Also note that:

- $data(\Delta_1)$ and $data(\Delta_2)$ hold if the formulae in the databases Δ_1 and Δ_2, respectively, are logical rewrites of some development information.
- $union(\Delta_1, \Delta_2) \vdash \bot$ holds if the union of the databases Δ_1 and Δ_2 classically implies inconsistency.
- *inconsistency-source*$(union(\Delta_1, \Delta_2), S)$ holds if S is a minimally inconsistent subset of the union of Δ_1 and Δ_2, as defined in section 6.1. The 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 meta-level, we can also include conditions in our rule that we haven't checked this problem at previous points S 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. We have identified at least four possible kinds of actions [Nus95]:

Ignoring the inconsistency completely and continuing development regardless. This may be appropriate in certain circumstances where the inconsistency is isolated and does not prevent further development from taking place.

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.

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.

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 "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 paper.

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

Bearing in mind the above work, we are also interested in supporting multi-perspective 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 decentralised approach in which multiple ViewPoints represent different participants and the views that they hold, we have also examined decentralised process modelling, enactment and support in this context. For example, we have attempted to analyse "work records" of individual ViewPoints, in order to identify some key "situations" to which we know how to react. We have implemented a prototype to support this which uses regular expressions to specify particular situations and rules to associate actions with these situations [LFKN95]. Further work is needed to incorporate the kinds of logical analysis we have described in this paper within the ViewPoints framework. Nevertheless the formal nature of our analysis makes it amenable to automated support.

8 Autmated 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. Numerous commercial CASE tools adopt this strategy, and simple prototypes can be constructed easily in logic-based languages such as Prolog.

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 [SK88], described in section 10) search for one of a pre-defined 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, debuggers and 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 paper. Nevertheless, tools that support automated reasoning and deduction are available (for example [Fit90]). 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 backgound" - trying to make inferences and provide useful analyses of inconsistent specifications while development participants are engaged in other development acticvities. Some theorem provers [Lin88] 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. Therefore, these tools have the capability of reasoning about why an inconsistency exists when a proof cannot be produced.

Our intention is to provide the kind of tool support described above within our multi-perspective software development environment, The Viewer [NF92]. We have already extended The Viewer to support decentralised process enactment, including consistency checking [LFKN95]. 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, pre-defined, situations (as described in section 7). We intend to supplement this with more sophisticated, Prolog-based, analysis and reasoning computational engines.

9 Applications

In order to validate our work, we have completed a series of examples that illustrate the reasoning, analysis and action described in this paper. 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. Our earlier work [NKF94] deliberately focussed on the partitioning of specifications into mangeable 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 indus-

trial 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 multi-perspective development approach.

In the following, we present excerpts from a case study to develop software for an order processing system. The intention is to illustrate the techniques we have described in this paper in the context of a "real world" example. The exposition is somewhat artificial in order to illustrate the issues and contributions presented in the paper.

9.1 Requirements document

Consider the requirements for an order processing system for a wine warehouse. During requirements elicitation a number of the staff involved in order processing are interviewed in order to generate requirements document. Stakeholder analysis determines that the only valid proceedures are those authorized by one of three "client authorities": the managing director, logistics manager or the chief wine taster. The requirements document contains information such as the following.

Managing Director

> Good customers are either government customers or companies that pay promptly.
>
> If we receive an order V for X cases of item Y from company Z, and Z is a good customer, then generate a warehouse request for X cases of item Y, otherwise do credit checks on Z.

Logistics Manager

> If we receive an order V for X cases of item Y from company Z, and Z has a good credit rating, then generate a warehouse request for X cases of item Y.
>
> If we receive an order V for X cases of item Y from company Z, and Z does not have a good credit rating, then do not generate a warehouse request for X cases of item Y.

Chief Wine Taster

> If we receive an order V for X cases of item Y, and Y is a wine that is near the end of its shelf life, then generate a warehouse request V for X cases of item Y.

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

9.2 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 represention scheme that some readers are not familiar with, we will instead just use QC logic (as presented in this paper), directly, to represent this specification information.

