Inconsistencies in Software Development: Towards a Reference Framework *

Gianpaolo Cugola and Carlo Ghezzi
[cugola,ghezzi]@elet.polimi.it
Dipartimento di Elettronica e Informazione
Politecnico di Milano
P.za Leonardo da Vinci 32
20133 Milano (Italy).
Tel.: +39-2-23993666
Fax: +39-2-23993411

April 9, 1997

Abstract

This paper describes a framework to formalize the concept of inconsistency in software development. This framework will be used to analyze and compare several approaches proposed in literature to deal with the different forms of inconsistencies that may arise in software development.

1 Introduction

During software development a number of "models" are produced. The documents describing the software requirements, the documents describing the software design and the source and object code are all examples of such models. All these models have to be internally consistent (i.e., they have not to assert contradictory facts) and they have to be consistent with each other.

The description of the process that should be followed to produce the software, is another kind of model that has also to be considered during software development. Even this model, like the previous ones, must be consistent internally and with the process that is actually followed (i.e., it must describe the actual process faithfully).

During software development inconsistencies inevitably arise. As an example, in the early stages of development, contradictory requirements may be specified by different developers having different views of the system; design may be inconsistent with the requirements; and, finally, errors during implementation may result in source code that is inconsistent with the requirements or with the design specification. Similarly, the model of the software process that should be followed may be inconsistent with the process actually followed because the people involved in the software development activities deviate from the modeled process, e.g., to cope with unforeseen situations.

Different approaches have been proposed to detect inconsistencies during software development, to analyze, and to eliminate them. Often these approaches differ not only in the way and time inconsistencies are detected, analyzed, and solved but even in the way the concept of inconsistency is defined. To analyze and compare such approaches and to reuse the same approach in different fields, we need a common definition of the concept of inconsistency.

*This work has been partially sponsored by the CNR project PROGRESS.

In this paper we are not trying to provide an approach that can be used to detect and solve inconsistencies. Rather, we focus on a definition that could be useful to compare the existing approaches to inconsistencies detection and management.

The paper is organized as follows: Section 2 describes the proposed framework. Section 3 uses the framework to compare different approaches to inconsistencies management. Finally, Section 4 draws some conclusions.

2 A framework to formalize the concept of inconsistency

Before introducing a new framework to formalize the concept of inconsistency, a preliminary question has to be answered: "why not simply use the standard definition of inconsistency in logic whose semantics is formally defined and universally accepted?" The answer is simple. The logical definition of inconsistency does not fit our needs for three main reasons.

- First, the logical definition of inconsistency may be used only if the models to compare are all expressed using the same logical language. Conversely, we are interested to the general case of comparing models given in different, not necessarily logical, languages.
- Second, often the logical definition of inconsistency conflicts with the intuitive meaning given to the word "inconsistency". In particular, it may happen that two models, which intuitively seem to be inconsistent, are logically consistent. As an example consider the following logical formulas, both describing a domain composed of a single elevator and a set of users.

$$\forall t, f, u(User(u) \wedge IsUserAtFloor(f, u, t) \wedge IsElevatorAtFloor(next(f), t) \wedge PressTheCallButton(u, t) \Rightarrow IsElevatorAtFloor(f, t + 10))$$

$$\forall t, f, u(User(u) \wedge IsUserAtFloor(f, u, t) \wedge IsElevatorAtFloor(next(f), t) \wedge PressTheCallButton(u, t) \Rightarrow IsElevatorAtFloor(next(f), t + 10))$$

  The first formula says that if a user at floor f calls the elevator that is at floor $f + 1$ then, after ten seconds, the elevator reaches floor f. The second formula says that if the same preconditions holds then, after ten seconds, the elevator is still at floor $f + 1$.

