Inconsistency Management in Software Engineering: Survey and Open Research Issues

George Spanoudakis and Andrea Zisman
Department of Computing,
City University
Northampton Square, London EC1V 0HB, UK
email: {gespan | a.zisman} @soi.city.ac.uk

Abstract

The development of complex software systems is a complex and lengthy activity that involves the participation and collaboration of many stakeholders (e.g. customers, users, analysts, designers, and developers). This results in many partial models of the developing system. These models can be inconsistent with each other since they describe the system from different perspectives and reflects the views of the stakeholders involved in their construction. Inconsistent software models can have negative and positive effects in the software development life-cycle. On the negative side, inconsistencies can delay and increase the cost of system development; do not guarantee some properties of the system, such as safety and reliability; and generate difficulties on system maintenance. On the positive side, inconsistencies can facilitate identification of some aspects of the system that need further analysis, assist with the specification of alternatives for the development of the system, and support elicitation of information about it.

The software engineering community has proposed many techniques and methods to support the management of inconsistencies in various software models. In this paper, we present a survey of these techniques and methods. The survey is organized according to a conceptual framework which views inconsistency management as a process composed of six activities. These activities are the *detection of overlaps*, *detection of inconsistencies*, *diagnosis of inconsistencies*, *handling of inconsistencies*, *tracking of inconsistencies*, and *specification and application of a management policy for inconsistencies*. This paper also presents the main contributions of the research work that has been conducted to support each of the above activities and identifies the issues which are still open to further research.

1. Introduction

The construction of complex software systems is characterised by the distribution of roles and responsibilities among autonomous or semi-autonomous *stakeholders* (e.g., customers, users, analysts, designers, developers, third parties). These roles and responsibilities may be organisationally defined, be the result of the advocated system development process, follow some separation of concerns about the system under development, or be in line and reflect the different capabilities of the stakeholders involved in the process. The distribution of responsibilities and roles often results in the construction of many partial models of the developing system (referred to as "software models" in the following). These models may be requirement specifications and domain analysis models, system architecture models, structural and behavioural system design models, models of the implementation structure of the system, and/or models of the deployment of the components of the system.

Software models normally describe the system from different angles and in different levels of abstraction, granularity and formality. They may also be constructed using different notations and are likely to reflect the perspectives and the goals of the stakeholders involved in their construction. Very often, these dimensions of heterogeneity lead to inconsistencies among the models. Inconsistencies arise because the models overlap – that is they incorporate elements which refer to common aspects of the system under development – and make assertions about these aspects which are not jointly satisfiable as they stand, or under certain conditions.

Inconsistencies may have both positive and negative effects on the system development life-cycle. On the negative side, they may delay and, therefore, increase the cost of the system development process, jeopardise properties related to the quality of the system (e.g. reliability, safety), and make it more difficult to maintain the system. On the positive side, inconsistencies highlight conflicts between the views, perceptions, and goals of the stakeholders involved in the development process (which must be dealt with in an accountable way or, otherwise, they may put in risk the acceptance and usability of the system), indicate aspects of the system which deserve further analysis, and facilitate the exploration of alternatives in system development and the elicitation of information about the system.

The above benefits of inconsistencies have not been only the wisdom of academic researchers; they have also been confirmed by empirical studies (Nissen et al., 1996; Boehm & Egyed, 1998; Egyed & Boehm 2000). It has, however, to be appreciated that these benefits arise only if inconsistencies are allowed to emerge as models evolve, tolerated for at least some period and used as drivers of managed interactions among the stakeholders that can deliver the above benefits (Nissen et al., 1996). However, more important perhaps than any of these positive consequences is the fact that inconsistencies are the inevitable result of the need to describe complex systems from different perspectives, distribute responsibilities to different stakeholders in the software development life cycle, and allow them to work autonomously without requiring a continual reconciliation of their models and views for, at least, certain periods of time. These benefits and needs indicate that inconsistencies need to be "managed", that is detected, analysed, recorded and possibly resolved.

The software engineering community has been concerned with the problem of inconsistencies in software models since the late eighties and has developed techniques, methods and tools which support the identification, analysis, and treatment of various forms of inconsistencies in models expressed in a wide range of modelling languages and notations, including:

- formal specification languages including first-order logic (Easterbrook et al., 1994; Finkelstein et al., 1994; Nuseibeh et al., 1994; Easterbrook and Nuseibeh, 1995; Spanoudakis et al, 1999), Z (Bowman et al., 1996; Boiten et al., 1999; Zisman et al., 2000), LOTOS (Bowman et al., 1999), KAOS (van Lamsweerde et al., 1998; vam Lamsweerde & Latelier, 2000)
- structured requirements templates (Robinson and Fickas, 1994; Easterbrook, 1991; Kotonya & Sommerville, 1996)
- state transition diagrams (Glintz, 1995) and state-based languages (Chan et al., 1998)
- conceptual graphs (Delugach, 1992)
- object-oriented languages UML (Clarke, et al., 1996; Spanoudakis & Finkelstein, 1997; Spanoudakis & Finkelstein ,1998; Cheung et al., 1998; Ellmer et al., 1999; Zisman et al., 2000; Spanoudakis & Kassis, 2000)

The objectives of this paper are: (a) to clarify the issues which arise in connection with the problem of inconsistencies in software models, (b) to survey the research work that has been conducted to address this problem (with the exception of work concerned with inconsistencies in the specification and enactment of software process models), (b) to identify which of the arising issues have been addressed by the research work in this area so far, and (c) to identify current research trends and establish the issues which are still open to further research. Note, that this paper does not cover research work related to inconsistencies in the modelling and execution of software processes (e.g. Cugola et al., (1996))

The paper is organised according to a conceptual framework which views the management of inconsistency as a process that involves six major activities, namely the detection of overlaps between software models, the detection of inconsistencies between software models, the diagnosis of inconsistencies, the handling of inconsistencies, the tracking of the findings, the decisions made and the actions taken in the process, and the specification and application of an inconsistency management policy.

The rest of this paper is organized as follows. Section 2 contains definitions of the main phenomena, which arise in inconsistency management, and describe each of the activities of this process in detail. Section 3 presents the research work concerned with the identification of overlaps. Section 4 describes work related to the detection of inconsistencies. Section 5 is concerned with the work about diagnosis of inconsistencies. Section 6 presents work on handling of inconsistencies. Section 7 explores work related to tracking of the inconsistency management process. Section 8 presents work regarding specification and application of the inconsistency management process. Section 9 discusses open issues for further research. Finally, section 10 summarizes existing work and provides conclusions of the survey.

2. Inconsistency management: basic definitions and process

One of the factors that make it hard to understand and contrast the findings and contributions of the various strands of research and techniques developed to address the problem of inconsistencies in software models has been the lack of a commonly used terminology by the various researchers in this area (Spanoudakis et al., 1996). In this section, we attempt to overcome this problem by establishing a framework which:

a) defines the main phenomena which relate to inconsistencies in software models, and
b) describes the main activities of the process of managing inconsistencies.

2.1 Main definitions

In Section 1, we informally described an inconsistency as a state in which two or more overlapping elements of different software models make assertions about the aspects of the system they describe which are not jointly satisfiable. In this section, we define *overlaps* and *inconsistencies* in more precise terms. However, our definitions are still deliberately broad in order to accommodate the different ways in which these phenomena of overlaps and inconsistencies have been realised in the literature.

Following Spanoudakis et at (1999), we define overlaps as relations between *interpretations* ascribed to software models by specific *agents*. An agent in this setting may be a person or a computational mechanism.

An interpretation is defined as follows:

Definition 1. The interpretation of a software model S specified as a set of interrelated elements E is a pair (I,U) where:

(a) U is a non empty set of sets of individuals, called the *domain* of the interpretation of the model;
(b) I is a total morphism which maps each element *e* of E onto a relation of degree *n* $R \subseteq U^n$ called the extension of *e* (n=1,2,…, |U|).

According to this definition, an interpretation maps each element of a model onto the set of individuals or the relationships between these individuals in a given domain which are denoted by the element. Thus, an interpretation reflects how an agent understands a software model in reference to a given domain.

Given the previous definition of an interpretation, overlaps between software models are defined as follows:

Definition 2. Given a pair of elements e_i and e_j of two software models S_i and S_j and two interpretations $T_{iA} = (I_{iA}, U_{iA})$ and $T_{jB} = (I_{jB}, U_{jB})$ of S_i and S_j ascribed to them by the agents A and B, respectively:

- e_i and e_j will be said not to overlap at all with respect to T_{iA} and T_{jB} if $I_{iA}(e_i)$? Ø, $I_{jB}(e_j)$? Ø, and $I_{iA}(e_i) \cap I_{jB}(e_j) = Ø$[1]
- e_i and e_j will be said to overlap totally with respect to T_{iA} and T_{jB} if $I_{iA}(e_i)$? Ø, $I_{jB}(e_j)$? Ø, and $I_{iA}(e_i)=I_{jB}(e_j)$
- e_i will be said to overlap inclusively with e_j with respect to T_{iA} and T_{jB} if $I_{iA}(e_i)$? Ø, $I_{jB}(e_j)$? Ø, and $I_{jB}(e_j) \subset I_{iA}(e_i)$
- e_i will be said to overlap partially with e_j with respect to T_{iA} and T_{jB} if $I_{iA}(e_i)$? Ø, $I_{jB}(e_j)$? Ø, $I_{iA}(e_i) \cap I_{jB}(e_j)$ **?** Ø, $I_{iA}(e_i) - I_{jB}(e_j)$? Ø, and $I_{jB}(e_j) - I_{iA}(e_i)$? Ø

According to this definition, two model elements overlap totally if they have identical interpretations, overlap partially if the have non identical but overlapping interpretations, overlap inclusively if the interpretation of one of the elements includes the interpretation of the other and do not overlap if they have disjoint interpretations.

As Spanoudakis et al (1999) have pointed out, in reality it is impossible to enumerate the domains of interpretations with sufficient completeness for establishing the interpretations of the models and identifying overlaps on the basis of these interpretations. This is because in many cases these domains are not of bounded size or change frequently. To overcome this problem, overlaps can be represented as relations between the sets of elements E_i and E_j of two software models S_i and S_j, the set of the agents who can identify them (A), and the set of overlap kinds (O_T):

[1] The symbols $\cap$, $\cup$ and $-$ denote the set intersection, union and difference operations, respectively.

$$OV \subset E_i \times E_j \times A \times O_T$$ [2]

An element of this relation *ov(c_i,c_j,a,t)* denotes that there is an overlap of type *t* between the elements c_i and c_j of two models S_i and S_j given the interpretation of S_i and S_j that is assumed by the agent a. The set of overlap kinds O_T includes the following elements: *no* (null overlap); *to* (total overlap); *po* (partial overlap); and *io* (inclusive overlap – c_i overlaps inclusively with c_j). Agents can identify overlap relations without having to define the interpretations underlying these relations. However, if an agent A identifies that an model element e_i overlaps inclusively with an element e_j, this agent essentially confirms that $I_A(e_j) \subset I_A(e_i)$ without having to define $I_A(e_j)$ and $I_A(e_j)$. Thus, the identity of the agent becomes a substitute for a precise account of the interpretation underlying the relation and the agent who asserted an overlap relation can be traced when this relation is involved in the derivation of an inconsistency.

An inconsistency is then defined in terms of overlaps as follows:

Definition 3. Assume a set of *software models* S_1 … S_n, the sets of *overlap relations* between them $O_a(S_i,S_j)$ (i=1,…,n and j=1,…,n), a *domain theory* D, and a *consistency rule* CR. S_1, …, S_n will be said to be inconsistent with CR given the overlaps between them as expressed by the sets $O_a(S_i,S_j)$ and the domain theory D if it can be shown that the rule CR is not satisfied by the models.

The domain theory D in this definition may express some general knowledge about the domain of the system that is described by the models as in (van Lamsweerde, 1998) and/or some general software engineering knowledge (e.g. interactions between quality system features and architectural system design patterns (Boehm et al., 1995, Boehm and In, 1996).

A consistency rule may be:

- a *well-formedness* rule – These are rules which must be satisfied by the models for them to be legitimate models of the language in which they have been expressed. An example of a rule of this kind is a well-formedness rule specified in the semantic definition of the Unified Modelling Language which requires that the graph formed by the generalisation relations in UML models must be acyclic (OMG, 1999).

- a *description identity* rule – These are rules which require the different elements of software models which totally overlap to have identical descriptions (Delugach, 1992; Spanoudakis & Finkelstein, 1997; Clarke et al., 1998).

- an *application domain* rule – These are rules that specify relations between the individuals in the domain of the system which are denoted by the model elements connected by the rule. An example of such a relation for a resource management software system could be that every user who has requested a resource will get it even if this will happen some time after the request (van Lamsweerde et al, 1998).

- a *development compatibility* rule – These are rules which require that it must be possible to construct at least one model that develops further two or more other models or model elements and conforms to the restrictions which apply to both of them. This model depending on the assumed development relation might be an implementation model (Bowman et al., 1996). An example of such a rule is the existence of a common unification of two abstract data types (Boiten et al., 1999). Similar rules are also used in (Ainsworth et al., 1996).

- a *development process compliance* rule – These are rules which require the compliance of software models with specific software engineering practices or standards followed in a project. Emmerich et al (1999) give examples of numerous such rules for the case of the PSS-05 standard (QSS, 1996). As an example consider one of these rules which requires that each user requirement must have an associated measure of priority in a model that describes a system that is to be delivered incrementally (Emmerich et al., 1999).

2.2 The inconsistency management process

[2] This representation of overlap relations is a simplification of the scheme proposed by Spanoudakis et al (1999).

Inconsistency management has been defined by Finkelstein et al (1996) as the process by which inconsistencies between software models are handled so as to support the goals of the stakeholders concerned. In the literature, there have been proposed two general frameworks describing the activities that constitute this process: one by Finkelstein et al (1996) and one by Nuseibeh et al (2000). Both these frameworks share the premise that an inconsistency in software models has to be established in relation to a specific consistency rule (as in Definition 3) and that the process of managing inconsistencies includes activities for detecting, diagnosing and handling them. These core activities are supplemented by other activities which are not shared by both frameworks such as the specification and application of inconsistency management policies in (Finkelstein et al, 1996).

In this article, we propose a set of activities which unifies the above frameworks and amends them in a way which, as far as we are concerned, reflects more accurately the operationalisation of the inconsistency management process by the various techniques and methods which have been developed to support it. These activities are:

- *Detection of overlaps*

This activity is performed to identify overlaps between the software models. The identification of overlaps is a crucial part of the overall process since models with no overlapping elements cannot be inconsistent (Easterbrook 1991, Leite & Freeman 1991, Delugach, 1992, Zave & Jackson 1993, Jackson 1995, Finkelstein et al 1996, Boiten et al 1999, Spanoudakis et al 1999). The identification of overlaps is carried out by agent(s) which are specified in the inconsistency management policy (see below).

- *Detection of inconsistencies*

This activity is performed to check for violations of the consistency rules by the software models. The consistency rules to be checked are established by the adopted inconsistency management policy (see below). This policy also specifies the circumstances which will trigger the checks.