Managing Director

$$\{a\} : \forall X \; \textit{government-customer}(X) \vee \textit{prompt-customer}(X) \leftrightarrow \textit{good-customer}(X)$$

$$\{b\} : \forall V, X, Y, Z \; \textit{order}(V, X, Y, Z) \wedge \textit{good-customer}(Z) \rightarrow \textit{warehouse-request}(V, X, Y, Z)$$

$$\{c\} : \forall V, X, Y, Z \; \textit{order}(V, X, Y, Z) \wedge \neg \textit{good-customer}(Z) \rightarrow \textit{credit-check}(V, X, Y, Z)$$

Logistics Manager

$$\{d\} : \forall V, X, Y, Z \; \textit{order}(V, X, Y, Z) \wedge \textit{good-credit-rating}(Z) \rightarrow \textit{warehouse-request}(V, X, Y, Z)$$

$$\{e\} : \forall V, X, Y, Z \; \textit{order}(V, X, Y, Z) \wedge \neg \textit{good-credit-rating}(Z) \rightarrow \neg \textit{warehouse-request}(V, X, Y, Z)$$

Chief Wine Taster

$$\{f\} : \forall V, X, Y, Z \; \textit{order}(V, X, Y, Z) \wedge \textit{end-shelf-life}(Y) \rightarrow \textit{warehouse-request}(V, X, Y, Z)$$

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

9.3 Inconsistency management

For clarity of presentation, we work through our example in the same order as the techniques described in sections 5, 6 and 7 were presented. We then briefly evaluate the techniques, highlighting strengths and weaknesses that have emerged from our case study.

9.3.1 Reasoning

If we want to check certain scenarios with regard to the preliminary specification, we must add further relevant facts to model each scenario. First, consider the following facts.

$$\{p\} : \neg \mathit{government\text{-}customer}(\mathit{Acme})$$
$$\{q\} : \neg \mathit{prompt\text{-}customer}(\mathit{Acme})$$
$$\{r\} : \neg \mathit{good\text{-}credit\text{-}rating}(\mathit{Acme})$$
$$\{s\} : \mathit{order}(278, 10, \mathit{Bordeaux})$$
$$\{t\} : \mathit{end\text{-}shelf\text{-}life}(\mathit{Bordeaux})$$

From this scenario, we can generate the following (inconsistent) inferences.

$$\{f, s, t\} : \mathit{warehouse\text{-}request}(\mathit{Acme})$$

$$\{e, r, s\} : \neg \mathit{warehouse\text{-}request}(\mathit{Acme})$$

We analyse this inconsistency in the next section (9.3.2).

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.

$$\{a, c, p, q, s\} : \mathit{credit\text{-}check}(\mathit{Acme})$$

So even though the assumptions are inconsistent, we can generate a useful inference. It is possible then, for example, to develop a definition for the creditCheck procedure, without necessarily having to resolve the inconsistencies in the preliminary specification.

9.3.2 Analysis: Qualifying inferences

The following inference are only existential inferences, and hence need to be treated with caution when the specification is revised or analysed further.

$$\{f, s, t\} : \mathit{warehouse\text{-}request}(\mathit{Acme})$$

$$\{e, r, s\} : \neg \mathit{warehouse\text{-}request}(\mathit{Acme})$$

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

$$\{a, c, p, q, s\} : \mathit{credit\text{-}check}(\mathit{Acme})$$

There are no free inferences from these assumptions, which is an indication of the preliminary nature of the specification.

9.3.3 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, $\{f, s, t\}$ and $\{e, r, s\}$. There are many possible sources of the inconsistency. However, if we assume the facts we added for the scenario, labelled from the set $\{p, q, r, s, t\}$ are not causing the problem, we order these above the set of labels, $\{a, b, c, d, e, f\}$, refering to the preliminary specification. Using this ordering, we obtain two likely sources of the inconsistency, namely $\{e\}$ and $\{f\}$. These two pieces of proceedural information were elicited from the Logistics Manager and the Chief Wine Taster, respectively. These participants in the requirements engineering process need to be consulted 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).

9.3.4 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 warehouse requests was probably caused by the conflicting rules laid down by the Logistics Manager and the Chief Wine Taster. 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). Meta-level inconsistency handling rules such as those described in section 7 can be used to specify such actions:

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

This (simplified) rule states that if a there is an inconsistency in the specification data, data(delta), 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 (i.e., the boss is right!).