  The two formulas assert contradictory facts but, unfortunately, they are logically consistent. The problem is that we did not provide information enough to logically prove the inconsistency of the two formulas. In particular we did not assert that:

$$\forall f(next(f) \neq f)$$

$$\forall f, t(IsElevatorAtFloor(f, t) \Rightarrow \nexists f'(f' \neq f \wedge IsElevatorAtFloor(f', t)))$$

  These formulas formalize some fundamental constraints of the domain we were specifying.

  The example shows that two or more specifications may be logically consistent even if intuitively they are inconsistent. This happens because the specification obtained by conjoining them is not *complete*, that is, it lacks some information that is necessary to prove the logical inconsistency.
- Finally, the logical definition of consistency does not take care of the problem of *designations*. When two logical models are put together to evaluate their consistency, it is assumed that each predicate or function symbol has the same meaning in both models. Unfortunately, this assumption is seldom satisfied. In general, the two models have been developed by different modelers who may have used the same predicate or function symbol with different meanings.

  Making the assumption that each symbol has the same meaning in all the specifications where it appears is one of the main sources of errors when the consistency of different specifications has to be evaluated.

For instance, consider the predicate symbol $doorClosed$. It can be used to mean that a door is closed (but not necessarily locked) or that it is closed and locked. Suppose that the former meaning is used in the first of the two formulas that follow and the latter meaning is used in the second. As a result, the two formulas are intuitively inconsistent (they assert contradictory facts) even if they are logically consistent.

$$doorClosed \wedge \neg doorLocked$$

$$doorClosed$$

The problem arises from the fact that the same symbol $doorClosed$ has a slightly different meaning in the two formulas. To derive a logical inconsistency we have to explicitly express the difference in the meaning given to the symbol $doorClosed$ by adding the following logical formula:

$$doorClosed'' \Leftrightarrow doorClosed' \wedge doorLocked'$$

where we used one or two quotes to distinguish between the symbols of the first model (i.e., formula) and the symbols of the second.

The problem shown by this example is related to a kind of incompleteness but of different nature than the incompleteness we were referring before in the elevator example. There, we were unable to logically prove the inconsistency of the two models because the model obtained by conjoining them lacks some *domain information* which is needed to prove the inconsistency. Conversely, the information we need here to logically prove the inconsistency of the two models does not depend on domain information, but it depends only on the meaning given to the symbols used into the two models.

In this paper we explore a new framework that tries to overcome the problems found in a strict, conventional approach based on logic, and still is based on sound, formal foundations.

2.1 Formal models and domains

The framework developed in this paper is based on the concept of a *formal model*. A formal model asserts some facts about a *domain* that constitutes the *scope* of the model [1]. A formal model is expressed using some formal language (i.e., a language whose semantics is formally defined)[1].

A domain is a particular part of the world that can be considered separately from the rest of the world [1]. It is composed of a set of entities that interact and evolve during time. At every instant, a domain is in a particular *state* that is characterized by the properties that hold for the different entities that compose the domain and the relationships that hold among these entities. Moreover, a domain is characterized by a set of underlying properties that express the physical constraints that have to hold in that particular domain. We name this properties *domain properties*. They restrict the set of admissible states and the set of feasible state transitions for the domain.

The following sentence is an example of model given in natural language:

> A lift passenger who wants to travel from the third to the seventh floor presses the Up button at the third floor. The button light must then be on, if it was not lit before. The lift must arrive reasonably soon. The direction of travel is indicated by an arrow illuminated when the lift arrives. The doors must open and stay open long enough for the passenger to enter the lift. The doors must never be open except when the lift is stationary at floor[2].

The scope of this model is the domain composed of the lift passengers, the lift, the floors, the lift buttons, the buttons' lights, and the arrows indicating the direction of travel. This domain is characterized by domain properties like the following:

[1]The concept of formal model given in this paper has not to be confused with the concept of model of a theory used in logic. In this paper the word "formal model" (or simply "model") will be used to denote a formal description of some part of the world.