- *Diagnosis of inconsistencies*

This activity is concerned with the identification of the *source*, the *cause* and the *impact* of an inconsistency. The source of an inconsistency is the set of elements of software models which have been used in the construction of the argument that shows that the models violate a consistency rule (Nuseibeh et al., 2000). The cause of an inconsistency in our framework is defined as the conflict(s) in the perspectives and/or the goals of the stakeholders which are expressed by the elements of the models that give rise to the inconsistency. The impact of an inconsistency is defined as the consequences that an inconsistency has for a system.

The source and the cause of an inconsistency have a very important role in the inconsistency management process since they can be used to determine what options are available for resolving or ameliorating an inconsistency and the cost and the benefits of the application of each of these options (see handling of inconsistencies below). Establishing the impact of an inconsistency in qualitative or quantitative terms is also necessary for deciding with what priority the inconsistency has to be handled and for evaluating the risks associated with the actions for handling the inconsistency which do not fully resolve it (see handling of inconsistencies below).

- *Handling of inconsistencies*

The handling of inconsistencies has been considered as a central activity in inconsistency management (van Lamsweerde et al., 1998; Robinson, 1997). This activity is concerned with:

(i) the identification of the possible *actions* for dealing with an inconsistency,
(ii) the evaluation of the *cost* and the *benefits* that would arise from the application of each these actions,
(iii) the evaluation of the *risks* that would arise from not resolving the inconsistency, and
(iv) the selection of one of the actions to execute.

- *Tracking*

This activity is concerned with the recording of: (a) the reasoning underpinning the detection of an inconsistency, (b) the source, cause and impact of it, (c) the handling actions that were considered in connection with it, and (d) the arguments underpinning the decision to select one of these options and reject the other. Keeping track of what has happened in the process makes the understanding of the findings, the decisions and the actions taken by those who might need to use or refer to the software models in subsequent stages of the development life-cycle of the system easier. This is especially true for those who may not have been involved in the development of the system and/or the process of managing the inconsistencies detected in it. However, a detailed tracking of this sort certainly imposes an information management overhead to the process of inconsistency management and the software development process. This overhead has to be carefully evaluated in relation to the expected benefits.

- *Specification and application of an inconsistency management policy*

There are numerous questions that need to be answered before and during the application of an inconsistency management process. These are questions about the agent that should be used to identify overlaps between particular kinds of software models, questions about when and how often inconsistencies should be detected, questions about the diagnostic checks that should be applied to breaches of specific consistency rules, questions about the techniques that should be applied to evaluate the cost, benefits and risks associated with inconsistency handling options, and questions about the stakeholders who will undertake responsibility for handling inconsistencies. The answers to these questions depend on the kind of the software models that the inconsistency management process will have to deal with, the general software development process advocated in a specific project (Nuseibeh et al., 2000) and the particular standards that this process wants to observe (Emmerich et al., 1999), and the characteristics of the software development team (for example availability of stakeholders during particular inconsistency management activities, willingness of stakeholders to engage in resolution negotiations).

Clearly to provide coherent and effective answers to the above questions it is necessary to have a *policy* about the inconsistency management that should be applied to a particular project (Finkelstein et al., 1996). This policy must specify:

(i) the agent(s) that should be used to identify the overlaps among the partial models
(ii) the consistency rules that should be checked against the models
(iii) the circumstances that will trigger the detection of the overlaps and the inconsistencies
(iv) the mechanisms that should be used for diagnosing inconsistencies and the circumstances that should trigger this activity
(v) the mechanisms that should be used for assessing the impact of inconsistencies and the circumstances that should trigger this activity
(vi) the mechanisms that should be used for assessing the cost, benefits and risks associated with different inconsistency handling options, and
(vii) the stakeholders who would have responsibility for handling inconsistencies

This activity also establishes the mechanisms for applying an inconsistency management policy in a project and monitoring this application to ensure that progress is being made with regards to the general objectives that the policy aims to achieve (Finkelstein et al.,1996).

In the subsequent sections of this paper, we present the main ways in which the various methods and techniques which have been developed to handle inconsistencies in software models support the above inconsistency management activities. In the end of each section, we present a table summarizing the existing techniques.

3. Detection of overlaps

The methods and techniques that have been developed to support the management of inconsistencies in software models detect overlaps based on representation conventions, shared ontologies, human inspection, and forms of similarity analysis of the models involved.

The different ways in which each of these approaches have been realised is discussed after presenting the properties of the different types of overlap relations defined in Section 2.1. Spanoudakis et al (1999) have shown that according to Definition 2:

- total overlap is a reflexive, symmetric and transitive relation
- inclusive overlap is an irreflexive, antisymmetric and transitive relation, and
- partial overlap is an irreflexive and symmetric relation

These properties give us a basis for establishing the exact type of the overlap relations that each of the approaches in the literature can identify.

3.1 Representation conventions

The simplest and most common representation convention is to assume the existence of a total overlap between model elements with identical names and no overlap between any other pair of elements. This convention is widely deployed by model checking (Heitmeyer et al., 1995; Chan et al., 1998) and other specialised model analysis methods and techniques (Clarke et al., 1998).

The same convention also forms the basis of the classical unification algorithms (see Knight (1989) for a survey) and predicate matching process which are used to detect overlaps in all the logic-based methods and techniques for inconsistency management (Easterbrook et al., 1994; Finkelstein et al., 1994; Ainsworth et al., 1996; Nissen et al., 1996; van Lamsweerde et al., 1998; Boiten et al., 1999; Emmerich et al., 1999, Robinson & Pawlowski, 1998). The classical unification algorithms find the *most general unifier* (if one exists) between terms expressed in some first-order logical language. A term in such a language is either a constant or a variable symbol, or a function symbol followed by a series of other terms separated by commas.

The unification algorithms perform a syntactic matching between the terms that they are given to unify. For terms which start with a function symbol the matching is successful only if these symbols are the same for both terms, the terms have the same number of subterms (arity), and the matching of their corresponding subterms is also successful. Variable symbols can be matched freely with constant or other variable symbols or terms which start with function symbols provided that the latter do not contain the variable symbol that they are matched with in any of their subterms (i.e., the *occur* check (Knight, 1989)). This matching process returns a mapping of the variables of the terms onto other terms, called *substitution*. The most general unifier is a substitution σ between two terms t_1 and t_2 for which there is always another substitution τ that can be applied to σ in order to translate it to any other unifier θ of t_1 and t_2, that is $\tau(\sigma)=\theta$[3].

As an example of classical term unification consider the predicates *student* and *project_supervisor* below:

- *student(x),*
- *student(a),*
- *student(y),*
- *project_supervisor(x,personal_tutor(x)),*
- *project_supervisor(y,x),* and
- *project_supervisor(a,personal_tutor(a))*

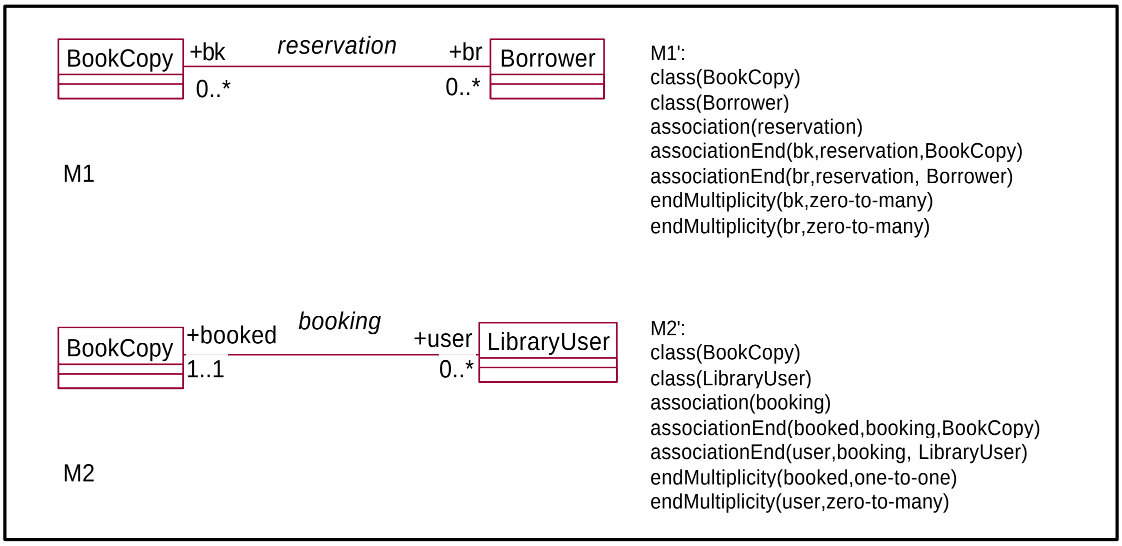
where "student" and "project_supervisor" are predicate symbols, "personal_tutor" is a function symbol, *x* and *y* are variable symbols and *a* is a constant symbol. In this example, the most general unifier of *student(x)* and *student(a)* is {x/a}, the most general unifier of *student(x)* and *student(y)* is {x/y} and the most general unifier of *project_supervisor(x,personal_tutor(x))* and *project_supervisor(a,personal_tutor(a))* is {x/a}[4].

It has to be appreciated that unification makes a number of matching assumptions which turn out to be weak given the forms of heterogeneity which may appear in independently constructed software models (see Section 1). These weak assumptions are: (a) that only terms with the same function symbol can be matched, (b) that only terms with the same arity can be matched, and (c) that only subterms which appear in the same relative position in the structure of a term can be matched. Assumptions analogous to (a), (b) and (c) are also used in matching predicates while making inferences using the standard inference rules of classical logic. In

[3] $\tau(\sigma)$ denotes the application of the substitution τ onto the result of the application of the substitution σ.

[4] The symbol "/" means "is replaced by".

this process, unification is attempted only between predicates with the same predicate symbol and arity, and only between the corresponding terms in the structures of two predicates. The assumption (a) in both predicate matching and term unification has been the subject of extensive criticism by many researchers in this area including Zave & Jackson (1993), Jackson (1997), Spanoudakis et al. (1999) and Boiten et al. (1999)) as it turns out to be inadequate even for simple forms of heterogeneity between models such as the presence of synonyms and homonyms.



```
M1':
class(BookCopy)
class(Borrower)
association(reservation)
associationEnd(bk,reservation,BookCopy)
associationEnd(br,reservation, Borrower)
endMultiplicity(bk,zero-to-many)
endMultiplicity(br,zero-to-many)
```

```
M2':
class(BookCopy)
class(LibraryUser)
association(booking)
associationEnd(booked,booking,BookCopy)
associationEnd(user,booking, LibraryUser)
endMultiplicity(booked,one-to-one)
endMultiplicity(user,zero-to-many)
```

Figure 1: Two UML class diagrams and their translations into a first-order language

To give an example, consider for instance the two UML class diagrams M1 and M2 and their translations M1'and M2' into the first-order logical language shown in Figure 1[5] and the consistency rule CR1 below:

CR1: $(\forall x_1)\ (\forall x_2)\ (\forall x_3)\ (\forall x_4)\ (\forall x_5)\text{: associationEnd}(x_1, x_2, x_3) \wedge \text{endMultiplicity}(x_1, x_4) \wedge \text{endMultiplicity}(x_1, x_5) \rightarrow (x_4 = x_5)$

CR1 requires all the assertions about the multiplicity of an association end in one or more models to be the same (i.e. a *description identity* rule). The models M1' and M2' are not inconsistent with respect to CR1 if the overlaps between their elements are detected using unification. Clearly, M1' and M2' would not satisfy CR1 if the association ends "bk" and "booked" in them were known to overlap totally. However, standard unification wouldn't be able to detect this total overlap since the non-identical constants "bk" and "booked" cannot be unified. Possible ways of overcoming this problem in the framework of first-order logic are discussed in Section 4.1.

In the case of first-order languages with function symbols, classical unification gives rise to a reflexive and symmetric relation between predicates but not transitive. In the above example, for instance, although the predicate *project_supervisor(x,personal_tutor(x))* can be unified with the predicate *project_supervisor(a,personal_tutor(a))* (mgu = {x/a}) and the predicate *project_supervisor(a,personal_tutor(a))* can be unified with the predicate *project_supervisor(y,x)* (mgu = {y/a,x/personal_tutor(a)}) there is no unifier for the predicates *project_supervisor(x,personal_tutor(x))* and *project_supervisor(y,x)*. Thus, in the case of such languages, it is not possible to classify the overlap relation between a pair of unified predicates in terms of the overlap types of Spanoudakis et al (1999) (a unification relation may be a total or a partial overlap). Note, however, that transitivity is guaranteed when unification is applied to models expressed in first-order languages with no function symbols as those used by Zave & Jackson (1993), Easterbrook et al (1994), Finkelstein et al (1994), van Lamsweerde et al (1998), and Spanoudakis et al (1999). Thus, in the case of such languages, it can be argued that unification detects total overlap relations.

[5] The first-order representation of the class diagram in Figure 1 follows a commonly suggested scheme for translating diagrammatic models into a first-order language. According to this scheme, the name of an element of the diagram becomes a constant that denotes the element and appears as an argument of a predicate that denotes the construct of the language that is instantiated by the element. Finkelstein et al. (1994) and Easterbrook & Nuseibeh (1995) use similar translation schemes.

3.2 Shared ontologies

An alternative approach for the identification of overlaps is to use shared ontologies. This approach requires the authors of the models to tag the elements in them with items in a shared ontology. The tag of a model element is taken to denote its interpretation in the domain described by the ontology and therefore it is used to identify overlaps between elements of different models. A total overlap in this approach is assumed when two model elements are "tagged" with the same item in the ontology (Leite & Freeman 1991, Robinson 1994, Robinson & Fickas 1994, Boehm & In 1996).

The ontologies used by Robinson (1994), and Robinson and Fickas (1994) in their Oz system are domain models which prescribe detailed hierarchies of domain objects, relationships between them, goals that may be held by the stakeholders, and operators which achieve these goals. The software models which can be handled by their techniques are constructed by instantiating the common domain models they propose. In searching for inconsistencies their techniques assume total overlaps between model elements which instantiate the same goal in the domain model. The ontology used by the QARCC system (Boehm & In 1996) is a decomposition taxonomy of software system quality attributes. This ontology also relates quality attributes with software architectures and development processes that can be used to achieve or inhibit them. For example, there is a quality attribute called "portability" in this ontology which is decomposed into the quality attributes "scalability" and "modifiability". Portability, according to QARCC’s ontology can be achieved through a "layered" system architecture and a "prototyping" approach to system development. The software requirements models that QARCC analyses for inconsistencies are described through "Win" conditions which are connected to quality attributes. An overlap between two Win conditions is established if the quality attribute that these conditions refer to can be realised or inhibited by a common architecture or system development process.

It has to be appreciated that in order to facilitate communication between stakeholders about a specific domain of discourse, without necessarily assuming the existence of a global shared theory amongst the stakeholders, ontologies have to provide definitions of the items they include and be general. Furthermore, the stakeholders need to "commit" themselves to the ontology. Commitment in this setting means that observable actions of the stakeholders are consistent with the definitions of the items in the ontology (Gruber 1993, Guarino 1994). As a consequence of ontology generality, software models have to add a lot of details to an ontology in order to describe a complex system with reasonable degree of completeness. Inevitably this leads to associations of many model elements with the same item in the ontology and as a result only coarse-grain overlaps can be identified by using these items. Furthermore, since ontologies incorporate item definitions, they are models themselves and as such they might be interpreted in different ways by different stakeholders! Thus, the same ontology item may be used for ascribing different meanings to different model elements. As a consequence, the existence of overlaps between elements associated with the same item in an ontology cannot be assumed with safety unless there is evidence that the stakeholders understood the ontology in the same way and committed themselves to it.

