Inconsistency Handling in Multi-Perspective Specifications

A. Finkelstein D. Gabbay A. Hunter J. Kramer B. Nuseibeh

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London, SW7 2BZ, UK
Email: {acwf, dg, abh, jk, ban}@doc.ic.ac.uk

IEEE Transactions on Software Engineering, 20(8): 569-578, August 1994.

Earlier versions available in:

Proceedings of 4th European Software Engineering Conference (ESEC '93), Garmisch-Partenkirchen, Germany, September 1993, 84-99, LNCS 717, Springer-Verlag;

and as:

Dept. of Computing (Imperial College) technical report no. DoC 93/2.

Inconsistency Handling in Multi-Perspective Specifications

A. Finkelstein D. Gabbay A. Hunter J. Kramer B. Nuseibeh

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London, SW7 2BZ, UK
Email: {acwf, dg, abh, jk, ban}@doc.ic.ac.uk

Abstract. The development of most large and complex systems necessarily involves many people - each with their own perspectives on the system defined by their knowledge, responsibilities, and commitments. To address this we have advocated distributed development of specifications from multiple perspectives. However, this leads to problems of identifying and handling inconsistencies between such perspectives. Maintaining absolute consistency is not always possible. Often this is not even desirable since this can unnecessarily constrain the development process, and can lead to the loss of important information. Indeed since the real-world forces us to work with inconsistencies, we should formalise some of the usually informal or extra-logical ways of responding to them. This is not necessarily done by eradicating inconsistencies but rather by supplying logical rules specifying how we should act on them. To achieve this, we combine two lines of existing research: the ViewPoints framework for perspective development, interaction and organisation, and a logic-based approach to inconsistency handling. This paper presents our technique for inconsistency handling in the ViewPoints framework by using simple examples.

1 Introduction

The development of most large and complex systems necessarily involves many people - each with their own perspectives on the system defined by their knowledge, responsibilities, and commitments. Inevitably, the different perspectives of those involved in the process intersect - giving rise to the possibility of inconsistency between perspectives and to a need for co-ordination. These intersections, however, are far from obvious because the knowledge from each perspective is expressed in different ways. Furthermore, because development may be carried out concurrently by those involved, different perspectives may be at different stages of elaboration and may be subject to different development strategies.

The problem of co-ordinating these different perspectives is partly 'organisational' and partly 'technical'. The organisational aspect requires that support is provided for ordering activities, interacting by passing information and resolving conflicts. The main technical aspect centres around the consistency relationships between these perspectives, given as partial specifications. Indeed checking consistency between perspectives and the handling of inconsistency creates many interesting and difficult research problems.

We do not believe that it is possible, in general, to maintain absolute consistency between perspectives at all times. Indeed, it is often not even desirable to enforce consistency, particularly when this constrains the specification unnecessarily or entails loss of design freedom by enforcing an early resolution [20]. Thus, there is a requirement for some form of inconsistency handling techniques in which inconsistency is tolerated and used to trigger further actions [15, 16].

In section 2 we provide a brief background to the ViewPoints framework, and in section 3 we provide an overview of inconsistency handling in this setting. In the subsequent sections, we illustrate and discuss the identification and handling of inconsistency using simple specification examples.

2 Background to the ViewPoints Framework

The integration of methods, notations and tools has generally been addressed by the use of a common data model, usually supported by a common, centralised database [37]. This has some advantages in providing a uniform basis for consistency checking. Multiple views can still be supported by the provision of mappings to and from the data model. However, we believe that the general use of centralised data repositories is a mistake for the long term. General data models are difficult to design and tend to be even more difficult to modify and extend when new tools are to be integrated [31, 36]. This is analogous to the search for some universal formalism. Therefore, although the approaches based on common data models have enabled us to make good progress in the provision of current CASE tools, we believe that such data models are too tightly integrated. Such inherent logical centralisation will be one of the major restrictions in the provision of tools that integrate more methods and notations, cover a larger part of the life-cycle and support use by large teams of software engineers.

To address this we have developed a novel framework that supports the use of multiple distributed perspectives in software and systems development [10, 13]. The primary building blocks used in this framework are “ViewPoints” (Fig. 1), combining the notion of a “participant” in the development process, and the idea of a “view” or “perspective” that the participant maintains. A ViewPoint template is a ViewPoint in which only the representation style and work plan have been elaborated. A ViewPoint is thus created by instantiating a template, thereby producing a ViewPoint specification for a particular domain.

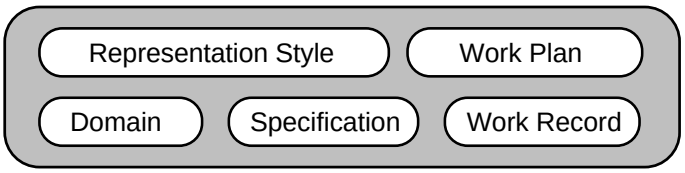

Fig. 1. A ViewPoint encapsulates representation, development and specification knowledge in its five “slots”. The representation style defines the notation deployed by the ViewPoint. The work plan describes the development actions, strategy and process used by the ViewPoint. The specification slot delineates the ViewPoint's domain in the chosen representation style, while the work record contains a development history of the specification.

The framework has been implemented [26] to allow the construction of partial specifications in a variety of formalisms. A work record of the development, including the development history and rationale for each specification, is also recorded. A typical systems engineering project would deploy a number of ViewPoints described and developed using a variety of different languages. ViewPoints are bound together by inter-ViewPoint relations that specify dependencies and mappings between system components (Fig. 2). ViewPoints be managed by maintaining local consistency within each ViewPoint and partial consistency between different ViewPoints.

