2010

Supporting change propagation in UML models

Hoa Khanh Dam
University of Wollongong, hoa@uow.edu.au

Michael Winikoff
University of Otago

Supporting change propagation in UML models

Abstract

A critical issue in software maintenance and evolution is change propagation: given a primary change that is made in order to meet a new or changed requirement, what additional, secondary, changes are needed? We have previously developed techniques for effectively supporting change propagation within design models of intelligent agent systems. In this paper, we propose how this approach is applied to support change propagation within UML design models. Our approach offers a number of advantages in terms of saving substantial time writing hard-coded rules, ensuring soundness and completeness, and at the same time capturing the cascading nature of change propagation. We will also present and discuss the results of an evaluation performed to assess the scalability of our approach.

Keywords

era2015

Disciplines

Physical Sciences and Mathematics

Publication Details

Dam, H. K. & Winikoff, M. (2010). Supporting change propagation in UML models. 26th IEEE International Conference on Software Maintenance (ICSM) (pp. 1-10). Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010): IEEE.

Supporting change propagation in UML models

Hoa Khanh Dam
School of Computer Science and Software Engineering
University of Wollongong
Wollongong, Australia
Email: hoa@uow.edu.au

Michael Winikoff
Department of Information Science
University of Otago
Dunedin, New Zealand
michael.winikoff@otago.ac.nz

***Abstract*—A critical issue in software maintenance and evolution is change propagation: given a primary change that is made in order to meet a new or changed requirement, what additional, secondary, changes are needed? We have previously developed techniques for effectively supporting change propagation within design models of intelligent agent systems. In this paper, we propose how this approach is applied to support change propagation within UML design models. Our approach offers a number of advantages in terms of saving substantial time writing hard-coded rules, ensuring soundness and completeness, and at the same time capturing the cascading nature of change propagation. We will also present and discuss the results of an evaluation performed to assess the scalability of our approach.**

I. Introduction

The ever-changing business environment demands constant and rapid evolution of software and consequently, change is inevitable if software systems are to remain useful. In this context, a critical issue is change propagation [1]: given a set of primary changes that have been made to software, what additional, secondary, changes are needed to maintain consistency within the system? For example, when adding a message to a sequence diagram, a corresponding method may need to be added to a class diagram.

Change propagation is very important in the process of maintaining and evolving a software system. The software maintainer has to ensure that the change is correctly propagated, and that the software does not contain any inconsistencies. Errors and bugs in software are partly due to unforeseen and uncorrected inconsistencies. Unfortunately, change propagation is a complicated and costly process, especially in complex software systems. The secondary changes may themselves introduce new inconsistencies, which may also trigger additional changes and so on. Although many approaches have been proposed, automated change propagation is still a significant technical challenge in software maintenance and evolution [1].

Furthermore, most of the existing change propagation approaches focus on source code (e.g. [2, 3]). However, as the importance of models in the software development process has been better recognised, it has become increasingly important to provide support for dealing with changes at the level of design models, particularly UML design models. Current modelling environments provide some support for fixing inconsistencies in a design model. For instance, IBM Rational Rose automatically detects inconsistencies between class diagrams and sequence diagrams, and suggests some potential resolutions for fixing such inconsistencies. However, those design tools do not adequately address change propagation problems: they do not capture completely possible changes and are only useful for trivial tasks [4, 5].

Previous work on change propagation within UML models has mostly focused on fixing inconsistencies, with most work aiming to automate inconsistency resolution by having predefined resolution rules (e.g. [6]) or by identifying specific change impact rules for all types of changes (e.g. [7]). However, these approaches suffer from the correctness and completeness issue. Since the rules are developed manually by the user, there is no guarantee that these rules are complete (i.e. that they generate all possible inconsistency resolutions) and correct (i.e. that the resolutions actually fix a corresponding inconsistency). In addition, a significant effort is required to manually hard-code such rules when the number of consistency constraints increases or changes.

The work in [8] addresses this issue by proposing an approach to automatically generate repair actions for consistency constraints expressed in xlinkit [9]. Recent work by Egyed *et al.* [5] proposes a mechanism for fixing inconsistencies in UML design models by automatically generating a set of concrete changes. Their approach uses pre-defined choice generation functions, which compute possible values for locations in the model, for instance, possible new names for a method. The generated options are checked against the constraints and are rejected if they do not in fact repair the constraint, or if they cause new constraint violations. However, this work has several major limitations. Firstly, they consider only a single change at a time, and consequently do not take into account the cases where a single change may not resolve all inconsistencies, or may even temporarily introduce new ones before reaching a consistent state. Secondly, the choice generation functions are written by hand, and may not be complete, meaning that the approach is incomplete: it only considers a subset of the possible ways of repairing a given constraint violation. Finally, their approach does not consider the creation of model elements, which in our opinion is an important part of change propagation.

The work we present in this paper overcomes the above issues by automatically generating inconsistency resolutions for UML design models. Consistency constraints are specified using the Object Constraint Language (OCL) [10], and pos-

26th IEEE International Conference on Software Maintenance in Timişoara, Romania
978-1-4244-8628-1/10/$26.00 ©2010 IEEE

sible inconsistency resolutions are represented in the form of repair plans (see section IV).

Our previous work has shown the effectiveness of our approach in supporting change propagation within agent-oriented design models [11–14]. This paper presents how our approach can be applied to deal with changes within UML design models. Although we use the same approach as in our previous work, this paper will contribute to showing in detail how our approach is in fact applicable to a UML setting, which potentially leads to a significantly wider range of applications. The key ideas of our approach are that (1) a particular style of representation for repair plans — inspired by Belief-Desire-Intention (BDI) style agents [15] — allows a large number of possible inconsistency resolutions to be represented compactly; furthermore, (2) these repair plans can be automatically generated from the OCL constraints; and finally, (3) using a cost calculation can reduce the number of options to be presented to the user.

Our approach has several advantages over existing approaches for fixing inconsistencies in UML models[1]. Firstly, tool developers save substantial time because they do not need to write resolution rules or choice generation functions. Secondly, designers are not required to do any work when consistency constraints change. Finally, our approach provides the designer with not just single change actions, but a series of repair actions (including creation of new model elements and/or relationships) which make the design consistent.

This paper is organised as follows. We begin with background on UML and OCL (Section II), and give an example system design (Section III). We then present our approach, firstly discussing how repair plans are generated (Section IV), and then discussing how they are used to perform change propagation (Section V). Section VI discusses our evaluation. Finally, we discuss related work in Section VII and then conclude (Section VIII).