3.3 Human inspection

A third general approach is to get the stakeholders to identify the overlap relations. Numerous methods and techniques rely on this approach (see Easterbrook (1991), Delugach (1992), Zave & Jackson (1993), Fiadeiro & Maibaum (1995), Jackson (1997), Bowman et al (1996), Boiten et al (1999)).

Synoptic (Easterbrook 1991), for instance, expects stakeholders to identify "strong" and "weak" correspondences between models which correspond to total and partial overlaps in our framework. The support provided by this tool takes the form of visual aid for browsing, graphical selection and recording of overlaps. Delugach (1992) requires stakeholders to specify "counterpart" relationships between model elements. These relationships correspond to total overlaps.

Zave & Jackson (1993) suggest the use of "description" graphs to relate the predicate symbols in the "signatures" of different software models. A model signature includes the symbols of the predicates in the model which cannot be defined in terms of other predicates. The decision about the exact predicates that should belong to the signature of a model is language-dependent. For example, in the case of representing a state-transition diagram in a first-order language if the predicates which represent the states of a diagram become members of the signature of the model, then the predicates which represent the transitions of the diagram will not and vice versa (Zave & Jackson 1993). A relation between two predicates in the description graph can be created only if there is an overlap between them and both models include assertions which incorporate the predicates. The relations specified in description graphs do not distinguish between different types of overlaps.

Jackson (1997) suggests the identification of overlaps by virtue of "designations". Designations constitute a way of associating non ground terms in formal models with ground terms which are known to have reliable and unambiguous interpretations (called "phenomena"). A designation associates a non ground term with a recognition rule which identifies the phenomena designated by it. The rule is specified in natural language. Designations can certainly help stakeholders in identifying overlaps but shouldn't be used as definitive indications of them. The reason is that the recognition rules of the designations might themselves admit different interpretations.

Boiten et al (1999) expect the authors of the Z schemas that their techniques are dealing with to identify "correspondence" relations between the variables which appear in these schemas (but not the predicates). These relations correspond to total overlaps and are then used in constructing possible compositions of two or more Z schemas (i.e., the basic way of exploring and handling inconsistencies in their approach). A similar approach is taken in (Bowman et al .,1996) for the Open Distributed Processing models expressed in LOTOS.

Fiadeiro & Maibaum (1995) suggest the representation of overlap relations between model elements using the formal framework of the category theory (Goguen & Ginali 1978). They formalise models as categories (directed graphs with a composition and identity structure) and use functors between these categories (i.e. functional mappings between the nodes and edges of the categories) to interconnect them. The functors are then checked if they preserve the structures and therefore the properties of the category elements that they interconnect. This check in our framework constitutes the detection of inconsistencies. The idea to use functors to represent overlaps has been supported by other authors (Easterbrook et al., 1998) who however, criticised the stance that it should be checked whether functors preserve the structures of the parts of the models they interconnect. This criticism has been on the grounds that such a check would be too strict in the development of large software systems.

Spanoudakis et al (1999) have also acknowledged the need to check the consistency of overlap relations but they propose a less strict check. According to them, a set of asserted overlap relations should be checked if they satisfy certain properties which arise from the formal definition of overlaps given in Section 2.1. For example, if it has been asserted that a model element *a* inclusively overlaps with a model element *b* but has no overlap with a third element *c* then it should be checked that there is no total, inclusive or partial overlap between *b* and *c*. In their view, this check should be performed before checking the consistency of the models involved. This check is particularly useful in cases of overlaps asserted by humans.

The main difficulty with the identification of overlaps using inspections by humans is that this identification becomes extremely time consuming even for models of moderate complexity.

3.4 Similarity analysis

The fourth general approach is to identify overlaps by automated comparisons between the models. This approach exploits the fact that modelling languages incorporate constructs which imply or strongly suggest the existence of overlap relations. For instance, the "Is-a" relation in various object-oriented modelling languages is a statement of either an inclusive overlap or a total overlap. This is because "Is-a" relations normally have a set-inclusion semantics, that is the subtype designates a proper or not proper subset of the instances of the supertype. Similarly, the implication ($\rightarrow$) between two predicates of the same arity constitutes a statement of inclusive overlap in a first order language.

The comparison methods that have been deployed to realise this approach search for structural and semantic similarities either between the models themselves (Spanoudakis & Finkelstein 1997) or between each model and abstract structures that constitute parts of domain specific ontologies (Maiden et al. 1995). In the "reconciliation" method of Spanoudakis & Finkelstein (1997) the detection of overlaps is formulated as an instance of the *weighted bipartite graph matching problem* (Papadimitriou & Steiglitz 1982). The nodes in the two partitions of the graph in their method denote the elements of the models being compared, the edges of the graph represent the possible overlap relations between these elements, and the weights of the edges are computed by distance functions which measure modelling discrepancies in the specifications of the elements connected by the edge with respect to different semantic modelling abstractions. The method detects partial overlaps between the elements of the two models by selecting the morphism between the two partitions of the overlap graph that has the lowest aggregate distance. Palmer & Fields (1992) also identify overlaps between software requirements models expressed in multimedia documents by using indexing and clustering techniques.

Overall, it should be noted that similarity analysis techniques tend to be sensitive to extensive heterogeneity in model representation, granularity and levels of abstraction.

Approach	Main Assumptions	Positive Features	Main Limitations
Representation conventions	• models expressed in some formal language	• relatively inexpensive way of overlap identification	• sensitive to simple forms of model heterogeneity • applicable only to models expressed in the same language
Shared ontologies	• existence of well-defined ontologies • models need to be related to the ontologies	• applicable to models expressed in different languages	• stakeholders may understand an ontology differently • identification of only coarse-grain overlaps
Human inspection	• stakeholders need to identify overlap relations	• certainty in overlap identification • applicable to models expressed in different languages	• labour intensive
Similarity analysis	• models need to be related through a common meta-model	• automatic identification of overlaps • applicable to models expressed in different languages	• sensitive to model heterogeneity • resulting overlaps are not always accurate

Table 1: Summary of Assumptions, Positive Features and Limitations of Approaches to Identification of Overlap

3.5 Summary

A summary of the assumptions, and the main positive features and limitations of the various approaches to the identification of overlaps is given in Table 1.

4. Detection of Inconsistencies

Our survey has indicated that, there have been four broad approaches to the detection of inconsistencies in software models. These are:

- the logic-based approach
- the model checking approach
- the specialised model analysis approach, and
- the human-centered collaborative exploration approach

The basic mechanisms used by each of these approaches and their merit are discussed next.

4.1 Logic-based detection

The logic-based approach to the detection of inconsistencies is characterised by the use of some formal inference technique to derive inconsistencies from software models expressed in a formal modelling language (e.g. first-order classical logic (Finkelstein et al., 1994; Easterbrook et al, 1994; Nuseibeh et al., 1994; Easterbrook and Nuseibeh, 1995a; Spanoudakis et al., 1999), real-time temporal logic (van Lamsweerde et al., 1998; vanLamsweerde & Letelier, 2000), Quasi-Classical (QC) logic (Hunter and Nuseibeh, 1998), Object Constraint Language (Spanoudakis and Kassis, 2000), assertional language of O-Telos (Nissen et al., 1996), Z (Bowman et al., 1996; Ainsworth et al., 1996)).

The methods and techniques which adopt this approach use the following reformulation of the definition of inconsistencies given in Section 2.1:

Definition 4. Assume a set of software models S_1 … S_n, sets of overlap relations between their elements $O_a(S_i,S_j)$ (i=1,…,n and j=1,…,n) and a *consistency rule* CR. S_1, …, S_n are inconsistent with respect to the rule CR when they overlap as indicated by the sets $O_a(S_i,S_j)$ if and only if:

$$\{F(G_1(S_1), \ldots, G_n(S_n), O_a), D\} +_L \neg CR$$

where

- O_a is the set of all the overlap relations asserted by an agent *a*: $O_a \equiv \cup_{\{i=1,\ldots,n\}\{j=1,\ldots,n\}} O_a(S_i,S_j)$
- G_i is a transformation that translates the model S_i from the original language in which it was expressed into the formal language assumed by the technique (i=1,…,n)
- F is a transformation from the vocabularies of the translated models (these are the sets of their predicate, function, variable and constant symbols) into a new common vocabulary which is used to represent the overlap relations between these symbols
- D are the axioms of some domain theory (see Section 2.1)
- $+_L$ is a consequence relation based on a particular set of inference rules L.

Definition 4 constitutes a parametrised re-formulation of definitions of inconsistencies in (Finkelstein et al., 1994; Bowman et al., 1996; Nissen et al., 1996; Hunter & Nuseibeh, 1998; Spanoudakis et al., 1999; Nuseibeh & Russo, 1999). It is also a special case of the Craig & Robinson's theorem of joint consistency according to which two theories T_i and T_j are inconsistent if and only if there is a formula A such that $T_i +$ A and $T_j + \neg$A ((Shoenfield, 1967), pg. 79).

Examples of transformations that could be used in the place of G for different non logic-based software modelling languages are given in (Zave & Jackson, 1993; Finkelstein et al., 1994).

Note also that, F becomes the *identity* transformation (and therefore can be ignored) if the overlaps between the models are identified by standard unification (see Section 3.1). Spanoudakis et al (1999) have proposed a transformation F that gets as input two models expressed in a first-order logical language (with no function symbols) and a set of overlap relations asserted by some agent between their constant and predicate symbols and translates them into a single model that can then be checked by normal theorem proving and unification for inconsistencies. Using this transformation a total overlap relation between the constants "bk" and "booked" in the models M1' and M2' of Figure 1, *ov(bk,booked,a,to)*, would be translated into the fact (bk = booked) (assuming the application of the intra-overlap algorithm presented in Spanoudakis et al (1999)). Then given the following translation of the rule CR1 (see Section 4.2):

CR1': $(\forall x_1)$ $(\forall x_2)$ $(\forall x_3)$ $(\forall x_4)$ $(\forall x_5)$: associationEnd$(x_1, x_2,x_3) \wedge$ associationEnd$(x_6,x_7,x_8) \wedge$ endMultiplicity$(x_1, x_4) \wedge$ endMultiplicity$(x_6, x_5) \wedge (x_1 = x_6) \rightarrow (x_4 = x_5)$

the inconsistency between M1' and M2', and CR1' that we discussed in Section 4.2 would be detectable (if the formulas of M1' and M2' are expanded by the set of *equality* axioms (Stepherdson, 1988)). It should be noted, however, that one of the limitations of this re-writing scheme is that it does not handle inclusive or partial overlaps between constant symbols and cannot handle overlaps relations between predicates of different arities.

Most of the techniques which advocate the logic-based approach operationalise the consequence relation $+_L$ by using theorem proving based on the standard inference rules of classical logic such as resolution, conjunct and negation elimination, instantiation of universally quantified formulas, and other rules for introducing negative information in the models such as the closed-word-assumption (CWA) (see (Hogger, 1990)). These techniques include (Finkelstein et al., 1994; Easterbrook et al, 1994; Nuseibeh et al., 1994; Easterbrook and Nuseibeh, 1995a; Spanoudakis et al., 1999). A theorem proving facility is also incorporated in the technique that checks the existence of a common unifying abstract data type (ADT) for two given ADTs expressed in Z that has been developed by Bowman et al (1996) and a similar technique described in (Ainsworth et al., 1996).

A similar approach, in terms of the inference mechanism used, is also taken by Nissen et al. (1996), Robinson and Pawlowski (1998), Emmerich et al (1999), and Spanoudakis and Kassis (2000).

Nissen et al (1996) assume software models and consistency rules expressed in O-Telos (Jarke et al., 1995). O-Telos is a variant of Telos (Mylopoulos et al., 1990), an object-oriented knowledge representation language that allows the specification of integrity constraints and deductive rules using its own assertion sub-language. Integrity constraints and deductive rules in O-Telos are associated with specific classes (or meta-classes) to restrict the relationships of their instances with other objects and to deduce information about these instances, respectively. The adequacy and efficiency of Telos in representing a wide range of software models has been demonstrated in research and industrial projects (Constantopoulos et al., 1995). Nissen et al (1996) specify consistency rules as "query classes". A query class in O-Telos is an object class defined as a subclass of one or more normal classes with detectable class membership. This means that the objects which are instances of all the superclasses of a query class become instances of it only if their description satisfies an integrity constraint that has been specified for the query class. Thus, if CR is a consistency rule that must be checked against specific kinds of software model elements in O-Telos, a query class may be created as a subclass of the classes that represent these kinds of elements having $\neg$CR as its associated integrity constraint. Robinson and Pawlowski (1998) also use O-Telos query classes to express development process compliance rules and check the compliance of requirements models against these rules as described above.

Spanoudakis & Kassis (2000) adopt a very similar approach in checking whether UML models satisfy certain well-formedness consistency rules. In their framework, the consistency rules are specified as *invariants* in the Object Constraint Language (OMG, 1999) and are associated with meta-classes in the UML meta-model. These meta-classes represent the various types of UML model elements and the relationships between them. An invariant is checked against all the elements of a model which are instances of the UML meta-class that it is associated with. The approach taken by Emmerich et al (1999) is also similar to it. Their system checks the compliance of structured requirements documents against consistency rules which determine relationships that should hold between them. The contents of documents are expressed as sets of relations. The rules expressible in their tool are expressed in AP5, an extension of Common Lisp, which supports the specification and checking of formulas similar to those that can be expressed in first-order logic (Cohen, 1992). These rules can make references to the relations used to represent the documents.

One of the criticisms against reasoning using a consequence relation ($\vdash_L$) based on the standard inference rules of classical logic is that reasoning is trivialised in the case of inconsistent information (Hunter & Nuseibeh, 1998). This is the result of allowing the introduction of arbitrary disjuncts in formulas in the inference process: if a formula α is known then $\alpha \vee \beta$ is valid consequence of it according to the standard set of inference rules of classical logic (Hunter and Nuseibeh, 1998). Thus, in the case where both α and $\neg\alpha$ are part of a model then any formula β can be derived from this model (first by obtaining $\alpha \vee \beta$ and then by resolving $\alpha \vee \beta$ with $\neg\alpha$). This problem of having arbitrary trivial inferences following from inconsistent information (termed as "ex falso quodlibet" in (Shoenfield (1967)) is significant in software engineering settings. The significance comes from the fact that software models – especially those developed in the early stages of the software development life-cycle when the stakeholders may have a vague understanding of the system under construction and its domain – are likely to incorporate inconsistent information (Krasner et al., 1987; Curtis et al., 1988; Waltz et al., 1993; Nuseibeh et al., 2000).

To address this problem Hunter and Nuseibeh (1998) have suggested the use of QC-logic (Besnard & Hunter, 1995). QC-logic overcomes the above problem by allowing no further inferences after the introduction of an arbitrary disjunct in a formula during theorem proving.