[2]This example is a partial rewriting of a problem statement given in [1], page 169.

- the lift is constrained to move so that it never goes from one floor to another without passing all the intermediate floors;
- if the winding motor polarity is set to up and the motor is activated, then the lift starts to rise;
- the lift doors take 2250 msecs to reach the fully closed from the fully open state.

It is useful to distinguish between two different kinds of models: static and dynamic. A *static model* describes the static properties of a domain. In a static model there is no concept of time. There are no events, there is no description of how things change during time. It describes the properties that have to be satisfied by all the states of the domain, abstracting from the properties that relate a state with the "next".

Given a set S of states of a domain, a static model M is *compatible* with this set of states if all the states in S satisfy the constraints stated by M. The states in S may satisfy other constraints too, but this is irrelevant. The only relevant things is that they do not violate the constraints specified by the static model. As an example, a static model asserting that a paper has a title, and is composed of an abstract and one or more sections, the first named "Introduction" and the last named "Conclusions", is compatible with all the states of the world that satisfy this constraint independently of all the other constraints that may be true in these states. Typical examples of static models are the entity-relationship diagrams and the class diagrams of an OO specification language.

Conversely, a *dynamic model* describes the way a domain evolves over time. In a dynamic model there is a time dimension. It describes how things happen and how things change and states the properties that must be satisfied by the legal sequences of states.

A dynamic model M is compatible with a set of *scenarios* of a domain. A scenario may be formally defined as a sequence of states that represents a particular evolution of the entities that belong to the domain. A dynamic model is compatible with a scenario if the scenario satisfies the constraints (both static and dynamic) stated by the model. The scenario, however, may describe additional events and satisfy additional constraints. As an example, a dynamic model which asserts that after pressing a light button the room becomes lighted is compatible with any scenario that satisfies this constraint: e.g., a person presses the light button and two lamps are switched on to light the room, or, alternatively, a person presses the light button and a single lamp lights the room. In fact, the model does not says anything about the person who presses the light button and the number of lamps that are switched on. Examples of dynamic models are the models expressed using state machines or Petri nets.

Each formal model contains a set of *terminal symbols*. They are all the symbols that compose the model and that are not part of the language. Each terminal symbol *designates* a particular element or property that is part of the domain described by the model [1].

The meaning of the same terminal symbol may vary from modeler to modeler and from model to model (even the same modeler may attach different meanings for the same terminal symbol in different models).

Knowing the exact meaning of the terminal symbols appearing in a model (i.e., the elements or properties they designate) is fundamental to decide if a particular state or scenario is compatible with the model.

For example, the state machine shown in Figure 1 models the operation of a pneumatic door. It contains the terminal symbols: `doorOpen`, `doorPartiallyClosed`, and `doorClosed` which informally describe the states in which the door is totally open, partially closed, or totally closed, respectively; and the symbols `pressTheOpenButton`, `pressTheCloseButton`, `doorTotallyClosedSignal`, and `doorTotallyOpenSignal`, which represent the events of pressing the open and close buttons and the events of receiving a signal from the micro-switches that stop the pneumatic mechanism, respectively. A scenario compatible with such model is the following: the door is closed, a person comes to the door and presses the open button. After a while the door starts opening. When it is completely opened the person enters through the door.

In general, we may say that the formal semantics of the language used to build a model, together with the meaning of the different terminal symbols used, defines two binary relations:

$$S_{rd} \subseteq S_m \times St$$

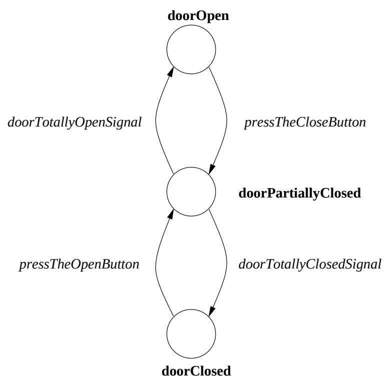

Figure 1: The specification of a pneumatic door

$$D_{rd} \subseteq D_m \times Sc$$

where S_m is the set of all the static models, St is the set of all states, D_m is the set of all dynamic models, and Sc is the set of all scenarios.