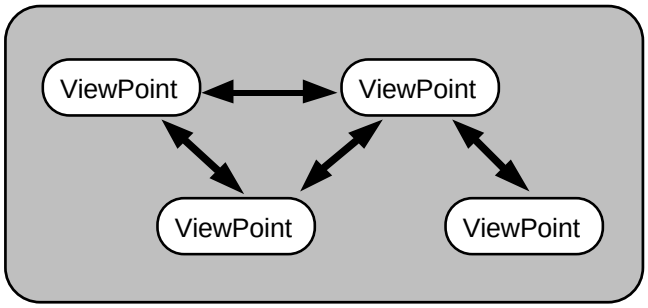

Fig. 2. A system specification in the ViewPoints framework is a configuration or collection of ViewPoints integrated together by one-to-one inter-ViewPoint rules. We can layer configurations of ViewPoints to reduce problems of scale [21], and use hypertext-like tools to navigate around large ViewPoint networks [18].

Thus, in contrast to the traditional approaches, the ViewPoints approach to specification development is inherently distributed. ViewPoints are loosely coupled, locally managed and distributable objects, with integration achieved via one-to-one inter-ViewPoint relationships defined as inter-ViewPoint rules in the work plans of templates from which the ViewPoints are instantiated [29].

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

3 Inconsistency Handling in the ViewPoints Framework

Given that inconsistency is often viewed as a logical concept, we believe that it is appropriate that inconsistency handling should be based on logic. The problem of inconsistency handling in the ViewPoints framework can then be viewed as being equivalent to inconsistency handling in distributed logical databases. For this we need to define rewrites from specification information, and inter-ViewPoint information, in a ViewPoint to a set of logical formulae. However, before describing how we rewrite, identify and handle inconsistency in this kind of data, we briefly discuss the general problems of inconsistency in logic. We use this as a means of motivating our approach.

Classical logic, and intuitionistic logic, take the view that anything follows from an inconsistency. Effectively, when an inconsistency occurs in a database, it becomes unusable. This has prompted the logic community to study such logics as relevant [1] and paraconsistent logics [7] that allow reasoning with inconsistent information. These isolate inconsistency by various means, but do not offer strategies for dealing with the inconsistency. Therefore there still remains the question of what do we do when we have two contradictory items of information in a database. Do we choose one of them? How do we make the choice? Do we leave them in and find a way “around” them?

Other logics, such as certain non-monotonic logics (for a review see [5]), resolve some forms of inconsistency, but do not allow the representation of certain forms of inconsistent data, or give no answer when present. There are also attempts at paraconsistent non-monotonic logics [6, 30, 35], but these again do not answer all the questions of handling inconsistency.

The logic programming and deductive database communities have focused on alternative approaches to issues of inconsistencies in data. These include integrity constraints (for example [33]) and truth maintenance systems [8, 15, 16]. For these, any attempt to introduce inconsistency in the database causes rejection of input, or amendment of the database. Therefore these also do not constitute solutions for the ViewPoints framework since neither allow us to represent and reason with inconsistent information nor allow us to formalise the desired actions that should result from inconsistency.

These approaches constitute a significant shortfall in the ability required to handle inconsistency in formal knowledge representation. In our approach, we attempt to shift the view of inconsistency from being necessarily "bad" to being acceptable, or even desirable, if we know how to deal with it [15, 16]. Moreover, when handling inconsistencies in a database, it is advantageous to analyse them within the larger context of the environment of the database and its use. When viewed locally, an inconsistency may seem undesirable, but within the larger environment surrounding the data, an inconsistency could be desirable and useful, if we know appropriate actions to handle it. Dealing with inconsistencies is not necessarily done by restoring consistency but by supplying rules telling one how to act when the inconsistency arises.

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

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

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

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

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

Fig. 3 schematically summarises the stages of rewriting, identification, and handling of inconsistency when checking two ViewPoints in the framework. To check the consistency of specifications in two ViewPoints, partial specification knowledge in each is translated into classical logic. Together with the inter-ViewPoint rules in each ViewPoint - which are also translated into logic - inconsistencies between the two ViewPoints may be identified. Meta-level rules are then invoked which prescribe how to act on the identified inconsistencies.