The major advantage that theorem proving has over its competitors as a reasoning mechanism for deriving inconsistencies is the fact that the reasoning rules that it deploys have an extensively studied, well-defined and sound semantics. Also, theorem proving can in principle be used to derive inconsistencies against any kind of consistency rule. Note, however, that theorem proving also has two major disadvantages. The first disadvantage is that first-order logic is semi-decidable (if a formula does not hold in a first-order model trying to prove it using theorem proving may never terminate). The second disadvantage is that theorem proving is computationally inefficient. These two disadvantages have been the basis of criticism (Zave & Jackson, 1993) against this approach.

Finally, there have been proposed two more formal techniques for the detection of inconsistencies. These are the derivation of "boundary conditions" by goal regression and the detection of inconsistencies using patterns of divergences. Both these techniques have been used to detect "divergences" in models of requirements expressed in the KAOS language (van Lamsweerde et al, 1998; van Lamsweerde and Letelier, 2000). This language advocates a goal-oriented approach to requirements specification. The modelling of a system starts by specifying the goals that the system should meet. Goals are refined into successive layers of subgoals by AND/OR refinements until the fulfillment of these subgoals may become the responsibility of an individual agent(s), for example a software agent, or an agent in the environment of the system. KAOS incorporates a real-time temporal logical language to specify goals formally.

A divergence in KAOS is a special kind of an inconsistency. It is defined as a state where some goal about the system cannot be satisfied given the rest of the goals about the same system, the overlap relations which exist between them, some domain theory about the system, and a boundary condition which arises due to some feasible agent behaviour (van Lamsweerde et al., 1998).

Goal regression is used in KAOS to identify boundary conditions that give rise to divergences. The technique is based on backward chaining and assumes that the goals and the domain theory relevant to a system are all specified as formulas of the form A → C. Backward chaining starts by taking the negation B of a goal G (B = ¬G) and tries to unify any subformula L in B with the implication part C of some formula in the domain theory or the other goals (the latter are tried first). If such a formula is found the most general unifier of L and C, *mgu(L,C)*, is applied to the premises of the implication A, and the resulting formula replaces L in B. This procedure is repeated until no more unifications can be found or until the stakeholders recognise the new form of formula B as a boundary condition that may arise under some scenario regarding the operation of the system. This scenario is an interaction with the system that can produce a behaviour H which entails B, H¦ B (van Lamsweerde & Letelier, 2000). The identification of such a scenario as suggested in (van Lamsweerde and Letelier, 2000) can be done manually, by planning or by model checking (see Section 5.2). In (van Lamsweerde & Letelier, 2000) van Lamsweerde and Letelier present a variant of goal regression that can be used to identify boundary conditions which are consistent with a domain theory and which together with this theory entail the negation of a goal. These conditions are called "obstacles" (van Lamsweerde & Letelier, 2000).

Goal regression is useful mostly in cases where the objective is to explore the potential for inconsistencies rather than to identify existing inconsistencies. In the special case of searching for obstacles which can obstruct the achievement of individual goals, the technique in effect explores exceptional behaviours and is used to "derive more complete and realistic goals, requirements and assumptions" (van Lamsweerde & Letelier, 2000). The main difficulty that we envisage in practical applications of goal regression is the lack of heuristic criteria that could direct the search for boundary conditions towards goals and domain formulas whose sub-formulas would be more likely to appear in prominent scenarios. This would certainly speed up the regression process for large complex domain theories and goal hierarchies. The developers of the technique admit this weakness (van Lamsweerde et al.,1998).

van Lamsweerde et al (1998) have also identified a number of frequent patterns of divergence which they have formally described and use to detect divergences in goal-oriented models. A divergence pattern associates the abstract syntactic forms of two goals with an abstract syntactic form of the boundary condition that gives rise to a divergence between them. Each pattern is matched against goals in the models and if the match is successful the pattern is instantiated to generate the boundary condition that gives rise to the divergence.

Consider for example the divergence pattern called "retraction" which is shown in Figure 2 and the goal (G1) and the domain assertion (D1) shown below (the formulas include operators of temporal logic whose semantics are informally explained in Figure 2):

(G1) Fire → ? Fire-Alarm-Rings
(D1) Fire-Alarm-Rings → Fire

The goal G1 specifies that if there is a fire the fire alarm of a system will ring at some point after it. The domain assertion (D1) specifies that when the fire alarm rings there is a fire. G1 and D1 match with the syntactic form of the divergent assertions in the "retraction" pattern of Figure 2: "Fire" matches with P and

"Fire-Alarm-Rings" matches with Q. These matches are used in the following instantiation of the boundary condition of the pattern:

$? [\text{Fire} \wedge (\neg \text{Fire-Alarm-Rings}\ \upsilon\ ?\ \neg \text{Fire})].$

This formula represents a potential state where there will be a fire but the file alarm will not ring until the fire is off.

Patterns of goals and obstacles that can obstruct them are also given in (van Lamsweerde & Letelier, 2000). Divergence and obstacle patterns have proved to be sound (i.e. the boundary condition or the obstacle that they specify entail the negation of the goal(s) of the pattern (van Lamsweerde et al., 1998; van Lamsweerde & Letelier, 2000). The main benefit arising from the use of divergence patterns is that they can detect boundary conditions for certain combinations of goals more efficiently than goal regression. However, as admitted by van Lamsweerde et al (1998) the current set of patterns has to be expanded to capture a larger part of the range of divergences that can be found in goal specifications for complex systems.

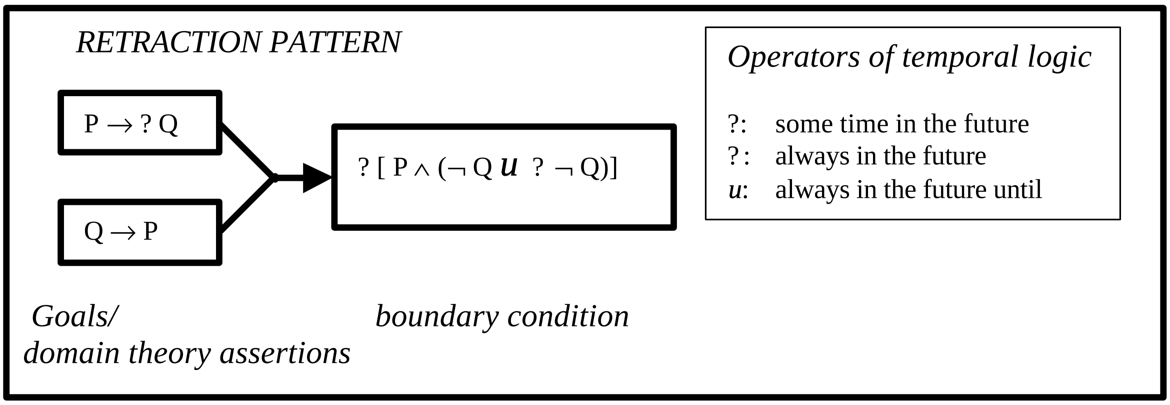

RETRACTION PATTERN
P → ? Q
Q → P
? [P ∧ (¬ Q u ? ¬ Q)]
Goals/ domain theory assertions
boundary condition
Operators of temporal logic
?: some time in the future
?: always in the future
u: always in the future until

Figure 2: The retraction divergence pattern (based on (van Lamsweerde et al., 1998))

4.2 Detection based on model checking

Model checking methods and techniques deploy, as indicated by their name, specialised model checking algorithms, for example SMV (McMillan, 1993) and Spin (Holzmann, 1997). Model checking algorithms were originally developed for and have been proved to be an effective means of detecting errors in hardware designs. Their use for checking the compliance of software models with consistency rules started in the mid-nineties. Two of the techniques that we surveyed deploy model checking algorithms. The first of these techniques was developed to check the consistency of software models expressed in the notations used by the SCR (Software Cost Reduction) method (Heitmeyer et al., 1995; Heitmeyer et al., 1996; Bharadwaj & Heitmeyer 1999). The second technique was developed to analyse requirements specification models expressed in the RSML (Requirements State Machine Language (Heimdahl & Leveson, 1996)) by Chan et al (1998).

In the SCR method a software system is described through a set of variables that take values in specific ranges. A SCR software model describes how the values of these variables change as the system moves from one state to another. A state is defined as a set of assignments of specific values to the variables. The system moves from one state to another when specific events occur in its environment. A SCR model is specified using a tabular representation scheme that constitutes an extension of state transition diagrams (STD). Models represented in this scheme may be checked against a variety of both well-formedness and application domain rules called "invariants" (see Section 2.1). Well-formedness rules in this case are concerned with the syntactic and type correctness of SCR models, and the absence of circular definitions and undesired non determinism. These rules are checked using forms of analysis that pertain to the particular model specification language of SCR. Invariants in this case determine legitimate combinations of values of variables at any state of the system (*state invariants*) or legitimate changes of values of variables after state transitions (*transition invariants*). Model checking is used to establish whether state invariants hold on all the possible states of the system and whether transition invariants hold after the transitions they are associated with (Bharadwaj & Heitmeyer 1999). The application of model checking requires the translation of SCR models into the languages used by the model checker. SCR provides automatic translation of its models into the languages used by SMV and Spin.

Chan et al (1998) have also developed a scheme which translates a requirements specification expressed in RSML into the language of the symbolic model verifier SMV. This language is based on binary decision diagrams (BDDs (Bryant 1986)). The scheme is described in (Chan et al., 1986).

The main problem with model checking arises with systems which have a non finite number of states. Model checking is also inefficient due to the explosion of the sequences of the state transitions that must be generated when checking consistency rules against systems specified by complex state transition diagrams (Easterbrook, 1997). Recent research tries to address this problem by using abstraction techniques such as *variable restriction* and *variable elimination* (Bharadwaj & Heitmeyer 1999).

4.3 Detection of inconsistencies based on specialised forms of automated analysis

Numerous inconsistency management methods and techniques use specialised ways of checking the consistency of software models including (Delugach, 1992; Robinson & Fickas, 1994; Glintz, 1995; Spanoudakis & Finkelstein, 1997; Spanoudakis & Finkelstein, 1998; Clarke et al., 1998; Cheung et al., 1998; Elmer et al., 1999; Zisman et al., 2000). Most of these techniques check the satisfiability of specific consistency rules.

The "reconciliation" method developed by Spanoudakis and Finkelstein (1997; 1998) checks the consistency of UML models against rules which require the totally overlapping elements in two UML models (OMG, 1999) to have identical descriptions. This method uses distance functions to compare model elements, and to identify and quantify the discrepancies in their descriptions (Spanoudakis & Constantopoulos, 1995). The technique developed by Clarke et al (1998) also checks whether or not overlapping elements of UML class diagrams have identical descriptions but is based on a much simpler matching mechanism. Delugach's technique (Delugach, 1991) checks whether totally overlapping elements of requirements models expressed as conceptual graphs have: (1) identical descriptions, or (2) any parts in their descriptions that can be mutually disproved. His technique is based on comparisons of conceptual graphs. In Oz (Robinson & Fickas, 1994) inconsistencies are detected between identical goals in different stakeholder perspectives which are to be realised by different design plans.

Glintz (1995) has developed a technique that checks behavioural software models expressed as statecharts (Harel, 1987) for deadlocks, reachability and mutual exclusiveness of states. Cheung et al (1998) have developed a technique that checks whether the sequence of the execution of operations that is implied by a UML statechart diagram is compliant with the sequence of the executions of operations implied by a UML sequence diagram. Their approach is based on the transformation of these diagrams into Petri-nets (Peterson, 1981).

Ellmer et al. (1999) have developed a technique for detecting inconsistencies in distributed documents with overlapping content. The documents represent either software models generated during software systems development life cycle, or general business documents (e.g. financial documents). The technique is based on the eXtensible Markup Language (Brayan, 1998) and related technologies and uses consistency rules to describe the relationships that are required to hold among the documents. Zisman et al (2000) present a language based on XML and related technologies to express these consistency rules. A consistency link generator has been developed to check the various consistency rules through the participating documents and associate related elements. The result of this checking is specified by "consistency links", represented in XLink (De Rose, 1998). A consistency link can be either "inconsistent" or "consistent" depending on whether or not the consistency rule that it was created for has failed. The technique has been tested for UML and Z software models.

4.4 Detection of inconsistencies based on human-centered collaborative exploration

Many of the techniques and methods that have been developed to support the management of inconsistencies between software models assume models or parts of models expressed in informal modelling languages (mainly supporting forms of structured text). These techniques include Synoptic (Easterbrook, 1991), QARCC

(Boehm & In, 1996), DealScribe (Robinson & Pawlowski, 1998), VORD (Kotonya & Sommerville, 1996). In these techniques the detection of inconsistencies is assumed to be the result of a collaborative inspection of the models by the stakeholders. A similar approach is also used as an option in the technique for divergence and obstacle management developed by van Lamsweerde et al (1998), and van Lamsweerde and Letelier (2000).

In Synoptic stakeholders are expected to fill the so-called conflict forms to describe a conflict that in their view exists between the overlapping model elements which are referenced in the form. A conflict in Synoptic may relate to non-identical descriptions of the same situation or the design of a system, or non-identical terms which have been used to denote the same aspect of the system.

As we discussed in Section 3.2, in QARCC a potential conflict is identified between "Win" conditions which are connected to two quality attributes *a* and *b* if the attribute *a* is realised by an architecture or system development process that inhibits *b* or vice versa. A potential inconsistency becomes the subject of further analysis only if the stakeholders who own the involved "Win" conditions decide to create an "issue" which confirms and explains the nature of the identified conflict.

In DealScribe (Robinson & Pawlowski 1998), the stakeholders are expected to identify conflicts between the so-called "root requirements" in their models. Root requirements are identified for each concept that is represented in the software models and have the property of being the most general requirements defined for the relevant concepts (requirements in DealScribe are organised in generalisation hierarchies). The stakeholders are expected to explore the interactions between all the possible pairs of root requirements in the models and indicate for each pair the nature of the interaction between its requirements elements. An interaction may be characterised as "very conflicting", "conflicting", "neutral", "supporting", or "very supporting". The former two characterisations need to be accompanied by a subjective measure of the probability that the envisaged conflict will occur during the operation of the system. Kotonya and Sommerville (1996) also expect the stakeholders to identify conflicts between requirements for system services in their VORD method. These conflicts may be related to constraints associated with the provision of these services.

Approach	Main Assumptions	Positive Features	Limitations
Logic-based	• models expressed in some formal language	• well-defined inconsistency detection procedures with sound semantics • applicable to arbitrary consistency rules	• first-order logic is semi-decidable • theorem proving is computationally inefficient
Model checking	• it must be possible to express or translate models in the particular state-oriented language used by the model checker	• well-defined inconsistency detection procedures with sound semantics	• not efficient due to explosion of states • only specific kinds of consistency rules (e.g. reachability of states) can be checked
Special Forms of Analysis	• models need to be expressed in a specific common language (e.g. conceptual graphs, UML, Petri Nets, XML) or be translated into it	• well-defined inconsistency detection procedures	• only specific kinds of consistency rules can be checked
Human-based collaborative exploration	• models (or parts of models) expressed in informal modelling languages	• only method for informal models	• labour intensive and difficult to use with large models

Table 2: Summary of Assumptions, Positive Features and Limitations of Different Approaches to the Detection of Inconsistencies

The detection of inconsistencies may also be left to the stakeholders in the techniques developed by van Lamsweerde and his colleagues (van Lamsweerde et al., 1998; van Lamsweerde and Letelier, 2000). Note that although these techniques have been developed to detect inconsistencies between formal models, their developers realise that in large models the detection of inconsistencies using their goal-regression method and their divergence or obstacles patterns may turn out to be inefficient. To address this problem, van Lamsweerde and his colleagues have developed a number of heuristics that could be used by the stakeholders to explore the possibility of divergences and obstacles (see Section 4.1) in connection with certain types of goals. Thus, for example, in the case of a goal regarding the confidentiality of a chunk of information and a second goal regarding the provision of some information the stakeholders are advised to check if the particular pieces of information which are referenced by the two goals are the same. If they are, the goals are divergent.