S_{rd} relates a static model with the states which are compatible with it, while D_{rd} relates a dynamic model with the scenarios which are compatible with it. For example, the scenario in which the door modeled in the previous example is closed, a person presses the open button and the door opens is compatible with the dynamic model of Figure 1, while a scenario in which, after pressing the open button, the door remains closed is not.

The relations S_{rd} and D_{rd} differ from the interpretation relation I defined in logic (which relates a logical formula with the "models" that satisfy it) in two aspects:

1. I relates a logical formula with an algebraic model (i.e., the interpretation of the formula) which does not have any direct relation with the domain described by the formula. Conversely, relation S_{rd} (resp. D_{rd}) relates a model with a state (resp. scenario) which is the "correct" and "complete" description of a domain.
2. Relations S_{rd} and D_{rd} embody the notion of designations. A model is in relation with a certain domain state or scenario only if it satisfies the constraints expressed by the model, interpreted according to the meaning of the terminal symbols that compose the model. A valid interpretation of the logical formula:

   $$\exists d(door(d) \wedge open(d))$$

   is given by a set U representing the universe of the discourse, together with two algebraic, unary, relations (i.e., two sets) $R_1 \subseteq U$ and $R_2 \subseteq U$ such that exists at least one element belonging to both the relations, and, finally, by a mapping from the symbol *door* to R_1 and from the symbol *open* to R_2. On the other hand, S_{rd} asserts that the static model expressed by the previous formula is compatible only with some states of some particular domains (the domains comprising doors).

In general, there is an infinite set of scenarios compatible with a dynamic model m, (i.e., the set $\{s : Sc|(m,s) \in D_{rd}\}$). Scenarios belonging to this set differ in the elements, events and constraints that are not described in the model. The same holds for the states that belong to the set of states compatible with a static model. Each model, in fact, is an abstract description of a domain that necessarily leaves out many details considered irrelevant by the modeler. For instance, the model of the previous example does not say anything about the time that elapses between the pressing of the open button and the complete opening of the door. Any scenario that differs only in the time elapsed between these two states is compatible with that model.

2.2 Intra-model inconsistency

A static (resp. dynamic) model m is *internally inconsistent* if there are no states (resp. scenarios) compatible with it. Formally:

$$\forall t, person(doorClosed(t) \wedge pressOpenButton(person, t) \rightarrow doorOpen(t+10)$$

$$\forall t, person(doorOpen(t) \wedge pressCloseButton(person, t) \rightarrow doorClosed(t+10)$$

Figure 2: A different specification of a pneumatic door

Definition 1 *A static model sm is internally inconsistent iff:*

$$\{st \in St | (sm, st) \in S_{rd}\} = \emptyset$$

Similarly:

Definition 2 *A dynamic model dm is internally inconsistent iff:*

$$\{sc \in Sc | (dm, sc) \in D_{rd}\} = \emptyset$$

As an example the static model:

$$doorClosed \wedge doorOpen$$

is internally inconsistent if the intuitive meaning is given to the terminal symbols *doorClosed* and *doorOpen*. Notice, however, that this model is logically consistent and this is another example of the fact that the logical definition of inconsistency may lead to conclusions that conflict with the intuition.