II. Background

The Unified Modelling Language (UML) has become the *de facto* modelling language for object-oriented software development. It has undergone various revisions and the latest version 2.2 has recently been released. However, UML 2.2 is not commonly supported in industry, in part, because of legacy models and tools [16]. Although we apply our approach to UML 1.4.2 [17], which was adopted as an ISO standard, as will be seen our ideas and results can easily be applied to other versions of UML.

UML has a set of diagrams which provide multiple perspectives of the system under development. Due to space limitation we consider only a fragment of the UML and focus on three major diagrams: class, sequence and statechart diagrams. Class diagrams are the most important structural diagrams which capture the fundamental concepts of object-oriented development: classes and their relationships. Sequence and statechart diagrams play an important part in modelling in UML since they capture the behavioural aspect. There is a great deal of overlap between the three types of diagram. Model elements defined in class diagrams are used in sequence and statechart diagrams. Sequence diagrams and statechart diagrams are complementary in that the former model interactions between objects, whilst the latter depict the behaviour of a single object. The three diagram types also cover a large range of modelling elements in UML. In addition, they are the diagram types most commonly described and studied in the literature [16]. We do not specifically address other types of diagram, such as use case and activity diagrams. However, we do not see any issues with supporting them since our approach operates at the level of metamodels using consistency constraints which can also be applied to such diagrams.

The remainder of this section briefly describes two of the four entities that are used in our change propagation approach, namely the **metamodel** and (OCL) **consistency constraints**. The other two components are **repair plan types** (see section IV) and **the model**, i.e. the design model for the application, such as the video on demand (VOD) system described in section III.

UML is defined using a metamodelling approach. Figure 1 depicts the relationships between major elements in a class diagram: *Class*, *Association*, *AssociationEnd* and *Operation*, and between two major elements in a sequence diagram: *ClassifierRole* (usually referred to as objects in a sequence diagram) and *Message*. According to the UML metamodel, a *Class* has many *AssociationEnds*, and at least two of those are needed to form an *Association*. In addition, a *Class* can own multiple *Operations* and can be a base of several *ClassifierRoles*, which can send or receive messages.

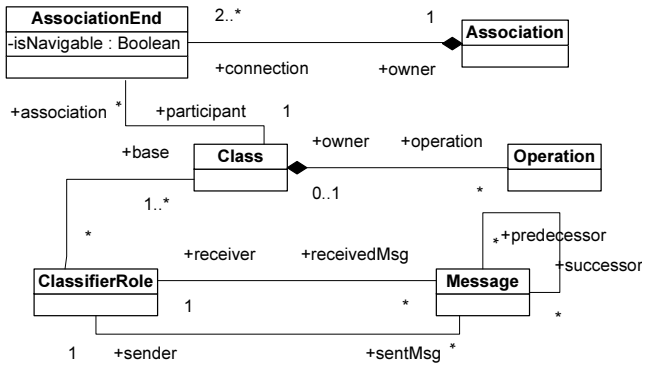

Fig. 1: An excerpt of UML metamodel

Figure 2 depicts the *key* elements of a state machine[2] (alternatively called a *statechart diagram*). The behaviour of each *Class* can be specified by multiple *StateMachines* (although one is sufficient for most purposes). A *StateMachine* contains a top-level *State* and a set of *Transitions*. All remaining states are transitively owned by a state machine through its top state and the state containment hierarchy. A *State* can be either a

[1]Extending our work to encompass code as well as design models is future work.

[2]For the sake of simplicity we abstract away other entities such as StateVertex, PseudoState, Guard, SubState, etc.

CompositeState (that contains other states), a *SimpleState* or a *FinalState*. Each *Transition* has a source *State* and a target *State* that is reached when the transition is taken. A transition can have at most one trigger, which is the *Event* that fires the transition. Each state has transitions departing from it and entering it.

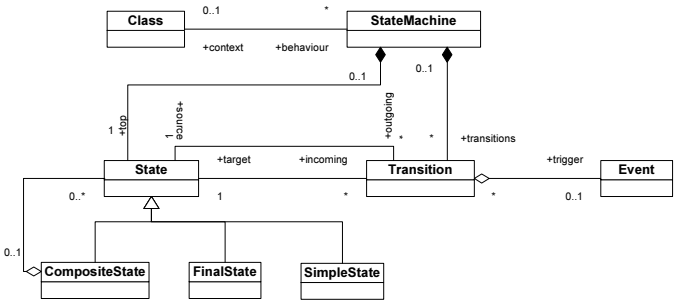

Fig. 2: An excerpt of UML metamodel concerning *Class*, *StateMachine*, *State*, *Transition*

Consistency constraints for UML specify conditions that a UML model must obey for it to be considered a valid UML model, e.g. syntactic well-formedness and coherence between different diagrams. Two such consistency constraints[3] on how UML class, sequence and statechart diagrams relate to each other are given below. The first constraint is a standard UML well-formedness constraint [17], whilst the second is not[4], but it is an example of a coherence constraint between two diagrams [18].

C1: The name of a message (in a sequence diagram) must match an operation in its receiver's class (in a class diagram).
Context Message inv c1:
self.receiver.base.operation→exists(op : Operation | op.name = self.name)

C2: For any incoming message in an object of a sequence diagram, there must exist a transition in the statechart diagram of the object's class that has the same name as the message.
Context Message inv c2:
self.receiver.base.behaviour.transitions→exists(tr : Transition | tr.name = self.name)

III. Example: VOD System

The example we use is a design of a real, albeit simplified, video on demand (VOD) system [16]. The VOD system allows a user to select a movie to play. The user is also able to play, pause and resume the movie. The class diagram (see Figure 3) represents the structure of the initial VOD system. There are three classes: "Display" for visualising movie streams and receiving user inputs, "Streamer" for downloading and decoding movies, and "Server" for providing data. The "Display" class has four operations: "select()" for choosing a movie, "stream()" for playing and retrieving the movie streams, "draw()" for rendering the received movie stream, and "stop()" for halting the movie being played. The "Streamer" class has only two operations: "stream()" for streaming the movie data received from the server, and "wait()" for halting the streaming process. Finally, the "Server" has two operations: "connect()", which is called by clients (e.g. the Streamer), and "handleRequest()" which deals with requests from clients.