4.5 Summary

A summary of the assumptions, and the main positive features and limitations of the various approaches to the detection of inconsistencies is given in Table 2.

5. Diagnosis of Inconsistencies

In Section 2.2 we described the diagnosis of inconsistencies as an activity whose objective is to establish the source, cause and impact of an inconsistency. Most of the techniques and methods for inconsistency management provide little or no support for this activity. Notable exceptions to this is the work of Hunter and Nuseibeh (1998) on QC-logic and the significance diagnosis framework of Spanoudakis and Kassis (2000). The former provides a mechanism for identifying the source of inconsistencies detected in formal software models and the latter provides a configurable framework for assessing the impact of inconsistencies detected in UML models. The DealScribe system developed by Robinson and Pawlowski (1998) and the VORD method (Kotonya and Sommerville, 1996) also provide schemes for making a quantitative assessment of the impact of inconsistencies detected in goal-oriented requirements models.

As part of their work on using QC-Logic to support reasoning from software models in the presence of inconsistencies, Hunter and Nuseibeh (1998) have also addressed the problem of identifying the "possible sources" of an inconsistency. In their work, this source is defined in terms of the set Δ of the original formulas in the software model(s) and the set P of the formulas used in the proof of the inconsistency (i.e. the derivation of the empty clause ($\perp$) from the formulas in Δ). More specifically, a possible source of an inconsistency is defined as any subset S of P whose formulas belong to Δ and for which the set of formulas ($\Delta \cap$ P) – S is a set of consistent formulas[6]. As an example (the example is taken from Hunter and Nuseibeh (1998)) consider the following set of labeled formulas Δ:

$\{a\}: \alpha \qquad \{b\}: \neg\alpha \vee \neg\beta \qquad \{c\}: \beta \qquad \{d\}: \gamma$

The empty clause can be obtained from the above formulas by resolving the formula labeled by {a} with the formula labeled by {b} to get the formula {a,b}[7]: $\neg\beta$ and then the formula labeled by {a,b} with the formula labeled by {c} to get the formula {a,b,c}: $\perp$. In the above example, Δ = {a,b,c,d}, P = {a,b,c} and the subsets of P which belong to Δ are {a}, {b}, {c}. The complements of these subsets with respect to ($\Delta \cap$ P) are {b,c}, {a,c}, and {a,b}, respectively. All these complement sets are consistent. Therefore the possible sources of the inconsistency are the formulas {a}, {b} and {c}.

Hunter and Nuseibeh (1998) suggest that some ordering of the formulas in the original models (i.e., the set Δ) could be used to order the different possible sources and thus to identify the most likely resource of an inconsistency. The ordering of the formulas in Δ may be one that reflects the belief of the stakeholders in the validity of the formulas in this set (Hunter & Nuseibeh 1998). They also suggest that the labels may be used so as to identify the stakeholders who provided the formulas. If that is so then the identification of the possible

[6] The definition of a possible source given in this paper is a simplification of the definition of a possible source given in (Hunter and Nuseibeh 1998) which is equivalent to it when the formulas in Δ have single labels.

[7] Recall from Section 3.1 that Hunter and Nuseibeh (1998) keep track of the formulas used in a proof by labeling formulas and taking as the label of a formula derived by the application of the resolution rule of inference the union of the labels of the formulas which were resolved.

source could also identify the stakeholders involved in the inconsistency. These stakeholders can subsequently be consulted to check whether there is a deeper conflict underpinning the inconsistency manifested in the models.

The framework of Spanoudakis and Kassis defines a set of "characteristics" that indicate the significance of the main types of elements in UML models (i.e., classes, attributes, operations, associations and messages) and incorporates belief functions which measure the extent to which it may be believed from the model that an element has a characteristic. Examples of the characteristics used in this framework are the "co-ordination capacity" of a class in a model (i.e. the capacity of a class in co-ordinating other classes in specific interactions within a system) and the "functional dominance" of a message (i.e. the ability of a message to trigger an entire interaction within a system). The framework provides a formal language (based on OCL) that can be used to specify criteria of significance and associate them with specific consistency rules. These criteria of significance are defined as logical combinations of the characteristics of the framework using a formal language called *S-expressions*. Consider, for example, a consistency rule requiring that for every message which appears in an interaction (sequence) diagram of a UML model there must be an association or an attribute defined from the class of the object that sends the message to the class of the objects that receives the message (this condition guarantees that the message can be dispatched). A criterion that could be defined and associated with this rule is that the message must have functional dominance in the interaction that it appears and the class of the object that sends it must have a co-ordinating capacity in that interaction. The framework in this case would calculate beliefs for the satisfiability of the significance criterion by the messages which breach the rule and rank the inconsistencies caused by each of these messages in descending order of the computed beliefs.

Technique	Main Assumptions	Positive Features	Limitations
QC-Logic (Hunter & Nuseibeh, 1998)	• applied to formal software models • formulas in models must be labelled	• automatic identification of possible source(s) of inconsistencies	• computationall y expensive
S-expressions (Spanoudakis & Kassis, 2000)	• models expressed in UML • stakeholders specify criteria to diagnose significance of inconsistencies	• fine-grain distinctions of significance based on reasoning with well-defined semantics • relatively inexpensive computations	• not possible to differentiate the significance of violations of different consistency rules
DealScribe (Robinson & Pawlowski, 1998)	• requirements models expressed in a goal-oriented proprietary language • based on subjective probabilities of conflicts identified by the stakeholders	• effective in rankingconflicting requirements	• scalability due to the need to provide subjective probabilities of all conflicts
VORD (Kotonya & Sommerville, 1996)	• informal requirements models • based on weights indicating the importance of requirements	• applicable to informal requirements models	• differentiation of requirements importance is not always possible

Table 3: Summary of Assumptions, Positive Features and Limitations of Inconsistency Diagnosis Techniques

Robinson and Pawlowski (1998) suggest the use of two simple measures as estimates of the impact of conflicting requirement statements, namely the requirement "contention" and "average potential conflict". As we discussed in Section 4, the root requirements statements, which can be expressed in DealScribe, are related to each other as very conflicting, conflicting, neutral, supporting and very supporting. The contention of a requirement statement is computed as the ratio of the number of the very conflicting or conflicting relations over the total number of relations that this statement has with other requirements statements. The average potential conflict of a statement is measured as the average of the subjective probabilities of conflict that have been associated with all the conflicting and very conflicting relations that have been asserted for it. Robinson and Pawlowski (1998) claim that the contention measure has been found to be very effective in ranking conflicting requirements in terms of significance and attempting their resolution in the derived order. Kotonya and Sommerville (1996) in their VORD method also expect the stakeholders to provide weights that indicate the order of importance of their requirements models. These weights are subsequently used to establish the importance of conflicts between these requirements.

A summary of the assumptions, and the main positive features and limitations of the various approaches to the diagnosis of inconsistencies is given in Table 3.

6. Handling of Inconsistencies

The handling of inconsistencies is concerned with the questions of *how* to deal with inconsistencies, *what* are the impacts and consequences of specific ways of dealing with inconsistencies, and *when* to deal with inconsistencies. Inconsistencies may be handled through actions, which may be taken under certain conditions. The actions to be taken depend on the type of an inconsistency (Nuseibeh et al., 2000). Actions can modify the models, overlap relations, or rules to restore or ameliorate inconsistencies, or notify stakeholders about inconsistencies and perform sorts of analysis that would safe further reasoning from models without however changing the models, the overlaps and the rules. Actions of the former kind are called *changing actions* and actions of the latter kind are called *non changing actions*. Changing actions can be further divided into *partial* and *full resolution* actions. The former are actions that ameliorate the inconsistencies, but do not fully resolve them. The latter are actions that resolve the inconsistencies. Both these kinds of actions can be automatically executed or executed only if selected by the stakeholders. The application of an action has a cost and certain benefits which need to be established before the action is taken, especially in cases where the stakeholders are expected to select one action from a pool of possible actions.

Many of the surveyed techniques provide some support for handling inconsistencies. Based on the type of the actions that they support these techniques may be distinguished into (a) those which support changing actions, and (b) those which support non changing actions. In the following subsections we present the ways in which these techniques deal with handling based on the above classification.

6.1 Changing Actions

Most of the techniques classified in this group use human interaction to support inconsistency handling. These techniques expect the stakeholders to define and select handling actions by evaluating the inconsistencies, and execute these actions. Examples of these techniques are described below.

Easterbrook (1991) proposed a general framework for comparing and resolving inconsistencies (viz. conflicts) to integrate software models. His method is interactive. The stakeholders get involved in the generation of solutions for handling inconsistencies after they have explored the overlaps between models and identify the inconsistencies between them. This phase is called "generative phase" in Synoptic. No guidance for generating resolutions is offered. The result of the generative phase is a list of options for resolution. The main goal in Synoptic is to identify the options that best resolve the issues related to the inconsistencies and link them together. The process of linking the options to the inconsistencies is performed by either displaying an option and asking the user to select related issues, or by displaying an issue and allowing the user to select the options. The resolution chosen is represented as a new model

In an evolving software development process, it is not possible to guarantee that an inconsistency that was resolved at a particular stage will not be re-introduced at other stages of the process. Easterbrook and Nuseibeh (1995a, 1995b) suggested an approach for handling inconsistencies in evolving specifications. The method is demonstrated through the ViewPoints framework (see Section 8) developed by (Nuseibeh & Finkelstein, 1992). The method requires the specification of various actions to change software models called ViewPoints when a specific consistency rule is violated. The set of actions is defined by the method designer, as part of the process of specifying consistency rules. Some actions will repair the inconsistency, while other actions will only ameliorate it by taking steps towards a resolution. Actions are selected by the stakeholder who owns the ViewPoint(s) that breach the rule. The actions taken are recorded.

The Oz tool (Robinson & Fickas, 1994) is able to detect and characterize inconsistencies, generate resolutions, and derive a final integrated model. The resolutions are generated using a small set of domain independent strategies, such as patching, re-planning, relaxation, reformulation, and augmentation, together with a domain specific model. The system has a component named Conflict Resolver that proposes actions which can resolve a conflict. Actions are identified using an analytic method which establishes trade-offs between weighted issues related to the inconsistency (the issues and their weights are defined by the stakeholders) or heuristics. The heuristic method uses abstraction hierarchies to find substitute or additional issues. Using a graphical interface, a stakeholder selects intermediate alternatives and automated negotiation methods. Following this idea, Robinson (1997) suggested two ways of resolving inconsistencies: structured-oriented resolution and value-oriented resolution. The former is related to the modification of the context in order to remove conflict. The latter modifies attribute values of objects in order to reduce the degree of inconsistency.

The "reconciliation" method (Spanoudakis & Finkelstein 1997) generates and proposes to the stakeholders actions that can partially resolve the different types of the modelling discrepancies (inconsistencies) which have been detected in partial object-oriented software models. The generation process is driven by the distances that the method computes between the two models. These distances indicate the type and the extent of the inconsistencies. The actions generated can resolve inconsistencies of specific types (e.g inconsistencies in the classification of model elements) and are associated with measures which indicate the extent to which that can ameliorate an inconsistency. The stakeholders are free to select or disregard the actions proposed by the method.

van Lamsweerde et al. (1998) have developed techniques and heuristics for conflict resolution by transforming software models and/or consistency rules (called "goals" in their KAOS framework) which give rise to "divergences" (see Section 4.1). Divergences in their framework can be resolved: (a) by introducing new goals, (b) by weakening (i.e. dropping conditions of) one or more of the existing goals that caused the divergence, (c) by refining goals to subgoals that are no longer divergent, or (d) by transforming goals into new goals that no longer causing a conflict. Their approach is based on divergence resolution patterns and heuristics.

Another approach has been proposed by Nuseibeh and Russo (1999) to allow inconsistency management in evolving specifications based on abduction. The approach uses abductive reasoning to identify changes that have to be executed in requirements specifications specified in QC logic (Hunter and Nuseibeh, 1998) that are inconsistent. The technique generates a list of actions to be executed in a specification in order to bring it in a consistent state. The generated actions eliminate literals from the formulas of the models. The literal to be removed can be the one causing the inconsistency, or a literal involved in the inference process that derives the literal causing the inconsistency, or both. The approach does not mention how and when to perform the elimination of literals neither proposes how to select a specific change out of all the possible changes. Providing support for the selection of the change to perform is important for large models where there are numerous alternative literals that could be removed. In such case, it would be necessary to evaluate the impact of each literal removal (Nuseibeh & Russo, 1998).

6.2 Non changing actions

Inconsistencies do not necessarily have to be resolved or ameliorated (see (Gabbay & Hunter, 1991; Gabbay & Hunter, 1993) for a similar perspective regarding inconsistencies in databases specified in temporal logic). The actions which may be taken in response to an inconsistency can be a reference to a user or an attempt to obtain more data or some sort of further analysis over the models.

This approach has been adopted by Finkelstein et al. (1994) to support inconsistency handling in the ViewPoint framework (Finkelstein et al. 1992). The approach uses meta-level axioms, which specify how to act depending on the inconsistency. The actions are expressed by action rules, which are triggered when inconsistencies are identified between two ViewPoints. These actions can invoke a truth maintenance system, require external actions such as 'getting information from a user', or invoke external tools. However, the described inconsistency handling approach has not been implemented for a distributed environment. In the approach, when an identified inconsistency is due to typographical error, the user is notified about the problem and is expected to deal with it.

Hunter and Nuseibeh (1998) suggest that in certain cases the handling that is required for inconsistencies should take the form of further analysis to identify the parts of the models that it would be safe to continue reasoning from in the presence of an inconsistency. They propose a specific form of such analysis which is based on QC logic. This analysis identifies maximally consistent subsets of formulas in inconsistent software models and qualifies inferences depending on the consistent subsets that they were made from. The result of this analysis is specified in a report, which is used to assist with the identification of possible actions that can be taken in the presence of inconsistencies.

In the approach proposed by Boehm and In (1996) the stakeholders are responsible to evaluate and select the potential attribute requirements conflicts and resolution options, identified by the QARCC tool. A similar approach is presented by Clarke et al. (1998) to support conflict resolution in UML class models, where the designer of the model is presented with a dialog displaying a conflict and the source of the conflict.