2.3 Inter-model inconsistency

Two static models sm_1 and sm_2 that describe the same domain are inconsistent if there are no states that are compatible with both[3]. Formally:

Definition 3 *Two static models sm_1 and sm_2 that describe the same domain are inconsistent iff there is no state st such that:*

$$(sm_1, st) \in S_{rd} \wedge (sm_2, st) \in S_{rd}$$

Similarly:

Definition 4 *Two dynamic models dm_1 and dm_2 that describe the same domain are inconsistent iff there is no scenario sc such that:*

$$(dm_1, sc) \in D_{rd} \wedge (dm_2, sc) \in D_{rd}$$

and finally:

Definition 5 *A dynamic model dm and a static model sm that describe the same domain are inconsistent iff it does not exist any scenario sc such that:*

$$(dm, sc) \in D_{rd} \wedge sc = (st_1, ..., st_n) \wedge \forall i (i \in [1, n] \Rightarrow (sm, st_i) \in S_{rd})$$

As an example, the dynamic model of Figure 2 is consistent with the dynamic model of Figure 1 because the scenario in which the open button is pressed at time t and the door opens after ten seconds is compatible with both models (we used the intuitive meaning for the terminal symbols that appear in both the models).

[3] Observe that two models that describe two different, non overlapping, domains are necessarily consistent because they cannot assert contradictory facts. This is true even by using our definition of inconsistency because any states (resp. scenario) of a domain is compatible with any static (resp. dynamic) model that describes a different domain.

2.4 Completeness and Correctness

The framework to characterize the concept of inconsistency given so far suffers from a main problem: a model that does not say anything about the domain it describes is consistent with any other internally consistent model and any scenario is compatible with it. In fact, consider the static and dynamic models described logically by the predicate $true$ (i.e., no constraints are given on states and scenarios). Any other internally consistent model is consistent with such models.

To make our framework more useful we introduce the concepts of completeness and correctness of a model.

Definition 6 *A static (resp. dynamic) model is complete with respect to a set of states (resp. scenarios) iff all of them are compatible with it.*

Definition 7 *A static (resp. dynamic) model is correct with respect to a set of states (resp. scenarios) S iff any state (resp. scenario) compatible with the model belongs to S.*

The model described logically by the predicate $true$ is complete with respect to any set of scenarios but it is correct only with respect to the set of scenarios that includes any possible scenario.

The concepts of completeness and correctness are very important in requirements engineering. In fact, a requirements specification is complete (resp. correct) with respect to the user requirements if it is complete (resp. correct) with respect to the set of scenarios (or states) that capture the user requirements.

2.5 Levels of abstraction and equivalence

Using the framework defined so far, it is possible to define when two models are equivalent and when they differ only with respect to the level of abstraction.

Definition 8 *Given two static models sm_1 and sm_2 that describe the same domain, sm_1 is said to be at a lower level of abstraction than sm_2 iff:*

$$\{st \in St | (sm_1, st) \in S_{rd}\} \subseteq \{st \in St | (sm_2, st) \in S_{rd}\}$$

Similarly:

Definition 9 *Given two dynamic models dm_1 and dm_2 that describe the same domain, dm_1 is said to be at a lower level of abstraction than dm_2 iff:*

$$\{sc \in Sc | (dm_1, sc) \in D_{rd}\} \subseteq \{sc \in Sc | (dm_2, sc) \in D_{rd}\}$$

Finally,

Definition 10 *Two static models sm_1 and sm_2 are equivalent iff:*

$$\{st \in St | (sm_1, st) \in S_{rd}\} \equiv \{st \in St | (sm_2, st) \in S_{rd}\}$$

and similarly:

Definition 11 *Two dynamic models dm_1 and dm_2 are equivalent iff:*

$$\{sc \in Sc | (dm_1, sc) \in D_{rd}\} \equiv \{sc \in Sc | (dm_2, sc) \in D_{rd}\}$$

3 Applying the framework

In this section we use the framework described so far to compare two approaches to inconsistency detection and management which cope with two different kinds of inconsistencies. The former approach, described in 3.1, faces the problem of inconsistencies in requirement engineering; the latter, described in 3.2, faces the problem of inconsistencies arising in Process-centered Software Engineering Environments (PSEEs) as the result of a deviation of the development process from its model. Such kinds of inconsistencies may cause the corruption of all the data maintained by the PSEE which is in charge of controlling the development process.