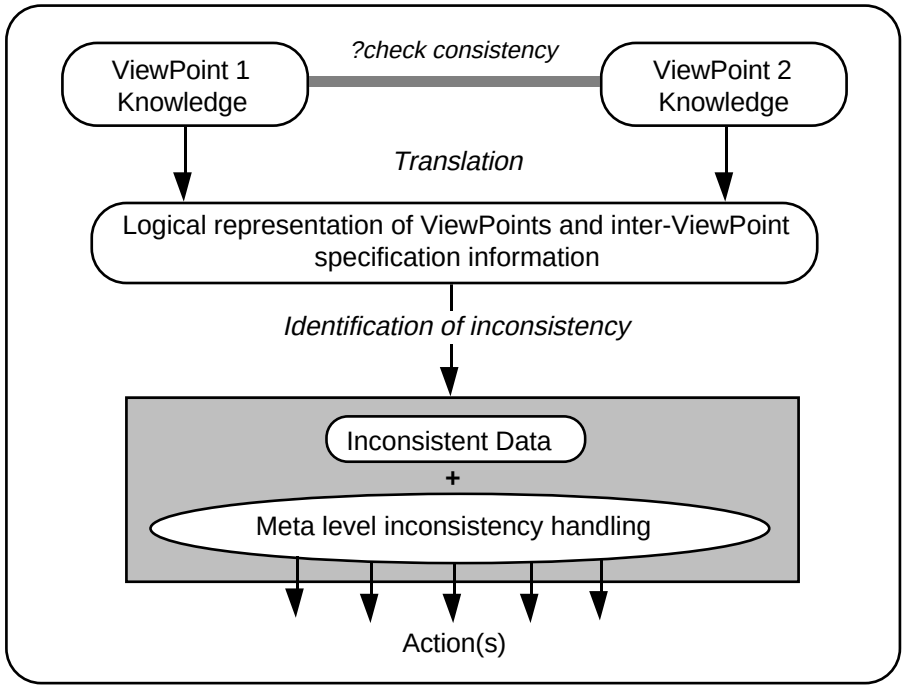

Fig. 3. Inter-ViewPoint communication and inconsistency handling in the ViewPoints framework. Selected ViewPoint knowledge in each of the communicating ViewPoints is translated into logical formulae and used to detect and identify inconsistencies. The meta-level rules may then be used to act upon these inconsistencies.

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

4 A Simple Example

To demonstrate our approach, we will use a simplified library example specified using a subset of the formalisms deployed by the requirements analysis method CORE [24]. The two formalisms we use are what we call agent hierarchies (Fig. 4) and action tables (Fig. 5a, b, c). An agent hierarchy decomposes a problem domain into information processing entities or roles called "agents"[1]. Agents may be "direct", if they process information, or "indirect", if they only generate or receive information without processing it. For each direct agent in the hierarchy, we construct an action table showing the actions performed by that agent, the input data required for the actions to occur and the output data produced by those actions. The destination and source agents to and from which the data flows are also shown (action tables may thus be regarded as a standard form of data flow diagrams). An arc drawn between two lines in an action diagram indicates the conjunction of the two terms preceding the joining lines.

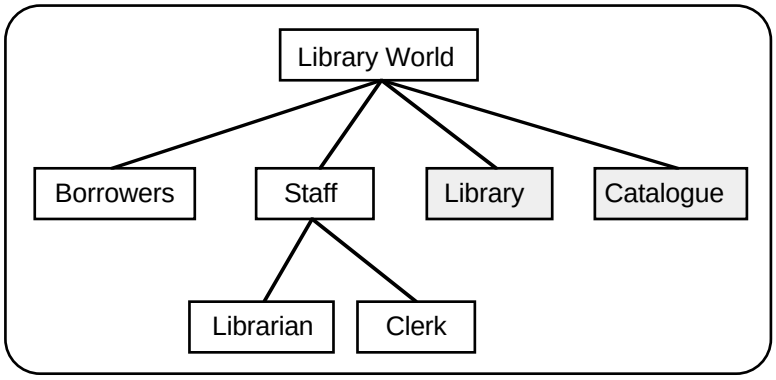

Fig. 4. An agent hierarchy decomposing the library problem domain into its constituent agents. Shaded boxes indicate indirect agents which require no further decomposition or analysis.

SOURCE	INPUT	ACTION	OUTPUT	DESTINATION
Borrower	book	check-in	book	Clerk
	card		card	
Library	book	check-out	book	Borrower

Fig. 5a. An action table elaborating the agent "Borrower". In ViewPoints terminology, the action table is part of a ViewPoint specification where the ViewPoint domain is "Borrower".

1 CORE uses the term "viewpoint" as part of its terminology, so we have renamed it "agent" to avoid the clash in nomenclature.

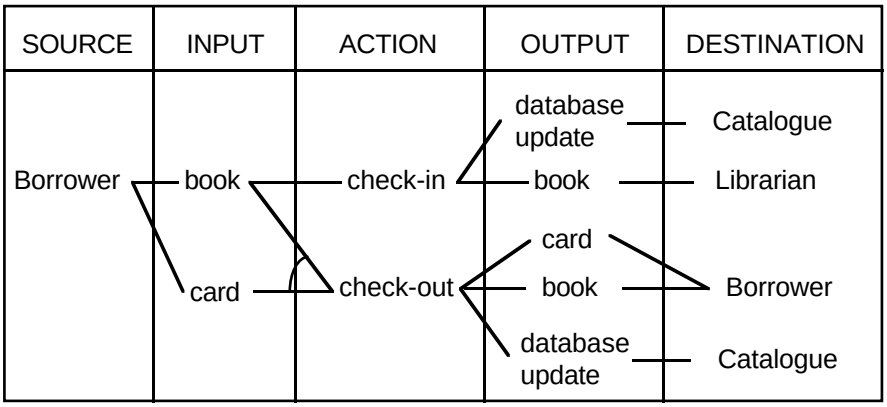

Fig. 5b. An action table elaborating the agent "Clerk". In ViewPoints terminology, the action table is part of a ViewPoint specification where the ViewPoint domain is "Clerk".

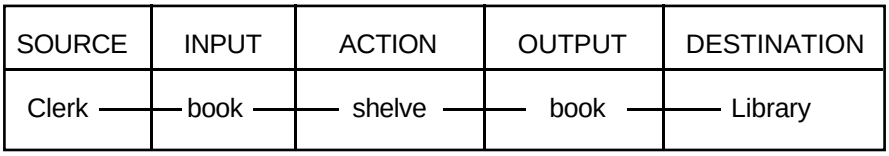

Fig. 5c. An action table elaborating the agent "Librarian". In ViewPoints terminology, the action table is part of a ViewPoint specification where the ViewPoint domain is "Librarian".