Technique	Main Assumptions	Positive Features	Limitations
Synoptic (Easterbrook, 1991)	• stakeholders are expected to define and select handling actions	• complete freedom to the stakeholders • support for informal models	• inconsistencies are handled by stakeholders • no support for generating handling actions
OZ System (Robinson & Fickas, 1994)	• use of set of domain independent strategies to generate handling actions	• identification of resolution actions	• high human interaction
Reconciliation method (Spanoudakis & Finkelstein,1997)	• use of distance metrics to indicating the type and extent of inconsistencies in proposing handling actions	• stakeholders have control of the handling activity • support for partial resolution • automatic generation of handling actions	• no support for automating application of actions
KAOS (van Lamsweerde et al., 1998)	• use of divergence resolution patterns	• stakeholders have control of the handling activity • sound semantics	• only specific kinds of divergences can be handled
ViewPoint framework (Finkelstein et al., 1994)	• stakeholders may specify the handling actions to be taken in violations of specific rules	• automatic partial of full resolution	• stakeholders not in full control of the process but may de-activate handling rules
QC-Logic (Hunter & Nuseibeh, 1998)	• analysis to identify maximally consistent parts of the models that are safe to continue reasoning	• useful analysis in cases of partial resolutions • sound semantics	• manual identification of related actions • computationally expensive

Table 4: Summary of Assumptions, Positive Features and Limitations of Techniques in Handling Inconsistencies

Another technique which handles inconsistencies through non-changing actions approach is that developed by Ellmer et al. (1999). This technique represents the existence of inconsistencies through "consistency links". Consistency links are generated after checking "consistency rules", as described in subsection 5. In order to leave the original documents as they are, the consistency links are stored in different documents. The users are expected to identify inconsistent elements in the participating documents (software models) by navigating through the consistency links. At the current stage, the approach does not suggest which actions should be taken when a consistent link of type "inconsistent" is identified.

A summary of the assumptions, and the main positive features and limitations of the various approaches to the handling of inconsistencies is given in Table 4.

7. Tracking

As outlined in Section 2, this activity is concerned with the recording of information about the consistency management process. The kinds of information which are tracked by various techniques and the structures used to do it are described below.

Most of the techniques for inconsistency management which are based on the ViewPoints framework (Nuseibeh and Finkelstein, 1992) have a structure to record information about the process. This structure is provided by what have been termed as "ViewPoint work record" which stores the trace of the development process of a ViewPoint[8]. These include the techniques by Finkelstein et al. (1994), and Easterbrook and Nuseibeh (1995a, 1995b). Easterbrook (1991) suggested also the use of a map with information about ViewPoints overlaps and conflicts, a list of options to resolve the inconsistencies, and links between the map and options.In their approach for consistency checking and conflict resolution using decentralised process models Leonhardt et al. (1995) also store the results of consistency checks in the work record slots of the ViewPoints which participate the process. Hunter and Nuseibeh (1998) proposed also the use of a "report" with the results of their inconsistency diagnosis analysis.

When checking for standards compliance (Emmerich et al., 1999) a diagnostic of non-compliant document elements and range of possible repairs that can be performed in order to guarantee the compliance is produced. This diagnostic is specified as functions and used by the engineers to assess the importance and difficulty of making the document compliant. The approach implements three such functions: LIST, which generates a list of non-compliant elements; STAT, which generates statistical analysis on non-compliant elements; and TRAV, which generates a filtered document with all non-compliant elements after traversing a document.

Technique	Main Assumptions	Positive Features	Limitations
ViewPoint framework (Finkelstein et al., 1994)	• models expressed according to instantiated ViewPoint template(s)	• automatic recording of all information related to detection and handling of inconsistencies	• strongly related to the ViewPoint development process • may generate a vast amount of information
Standards compliance (Emmerich et al, 1999)	• informal requirements models expressed in proprietary format	• generates a report of all the model elements which violate consistency rules	• does not keep track of the entire inconsistency management process
XML technique (Ellmer et al., 1999)	• models expressed in or translated to XML	• use of special XML documents and hyperlinks	• requires knowledge of XML
DealScribe (Robinson & Pawlowski, 1998)	• goal-oriented requirements models expressed in proprietary format	• automatic check of the history of models compliance	
QARCC (Boehm & In, 1996)	• informal models	• provides a structure for recording conflicts, their handling and the alternative handling options considered	• the record has to be created manually by the stakeholders
Reconciliation method (Spanoudakis & Finkelstein, 1997)	• models expressed in UML	• keeps a record of the enactment of the built-in model of the process of reconciliation	• may generate a vast amount of information

[8] The inconsistency management process is seen as part of the overall ViewPoint development process in this approach as we discuss in Section 8.

Table 5: Summary of Assumptions, Positive Features and Limitations of Techniques w.r.t Tracking

The approach proposed by Ellmer et al. (1999) keeps information about consistency management in various XML documents. This approach uses different XML documents for the consistency rules and for the consistency links (Zisman et al., 2000).

One of the goals of the requirements dialog meta-model (DMM) in the DealScribe tool (Robison & Pawlowski, 1999) is to allow tracking and reporting on development issues. DealScribe is composed of two tools: HyperNews and ConceptBase. The former has a World Wide Web interface and allows a discussion system like Usenet News. The former is a deductive database that defines the dialog meta-model and stores the dialog history as instances of DMM. The tool contains a dialog goal model composed of set of goals specified by logical conditions. The dialog goal model is used to automatically check the history for compliance.

Tracking is also supported by the "reconciliation" method proposed by Spanoudakis and Finkelstein (1996, 1997). In their approach, it is possible to keep a trace of the enactment of the software process model that their method incorporates to describe the activity of reconciling models. This process model provides a structure for holding information about overlap relations, inconsistencies, and actions taken by the stakeholders to resolve these inconsistencies. Boehm and In (1996) also use a specific structure for storing information about inconsistencies (called "issues"), possible ways of resolving them (called "options") and the decisions made about their resolutions (called "agreements").

A summary of the assumptions, and the main positive features and limitations of the various techniques in keeping track of the inconsistency management process is given in Table 5.

8. Specification and Application of Inconsistency Management Policies

In Section 2.2, we argued about the need to have an inconsistency management policy that specifies the techniques that could be used to carry out the activities of overlap and inconsistency detection, diagnosing and assessing the impact of inconsistencies, and handling inconsistencies. The variety of the possible ways which may be used to carry out each of these inconsistency management activities that were presented in Sections 4-7 must have reinforced this point. Each of these ways is applicable under certain conditions, works best in settings which satisfy its own assumptions and has its own advantages and disadvantages. Thus, the stakeholders are faced with numerous options to select from and need guidance for making mutually coherent decisions which serve best the objectives of the overall process.

In this section, we discuss the support that is available for the specification and monitoring of an inconsistency management policy by the methods and techniques that we have surveyed. More specifically, we examine whether the reviewed techniques and methods:

(i) have an implicit or explicit inconsistency management process specification and in the latter case how this process is specified
(ii) allow the modification of their inconsistency management process by the stakeholders
(iii) enforce their inconsistency management processes or guide the stakeholders in enacting them
(iv) embed their inconsistency management process within a broader software development process or method

In the techniques built upon or derived from the ViewPoints[9] framework (Nuseibeh et al., 1994; Easterbrook et al., 1994; Finkelstein et al., 1994; Easterbrook & Nuseibeh, 1995), the inconsistency management process is seen as part of the overall ViewPoints development process. This process is expressed in process rules of the form [<situation> <action>]. The meaning of these rules is that the <action> may be taken if the <situation> is satisfied. Process rules of this kind are generally used to define ViewPoints development actions and the situations under which these actions can be applied (Nuseibeh et al., 1996). In the case of

[9] A ViewPoint is defined (Nuseibeh et al., 1994) as a locally managed software model that can be created by instantiating a template which encapsulates the specification of the language in which its instances are described (called "style") and the process by which these instances can be developed (called "work plan"). Each ViewPoint also keeps a record of the actions that have been applied to it or taken in connection with it in its development history (called "work record").

inconsistency management, rules of the same kind may be used to define: (a) when the consistency rules should be checked (the <action> in this case becomes the consistency rule to check and the <situation> defines the conditions under which the check should be triggered); or (b) the actions that may be executed when a consistency rule is violated (the <situation> in this case refers to event of having detected the violation of the rule). The action in the latter case may be a handling action or an inconsistency diagnosis action or an action that assesses the impact of an inconsistency.

Leonhard et al (1995) describe a scheme that supports a decentralised enactment of inconsistency management process models of the ViewPoints framework. In their operationalisation of the approach, a <situation> is defined by a regular expession which refers to events in the development history of a software model that is encapsulated within a ViewPoint. An <action> in this case may be an "execute", "display" or "recommend" action. "Execute" actions are used by a model which wishes to check a consistency rule to request from another model (also encapsulated in a ViewPoint) whose parts are referenced by the consistency rule to supply information about these parts which is required for the execution of the check. The latter model depending on the existence of a process rule that would enable it to recognise and act upon the request event may accept the request and send the necessary information or reject it. If it accepts the model that issued the original request checks the rule, informs its collaborating model about the result of the check, and finally appends this result to its work record. This model of enacting the inconsistency management process clearly requires some compatibility between the process models held in the different software models. Leonhard et al (1995) admit this requirement but offer no general advise on how to construct compliant local process models that could lead to effectively co-ordinating software models.

The method developed by Emmerich et al (1999) advocates a similar approach to that taken in the ViewPoints framework. This method has a number of explicit and modifiable inconsistency management process models (called "policies") which cover the activities of inconsistency detection, diagnosis, significance assessment and handling. The check of groups of consistency rules in policies is triggered by the recognition of events which occur in the models referenced by the rules. These events are specified by the policy in terms of actions that can be performed in models. An event occurs when there is an attempt to perform the action(s) that the event references in a specific model. Events are specified using the event specification language of FLEA which is also used to monitor the events. When the check of a consistency rule fails a "diagnostic" which is associated with it in a policy is executed. A diagnostic is a script that produces information that could help the stakeholders to handle the inconsistencies. This information may refer to the source of the inconsistency, the significance of it or the actions that could be taken to resolve it. Policies in this method may be executed in three different modes, namely the "error", "warning" and "guideline" mode. When executed in the "error" mode the policy is enforced, that is the consistency rules are automatically checked and the model changing actions that triggered the check are aborted if they would breach the rules. In the "warning" mode, the checks are again automatically triggered but the stakeholders are allowed to perform the actions that triggered the checks even if the rules would be violated. Finally, in the "guideline" mode the stakeholders are advised but not forced to execute a check if the events that should trigger it have occurred. The inconsistency management process of this method is not embedded within a broader software development process or method.

DealScribe (Robinson & Pawlowski, 1998) has an explicit and modifiable inconsistency management process which covers the activities of inconsistency detection and handling. The stakeholders can define process compliance development rules (called goals), the operations that should be executed to monitor them, the operations that should be executed to handle their violations, and the conditions under which each of these operations should be invoked. These goals and operations are defined as O-Telos objects and the conditions as O-Telos constraints (see Section 5.1). The stakeholders can instruct DealScribe to create and start monitors that check the goals and handle their violations as defined by the operations associated with them. Once a monitor is started it acts according to the operations defined for the goal it is meant to monitor and the stakeholders cannot intervene in this process. However, they may at any point instruct the system to stop any of the created monitors (by issuing stop monitor statement). The inconsistency management process of DealScribe is not embedded within a broader software development process or method.

The "reconciliation" method (Spanoudakis & Finkelstein, 1998) also has an explicit, non-modifiable model of its process of inconsistency management specified using the contextual process modelling approach described in (Pohl, 1996). This process model describes the ways of carrying out the activities of overlap and inconsistency detection, and the activity of inconsistency handling. The enactment of the process is performed by a process enactment engine which monitors the state of the inconsistency management process and proposes to the stakeholders the actions that they are allowed to take at the current state of the process. The stakeholders have the right to reject all the suggested actions and take a different action. The inconsistency management process of "reconciliation" is not embedded within a broader software development process or method.

Oz (Robinson & Fickas, 1994) and QARCC (Boehm and In, 1996) have implicit models of their inconsistency management processes which cover the activities of overlap and inconsistency detection and inconsistency handling. None of these models is enforced. The inconsistency management process in QARCC is part of the WinWin spiral model of software development (Boehm & Ross, 1989). van Lamsweerde and his colleagues (van Lamsweerde et al., 1998; van Lamsweerde and Letelier, 2000) suggest that their divergence and obstacle management techniques should be used as part of the goal elaboration phase of their goal-driven requirements engineering process but provide no explicit fine-grain process model for the use of these techniques.

The rest of the techniques and methods that we surveyed do not have an explicit model of the inconsistency management process. In (Clarke et al., 1998) there is an acknowledgement of the need to describe the overlap detection and the inconsistency handling activity through a set of explicitly defined overlap detection and inconsistency resolution rules but this is left to future work.

Technique	Main Assumptions	Positive Features	Limitations
ViewPoint framework (Finkelstein et al., 1994; Leonhard et al., 1995)	• the inconsistency management process is part of the overall ViewPoint development process • the process is specified by rules of the form *if <situation> Then <action>*	• explicit and modifiable model of the process • decentralized enactment of the process	• requires compatibility between the process models held in different software models
Standards compliance (Emmerich, 1999)	• use of explicit process models called "policies"	• explicit and modifiable model of the process • covers inconsistency detection, diagnosis, and handling	• process not embedded within a broader software development process or method
DealScribe (Robinson & Pawlowski, 1998)	• the stakeholders can define goals, operations to monitor the goals, operations to handle violations, and conditions to invoke the operations	• explicit and modifiable model of the process • covers detection and handling inconsistencies	• process not embedded within a broader software development process or method
Reconciliation method (Spanoudakis & Finkelstein, 1997)	• explicit specification of the inconsistency management process using the "contextual" software process modelling approach • execution and tracking of the process through a process enactment engine	• explicit model of the process which is not enforced • covers overlap identification, detection diagnosis, and handling inconsistencies	• process not embedded within a broader software development process or method
OZ System (Robinson & Fickas, 1994)	• use of implicit process	• the process is not enforced • covers overlap identification, inconsistency detection and handling	• implicit model of the process • process not embedded within a broader software development process or method
QARCC (Boehm and In, 1996)	• use of implicit process	• the process is not enforced • the process is part of the spiral WinWin model of software development	• implicit model

Table 6: Summary of assumptions, positive features and limitations of techniques w.r.t process specification and application

A summary of the assumptions, and the main positive features and limitations of the various techniques in specifying and applying an inconsistency management process is given in Table 6.

9. Open Research Issues

As presented in this survey, in the last decade there has been a substantial body of research that has tackled a number of significant problems and issues, which arise in the context of managing inconsistencies in software models. A summary of the techniques discussed in this survey is shown in the Tables 7a and 7b. In this table, the rows represent the techniques, the columns represent the different activities, and the cells summarise the support level of each of these techniques for the respective activity. An empty cell indicates that a technique does not support the relevant activity.

Despite the considerable work that has been concerned with the problem of inconsistency management in software models, important issues need to be studied further and addressed in all the six individual activities of the inconsistency management process. These issues are discussed in the following subsections.

9.1 Open issues in the detection of overlaps

In our view, there are issues, which deserve further research in the detection, synthesis, representation and management of overlaps.