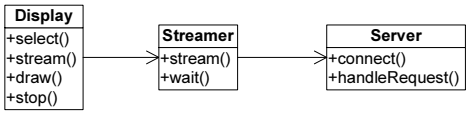

Fig. 3: Class diagram for the VOD system (redrawn based on [16])

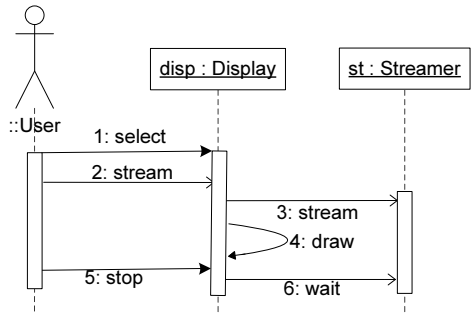

Fig. 4: A sequence diagram for instances of classes Display and Streamer (redrawn based on [16])

The sequence diagram (see Figure 4) depicts a typical scenario of interactions between the user, a Display object ("disp") and a Streamer object ("st")[5]. The user selects a movie that she wants to see (message 1). She then starts playing the selected movie — in the sequence diagram the user sends a message "stream" to the Display (message 2). The Display then retrieves the movie stream from the Streamer (message 3) and renders the movie (message 4). When the user wants to stop viewing the movie (message 5), the Display notifies the Streamer to stop streaming (message 6).

The two statechart diagrams (see Figure 5) describe the behaviour of the two classes: "Display" and "Streamer". As can be seen, the behaviour of the "Streamer" class simply changes between the waiting and the streaming states depending on whether it is triggered by the "wait" or "stream" event. Meanwhile, the behaviour of the "Display" class ranges over three different states: "Idle", "Ready" and "Playing".

IV. Generating repair plan types

After consistency constraints are written (either by tool developers or tool administrators), the repair plan types are automatically generated from the constraints and form a library of repair plan types. It is important to emphasise that this

[3]It is noted that we extensively use the shorthand of the *collect* OCL operation here. For instance, in the first constraint *self.receiver.base* refers to the set of base classes of the *self*'s receiver and *self.receiver.base.operation* results in the set of operations of all those base classes.

[4]It is noted that this constraint is not necessarily universally agreed upon.

[5]A Server object is also involved in these interactions but we do not show it here.

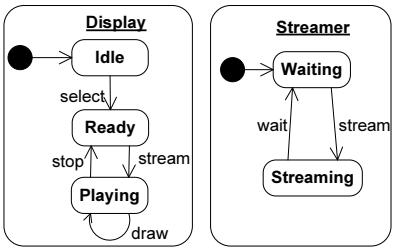

Fig. 5: Statechart diagrams for classes Display and Streamer (redrawn based on [16])

generation is done *once*, when the constraints are specified. It only needs to be re-done should the consistency constraints change. In addition, the tool developers or tool administrators are allowed to use their domain knowledge and expertise to modify generated repair plans or remove plans that should not be executed (however, doing this risks compromising soundness and completeness).

Let us consider the first constraint, denoted as $\mathsf{C1}(self)$, that we have defined above. By analysing its definition we are able to systematically identify several ways of fixing $\mathsf{C1}(self)$. In fact, we have defined a systematic process that translates OCL constraints into a set of repair plans. This process (which cannot be presented here due to lack of space — see [19, Chapter 6] for details) is provably correct and generates a complete set of repair plans[6]. In this case, since the constraint is of the form $\exists x \in e \bullet c(x)$ it can be fixed by either selecting an existing item $y \in e$ and making $c(y)$ true, or by adding an element z to e and ensuring that $c(z)$ is true; in this case z may be an existing element in the model or a newly created element. More specifically, to fix $\mathsf{C1}(self)$ we can:

- Take an operation *op* in *self.receiver.base.operation*, and make $op.name = self.name$ true (which can be achieved by either renaming *op* or renaming *self*).
- Take an existing operation *op* which does not belong to *self.receiver.base.operation*, add *op* to *self.receiver.base.operation*, and make $op.name = self.name$ true.
- Create a new operation *op*, add *op* to *self.receiver.base.operation*, and make $op.name = self.name$ true.

As mentioned earlier, we represent repair plan types as plans, inspired by Belief-Desire-Intention (BDI) agent systems [15]. In such systems, a plan consists of three parts: a triggering event, a context condition, and a plan body. When an event is posted, it is matched against the triggering event of the plans in the plan library, and those plans that match it are considered to be *relevant*. The context condition of the relevant plans are then checked[7] to see which of the relevant plans are *applicable* in the current state. One of the applicable plans is selected, and its body is executed. The plan body may include posting events that are handled through this process. This approach is a good match for modelling change propagation because (a) it naturally models cascades in the form of events posted within plans; and (b) it naturally captures that a given violated constraint (which we model as an event) can have multiple ways in which it can be fixed.

The plan syntax that we use is an extension of AgentSpeak(L) [15]: a plan is written as $e : c \leftarrow b$ where e is the event (corresponding to a constraint to be fixed); c is the context condition, a Boolean condition; and b is the plan body: a sequence of steps. Each step can be an action on the model (e.g. creating an entity, adding/deleting a link, changing the attributes of an entity, or deleting an entity), or posting an event (corresponding to a sub-constraint) written as $!e$.

The above options for repairing $\mathsf{C1}(self)$ can then be written in our repair plan syntax as follows. We use "c1True(self)" to represent making the constraint $c1$ true.

Plan P1: *c1True(self) : op ∈ self.receiver.base.operation ← !c1'True(op, self)*

Plan P2: *c1True(self) : op ∈ Set(Operation) ∧ op ∉ self.receiver.base.operation ← !(Add op to self.receiver.base.operation) ; !c1'True(op, self)*

Plan P3: *c1True(self) ← Create an operation op ; !(Add op to self.receiver.base.operation) ; !c1'True(op, self)*

In the bodies of plans P1, P2 and P3, an event $c1'True(op, self)$ is posted. This event corresponds to making $op.name = self.name$ true, which is handled by the following plans.

Plan P4: *c1'True(op, self) : self.name ≠ op.name ∧ self.name ≠ null ← Rename op.name to self.name*

Plan P5: *c1'True(op, self) : self.name ≠ op.name ∧ op.name ≠ null ← Rename self.name to op.name*

Plan P6: *c1'True(op, self) : self.name = op.name ← true*[8]

Plan P6 applies in the case where an attempt to repair $c1'$ (i.e. to ensure that $self.name = op.name$) turns out to be unnecessary, because it has been achieved by other actions of the repair plan. The repair plan generation process generates similar plans for other constraints, but P6 is the only one that is used in our example, and so the other "do nothing" plans have been elided.