9.4 Evaluation

The application of the inconsistency management techniques to the above scenario supports the thesis that our logic-based approach provides simple me-

chanical tools for living with, even making use of, inconsistent specification information. The techniques clearly allow continued classical reasoning in the presence of inconsistency, and provide many additional analysis and tracking tools so necessary 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 invaluable to the specification developer.

It is clear however, that deciding and modelling what kinds of action are most appropriate in the presence of different kinds of inconsistency are still difficult and open issues. Nevertheless, the additional analysis and reasoning that our techniques provide can only be regarded as welcome input to the decision-action process.

10 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 recognised the need to formalise this notion. For example, Hagensen and Kristensen have explicitly explored the consistency perspective in software development [HK92]. 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 modelled in terms descriptions, interpretations and relations, are also proposed.

We are not aware of any related work on explicitly analysing inconsistent specifications in the manner we have described. However, a number of researchers have recognised 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) [SK88]. 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 [Bal91]. 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 [NG92]. This approach favours software development architectures where impending or proposed changes - as well as changes that have already occurred - are "announced". This allows the consistency requirements of a system to be "lazily" maintained as it evolves. The approach is a compromise between the optimistic view in which inconsistencies are assumed to occur infrequently and can thus be handled individually when they arise, and a pessimistic approach in which inconsistencies are prevented from ever occurring. A compromise approach is particularly realistic in a distributed development setting where conflicts or "collisions" of changes made by different developers may occur. Most of the analysis supported by this approach is pre-emptive 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 specialised language that is best suited for describing its intended area of concern [ZJ93]. 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 demonstratedusing 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 paper. 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 [Bor85]. 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 modularised exception handling [Fea95] 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 [Dow93]. 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. [CNGM95] 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 paper also provides more sophisticated options than truth maintenance (such as [Doy79, Kle86]) for managing inconsistency specifications. These include: (1) paraconsistent reasoning with sets of inconsistent formulae; (2) labelling strategies to allow inconsistent formulae to be tracked, and likely sources identified using meta-level 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 non-monotonic logics (for a review, see [KC93]). Whilst they constitute important developments that could be incorporated in our framework, they are not directly oriented to the inconsistency management issues that we consider with in this paper. In the main they are focussed on resolving inconsistency by finding the best possible inferences for any given set of information, whereas we really need to be able to analyse inconsistent information, consider options, and track information to find likely sources of inconsistency.

11 Discussion, summary and conclusions

Our earlier work began by providing a framework for multi-perspective 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 recognised 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 analyse inconsistencies in this context. This paper addressed such inconsistency handling activities formally.

In this paper, we 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 trivialisation 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 multi-perspective 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 labelled 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.

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

12 Acknowledgements

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 and Jonathan Moffett. This work was partially funded by the UK EPSRC as part of the Voila project (GR J15483), the CEC as part of the Basic Research Actions Promoter and Drums II, and the ISI project (ECAUS003).

References

[AC91] W Atkinson and J Cunningham. Proving properties of safety-critical systems. *IEE Software Engineering Journal*, 6(2):41–50, 1991.

[Bal91] R Balzer. Tolerating inconsistency. In *Proceedings of 13th International Conference on Software Engineering (ICSE-13)*, pages 158–165. IEEE Computer Society Press, 1991.

[BDP93] S Benferhat, D Dubois, and H Prade. Argumentative inference in uncertain and inconsistent knowledge bases. In *Proceedings of Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 1993.

[Bes91] Ph Besnard. Paraconsistent logic approach to knowledge representation. In M de Glas M and D Gabbay D, editors, *Proceedings of the First World Conference on Fundamentals of Artificial Intelligence*. Angkor, 1991.

[BH95] Ph Besnard and A Hunter. Quasi-classical logic: Non-trivializable classical reasoning from inconsistent information. In C Froidevaux and J Kohlas, editors, *Symbolic and Quantitative Approaches to Uncertainty*, volume 946 of *Lecture Notes in Computer Science*, pages 44–51, 1995.

[Bor85] A Borgida. Language features for flexible handling of exceptions in information systems. *Transactions on Database Systems*, 10(4):565–603, 1985.

[CCB90] M Costa, R Cunningham, and J Booth. Logical animation. In *Proceedings of the twelth International conference on software engineering*, pages 144–149, Nice, 1990. IEEE Computer Society Press.