The detection of overlap is complicated by software models expressed in different languages; being at different levels of abstraction, granularity and formality, and deploying different terminologies. Stakeholders certainly cannot be expected to carry this activity without support and representation conventions fail when models are constructed independently. Our work on similarity analysis (Spanoudakis & Constantopoulos, 1995; Spanoudakis & Finkelstein, 1997, Spanoudakis & Finkelstein, 1998) clearly reflects our assumption that effective overlap identification is possible by developing specialised methods or algorithms for software models expressed in specific modelling languages. Forms of automated analysis can be used as "quick and dirty" identification mechanisms whose results should then be confirmed by human inspection. For large models, this inspection won't be feasible unless it is focused on specific sets of overlap relations. The identification of these sets is an open research issue. Possibilities that may be exlored in this direction include the inspection of overlap relations which have given rise to significant inconsistencies and/or overlap relations which are referenced by consistency rules expressing critical properties of a system (e.g. safety critical properties, reliability properties).

We also envisage that in many settings a combination of different overlap identification techniques will be required, and overlap relations identified by different agents will need to be synthesised in an attempt to reduce the cost and complexity of overlap identification. For instance, in cases where a shared ontology has been evidently used in a coherent way for some time, it makes sense to use it for identifying coarse-grain overlaps and then refine them using some other method (e.g. specification matching or inspection). In cases where overlap relations have been identified by different agents but not jointly, it is necessary to be able to check if these relations are consistent, and to synthesise them in a coherent and systematic way prior to informing consistency checking. It is also necessary to be able to assess our confidence in the subsequent consistency checking.

Finally, to facilitate consistency checking with respect to different sets of overlap relations, these relations need to be represented in forms suitable for consistency checking and separately from the software models that they relate (Spanoudakis et al., 1999). Easterbrook et al. (1998) have suggested the use of category theory as a means of representing and reasoning about overlaps between models. This is certainly a direction worth pursuing especially in cases where the overlap relations connect models specified in different languages.

However, representing overlaps separately from the models they relate raises interesting questions about what happens to these relations when the models change, and especially in cases where the agents that identified these relations are not those who made the changes.

Technique	Overlap detection	Inconsistency detection	Diagnosis	Handling	Tracking	Policy
Model checking (Heitmeyer et al., 1995; Chan et al, 1998)	elements with the same name	use of specialized model checking algorithms				
Specialized techniques (Clarke et al., 1998)	elements with the same name	checking of specific rules concerned with UML class models		conflict resolution in UML class models		
OZ System (Robinson, 1994; Robinson & Fickas, 1994)	elements associated with the same goal in a domain model	checks for identical goals in different stakeholder perspectives		use of domain independent strategies (e.g. patching, relaxation)		covers the activities of overlap and inconsistency detection and handling
QARCC (Boehm & In, 1996)	based on common 'Win' conditions	responsibility of stakeholders analysis and explanation of the nature of a conflict		generation of resolution options; guidance in selecting from them		
Synoptic (Easterbrook, 1991)	human inspection (visual aids are provided)	responsibility of stakeholders		generation of handling options by the stakeholders		
Reconciliation method (Spanoudakis & Finkelstein, 1997)	based on the weighted bipartite graph matching	use of distance functions to detect modelling discrepancies		generation of resolution options; guidance in selecting from them	tracking of the enactment of the process model of the method	explicit model covering the activities of overlap and inconsistency detection and handling
ViewPoint framework (Finkelstein et al., 1994)	unification	based on theorem proving		automatic handling based on action rules	tracking of rules checked, and executed handling actions	covers the activities of inconsistency detection and handling
Delugach, 1992	human inspection	based on comparison of conceptual graphs				
QC-Logic (Hunter & Nuseibeh, 1998)	unification	theorem proving (special set of inference rules)	detection of source of inconsistencies	detection of consistent subsets of models		

Table 7a: Summary of Inconsistency Management Techniques – Part 1

Technique	Overlap detection	Inconsistency detection	Diagnosis	Handling	Tracking	Policy
Nissen et al. (1996)	unification	checking of integrity constraints				
XML technique (Ellmer et al., 1999)	matching similar to unification	special algorithm for checking rules expressed in XML syntax		navigation through "consistency links"	use of special XML documents and hyperlinks	
S-expressions (Spanoudakis & Kassis, 2000)	matching similar to unification	check of OCL rules against UML models	based on criteria formally defined by the stakeholders			
Standard compliance (Emmerich, 1999)	matching similar to standard unification	special algorithm for checking rules expressed in proprietary format	diagnostics of compliance			covers the activities of inconsistency detection, diagnosis, and handling
KAOS (van Lamsweede et al., 1998)	unification	derivation of boundary conditions by goal regression use of patterns of divergence		resolution based on patterns and heuristics		
DealScribe (Robinson & Pawlowski, 1998)	unification	stakeholders identify conflicts between "root requirements" goal monitoring	based on "contention" and "average potential conflict"		use of HyperNews and ConceptBase	covers the activities of inconsistency detection and handling
VORD (Kotonya & Sommerville, 1996)		by the stakeholders	based on measures of importance provided by the stakeholders			
Nuseibeh & Russo (1999)				generation of handling actions based on abduction		

Table 7b: Summary of Inconsistency Management Techniques – Part 2

9.2 Open research issues in the detection of inconsistencies

The most important open research issues regarding the detection inconsistencies are related to the efficiency and scalability of this process. The detection has to be efficient and scalable because in real projects it will have to deal with a large number of complex software models and many consistency rules. None of the current approaches to detection including theorem proving, model checking, special forms of automated analysis and collaborative exploration can really claim that it sufficiently addresses these issues of scalability and efficiency as far as we are concerned. One possible way of tackling this problem is to try to reduce the original models into versions that could be safely used to reason about specific consistency rules. This approach of model reduction has been explored in the context of techniques which employ model checking algorithms and has been termed as "abstraction" (Bharadwaj & Heitmeyer, 1999). The most difficult aspect of model reduction is the ability to establish that the reduced model is equivalent to the original one with regards to the inferences that may be made from it in connection with a specific consistency rule.

Efficiency and scalability become even more important when inconsistencies need to be detected in software models which evolve. Obvioulsy, changes in software models may make it necessary to go through another inconsistency detection cycle. In such circumstances, it is important to be able to identify the parts of the models that have been changed and the rules that refer directly or indirectly to these parts in order to avoid unnecessary checks. This is particularly crucial for techniques which detect inconsistencies based on inspections by the stakeholders.

9.3 Open research issues in diagnosis

It must have been clear from the discussion in Sections 5 and 6 that although diagnostic information about the source, the cause and the impact of an inconsistency is essential for deciding how and when to handle inconsistencies the support which is available for this activity is limited.

The approach taken by (Hunter and Nueseibeh, 1998) regarding the identification of the source of an inconsistency is interesting but computationally expensive (since all the consistent subsets of the model formulas that have been used in the derivation of an inconsistency need to be computed). To this end, the developers of this technique have suggested that diagnosis should try to identify the most likely source(s) of an inconsistency rather than all the possible sources (Hunter and Nuseibeh, 1998). We fully agree with them that this is a direction worth pursuing. Possible issues to explore in this direction include the use of schemes for reasoning on the basis of: (a) beliefs that the stakeholders may have about the validity of the information provided by the different formulas (model elements), and/or (b) beliefs about the validity of a certain formula computed from the use of the formula in other forms of reasoning with the model.

The approach taken by Spanoudakis and Kassis (2000) for the assessment of an impact of an inconsistency is also a direction that could be pursued for other non object-oriented modelling schemes. In that case, however, the stakeholders would need concrete guidelines about the sorts of the significance criteria that they should associate with specific consistency rules. Research in this direction should also explore whether or not the significance predictions that may be made on the basis of evolving software models are confirmed in later stages of the software development life-cycle.

Finally, substantial research is required to support the stakeholders in establishing the cause of an inconsistency. This research could certainly benefit from work on requirements traceability which provides schemes for annotating model elements with the stakeholders who contributed to their construction in different capacities (Gotel & Finkelstein, 1994; 1995). Schemes of this sort could be used to locate the stakeholders who have constructed the relevant model elements or whose perspectives are expressed in them. However, this would be only one step towards the objective which is to identify if there is a deeper conflict between the model contributors that has been manifested as an inconsistency in the models. Any work that would attempt to make the next step in this direction would have to start from an empirical investigation of the circumstances under which deeper stakeholder conflicts are manifested as inconsistencies between models (see for example (Boehm & Egyed, 1998)).

9.4 Open research issues in handling

Most of the approaches proposed for handling inconsistencies assume that stakeholders will be responsible for selecting a handling action without however supporting them to make this selection. In the majority of the cases the information presented to the stakeholders is a list of possible actions to be taken. Normally, the stakeholders select an action from the list intuitively, based on their experience and knowledge about the system being developed. This lack of support becomes a problem in cases where the stakeholders are given a large number of actions to select form.

As we argued in Section 2.2 a selection should be based on the cost, risk and benefit of applying an action and therefore it is essential to develop ways of evaluating or measuring these factors. This evaluation relates to the diagnosis of the inconsistencies but even those techniques which support diagnosis do not relate their findings with the above selection factors. The estimation of the risk and benefits associated with an action can also be based on an analysis of the effects that the application of an action may have on the satisfiability of consistency rules other than those that the action is meant to satisfy. In handling, it is also necessary to identify the optimal time to perform the various actions associated to the different types of inconsistencies.

Another issue that needs further investigation in handling is related to the languages that may be used to specify actions (Hunter & Nuseibeh, 1998). Existing techniques seem to be using schemes with not well-defined semantics to specify actions that are to be performed on models which are expressed in informal languages and actions that do not change the models. Finally, handling has to address the issue of efficient generation of actions.

9.5 Open research issues in tracking

Only few of the approaches for managing inconsistencies support tracking of the whole process. The tracking of information is a very important activity for the management process and should be performed for the activities of overlap and inconsistency detection, inconsistency diagnosis, and handling. It is also necessary to have efficient mechanisms for managing the vast amount of information collected during the process.

Another important issue in tracking relates to the ability to represent the context in which certain actions were taken in the process. Context information is important for the stakeholders and for more advanced forms of analysis connected to certain activities of the process such as the handling of inconsistencies.

9.6 Open research issues in the specification and monitoring of the inconsistency management process

The research in the area of software process modelling has generated many alternative schemes that can be used to specify and enact processes for inconsistency management (see (Pohl, 1996) for an overview). Some of these schemes, as we discussed in Section 8, have already been successfully used by inconsistency management techniques to represent their underpinning processes (e.g. (Pohl, 1996)). Other techniques have opted for more light-weight schemes (Leonhard et al (1995)). Overall, the way of representing processes in this area seems to be a relatively well-understood issue with the exception of the schemes for defining inconsistency handling actions which do not change the models (see Section 9.4 above).

In our view, the two most important issues which have to be investigated for this activity include: (a) the relationship between the inconsistency management process with the more general software development process followed in a project, and (b) the construction of efficient process monitoring mechanisms. A process for inconsistency management can be effective and relatively easy to follow only if it is properly embedded in a general software development process. A deep understanding of the roles of the participants, procedures, phases and milestones used by the general software development process in a project is required before trying to embed an inconsistency management process into it. Existing research on inconsistency management appears to have neglected this issue.

The second issue relates to the way that an inconsistency management process is being monitored. It seems to us that this monitoring needs to be both decentralised and centralised at different stages of the software development life-cycle. For instance, decentralised monitoring will be probably required when the stakeholders work on their models independently and centralised monitoring will be required during group activities whose objective will be to establish overlaps and/or handle inconsistencies. In the former case, the construction of compliant distributed local inconsistency management process models is necessary but not well-understood (Leonhardt et al., 1995).

10. Conclusions

In this paper, we have presented a survey of the research work that has been conducted to deal with the problem of managing inconsistencies in software models. This survey has been presented according to a conceptual framework that views inconsistency management as a process which incorporates activities for detecting overlaps and inconsistencies between software models, diagnosing and handling inconsistencies, tracking the information generated along the way, and specifying and monitoring the exact way of carrying out each of these activities.

We have tried to survey all the research work that has been brought to our attention, identify which of the issues that arise in the process of managing inconsistencies have been addressed by this work, which are the main contributions that have been made, and what are the issues which seem to be open to further research for the time being. We have attempted to be unbiased in our discussion but have probably failed! Thus, the readers of this paper are certainly advised to study the literature on the subject themselves using this survey only as roadmap to this literature if they wish.

Our overall conclusion from this survey is that the management of inconsistency in software models has been an active and still rapidly evolving field in the more general area of software engineering. Existing research has certainly made significant contributions to the clarification of the issues which arise in this field and has delivered techniques which address a considerable number of these issues. However, it has to be appreciated that managing inconsistency is by no means an easy problem. This is because it needs to deal with models expressed in different modelling languages (ranging from completely informal to formal), constructed through different software development processes, and developed to describe systems in different domains. Most of the techniques that have been developed are aimed at dealing only with particular manifestations of this problem and do not address all the issues of it. To this end, we believe that more research is required. The arguments underpinning this claim are those that we discussed in the section on the open research issues.

Acknowledgements

The research that led to this survey article has been partially funded by the British Engineering and Physical Sciences Research Council (EPRSC) under the grant IMOOSD (GR/M57422). We also wish to thank Dr. Stephen Morris whose comments have helped us to improve this paper both in content and presentation.

References

Ainsworth M., Riddle S., Wallis P., 1996. "Formal Validation of Viewpoint Specifications", IEE Software Engineering Journal, Vol. 11, No. 1, pp. 58-66.

Besnard P., Hunter A., 1995. "Quasi-Classical Logic: Non-trivialisable reasoning from inconsistent information", In Symbolic and Quantitative Approaches to Uncertainty, (eds) Froidevaux C. and Kohlas J., LNCS 946, Springer-Verlag, pp. 44-51.

Bharadwaj R., Heitmeyer C., 1999. "Model Checking Complete Requirements Specifications Using Abstraction", Automated Software Engineering, 6, pp. 37-68.

Boehm B., Bose P., Horowitz E. & Lee M.J., 1995. "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach", Proceedings of the 17th International Conference on Software Engineering, IEEE CS Press, Los Alamitos, California, pp. 243-253.

Boehm B., Egyed A., 1998. "Software Requirements Negotiation: Some Lessons Learned", Proceedings of the 20th International Conference on Software Engineering.

Boehm B., In H., 1996. "Identifying Quality Requirements Conflicts", IEEE Software, March 1996, pp. 25-35.

Boehm B., Ross R., 1989. "Theory W Software Project Management: Principles and Example", IEEE Transactions on Software Engineering, pp. 902-916.

Boiten, E., Derrick, J., Bowman, H., and Steen, M., 1999. "Constructive Consistency Checking for Partial Specification in Z". Science of Computer Programming, 35(1), September, pp. 29-75.

Bowman H., Derrick J., Linington P. & Steen M., 1996. "Cross-viewpoint Consistency in Open Distributed Processing", IEE Software Engineering Journal, Vol. 11, No. 11, pp. 44-57.

Bray, T., Paoli, J., and Sperberg-McQueen, C.M., 1998. "Extensible Markup Language (XML) 1.0 Specification". http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium.

Chan W. et al., 1998. "Model checking large software specifications", IEEE Transactions on Software Engineering, Vol 24, No. 7, pp. 498-520.

Clarke S., Murphy J., Roantree M., "Composition of UML Design Models: A Tool to Support the Resolution of Conflicts", Proceedings of the Int. Conference on Object-Oriented Information Systems, 1998, Springer Verlang, pp. 464-479

Constantopoulos P., Jarke M., Mylopoulos J. and Vassiliou Y., "The Software Information Base: A Server for Reuse" , VLDB Journal, Vol. 4, No. 1, Jan. 1995, pp. 1-43.