There is another event, *Add op to self.receiver.base.operation*, that is posted within the body of plans P2 and P3. Adding the operation *op* to the set of operations $self.receiver.base.operation$ can be achieved in several ways, including[9] making *op* an operation of the class $self.receiver.base$, or changing $self.receiver.base$ to an existing class that owns the operation *op*. These are expressed in terms of the following repair plans.

Plan P7: *Add op to self.receiver.base.operation ← Connect self.receiver.base with op*

[6] A formal proof of correctness and completeness of the derived repair plans can also be found in [19, Chapter 6].

[7] In fact they are evaluated and each solution yields a plan instance. For example, if the context condition is $x \in \{1, 2\}$ then there will be two plan instances, each of which corresponds to one of the two possible values of x.

[8] A body of "true" means that this plan does nothing.

[9] Other options are less reasonable, e.g. creating a new class, adding *op* to be one of the new class's operations, and making the receiver's class of message *self* to be the new class.

Plan P8: *Add op to self.receiver.base.operation : x ∈ Set(Class) ∧ op ∈ x.operation ← !(Change self.receiver.base to x)*

Similarly, changing *self.receiver.base* to an existing class *x* that owns the operation *op* can be achieved in different ways and is consequently represented as an event. The plans that are able to handle this event include making *x* be the base of *self.receiver* (plan 9) or making the receiver of message *self* be an object that is an instance of class *x* (plan 10).

Plan P9: *Change self.receiver.base to x ← Connect self.receiver to x*

Plan P10: *Change self.receiver.base to x : o ∈ Set(ClassifierRole) ∧ o.base = x ← Connect self to o*

Figure 7 summarises the repair plans (and subplans) for constraint C1 (it is placed in section V-B, where it is used). Using a similar approach [19, Chapter 6] we can automatically derive repair plans for constraint C2.

V. Change Propagation Process

The designer uses the tool to make some primary changes to the design model and then the designer invokes the tool to start propagating changes. The process of change propagation in our framework then proceeds as follows:

A) We check whether the constraints hold in the design model.
B) We use the library of repair plan types to generate plan instances (i.e. repair options) for the violated constraints.
C) We calculate the cost of the different repair plan instances. Since we recognise that fixing one violated constraint may also repair or violate others as a side effect, the cost of a repair plan includes the cost of its actions (using basic costs defined by the designer), the cost of any other plans that it invokes directly, and also the cost of fixing any constraints that are violated after executing this repair plan.
D) We present the cheapest[10] repair plan instances[11] to the designer and ask for their selection; and the selected repair plan instance is executed, updating the design model. We assume here that repair plans which lead to fewer changes to the model, and thus have lower costs, are preferable.

We now describe how each of the steps in this process is applied to our VOD example. In the current VOD design (section III), the Display and Streamer classes have two different methods with the same name "stream". In order to avoid the confusing dual use of the term "stream", the designer makes the following *primary* changes. It is emphasised that these changes are proposed by [16] and intentionally include design errors for the purpose of illustrating how undesirable inconsistencies are identified and resolved.

[10]An alternative to only presenting the cheapest options is to show a ranked list of repair options, or to show only the cheapest options but allow the user to see more options if desired.

[11]Note that a given repair plan instance may perform a number of changes to the model.

A1: Renaming the method "stream()" of class Display to "play()".
A2: Renaming the message "3:stream" to "3:play" in the sequence diagram.
A3: Renaming the state transition named "stream" to "play" in the Display's statechart.

Figure 6 shows how the above changes are made to the initial UML model (each change is marked with a ✓ sign). We now consider how our framework performs change propagation by restoring consistency in this design.

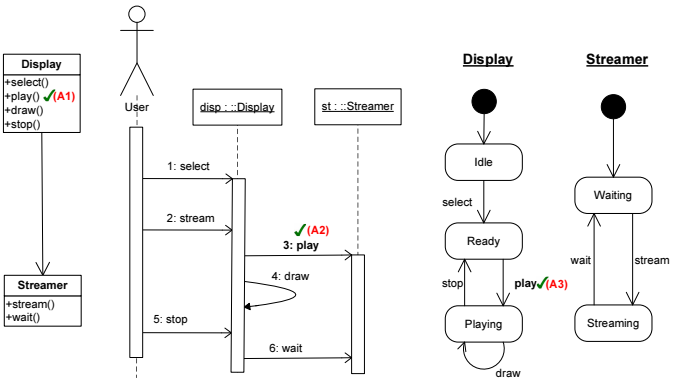

Fig. 6: Design of the VOD system after primary changes are made

A. Check constraints

After the designer completes making the primary changes on the design of the initial system, the first step is checking constraints, which involves the instantiation of pre-defined constraints. For instance, the two constraints defined at the end of section II are instantiated with respect to each instance of the constraints' context. There are 6 instances of the first constraint, each corresponding to a message in the sequence diagram. Each constraint instance is evaluated to check for violation. For example, with respect to the constraint instance C1("2 : *stream*") the evaluation first computes *self.receiver.base.operation* where *self.receiver* is the object "disp" (this object is on the receiving end of the message as shown by the arrowhead), *receiver.base* is the class "Display" (object "disp" is an instance of class "Display"), and *base.operation* is {"*select*()", "*play*()", "*draw*()", "*stop*()"} (the set of operations of the class "Display"). The evaluation then returns false because there does not exist any operation in the set *base.operation* that has the same name (i.e. *stream*) as message "2:stream".

Following a similar approach, we identify the following constraints that are also violated after the primary changes are made: C1("3 : *play*"), constraint C1 evaluated on message "3:play"; C2("2 : *stream*"), constraint C2 evaluated on message "2:stream"; and C2("3 : *play*"), constraint C2 evaluated on message "3:play".