In ViewPoints terminology, figures 4 and 5 are contained in the specification slots of the different ViewPoints of the overall system specification. Thus, the specification shown in Fig. 5c for example, would appear in the ViewPoint outlined schematically in Fig. 6. Note that the domain of the ViewPoint is "Librarian" which indicates the delineation of the action table in the specification slot.

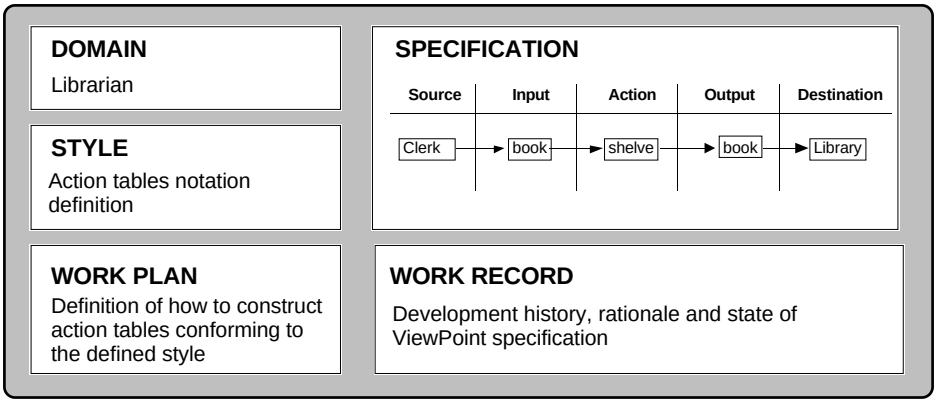

Fig. 6. A schematic outline of a ViewPoint containing the specification shown in Fig. 5c.

The agent hierarchy and action table formalisms are related in a number of ways (as specified by the CORE method designer). Two such relations are described informally by the following rules.

Rule 1 (between an agent hierarchy and actions tables): Any “source” or “destination” in an action table, must appear as a leaf agent in the agent hierarchy.

Rule 2 (between action tables): The output (Z) produced by an action table for an agent (X) to a destination (Y), must be consumed as an input (Z) from a source (X) by the action table for the original destination (Y).

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

5 Identification of Inconsistency

To undertake a partial consistency check between two or more ViewPoints, we form a logical database (D, E), where D contains formulae representing the partial specifications in these ViewPoints, and E contains formulae representing environmental information such as the definitions for the inter-ViewPoint relations. For the remainder of this paper we assume the language for (D, E) is first-order classical logic.

We start by looking at the action tables in Fig. 5. For these we consider the pre-conditions and post-conditions to an action. We use the function pre(X, Y) to denote X is a source and Y is a 'conjunction' (denoted by ‘&’) of inputs that are consumed by some action. Similarly, post(Y, X) denotes that X is the destination of the 'conjunction' of outputs, Y, produced by some action. Now we denote the part of the action table associated with an action B as follows:

table(A, P, B, Q)

where, A is an agent name (the ViewPoint domain),
P is a 'conjunction' of pre-conditions for action B, and
Q is a 'conjunction' of post-conditions for action B.

Note that for this definition we are using 'conjunction' as a function symbol within first-order classical logic.

We can now represent each table in Fig. 5 using the above predicates. First we consider the information in Fig. 5a.

(1) table(borrower, pre(borrower, book), check_in, post(book, clerk)).
(2) table(borrower, pre(borrower, card) & pre(library, book), check_out,
post(book&card, borrower)).

Similarly, we represent the information in Fig. 5b as,

(3) table(clerk, pre(borrower, book), check_in,
post(database_update, catalogue) & post(book, library)).
(4) table(clerk, pre(borrower, book&card), check_out,
post(book&card, borrower) & post(database_update, catalogue)).

Finally we represent the information in Fig. 5c as,

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

Obviously there are many ways we could present the information in figures 5a-5c in logic. However, it is straightforward to define a rewrite that can take any such table and return logical facts of the above form. We can also capture the inter-ViewPoint relations in classical logic. So rule 2 (between action tables) is represented by,

(6) for all A, B_i , C_i [[table(A, _, _, post(C_1, B_1) & .. & post(C_m, B_m))]
[table(B_1, X_1, _, _) and ... and table(B_m, X_m, _, _)]]
where $1 \leq i \leq m$ and pre(A, C_i) is a conjunct in X_i.

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

The formulae (1) - (5) are elements in D, and the formula (6) is an element in E. Since we have represented the information in D as a set of classical logic facts, we can use the Closed World Assumption [32] to capture the facts that do not hold in each ViewPoint specification. The Closed World Assumption (CWA) essentially captures the notion that if a fact A is not a member of a list of facts then ¬A (not A) holds. So for example, using the CWA with the borrower domain we can say that the following arbitrary formula does not hold,

(7) table(borrower, pre(borrower, book), check_in, post(card, clerk)).

In other words we can infer the following,

(8) ¬table(borrower, pre(borrower, book), check_in, post(card, clerk)).

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

(9) ¬table(clerk, pre(borrower, book), check_in,
post(database_update, catalogue) & post(book, library)).

Using a classical logic theorem prover with the CWA, we can easily show that (D, E) is inconsistent and that the formulae (1), (6) and (9) cause the inconsistency - since (1) and (6) give,

table(clerk, pre(borrower, book), check_in, X, Y)

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

¬table(clerk, pre(borrower, book), check_in, X, Y)

for any terms X, Y in the language. We address the handling of this situation in section 6.