Cugola, G., Di Nitto, E., Fuggetta, A. & Ghezzi, C., 1996. "A Framework for Formalizing Inconsistencies and Deviations in Human-centered Systems" ACM Transactions on Software Engineering and Methodology (to appear).

Curtis B., Krasner H. & Iscoe N., 1988. "A Field Study of the Software Design Process for Large Systems", Communications of the ACM, Vol. 31, No. 11, pp. 1268-1287.

Delugach H., 1992. "Analyzing Multiple Views Of Software Requirements", in Conceptual Structures: Current Research and Practice, P. Eklund, T. Nagle, J. Nagle and L. Gerholz, eds., Ellis Horwood, New York, pp. 391-410, 1992.

DeRose, S., Maler, E., Orchard, D., and Trafford, B., 2000. "XML Linking Language (XLink)". Working Draft http://www.w3.org/TR/ WD-xlink- 20000119, World Wide Web Consortium.

Easterbrook S., 1991. "Handling Conflict between Domain Descriptions with Computer-Supported Negotiation", Knowledge Acquisition, 3, pp. 255-289.

Easterbrook S., 1993. "A Survey of Empirical Studies of Conflict", CSCW: Cooperation or Conflict ?, Easterbrook S. (ed), Springer-Verlag, pp. 1-68.

Easterbrook S., 1997. "Model Checking Software Specifications: An Experience Report", NASA/WVU Software Research Laboratory (available from http://research.ivv.nasa.gov)

Easterbrook S., Callahan J., and Wiels V., 1998. "V & V Through Inconsistency Tracking and Analysis", Proceedings of International Woskshop on Software Specification and Design, Kyoto, Japan.

Easterbrook S., Finkelstein A., Kramer J. and Nuseibeh B., 1994. "Co-Ordinating Distributed ViewPoints: the anatomy of a consistency check", International Journal on Concurrent Engineering: Research & Applications, 2,3, CERA Institute, USA, pp. 209-222.

Easterbrook S., Nuseibeh B., 1995a. "Using ViewPoints for Inconsistency Management", IEE Software Engineering Journal.

Easterbrook S., Nuseibeh B., 1995b. "Managing Inconsistencies in an Evolving Specification", Proceedings of the 2nd International Symposium on Requirements Engineering, IEEE Press, pp. 48-55.

Egyed, A., Boehm, B. 2000, "Comparing Software System Requirements Negotiation Patterns", Journal for Systems Engineering (to appear).

Ellmer, E., Emmerich, W., Finkelstein, A., Smolko, D., and Zisman, A., 1999. "Consistency Management of Distributed Document using XML and Related Technologies", UCL-CS Research Note 99/94, (submitted for publication).

Emmerich, W., 1996. "An Architecture for Viewpoint Environments based on OMG/CORBA" Joint Proceedings of the Sigsoft '96 Workshops – Viewpoints '96, ACM Press, pp. 207-211.

Emmerich W., Finkelstein, F. and Montangero, C., Antonelli, S., Armitage, S., 1999. "Managing Standards Compliance". IEEE Transactions on Software Engineering, 25, 6, pp.

Fiadeiro, J. and Maibaum, T., 1995. "Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality", Proceedings of the Symposium on the Foundations of Software Engineering (FSE 95), ACM Press, pp. 1-8.

Finkelstein, A., Spanoudakis, G. & Till D., 1996. "Managing Interference" Joint Proceedings of the Sigsoft '96 Workshops – Viewpoints '96, ACM Press, pp. 172-174.

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B., 1994. "Inconsistency Handling In Multi-Perspective Specifications" IEEE Transactions on Software Engineering, 20, 8, pp. 569-578.

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L. & Goedicke, M., 1992. "Viewpoints: a framework for integrating multiple perspectives in system development" International Journal of Software Engineering and Knowledge Engineering, 2, 1, pp. 31-58.

Finkelstein, A. & Sommerville I., 1996. "The Viewpoints FAQ", Software Engineering Journal, Vol. 11, No. 1, pp. 2-4.

Gabbay, D. and Hunter, A., 1991. "Making Inconsistency Respectable 1: A Logical framework for inconsistency in reasoning". In Foundations of Artificial Intelligence Research. Lecture Notes in Computer Science, LNCS 535, pages 19-32, Springer Verlag.

Gabbay, D. and Hunter, A., 1993. "Making Inconsistency Respectable: Part 2 - Meta-level handling of inconsistency". In *Symbolic and Qualitative Approaches to Reasoning and Uncertainty* (ECSQARU'93), LNCS 746, Springer-Verlag, pp. 129-136

Goguen J., Ginali S., 1978. "A Categorical Approach to General Systems Theory", In Applied General Systems Research, (ed) G. Klir, Plenum, pp. 257-270.

Gruber T., 1993. "Towards Principles for the Design of Shared Ontologies Used for Knowledge Sharing", Proceedings of International Workshop on Formal Ontology, Padova, Italy (also available as Technical Report KSL 93-04, Knowledge Systems Laboratory, Stanford University).

Guarino, N., 1994. "The Ontological Level", In Philosophy and the Cognitive Sciences, (eds) R. Casati, B. Smith and G. White, Holder-Pichler-Tempsky, Vienna.

Glinz M., 1995. "An Integrated Formal Model of Scenarios Based on Statecharts", Proceedings of the 5th European Software Engineering Conference (ESEC '95), LNCS 989, Springer-Verlag, pp.254-271.

Harel D., 1987. "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, Vol. 8, pp. 231-274.

Harrison W., Kilov H., Ossher H., Simmonds I., 1996. "From dynamic supertypes to subjects: A natural way to specify and develop systems", IBM Systems Journal, Vol. 35, No. 2, pp. 244-256

Heimdahl M.P.E, Leveson N., 1996. "Completeness and Consistency in Hierarchical State-Based Requirements", IEEE Transactions in Software Engineering, Vol. 22, No. 6, pp. 363-377

Heitmeyer C., Labaw B., Kiskis D., 1995. "Consistency Checking of SCR-Style Requirements Specifications", Proceedings of the 2nd International Symposium on Requirements Engineering (RE '95), IEEE CS Press, pp. 56-63.

Heitmeyer C., Jeffords R., and Kiskis D., 1996. "Automated Consistency Checking Requirements Specifications", ACM Transactions on Software Engineering and Methodology, Vol 5, No 3, pp. 231-261.

Hogger C., 1990. "Essential of Logic Programming", Graduate Texts in Computer Science Series, Clarendon Press, Oxford, ISBN 0-19-853832-4.

Holzmann J., 1997. "The model checker SPIN", IEEE Transactions on Software Engineering, Vol. 23, No 5, pp. 279-295

Hunter, A. & Nuseibeh, B., 1995. "Managing Inconsistent Specifications: reasoning, analysis and action" Department of Computing Technical Report Number 95/15, Imperial College, London, UK.

Hunter, A. and Nuseibeh, B., 1997. "Analysing Inconsistent Specifications". Proceedings of 3rd International Symposium on Requirements Engineering (RE 97), Annapolis, USA, January, IEEE CS Press, pp. 78-86.

Hunter A. & Nuseibeh B., 1998. "Managing Inconsistent Specifications: Reasoning, Analysis and Action", ACM Transactions on Software Engineering and Methodology, Vol. 7, No. 4, pp. 335-367.

ISO/IECJTC1/SC21/WG7, 1995. "Open Distributed Processing–Reference Model: Part 2: Foundations", ISO/IEC 10746-2:ITU-T Recommendation X.902, February 1995.

Jackson M., 1997. "The meaning of requirements", Annals of Software Engineering, Vol. 3, pp. 5-21.

Knight K., 1989. "Unification: A Multidisciplinary Survey", ACM Computing Surveys, Vol. 21, No 1, pp. 93-124.

Krasner H., Curtis B., Iscoe N., 1987. "Communication Breakdowns and Boundary Spanning Activities on Large Programming Projects", Empirical Studies of Programmers: 2nd Workshop, (eds) Olson G., Sheppard S., Soloway E., Ablex Publishing Corporation.

Kotonya, G., Sommerville, I., 1999. "Requierements Engineering with Viewpoints". Software Engineering Journal, vol. 11, n. 1, January, pp. 5-18.

Leite J., Freeman P.A., 1991. "Requirements Validation through Viewpoint Resolution" IEEE Transactions on Software Engineering, Vol. 12, No. 12, pp. 1253-1269.

Leonhardt, U., Finkelstein, A., Kramer, J., & Nuseibeh, B., 1995. "Decentralised Process Enactment in a Multi-Perspective Development Environment", Proceedings of 17th International Conference on Software Engineering (ICSE-17), Seattle, Washington, USA, 24-28th April, IEEE CS Press, pp. ???.

Maiden N., Assenova P., Jarke M., Spanoudakis G. et al., 1995. "Computational Mechanisms for Distributed Requirements Engineering", Proceedings of the 7th International Conference on Software Engineering & Knowledge Engineering (SEKE '95), Pitsburg, Maryland, USA, pp. 8-16.

McDermid J.A., Vickers A.J. & Wilson S.P., 1996."Managing Analytical Complexity of Safety Critical Systems using Viewpoints", Joint Proceedings of the Sigsoft '96 Workshops – Viewpoints '96, ACM Press, pp. 272-274.

McMillan, L., 1993. "Symbolic Model Checking", Kluwer Academic Publishers

Mullery G., 1985. "Acquisition–Environment", Distributed Systems: Methods & Tools for Specification, Paul M. and Siegert H. (eds), LNCS, 190, Springer-Verlag.

Mullery G., 1996. "Tool Support For Multiple Viewpoints", Joint Proceedings of the Sigsoft '96 Workshops – Viewpoints '96, ACM Press, pp. 227-231.

Mylopoulos J. 1990. "Telos: Representing Knowledge about Information Systems", ACM Transactions on Information Systems, pp. 325-362.

Nissen H., Jeusfeld M., Jarke M., Zemanek G., Huber H., 1996. "Managing Multiple Requirements Perspectives with Metamodels", IEEE Software, March 1996, pp. 37-47.

Nuseibeh B., 1996. "Towards a Framework for Managing Inconsistency Between Multiple Views" Proceedings Viewpoints 96: International Workshop on Multi-Perspective Software Development, ACM Press, pp. 184-186.

Nuseibeh B., Finkelstein A., 1992. "Viewpoints: a vehicle for method and tool integration" Proc. 5th International Workshop on CASE - CASE 92, IEEE CS Press, pp. 50-60.

Nuseibeh B., Finkelstein A., Kramer J., 1996. "Method Engineering for Multi-Perspective Software Development", Information and Software Technology Journal.

Nuseibeh B., Kramer J., Finkelstein A., 1994. "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification", IEEE Transactions on Software Engineering, Vol. 20, No. 10, pp. 760-773.

Nuseibeh B. and Russo A., 1999. "Using Abduction to Evolve Inconsistent Requirements Specifications", Austrialian Journal of Information Systems, 7(1), Special Issue on Requirements Engineering, ISSN: 1039-7841.

Nuseibeh, B., Easterbrook, S., and Russo, A., 2000. "Leveraging Inconsistency in Software Development". IEEE Computer, April.

OMG, 1999. OMG Unified Modelling Language Specification, V. 1.3a. Available from:ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf.

Ossher H., Kaplan M., Harrison W., Katz A., Kruskal V., 1995."Subject-Oriented Composition Rules", Proceedings of International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '95), ACM SIGPLAN Notices, pp. 235-251.

Papadimitriou C., Steiglitz K., 1982. "Combinatorial Optimisation: Algorithms and Complexity", Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Peterson J., 1981. "Petri Net Theory and the Modelling of Systems", Prentice Hall.

Piwetz, C. & Goedicke, M., "A Module Concept for ViewPoints" Joint Proceedings of the Sigsoft '96 Workshops – Viewpoints '96, ACM Press, pp. 247-251.

Pohl K., 1996. "Process-Centred Requirements Engineering", Advanced Software Development Series, (ed) Krammer J., Research Studies Press, ISBN 0-86380-193-5, London.

Robinson W., 1994. "Interactive Decision Support for Requirements Negotiation", Concurrent Engineering: Research & Applications, 2, pp. 237-252.

Robinson W., Fickas S., 1994. "Supporting Multiple Perspective Requirements Engineering", Proceedings of the 1st International Conference on Requirements Engineering (ICRE 94), IEEE Computer Society Press, pp.206-215.

Robinson W., 1997. "I Didn't Know My Requirements were Consistent until I Talked to My Analyst", Proceedings of 19th International Conference on Software Engineering (ICSE-97), IEEE Computer Society Press, Boston, USA, May 17-24.

Robinson W. and Pawlowski S., 1999. "Managing Requirements Inconsistency with Development Goal Monitors", IEEE Transactions on Software Engineer, November/December.

Shoenfield J., 1967. "Mathematical Logic", Addison Wesley.

Spanoudakis G., Constantopoulos P., 1995. "Integrating Specifications: A Similarity Reasoning Approach", Automated Software Engineering Journal, Vol. 2, No. 4, pp. 311-342.

Spanoudakis G., Finkelstein A., 1997. "Reconciling Requirements: a method for managing interference, inconsistency and conflict" Annals of Software Engineering, Special Issue on Software Requirements Engineering,

Spanoudakis G., Finkelstein A. 1998. "A Semi-automatic process of Identifying Overlaps and Inconsistencies between Requirement Specifications", In Proceedings of the 5th International Conference on Object-Oriented Information Systems (OOIS 98), pp. 405-424.

Spanoudakis G., Finkelstein A., Till D., 1999. "Overlaps in Requirements Engineering", Automated Software Engineering Journal, Vol 6, pp. 171-198

Spanoudakis G., Kassis K., 2000. "An Evidential Framework for Diagnosing the Significance of Inconsistencies in UML Models", Proceedings of the International Conference on Software: Theory and Practice, World Computer Congress 2000, Bejing, China, pp. 152-163

van Lamsweerde A., 1996. "Divergent Views in Goal-Driven Requirements Engineering", Joint Proceedings of the Sigsoft ‘96 Workshops – Viewpoints ‘96, ACM Press, pp. 252-256.

van Lamsweerde A., Darimont R., & Letier E., 1998. "Managing Conflicts in Goal-Driven Requirements Engineering", IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development, November.

van Lamsweerde A, Letier E., 2000, "Handling Obstacles in Goal-Oriented Requirements Engineering", IEEE Transcactions on Software Engineering, Vol 6, Special Issue on Exception Handling

Waldinger R., 1977. "Achieving Several Goals Simultaneously", In Machine Intelligence, Vol 8, (eds) E. Elcock and Mitchie D., Ellis Horwood, 1977

Walz B., Elam J., Curtis B., 1993. "Inside a Software Design Team: Knowledge Acquisition, Sharing and Integration", Communications of the ACM, Vol. 36, No. 10, pp. 63-77.

Zave P., Jackson M., 1993. "Conjunction as Composition", ACM Transactions on Software Engineering and Methodology, Vol. 2, No. 4, pp. 379-411.

Zisman A., Emmerich W., and Finkelstein A. , 2000. "Using XML to Specify Consistency Rules for Distributed Documents", 10th International Workshop on Software Specification and Design (IWWSD-10), Shelter Island, San Diego, California, November (to appear).