P1	c1True(self) : op ∈ self.receiver.base.operation ← !c1'True(op, self)
P2	c1True(self) : op ∈ Set(Operation) ∧ op ∉ self.receiver.base.operation ← !(Add op to self.receiver.base.operation) ; !c1'True(op, self)
P3	c1True(self) ← Create an operation *op* ; !(Add op to self.receiver.base.operation) ; !c1'True(op, self)
P4	c1'True(op, self) : self.name ≠ op.name ∧ self.name ≠ null ← Rename *op.name* to *self.name*
P5	c1'True(op, self) : self.name ≠ op.name ∧ op.name ≠ null ← Rename *self.name* to *op.name*
P6	c1'True(op, self) : self.name = op.name ← true
P7	Add op to self.receiver.base.operation ← Connect *self.receiver.base* with *op*
P8	Add op to self.receiver.base.operation : $x \in Set(Class) \wedge op \in x.operation$ ← !(Change *self.receiver.base* to *x*)
P9	Change *self.receiver.base* to *x* ← Connect *self.receiver* to *x*
P10	Change *self.receiver.base* to $x : o \in Set(ClassifierRole) \wedge o.base = x$ ← Connect *self* to *o*

Fig. 7: Example repair plans for constraint C1

B. Generate repair plan instances

After violated constraints are identified, the next step in our change propagation framework is generating plan instances for each of the violated constraints. It is important to note that each repair plan type can generate multiple (i.e. zero or more) plan instances, depending on its context condition. For instance, let us consider the repair plan instances for constraint C1("2 : *stream*") (where *self* = "2 : *stream*" and *self.name* = "*stream*") (refer to Figure 7 for the repair plan types). Since *self.receiver.base.operation* = {"*select*()", "*play*()", "*draw*()", "*stop*()"}, repair plan P1 generates 4 plan instances (plans $P1_1$, $P1_2$, $P1_3$, and $P1_4$ in Figure 8), one for each of the existing operations in the "Display" class. Each instance of Plan P1 posts event $c1'True$ which can be handled by three different plans P4, P5, and P6. However, plan type P6 does not generate any plan instance because its context condition does not hold (none of the operations *op* in the "Display" class has the same name as message *self*, i.e. "2:stream"). Therefore, there are 8 possible options to repair constraint C1("2 : *stream*") using plan type P1 (see Figure 8).

1) Rename operation "select()" to *stream*.
2) Rename operation "play()" to *stream*.
3) Rename operation "draw()" to *stream*.
4) Rename operation "stop()" to *stream*.
5) Rename message "2:stream" to *select*.
6) Rename message "2:stream" to *play*.
7) Rename message "2:stream" to *draw*.
8) Rename message "2:stream" to *stop*.

Similarly, let us consider the instances generated by plan P2. Note that there are two existing operations that do not belong to the Display class: "stream()" and "wait()"[12]. Figure 8 shows how plan instances are generated with regard to the case *op* = "*stream*()". In summary, plan P2 with respect to *op* = "*stream*()" gives the following repair options to fix C1("2 : *stream*"):

9) Connect operation "stream()" to class "Display", i.e. add method "stream()" to class "Display".
10) Connect object "disp" to "Streamer" class, i.e. add class "Streamer" to the set of bases of object "disp".
11) Connect message "2:stream" with object "st:Streamer", i.e. changing the receiver of message "stream" to object "st".

Similarly, plan P2 with respect to *op* = "*wait*()" gives four repair options (not shown in Figure 8): options 10 and 11 above (because the change proposed is at the class level, the same change is proposed for *op* = "*wait*()"); and the following two options:

12) Connect operation "wait()" (of class "Streamer") to class Display, and rename operation "wait()" to *stream*.
13) Connect operation "wait()" (of class "Streamer") to class Display, and rename message "stream" to *wait*.

Plan P3 (not shown in Figure 8) involves the creation of a new operation and it has only one instance:

14) Create a new operation, add it to class "Display" and name it "stream".

Overall, there are 14 different options for fixing constraint C1("2 : *stream*"). Repair plan instances for the other three violated constraint instances are also created in a similar way.

C. Calculate cost[13]

The next step in our framework is calculating the cost of each repair option before presenting the cheapest ones to the user for selection. Similarly to the notion of edit distance on strings, we define the cost in terms of the number of operations (i.e. creation, deletion, connection, disconnection, or modification of model elements) needed to transform one model into the other. In order to convert the cost to a single number we need to define "exchange rate" values (termed *basic costs*) that specify (for instance) how many creation operations are equivalent in cost to a deletion operation. These numbers do not correspond to any real cost and are somewhat arbitrary, and thus we allow the designer to specify them. In this example we use the following basic costs: the cost of creation is 0, the cost of connection, disconnection, and modification is 1, and the cost of deletion is 2. Other basic costs can be used but may give a different outcome.

The cost of a repair option is then defined as the sum of the costs of its repair actions and the costs of fixing violated constraints existing after the repair option is executed. The former cost component is calculated by simply summing the cost of each primitive action in a repair option. On the other hand, in order to work out the latter cost we need to simulate the execution of a repair option. For example, the cost of the first repair option (renaming operation "select()" to *stream*) is the cost of the renaming action (which is equal

[12]There are also repair plan instances generated with regard to the two operations in the "Server" class. However, due to space limitation, we do not consider them here.

[13]Here we briefly describe a mechanism for plan selection based on cost calculation which applied for this example. A more detailed presentation, including algorithms and complexity results, can be found in [19, Chapter 7]