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

(10) tree(library_world, borrower)
(11) tree(library_world, staff)
(12) tree(library_world, library)
(13) tree(library_world, catalogue)
(14) tree(staff, librarian)
(15) tree(staff, clerk)

with the first argument to the predicate 'tree' representing a parent in the [illegible]ent hierarchy, and the second argument representing its child. To t[illegible] axioms of reflexivity, transitivity, anti-symmetry, up-linearity, and leaf. Recall that

(up-linearity) for all X, Y, Z, [tree(X, Z) and tree(Y, Z) tree(X, Y) or tree(Y, X)]

(leaf) for all X, Y [tree(X, Y) and ¬(there exists a Z such that tree(Y, Z)) leaf(Y)]

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

(16) tree(library_world, clerk).

Another inter-ViewPoint relation (rule 1 between ViewPoints deploying different formalisms[illegible]ws,

(17) for all A_i [[table([illegible]re(A_1, _) & .. & pre(A_m, _), _, _)]
there exists X such that [leaf(X) and X = A_i]]
where 1 ≤ i ≤ m

(18) for all B_i [[table([illegible]post(_, B_1) & .. & post(_, B_m))]
there exists X such that [leaf(X) and X = B_i]]
where 1 ≤ i ≤ m

As before, formulae (10) - (15) plus the axioms of reflexivity, transitivity, antisymmetry, leaf and up-linearity are elements in D, while (17) and (18) are elements in E. Remember that we are assuming that rules 1 and 2, expressed in logic as (6), (17) and (18) above, are correctly defined by the method designer.

Now suppose that formula (3) had not been inserted in the relevant ViewPoint, and again using the CWA together with a classical logic theorem prover, we can see how insufficient information in (D, E) leads to inconsistency.

We now turn our attention to the situation where[illegible]e have too much information in a pair of partial specifications. Suppose in ViewP[illegible]nt 1, we have the following information,

(19) reference_book(childrens_dictionary)
(20) for all X, reference_book(X) ¬lendable(X)

and suppose in ViewPoint 2, we have the following in[illegible]mation,

(21) childrens_book(childrens_dictionary)
(22) for all X, childrens_book(X) lendable(X)

Taking the formulae in (19) - (22) as elements in D, we have an inconsistency in (D, E) resulting from too much information. Such a situation is common in developing specifications, though the causes are diverse. We address the handling of this situation in the next section.

To summarise, in this section we have advocated the use of classical logic together with the CWA to provide a systematic and well-understood way of finding inconsistency in specifications. It will not be possible to provide a universal and meaningful rewrite from any software engineering formalism into classical logic. However, for partial consistency checking it is often possible to compare some of the specification information, plus other information such as inter-ViewPoint relations, for two, or maybe more, ViewPoints.

6 Acting on Inconsistency

For the meta-level inconsistency handling we use an action-based meta-language [16] based on linear-time temporal logic. We use a first-order form where we allow quantification over formulae (Note that we are not using a second-order logic - rather we are treating object-level formulae as objects in the semantic domain of the meta-level.). Furthermore, we use the usual interpretation over the natural numbers - each number denotes a point in time. Using this interpretation we can define operators such as LAST^n and NEXT^n where LAST^n A holds at time t if A holds at t-n, and NEXT^n A holds at time t if A holds at t+n.

Using temporal logic, we can specify how the databases should evolve over time. In this way, we can view the meta-level handling of inconsistent ViewPoint specifications in terms of satisfying temporal logic specifications. So if during the course of a consistency check between two ViewPoints an inconsistency is identified, then one or more of the meta-level action rules will be fired. Furthermore, since we use temporal logic, we can record how we have handled the ViewPoints in the past.

The meta-level axioms specify how to act according to the context of the inconsistency. This context will include the history behind the inconsistent data being put into the ViewPoint specification - as recorded in the ViewPoint work record - and the history of previous actions to handle the inconsistency. The meta-level axioms will also include implicit and explicit background information on the nature of certain kinds of inconsistencies, and how to deal with them.

To illustrate the use of actions at the meta-level, we now return to the examples introduced in section 5. For handling the inconsistency resulting from formulae (1), (6) and (9), a simplified solution would be to incorporate the kind of meta-level axiom (23) into our framework. For this we provide the following informal definitions of the key predicates:

- data(vp1, Δ_1) holds if the formulae in the database Δ_1 are a logical rewrite of selected information in ViewPoint vp1.
- union(Δ_1, Δ_2) false holds if the union of the databases Δ_1 and Δ_2 implies inconsistency.
- inconsistency_source(union(Δ_1, Δ_2), S) holds if S is a minimal inconsistent subset of the union of Δ_1 and Δ_2.
- likely_spelling_problem(S) holds if the cause of the inconsistency is likely to result from typographical errors in S. Since we are using a temporal language at the meta-level, we can also include conditions in our rule that we haven't checked this problem

at previous points S in time. This means that our history affects our actions.

- tell_user("is there a spell problem?", S) if the message together with the data in S is outputted to the user. In software process modelling terminology [12], this is equivalent to a tool invocation say, such as a spell-checker or other tool [11].

Essentially, this rule captures the action that if S is the source of the inconsistency and that the likely reason that S is inconsistent is a typographical error, then we tell the user of the problem. We assume that the user can usually deal with this kind of problem once informed. However, we should include further meta-level axioms that provide alternative actions, in case the user cannot deal with the inconsistency on this basis. Indeed, it is likely that for handling inconsistency between different formalisms such as in (1), (6) and (9), there will be a variety of possible actions. This meta-level axiom also has the condition that this action is blocked if likely_spelling_problem(S) has been identified in either of the two previous two steps. This is to stop the same rule firing if the user wants to ignore the problem for [redacted] steps.