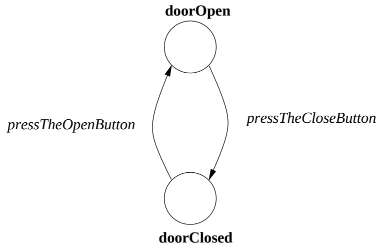

Figure 3: Another specification of the pneumatic door of the previous example

3.1 Inconsistencies in ViewPoints

In [2, 3, 4, 5] the authors propose a solution to the problem of detecting inconsistencies that arise during software development among a set of models that describe different views of the same system (i.e., a set of *ViewPoints*). To identify such inconsistencies they introduce the concept of a *consistency rule*. These rules are expressed using a logical language and relate the different views that compose the description of a system. The consistency rules are used both to detect inconsistencies among different views and to define the concept of an inconsistency among different ViewPoints. Two ViewPoints are said to be "inconsistent" if at least one of the consistency rules that relate them is broken. The software engineers that set up the modeling environment, together with the modelers that use it, are responsible for defining such rules.

Defining an inconsistency as the cause that breaks a consistency rule is quite different from defining an inconsistency as the result of the fact that two models do not agree in the scenarios (or states)that are compatible with them. The relations S_{rd} and D_{rd}, which relate the models to the states and scenarios compatible with them, embody both the semantics of the language used to compose the model and the meaning given to the terminal symbols that appear into the model. On the other hand, the consistency rules for viewpoints are used to directly relate two different models, possibly expressed using different languages.

The main consequence of this fact is that, to be effective, the consistency rules should capture:

1. the semantics of the different models (which, as we said before, depends on the semantic of the language used to compose the models but also on the meaning given by the modelers to the terminal symbols that appear into their models), and
2. all the domain properties.

In general these requirements cannot be fulfilled. First, because because the software engineers that set up the ViewPoint environment (who write down most of the consistency rules) cannot know in advance the meaning that the different modeler will give to the terminal symbols they will use into their models and, second, because it is not possible to express all the domain properties.

Moreover, depending on the consistency rules defined, it is also possible to detect an inconsistency between two models that are simply at a different level of abstraction. As an example, suppose you have the following consistency rule, which applies to models expressed using finite state machines:

> "If a transition between two states is described in one model, and both states are described in another model, then the transition should also be described in the second model" [2].

According to these consistency rule the models in Figure 1 and 3 are inconsistent. On the other hand, intuitively they are not; they are simply at a different level of abstraction.

In conclusion, the approach of using consistency rules to **define** the consistency of two or more models may lead to a counter-intuitive definition of consistency. On the other hand, the consistency rules are useful to **detect** the inconsistencies among different models but the users have to keep in mind the limitations of this approach pointed out by the previous discussion and examples.

3.2 Inconsistencies in PSEEs

In Process-centered Software Engineering Environments (PSEEs)[6], the users may need to deviate from the modeled process to cope with unforeseen situations that are not specified in the process model. This deviations lead to an inconsistency between the modeled process and the process actually followed. The same happens in workflow systems and, more in general, in all the *human centered systems (HCS)* [7]. These environments are characterized by the fact that a process support system is used to enact a process model in order to support a group of people in carrying out a set of work procedures.

In [7] we give a precise formalization of the concepts of inconsistency and completeness in process environments. In this section we will reformulate such definitions using the framework presented so far.

The definition of inconsistency of a process environment given in [7] may be reformulated using the framework presented in this paper by saying that at a specific time t a HCS is *environment-level inconsistent* if the static model of the world kept by the process support system at time t is inconsistent with respect to the ideal model of the world that correctly and completely describes the state of the world at time t.

Similarly, it is possible to reformulate the definition of completeness for a process environment given in [7] by saying that a process environment is *transition complete* (see [7]) if the process model it enacts is complete with respect to the set of scenarios that describe all the possible evolution of the supported process.