[CNGM95] G Cugola, E Di Nitto, C Ghezzi, and M Mantione. How to deal with deviations during process model enactment. In *Proceedings of 17th International Conference on Software Engineering (ICSE-17)*, pages 265–273, Seattle, USA, 1995. ACM Press.

[dC74] N C da Costa. On the theory of inconsistent formal systems. *Notre Dame Journal of Formal Logic*, 15:497–510, 1974.

[Dow93] M Dowson. 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).

[Doy79] J Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[EGH95] M Elvang-Goransson and A Hunter. Argumentative logics: Reasoning from classically inconsistent information. *Data and Knowledge Engineering Journal*, 16:125–145, 1995.

[EN95a] S Easterbrook and B Nuseibeh. Managing inconsistencies in an evolving specification. In *Proceedings of 2nd International Symposium on Requirements Engineering (RE '95)*, pages 48–55, York,UK, 1995. IEEE CS Press.

[EN95b] S Easterbrook and B Nuseibeh. Using viewpoints for inconsistency management. *IEE Software Engineering Journal*, 1995. To appear.

[Fea95] M Feather. Modularized exception handling. Technical report, USC/Information Sciences Institute, Marina del Rey, California, USA, 1995.

[FGH+94] A Finkelstein, D Gabbay, A Hunter, J Kramer, and B Nuseibeh. Inconsistency handling in multi-perspective specifications. *Transactions on Software Engineering*, 20(8):569–578, 1994.

[Fit90] M Fitting. *First-order Logic and Automated Theorem Proving.* Springer, 1990.

[FKN+92] A Finkelstein, J Kramer, B Nuseibeh, L Finkelstein, and M. Goedicke. 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 SEE*, 2(1):31–57, 1992.

[GH91] D Gabbay and A Hunter. Making inconsistency respectable 1: A logical framework for inconsistency in reasoning. In *Fundamentals of Artificial Intelligence*, volume 535 of *Lecture Notes in Computer Science*, pages 19–32. Springer, 1991.

[GH93] D Gabbay and A Hunter. 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*, pages 129–136. Springer, 1993.

[GH95] D Gabbay and A Hunter. Negation and contradiction. In *What is negation?* Oxford University Press, 1995.

[HK92] T M Hagensen and B B Kristensen. 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, 1992.

[KC93] P Krause and D Clark. *Representing Uncertain Knowledge*. Intellect, 1993.

[Kle86] J De Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[LFKN95] U Leonhardt, A Finkelstein, J Kramer, and B Nuseibeh. Decentralised process enactment in a multi-perspective development environment. In *Proceedings of 17th International Conference on Software Engineering (ICSE-17)*, pages 255–264, Seattle,USA, 1995. IEEE CS Press.

[Lin88] P A Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal (special issue on mechanical support for formal reasoning)*, 3(1), 1988.

[NF92] B Nuseibeh and A Finkelstein. Viewpoints: A vehicle for method and tool integration. In *Proceedings of 5th International Workshop on Computer-Aided Software Engineering (CASE '92)*, pages 50–60, Montreal, Canada, 1992. IEEE Computer Society Press.

[NG92] K Narayanaswamy and N Goldman. Lazy consistency: A basis for cooperative software development. In *Proceedings of International Conference on Computer-Supported Cooperative Work (CSCW '92)*, pages 257–264. ACM SIGCHI and SIGOIS, 1992.

[NKF94] B Nuseibeh, J Kramer, and A Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *Transactions on Software Engineering*, 20(10):760–773, 1994.

[Nus95] B Nuseibeh. On managing inconsistency in software development. Technical report, Department of Computing Imperial College, London, 1995.

[Poo85] D Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27–47, 1985.

[RBP+91] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorenson. *Object-Oriented Modelling and Design*. Prentice Hall, 1991.

[Rya92] M Ryan. Representing defaults as sentences with reduced priority. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*. Morgan Kaufmann, 1992.

[SK88] R W Schwanke and G E Kaiser. Living with inconsistency in large systems. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 98–118. B G Teubner, 1988.

[ZJ93] P Zave and M Jackson. Conjunction as composition. *Transactions on Software Engineering and Methodology*, 2(4):379–411, 1993.