(23) data(vp1, Δ_1) and [redacted]ata(vp2, Δ_2)
and union(Δ_1, Δ_2) [redacted] false
and inconsistency_s[redacted]urce(union(Δ_1, Δ_2), S)
and [redacted]em(S)
and ¬LAST^1 likely_spelling_problem(S)
and ¬LAST^2 like[redacted]_spelling_problem(S)
NEXT tell_user("is there a spell problem?", S).

In a similar fashion, we can define appropriate meta-level axioms for handling the inconsistency resulting from formulae (9), (10), (17) and (18) in the above examples.

For handling the problem of too much information occurring in formulae, such as for example (18) - (21), a simplified solution would be to incorporate the kind of meta-level axiom (24) into our framework, where likely_conflict_between_specs_problem(Δ_1, Δ_2) holds if the inconsistency arises from just information in the specification. In other words this inconsistency does not arise because the method or tools have been used incorrectly, but rather, it arises from the incorrectly specifying system.

(24) data(vp1, Δ_1)
[redacted]
and union(Δ_1, Δ_2) [redacted]alse
and likely_conflict_between_specs_problem(Δ_1, Δ_2)
NEXT tell_user("is there a conflict between specifications?", (Δ_1, Δ_2)).

These definitions for the meta-level axioms have skipped over many difficult technical problems, including the general problems of decidability and complexity of such axioms, and the more specific problems of say defining the predicates "inconsistency_source", "likely_spelling_problem", and "likely_conflict_between_specs_problem". Also, we have skipped over the many ways that this approach builds on a variety of existing work by various authors in database updates, integrity constraints, database management systems and meta-level reasoning. Nevertheless, we have illustrated how a sufficiently rich meta-level logic can be used to formally capture intuitive ways of handling inconsistencies in our (D, E) databases. Moreover, such meta-level axioms may also be used to describe, guide and manage the multi-ViewPoint development process in this setting. The

advantage over traditional approaches to process modelling [11] however, is that our technique allows very fine-grain modelling - at a level of granularity much closer to the representations deployed by the various ViewPoints [27].

7 Viability of Inconsistency Handling

Since the proposed system uses temporal logic, it is based on a well-developed theoretical basis. It is straightforward to show that this meta-level language inherits desirable properties of first-order until-since (US) temporal logic such as a complete and sound proof theory, and of semi-decidability. This temporal logic is sufficiently general for our purposes. Assuming that time corresponds to a linear sequence of natural numbers, we have all the usual temporal operators including NEXT^n, LAST^n, SOMETIME-IN-THE-FUTURE, SOMETIME-IN-THE-PAST, and ALWAYS. Similarly, if we assume time corresponds to a linear sequence of real numbers, we have many of these operators.

Furthermore for some sufficiently general subsets of US temporal logic there are viable model building algorithms, such that if the meta-level specification is consistent then the algorithm is guaranteed to find a model of the specification [3]. Using these properties we execute temporal logic specifications to generate a model [14]. This has led to the approach of Executable Temporal Logics - which have been implemented and applied in a variety of applications [4, 9, 22]. In the approach of executable temporal logics we view temporal logic specifications as programs. The model generated by executing the program is then the output from the program.

Though we have not yet implemented the described inconsistency handling for the ViewPoints framework, some of the components required have been implemented. Currently we have an implementation of the ViewPoints framework without the logic-based inconsistency handling technique described in this paper. Called *The Viewer* [26], it provides tool support for the construction of ViewPoint specifications in a variety of formalisms such as those in figures 4 and 5. Tool support for in-ViewPoint consistency checking is also provided. We also have an implementation of first-order executable temporal logic, and we have a first-order theorem prover for consistency checking [17]. We now need to implement the rewrites from the ViewPoints formalism to classical logic and to axiomatise meta-level actions for handling inconsistency.

Finally, in a distributed development setting, issues relating to inter-ViewPoint communication, co-ordination and synchronisation become even more significant. We have proposed a preliminary model for such communication and investigated protocols and mechanisms for exchanging data between ViewPoints [29]. However, the application of such protocols with the inconsistency handling techniques described here, is beyond the scope of this paper. Nevertheless, we have explored the method engineering process in the ViewPoints framework during which inter-ViewPoint and inconsistency handling rules are defined [28].

8 Discussion and Related Work

System specification from multiple perspectives using many different specification languages has become an area of considerable interest. Recent work by Zave & Jackson [40] proposes 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. Zave & Jackson's work complements our approach, but it does appear to differ in that they assume they can use classical logic as an underlying universal formalism. Also they do not consider the handling of inconsistent specifications.

Other authors have also considered multi-perspective or multi-language specifications, but again do not consider the *handling* of inconsistencies. Wileden et al. [39] describe specification level *interoperability* between specifications or programs written in different languages or running on different kinds of processors. The interoperability described relies on remote procedure calls and ways that interoperating programs manipulate shared typed data. The work serves as a basis for "the disciplined and orderly marshaling of interoperable components" to eradicate inconsistencies in the overall system specification or program. Wile [38] on the other hand uses a common syntactic framework defined in terms of grammars and *transformations* between these grammars. He highlights the difficulties of consistency checking in a multi-language framework, which suggests that, again, the handling of inconsistencies, once detected, has not been addressed in his work.