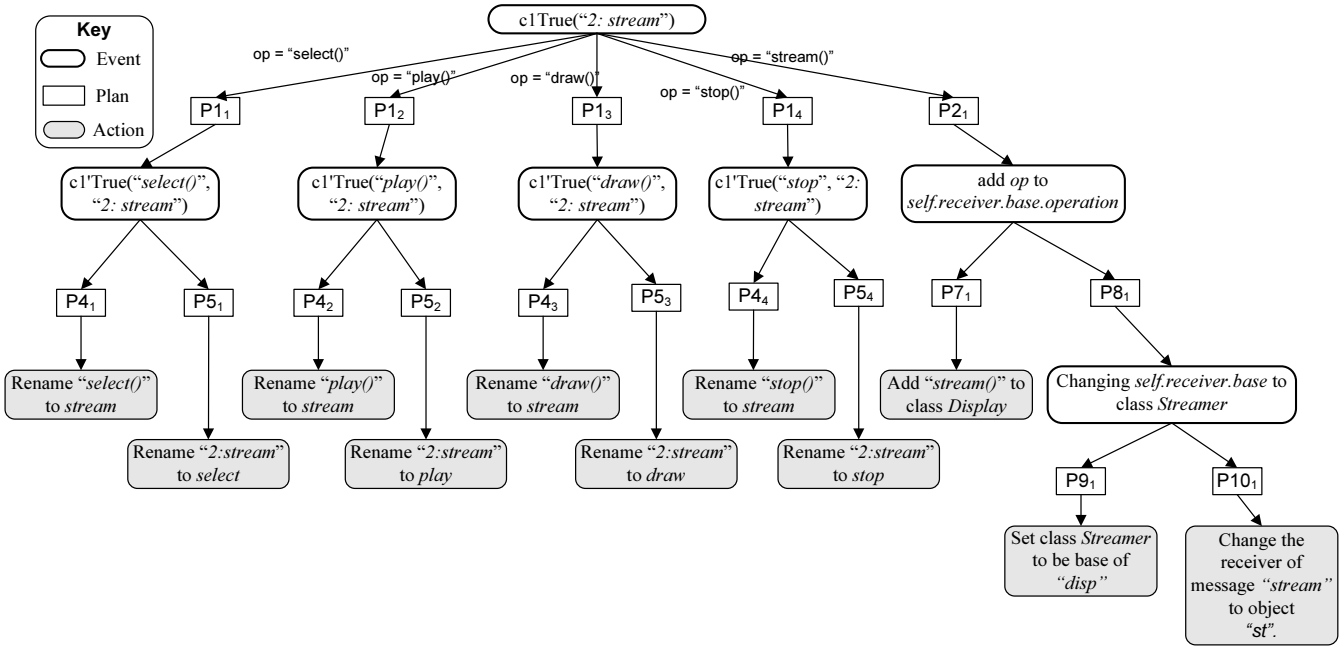

Fig. 8: Repair plan instances for fixing constraint C1("2 : *stream*") with respect to plan types $P1$ and $P2$

Option	Cost
1	1 + C1("1:select") + C1("3:play") + C2("2:stream") + C2("3:play")
2	1 + C1("3:play") + C2("2:stream") + C2("3:play")
3	1 + C1("3:play") + C1("4:draw") + C2("2:stream") + C2("3:play")
4	1 + C1("3:play") + C1("5:stop") + C2("2:stream") + C2("3:play")
5	1 + C1("3:play") + C2("2:stream") + C2("3:play")
6	1 + C1("3:play") + C2("3:play")
7	1 + C1("3:play") + C2("2:stream") + C2("3:play")
8	1 + C1("3:play") + C2("2:stream") + C2("3:play")
9	1 + C1("3:play") + C2("2:stream") + C2("3:play")
10	1 + C1("3:play") + C2("3:play") + C2("2:stream")
11	1 + C1("3:play") + C2("2:stream") + C2("3:play")
12	2 + C1("3:play") + C1("6:wait") + C2("2:stream") + C2("3:play")
13	2 + C1("3:play") + C2("2:stream") + C2("3:play")
14	2 + C1("3:play") + C2("2:stream") + C2("3:play")

TABLE I: The cost of repair options for fixing C1("2:stream")

to the cost of a modification, i.e. 1) plus the cost of fixing violated constraints. Assume that the first repair option is executed, then constraints C1("3:play"), C2("2:stream") and C2("3:play") are still violated. In addition, there is one new violated constraint: C1("1:select") – the message "1:select" does not correspond to any operation in class "Display". Therefore, the cost of the first repair option is 1 plus the costs of fixing constraints C1("1:select"), C1("3:play"), C2("2:stream"), and C2("3:play"). Table I shows the cost of the first repair option (the first row) as well as the cost of other repair options for fixing C1("2:stream").

The first repair option is an example where fixing an inconsistency may result in creating new inconsistencies. Repair option 6 is, on the other hand, an example where fixing an inconsistency may also lead to repairing other inconsistencies. In fact, this repair option fixes not only constraint C1("2:stream") but also constraint C2("2:stream").

In this example, it is easy to see that at this stage repair option 6 gives the cheapest cost. However, in other cases a full simulation, i.e. executing all repair plans, is needed to determine the cheapest repair option. Pruning techniques are applied to improve the performance as well as to detect cycles; this is discussed in more detail in [19, Chapter 7]. We briefly note here that repair option 6 (similarly to other repair options) needs to expand to include plans that fix constraints C1("3:play") and C2("3:play"). We then need to follow the same process to generate repair plan instances and calculate cost for those two constraint instances. However, it is easy to see that a cheapest repair option for both C1("3:play") and C2("3:play") is renaming message "3:play" to "stream".

D. Select one plan to execute and execute plan

After the cost of each repair plan is calculated, the next step is presenting the cheapest plan or plans with their constituent repair actions for user selection. However, since we have done full planning in the previous step, what we present to the user is repair plans that are able to fix not only one constraint but *all* of the relevant constraints, i.e. C1("2:stream"), C1("3:play"), C2("2:stream"), and C2("3:play") in our example. In other words, our approach supports making multiple changes at a time, which may be intuitive from the user's point of view. The user chooses one of the repair plans and the framework will execute the plans to apply (secondary) changes to the model. These changes will make the model become consistent with

respect to the relevant constraints. For instance, in our VOD example there is only one repair plan that has the cheapest cost of 2: renaming message "2:stream" to "play", and renaming message "3:play" to "stream". This repair plan is presented to the user and if it gets chosen it will be executed. The model will then become consistent with respect to constraints C1 and C2.

VI. Scalability

In the previous section, we showed that our framework is able to produce good recommendations for a simple VOD system. Furthermore, our evaluation with agent-oriented designs also showed that our approach generates good recommendations for secondary changes [14]. However, a key question that we need to address is whether our approach scales to larger designs.

Generation of repair plans is performed at "compile" time, and is thus not an issue. At runtime, the following steps are performed: (A) check design for consistency with respect to provided OCL constraints and a metamodel; (B) generate repair plan *instances* for violated constraints; (C) compute cost of different repair options; and (D) execute selected repair plan, where selection is either choosing the cheapest plan or asking the user (if there is more than one cheapest plan). We now consider these steps and argue that our approach is scalable.

For the first step, checking a UML design for consistency against a set of OCL constraints can be done quickly, even for large designs, as shown by Egyed [18].

For the second step, the repair plans that generate many instances are the ones which have a context condition of the form $x \in Type(SE)$ where *SE* is a set of model elements, e.g. plan P2. In some cases these rules will be disabled by the tool developers or tool administrators because they do not make sense: for instance, in our VOD example it is not feasible to add an existing operation to a class if it already belongs to another class. More generally though, it is possible to improve efficiency by being "lazy" and instead of selecting an element from $Type(SE)$, leaving x undetermined and allowing subsequent constraints to narrow down the selection. We leave this for future work.

The third step involves computing the cost of different repair options, and potentially takes the most time since it uses look-ahead. However, as [18] observed, in practice the consistency rules that are used are *local* in scope: the truth of a given constraint is determined by a relatively small number of elements. In particular, this number does not appear to increase as the model grows. A consequence of this local scope observation is that *fixing* a violated constraint (ignoring for the moment the possibility of cascades) also requires only changes to a relatively small number of elements, and hence is scalable.