In [8] we described SENTINEL, a PSEE that is able to detect inconsistencies between the process model it enacts and the process actually followed. Inconsistencies are defined as in [7] and a logical approach is followed to detect them.

In SENTINEL a software development process is described as a collection of *taskTypes*. Each taskType is modeled as a state machine. State transitions are atomic; they are characterized by a *precondition*, called ENTRY, and a *body*. The ENTRY defines a property that has to be satisfied before executing the corresponding transition.

SENTINEL enacts a process model by executing the transitions whose precondition is true. User interaction with the PSEE are modeled through a special kind of transition: the *exported transition*. Exported transitions are defined as any other transition, i.e., they are characterized by a precondition and a body. They differ from normal transitions in the way they are executed by the PSEE: a normal transition is automatically executed when its ENTRY becomes true while an exported transition can be executed only if the user requests it. The peculiarity of SENTINEL is that if the ENTRY of an exported transition is not verified, the user may still force its execution.

By forcing the execution of transition whose precondition is not verified, the users are allowed to deviate from the predefined set of legal behaviors to perform activities that were not modeled. SENTINEL is able to maintain a correct and complete model of the activities actually followed in the environment and supports the users even in performing unforeseen activities.

Having a correct and complete model A_m of the activities actually performed, SENTINEL is able to find inconsistencies between this model and the model E_m it enacts. In particular, both A_m and E_m are expressed using a first order temporal logic [4] and SENTINEL is able to detect logical inconsistencies between them.

4 Conclusions

The paper presented a framework to formally define the concept of internal inconsistency, completeness and correctness of a formal model and the concept of inconsistency between two formal models.

[4] E_m is not directly expressed in logic but it is automatically translated into logic.

The definition of inconsistency given is not suited to help in detecting and managing inconsistencies but it may be used to compare the different approaches presented in the literature to cope with inconsistencies in different fields of the software engineering.

We outlined an application of the framework to the description of two different approaches dealing with the problem of detecting and tolerating inconsistencies.

Future effort will be devoted to apply the framework presented in this paper in comparing and evaluating other solutions to the problem of inconsistency detection and management. We will also try to use the framework presented to guide the adoption of the various approaches to inconsistency management into fields different from that where they were developed.

References

[1] M. Jackson, *Software Requirements & Specifications.* ACM Press and Addison-Wesley, 1995.

[2] S. Easterbrook and B. Nuseibeh, "Managing inconsistencies in an evolving specification," in *Proceedings of 2nd International Symposium on Requirements Engineering (RE 95)*, (York, UK), pp. 48–55, IEEE CS Press, 27-29th March 1995.

[3] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh, "Coordinating distributed viewpoints: The anatomy of a consistency check," *nternational Journal on Concurrent Engineering: Research & Applications*, vol. 2, no. 3, pp. 209–222, 1994.

[4] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency handling in multi-perspective specifications," *Transactions on Software Engineering*, vol. 20, pp. 569–578, August 1994.

[5] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A framework for expressing the relationships between multiple views in requirements specification," *Transactions on Software Engineering*, vol. 20, pp. 760–773, October 1994.

[6] A. Finkelstein, J. Kramer, and B. Nuseibeh, eds., *Advances in Software Process Technology.* Research Studies Press, J. Wiley, 1994.

[7] G. Cugola, E. Di Nitto, A. Fuggetta, and C. Ghezzi, "A framework for formalizing inconsistencies in human-centered systems," *ACM Transaction On Software Engineering and Methodologies (TOSEM)*, July 1996.

[8] G. Cugola, E. Di Nitto, C. Ghezzi, and M. Mantione, "How to deal with deviations during process model enactment," in *Proceedings of the 17th International Conference on Software Engineering*, (Seattle (Washington - USA)), Apr. 1995.