Traditionally, multiparadigm languages, which deploy a common multiparadigm *base language*, have been used to combine many partial program fragments [19], while more recently the use of a single, common *canonical representation* for integrating so-called "multi-view" systems has been proposed [23]. Both these approaches to integration do not support the notion of transient inconsistencies.

Schwanke and Kaiser [34] suggest that during large systems development, programmers often circumvent strict consistency enforcement mechanisms in order to get their jobs done. They propose an approach to "living with inconsistency" during development, and describe a configuration management (CONMAN) programming environment that helps programmers handle inconsistency by: (1) identifying and tracking six different kinds of inconsistencies (without requiring them to be removed), (2) reducing the cost of restoring type safety after a change (using a technique called "smarter recompilation"), and (3) protecting programmers from inconsistent code (by supplying debugging and testing tools with inconsistency information).

Balzer [2] proposes another approach that addresses the handling of certain kinds of inconsistency. Here, the notion of relaxing constraints and tolerating inconsistencies is discussed, and a simple technique that allows inconsistencies to be managed and tolerated is presented. Inconsistent data is 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. Our approach goes a further by explicitly specifying the actions that may be performed in order to handle the inconsistencies.

Finally, Narayanaswamy and Goldman [25] propose the notion of "lazy" consistency as the basis for cooperative software development. 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 a system evolves. The approach is a compromise between the optimistic view in which inconsistencies are assumed to occur infrequently and can thus be handled individually when they arise, and a pessimistic approach in which inconsistencies are prevented from ever occurring. A compromise approach is particularly realistic in a distributed development setting where conflicts or "collisions" of changes made by different developers often occur. Lazy consistency maintenance supports activities such as negotiation and other organisational protocols that support the resolution of conflicts and collisions.

In conclusion, the work presented in this paper has outlined how we intend addressing the important issues surrounding inconsistency handling in multi-perspective specifications. The work brings together two promising lines of research: multi-perspective software development in the ViewPoints framework *and* inconsistency handling using classical and action-based temporal logics. While we recognise that a number of research issues remain, we believe that our work provides a sound alternative approach to software development.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments on earlier versions of the paper. This work was partly funded by the CEC ESPRIT BRA project DRUMS II and the UK DTI project ESF. An earlier version of this paper appeared in the Proceedings of the 4th European Software Engineeing Conference (LNCS 717, Springer-Verlag).

References

[1] A. R. Anderson and N. D. Belnap (1976); *The Logic of Entailment*; Princeton University Press, USA.

[2] R. Balzer (1991); "Tolerating Inconsistency"; *Proceedings of 13th International Conference on Software Engineering (ICSE-13)*, Austin, Texas, USA, 13-17th May 1991, 158-165; IEEE Computer Society Press.

[3] H. Barringer, M. Fischer, D. Gabbay, G. Gough and R. Owens (1989); "MetateM: A Framework for Programming in Temporal Logic"; *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems*, LNCS 430, Springer-Verlag.

[4] H. Barringer, M. Fischer, D. Gabbay and A. Hunter (1991); "Meta-reasoning in Executable Temporal Logic"; *Proceedings of 2nd International Conference on the Principles of Knowledge Representation and Reasoning*, 40-49; Morgan Kaufmann.

[5] J. Bell (1990); "Non-Monotonic Reasoning, Non-Monotonic Logics, and Reasoning About Change"; *Artificial Intelligence Review*, 4, 79-108.

[6] H. Blair and V. Subrahmanian (1989); "Paraconsistent Logic Programming"; *Theoretical Computer Science*, 68, 135-154.

[7] N. D. da Costa (1974); "On the Theory of Inconsistent Formal Systems"; *Notre Dame Journal of Formal Logic*, 15, 497-510.

[8] J. Doyle (1979); "A Truth Maintenance System"; *Artificial Intelligence*, 12, 231-272.

[9] M. Finger, P. McBrien and R. Owens (1991); "Databases and Executable Temporal Logic"; *Proceedings of ESPRIT Conference*, CEC.

[10] A. Finkelstein, M. Goedicke and J. Kramer (1990); "ViewPoint Oriented Software Development"; *Proceedings of 3rd International Workshop on Software Engineering and its Applications*, Toulouse, France, 3-7th December 1990, 337-351; Cigref EC2.

[11] A. Finkelstein, J. Kramer and M. Hales (1992); "Process Modelling: A Critical Analysis"; *(In) Integrated Software Engineering with Reuse: Management and Techniques;* P. Walton and N. Maiden (Eds.); 137-148; Chapman & Hall and UNICOM, UK.

[12] A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.) (1994); *Software Process Modelling and Technology*, Advanced Software Development Series, Research Studies Press Ltd. (Wiley), Somerset, UK.

[13] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein and M. Goedicke (1992); "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development"; *International Journal of Software Engineering and Knowledge Engineering*, 2(1): 31-58, March 1992; World Scientific Publishing Co.

[14] D. Gabbay (1989); "Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems"; *Proceedings of Colloquium on Temporal Logic in Specification*, LNCS 398, Springer-Verlag.

[15] D. Gabbay and A. Hunter (1991); "Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Part 1 - A Position Paper"; *Proceedings of the Fundamentals of Artificial Intelligence Research '91*, 19-32; LNCS, 535, Springer-Verlag.

[16] D. Gabbay and A. Hunter (1992); "Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning: Part 2"; *(In) Symbolic and Quantitative Approaches to Resaoning and Uncertainty;* 129-136; LNCS, Springer-Verlag.

[17] D. Gabbay and H. Ohlbach (1992); "Quantifier Elimination in Second Order Predicate Logic"; *Proceedings of 3rd International Conference on the Principles of Knowledge Representation and Reasoning*, 35-43; Morgan Kaufmann.