Let us now consider the possibility of cascades, i.e. where it is possible for fixing a constraint to break other constraints, and hence require further fixes. There are two cases to consider. The first is where there exists a simple local fix. In this case, even when other repair options might involve cascades, these options will not be explored, since they will be pruned in favour of the local (and hence cheaper) options. For both the ATM (discussed below) and the VOD designs, this was the case: given constraint C_1, where a plan for fixing it breaks another constraint C_2, then at worst the cost calculation partially explored C_2, and *never* explored any constraints broken by fixing C_2. The second case to consider is where extensive change propagation is needed, and clearly in this case cascades will occur, and will need to be explored. In this case the tool is arguably of most value, since the secondary changes are non-local, and consequently are difficult to manage manually.

In order to empirically explore the scalability of the third step we used a case study of a larger example: the design of an Automatic Teller Machine (ATM). The initial ATM design (from [7]) covers basic functionalities: the customer inserts his/her card, enters a PIN and then can perform transactions such as withdrawal and deposit; the ATM also prints receipts for all transactions. The design contains a class diagram (18 classes such as ATM, Bank, Transaction) and 10 sequence diagrams (corresponding to 10 use cases such as GetPIN, PrintReceipt). Classes are related by inheritance (6), association (11) and dependency relationships (2). The initial design has been encoded into the prototype that we have implemented.

We introduced the same realistic requirement change as in [7] to the initial design: *the ATM needs to keep track of how many times per session a user attempts to enter the PIN - after 3 invalid PINs the card will be retained*. We also assume that the designer makes some initial changes to the design: adding 3 messages *incrementNumTries*, *resetNumTries* and *displayRetainCard* to the existing sequence diagrams. Our prototype propagated the changes by adding 3 new methods *incrementNumTries*, *resetNumTries* and *displayRetainCard* to relevant classes in the class diagram.

To test our approach, we considered three model sizes: the (original) *full* model; a *medium* model consisting of the main classes that the changes affect and other classes associated with the main classes, as well as sequence diagrams that the main classes participate in; and a *small* model comprising only the main classes and the sequence diagrams that are affected by the initial changes. Figure 9 gives the sizes of these models (top part), the number of plan instances generated[14] for the three constraints[15] (middle part) and the running time[16] for calculating costs (bottom). As can be seen, the execution time increases as the model grows. In particular, the number of plan instances is dominated by plan P2 which produces a plan instance for each operation in the model, and hence the

[14]Note that, because of the cost calculation, the number of options presented to the user is considerably less than this.

[15]Two constraints are C1 and C2 as presented in section II. The other constraint ensures that the message calling direction in a sequence diagram matches the class association in a class diagram.

[16]All experiments reported in this paper were performed on a Mac running OS X v10.4.7 and Java v1.5.0_06, with a 1.67Ghz CPU and 2GB RAM. Tests were run with one user logged in and no applications running (except for the terminal). Times are an average of 9 runs (we ignored the first run, since it was inconsistent due to JVM startup).

execution time grows as the number of operations increases. However, although the number of operations in the model significantly increases (approximately 3 times between model sizes), the execution time does not increase at the same rate. This can be explained by the fact that we do not explore all possible combinations. Rather, as soon as cheaper options are identified, the search tree is pruned off.

A key point is that for the full model the tool took less than a second. Furthermore, the tool's execution time is dominated by constraint checking[17], and its efficiency could be significantly improved by implementing Egyed "instant consistency checking" [18], and by adding "lazy" generation for plan instances with context conditions of the form $x \in Type(SE)$.

	Full	Medium	Small
Model Size:			
Classes	18	10	2
Operations	63	28	8
Sequence Diagrams	10	6	3
Messages	37	20	9
Number of plans:			
c(incrementNumTries)	65	31	11
c(resetNumTries)	63	29	9
c(displayRetainCard)	64	30	10
Total	192	90	30
Runtime (ms)	917.9	538.8	395.8

Fig. 9: ATM case study results

Finally, the fourth step at runtime is executing a selected repair plan which is simply a matter of running the plan and performing the changes, and this is quite cheap in terms of the computational costs, since costs have already been computed for the plan and all sub-plans.

VII. Related Work

Since UML has become the *de facto* notation for object-oriented software development, most research work in consistency management has focused on problems relating to consistency between UML diagrams and models [20]. Such approaches have been advocated with the recent emergence of model-driven evolution [4]. Several approaches (e.g. [21]) strive to define fully formal semantics for UML by extending its current metamodel and applying well-formedness constraints to the model. Other approaches transform UML specifications to some mathematical formalism such as Description Logic [22]. The consistency checking capabilities of such approaches rely on the well-specified consistency checking mechanism of the underlying mathematical formalisms. However, traceability may be a problem: to what extent can a reported inconsistency be traced back to the original model? Furthermore, the identification of transformations that preserve and enforce consistency still remains a critical issue.

[17]Part of the cost calculation involves checking which constraints are violated after a repair plan is simulated.

Recently, Egyed [18] proposed a very efficient approach to check for inconsistencies (i.e. violations of consistency rules) in UML models. His approach scales up to large, industrial UML models by tracking which entities are used to check each consistency rule, and then using this information to determine which rules might be affected by a change, and only re-evaluate these rules. This work is complementary to our work: it provides a rapid means of checking consistency (which supports the first step of our approach), but does not tackle the issue of *how* to restore consistency.

There are approaches that go further than just detecting inconsistencies. Several approaches provide developers with a software development environment which allows for recording, presenting, monitoring, and interacting with inconsistencies to help the developers resolve those inconsistencies [23]. Other works also aim to automate inconsistency resolution by having pre-defined resolution rules (e.g. [6]) or identifying specific change propagation rules for all types of changes (e.g. [7]). However, these approaches suffer from the correctness and completeness issue since the rules are developed manually by the user.

In order to deal with this issue, Nentwich *et al.* [8] proposed an approach for automatically generating repair options by analysing consistency rules expressed in first order logic and models expressed in xlinkit. However, they did not take into account dependencies among inconsistencies and potential interactions between repair actions for fixing them. In other words, their work considers repair actions as independent events, and thus does not explicitly deal with the cascading nature of change propagation.

VIII. Conclusions and Future Work

In this paper, we have presented an approach to support change propagation during the maintenance and evolution of UML design models. Change options are represented in terms of repair plan types which allows us to compactly represent a large number of repair options, and captures nicely the cascading nature of repairing constraint violations, and the way that a given constraint violation may be repaired in a number of ways. We also proposed the use of a notion of repair cost to provide an effective way of accounting for the side-effects of a repair plan.

Using our approach, which has been implemented, tool developers do not need to write resolution rules or choice generation functions and consequently save substantial time. More importantly, they avoid the issues of soundness and completeness, since repair plan types are automatically generated from OCL consistency constraints, and are guaranteed to be sound and complete. Our approach also takes into account the cascading nature of change propagation in terms of considering the side-effects of fixing a given inconsistency.

We have used a simple design of a VOD system to illustrate how our approach can be applied in the context of UML design models. In addition, we performed a case study on the UML design of an ATM system to evaluate the scalability of our approach. Furthermore, we have previously evaluated the

effectiveness and efficiency of our approach in the context of agent-oriented designs [11, 14]. This evaluation showed that the approach did scale to larger designs [11]; and that it was useful for a range of change scenarios [14] drawn from actual changes made to a real agent-based application. In particular, the use of cost was effective in reducing the number of options presented to the user.

The evaluation's results also lead us to some potential future work. Although we have conducted some evaluation to assess scalability, there is a need for more extensive evaluation with larger models. In addition, in order to fully understand the applicability of our approach to UML models, a more extensive investigation involving the whole UML metamodel (including other types of UML diagrams such as use case and activity diagrams) and consistency constraints is needed. Furthermore, an interesting topic for future work is to apply our approach to change propagation between design models and source code. Finally, the approach and tool need to be evaluated with human users to fully assess its effectiveness and usability.

The evaluation assessing the effectiveness of our approach [14] showed that in some cases the tool, which is integrated with the Prometheus Design Tool (http://www.cs.rmit.edu.au/agents/pdt) for supporting agent-oriented design, proposed a large number of repair options, which makes it difficult for the user to decide which one to use. One approach to avoid presenting the user with a long list of repair options is to use "staging" questions, i.e. to ask a series of questions that cumulatively specify the desired repair option.

Assigning costs to basic repair actions (e.g. creation, connection, disconnection, etc.) is a means for the user to adjust the change propagation process. However, our evaluation indicated several places where the outcome may be sensitive to the basic costs. As a topic for future work, we want to perform a more thorough exploration of the effects of varying the basic costs.

Since the tool is still a research prototype, one area of future work is to develop an industry-grade tool. This would include a range of efficiency improvements, such as implementing Egyed's "instant consistency checking" [18].

In summary, although the results obtained from the evaluation we conducted are relatively preliminary and necessarily limited, they are quite encouraging and serve as concrete indication that the approach developed is promising.

REFERENCES

[1] T. Mens, "Introduction and roadmap: History and challenges of software evolution," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer Berlin Heidelberg, 2008, ch. 1, pp. 1–11.

[2] V. Rajlich, "A methodology for incremental changes," in *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Process in Software Engineering*, Cagliary, Italy, May 2001, pp. 10–13.

[3] R. Arnold and S. Bohner, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996, ISBN: 978-0-8186-7384-9.

[4] A. van Deursen, E. Visser, and J. Warmer, "Model-driven software evolution: A research agenda," in *Proceedings 1st International Workshop on Model-Driven Software Evolution (MoDSE)*, D. Tamzalit, Ed. University of Nantes, 2007, pp. 41–49.

[5] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 99–108.

[6] W. Liu, S. Easterbrook, and J. Mylopoulos, "Rule based detection of inconsistency in UML models," in *UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development*, L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, Eds., 2002, pp. 106–123.

[7] L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sowka, "Automated impact analysis of UML models," *Journal of Systems and Software*, vol. 79, no. 3, pp. 339–352, March 2006.

[8] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 455–464.

[9] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a consistency checking and smart link generation service," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 151–185, 2002.

[10] Object Management Group, "Object Constraint Language (OCL) 2.0 Specification," http://www.omg.org/spec/OCL/2.0/PDF/, 2006.

[11] K. H. Dam and M. Winikoff, "Cost-based BDI plan selection for change propagation," in *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller, and Parsons, Eds., Estoril, Portugal, May 2008, pp. 217–224.

[12] K. H. Dam, M. Winikoff, and L. Padgham, "An agent-oriented approach to change propagation in software evolution," in *Proceedings of the Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society, 2006, pp. 309–318.

[13] K. H. Dam and M. Winikoff, "Generation of repair plans for change propagation," in *Agent-Oriented Software Engineering VIII*, ser. Lecture Notes in Computer Science, M. Luck and L. Padgham, Eds., vol. LNCS 4951. Springer Berlin / Heidelberg, April 2008, pp. 132–146.

[14] ——, "Evaluating an agent-oriented approach for change propagation," in *Proceedings of the Ninth International Workshop on Agent Oriented Software Engineering*, M. Luck and J. J. Gomez-Sanz, Eds., Estoril, Portugal, May 2008, pp. 61–72.

[15] A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *MAAMAW '96: Proceedings of the 7th European workshop on Modelling Autonomous Agents in a Multi-Agent World: Agents breaking away*, ser. LNCS, vol. 1038. Springer-Verlag, 1996, pp. 42–55.

[16] A. Egyed, "Fixing inconsistencies in UML models," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 292–301.

[17] Object Management Group, "Unified Modeling Language (UML) 1.4.2 specification (ISO/IEC 19501)," http://www.omg.org/spec/UML/ISO/19501/PDF/, 2005.

[18] A. Egyed, "Instant consistency checking for the UML," in *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: ACM, 2006, pp. 381–390.

[19] K. H. Dam, "Supporting software evolution in agent systems," Ph.D. dissertation, RMIT University, School of Computer Science and IT, 2009.

[20] L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, Eds., *UML 2003, Modeling Languages and Applications. Workshop on Consistency Problems in UML-based Software Development II*, no. 2003:06. Ronneby: Blekinge Institute of Technology, 2003.

[21] J.-P. Bodeveix, T. Millan, C. Percebois, C. L. Camus, P. Bazex, and L. Feraud, "Extending OCL for verifying UML models consistency," in *UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development*, L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, Eds., no. 2002:06. Ronneby: Blekinge Institute of Technology, 2002, pp. 75–90.

[22] T. Mens, R. V. D. Straeten, and J. Simmonds, "A framework for managing consistency of evolving UML models," in *Software Evolution with UML and XML*, H. Yang, Ed. Idea Group Publishing, 2005, pp. 1–31.

[23] J. Grundy, J. Hosking, and W. B. Mugridge, "Inconsistency management for multiple-view software development environments," *IEEE Transactions on Software Engineering*, vol. 24, no. 11, pp. 960–981, 1998.