[18] P. Graubmann (1992); "The HyperView Tool Standard Methods"; *REX technical report*, REX-WP3-SIE-021-V1.0; January 1992; Siemens, Germany.

[19] B. Hailpern (1986); "Special issue on multiparadigm languages"; *Software*, 3(1): 6-77, January 1986; IEEE Computer Society Press.

[20] J. Kramer (1991); "CASE Support for the Software Process: A Research Viewpoint"; *Proceedings of 3rd European Software Engineering Conference (ESEC 93)*, Milan, Italy, October 1991, 499-503; LNCS 550, Springer-Verlag.

[21] J. Kramer (1991); "A Configurable Framework for Method and Tool Integration"; *Proceedings of European Symposium on Software development Environments and CASE Technology*, Königswinter, Germany, June 1991, 233-257; LNCS, 509, Springer-Verlag.

[22] J. Krogstie, P. McBrien, R. Owens and A. Selvit (1991); "Information systems Development Using a Combination of Process and Rules-Based Approaches"; *Proceedings of the International Conference on Advanced Information Systems Engineering*, LNCS, Springer-Verlag.

[23] S. Meyers and S. P. Reiss (1991); "A System for Multiparadigm Development of Software Systems"; *Proceedings of 6th International Workshop on Software Specification and Design (IWSSD-6)*, Como, Italy, 25-26th October 1991, 202-209; IEEE Computer Society Press.

[24] G. Mullery (1985); "Acquisition - Environment"; *(In) Distributed Systems: Methods and Tools for Specification;* M. Paul and H. Siegert (Eds.); LNCS, 190, Springer-Verlag.

[25] K. Narayanaswamy and N. Goldman (1992); ""Lazy" Consistency: A Basis for Cooperative Software Development"; *Proceedings of International Conference on Computer-Supported Cooperative Work (CSCW '92)*, Toronto, Ontario, Canada, 31st October - 4th November, 257-264; ACM SIGCHI & SIGOIS.

[26] B. Nuseibeh and A. Finkelstein (1992); "ViewPoints: A Vehicle for Method and Tool Integration"; *Proceedings of 5th International Workshop on Computer-Aided Software Engineering (CASE '92)*, Montreal, Canada, 6-10th July 1992, 50-60; IEEE Computer Society Press.

[27] B. Nuseibeh, A. Finkelstein and J. Kramer (1993); "Fine-Grain Process Modelling"; *Proceedings of 7th International Workshop on Software Specification and Design (IWSSD-7)*, Redondo Beach, California, USA, 6-7 December 1993, 42-46; IEEE Computer Society Press.

[28] B. Nuseibeh, A. Finkelstein and J. Kramer (1994); "Method Engineering for Multi-Perspective Software Development"; *Information and Software Technology (forthcoming)*, : Spring 1994; Butterworth Heinemann.

[29] B. Nuseibeh, J. Kramer and A. Finkelstein (1993); "Expressing the Relationships Between Multiple Views in Requirements Specification"; *Proceedings of 15th International Conference on Software Engineering (ICSE-15)*, Baltimore, Maryland, USA, 17-21 May 1993, 187-200; IEEE Computer Society Press.

[30] T. Pequendo and A. Buschbaum (1991); "The Logic of Epistemic Inconsistency"; *Proceedings of 2nd International Conference on the Principles of Knowledge Representation and Reasoning*, 453-460; Morgan Kaufmann.

[31] J. N. Pocock (1991); "VSF and its Relationship to Open Systems and Standard Repositories"; *Proceedings of European Symposium on Software development Environments and CASE Technology*, Königswinter, Germany, June 1991, 53-68; LNCS, 509, Springer-Verlag.

[32] R. Reiter (1978); "On Closed World Databases"; *(In) Logic & Databases;* H. Gallaire and J. Minker (Eds.); Plenum Press.

[33] F. Sadri and R. Kowalski (1986); "An Application of General Theorem Proving to Database Integrity"; *Tecnical report*, Department of Computing, Imperial College, London, UK.

[34] R. W. Schwanke and G. E. Kaiser (1988); "Living With Inconsistency in Large Systems"; *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, Germany, 27-29 January 1988, 98-118; B. G. Teubner, Stuttgart.

[35] G. Wagner (1991); "Ex contradictione nihil sequitor"; *Proceedings of 12th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann.

[36] A. I. Wasserman (1990); "Tool Integration in Software Engineering Environments"; *Proceedings of International Workshop on Environments*, Chinon, France, September 1989, 137-149; LNCS, 467, Springer-Verlag.

[37] A. I. Wasserman and P. A. Pircher (1987); "A Graphical, Extensible Integrated Environment for Software Development"; *SIGPlan Notices (Proceedings of 2nd Symposium on Practical Software Development Environments)*, 22(1): 131-142, January 1987; ACM Press.

[38] D. S. Wile (1992); "Integrating Syntaxes and their Associated Semantics"; *Technical report*, RR-92-297; 1992; USC/Information Sciences Institute, University of Southern California, Marina del Rey, California, USA.

[39] J. C. Wileden, A. I. Wolf, W. R. Rosenblatt and P. L. Tarr (1992); "Specification-Level Interoperability"; *Communications of the ACM*, 34(5): 72-87, May 1991; ACM Press.

[40] P. Zave and M. Jackson (1993); "Conjunction as Composition"; *Transactions on Software Engineering and Methodology*, 2(4): 379-411, October 1993; ACM Press.