Generation of Repair Plans for Change Propagation

Khanh Hoa Dam and Michael Winikoff

School of Computer Science and Information Technology,
RMIT University,
GPO Box 2476V, Melbourne, VIC 3001, Australia
{kdam,winikoff}@cs.rmit.edu.au

Abstract. One of the most critical problems in software maintenance and evolution is propagating changes. Although many approaches have been proposed, automated change propagation is still a significant technical challenge in software engineering. In this paper we present an agent-oriented change propagation framework based on fixing inconsistencies when primary changes are made to design models. A core piece of the framework is a new method for generating repair plans from OCL constraints that restrict these models.

1 Introduction

Software evolution is critical in the life-cycle of successful software systems, especially those operating in highly volatile domains such as banking, e-commerce and telecommunications. A basic operation of software evolution is change: in order to adapt the system to the desired requirements (be they new, modified, or an environmental change) the system is changed [1]. In practice, the software engineer usually starts making some primary changes that he/she can easily identify based on the characteristics of the change requests and/or his/her knowledge and expertise. However, these primary changes are not enough to make the design meet the change requests and may also create inconsistencies. As a result, additional, secondary, changes might be needed.

The process of determining and making secondary changes is termed *change propagation*. Since complex software systems consist of many artefacts, both design and code, and since there are usually many options when making secondary changes, change propagation is a complicated, labour-intensive, and expensive process. Hence, there is a need for tools that provide more effective *automated* support for change propagation. We do not believe that change propagation can be fully automated, since there are decisions that involve tradeoffs where human expertise is required. However, it is possible to provide tool support to assist with tracking dependencies, determining what parts of the system are affected by a given change, and, as in this paper, determining and making secondary changes.

We are basing our work on the conjecture that, given a suitable set of consistency constraints, *change propagation can be done by fixing inconsistencies in a design*. In other words, we propagate changes by finding places in a design where the desired consistency constraints are violated, and fixing them until no inconsistency is left in the design. For example, an agent type is added, and then consequently other agents need to be modified to communicate with the new agent type.

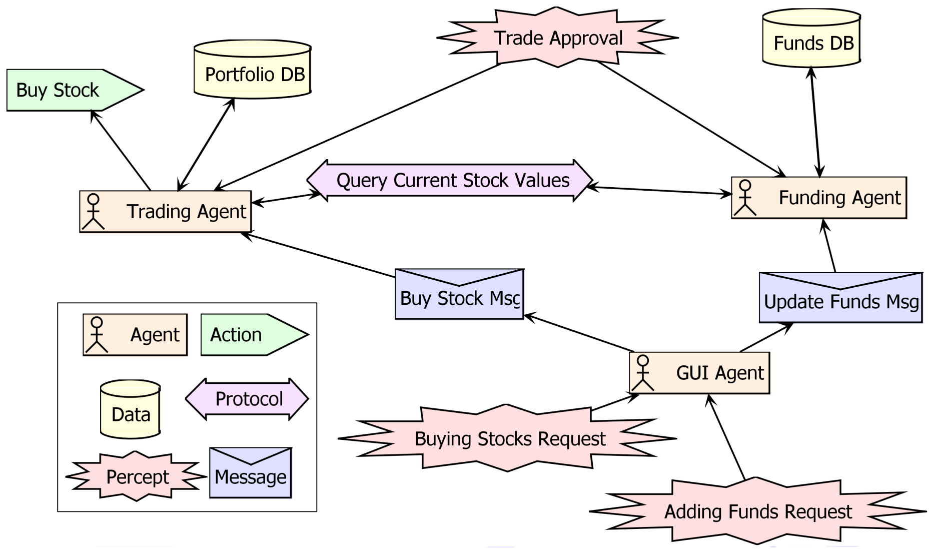


Fig. 1. System overview diagram for a stock trading management system

According to a survey in [2], handling inconsistencies has received much attention in mainstream software engineering. However, most existing work either fails to advocate effective automation (e.g. [3, 4]) or fails to reflect the cascading nature of repairing inconsistencies (e.g. [5]). In agent-oriented software engineering, there has not been much work addressing the maintenance aspect, especially the change propagation issue in the development of agent designs. In our previous work [6], we have shown how an *agent-oriented* approach, specifically the BDI agent architecture, is a suitable approach for performing change propagation and illustrated its capacity to deal with consistency management in the context of the *Prometheus* methodology [7]. In this paper we present a framework for change propagation which extends our previous work (section 3) and introduce a mechanism for automatic repair plan generation from Object Constraint Language (OCL) [8] constraints (section 4). We then discuss some related work (section 5) before concluding and outlining our future work (section 6).

2 A running example

Throughout this paper we use a running example which comprises an initial design for a simple stock trading management system (STMS). The design was developed using the Prometheus methodology [7]. Figure 1 shows a system overview diagram of the existing STMS. The system currently has three agents: a "*GUI Agent*" for handling users' requests such as buying stock or adding funds, a "*Funding Agent*" responsible for managing users' funds, and a "*Trading Agent*" for performing transactions such as buying stocks. In addition, "*GUI Agent*" has the "*Handling adding-funds request*" plan triggered by "*Adding Funds Request*" percept and the "*Handling buying-stocks request*" plan triggered by "*Buying Stocks Request*" percept.

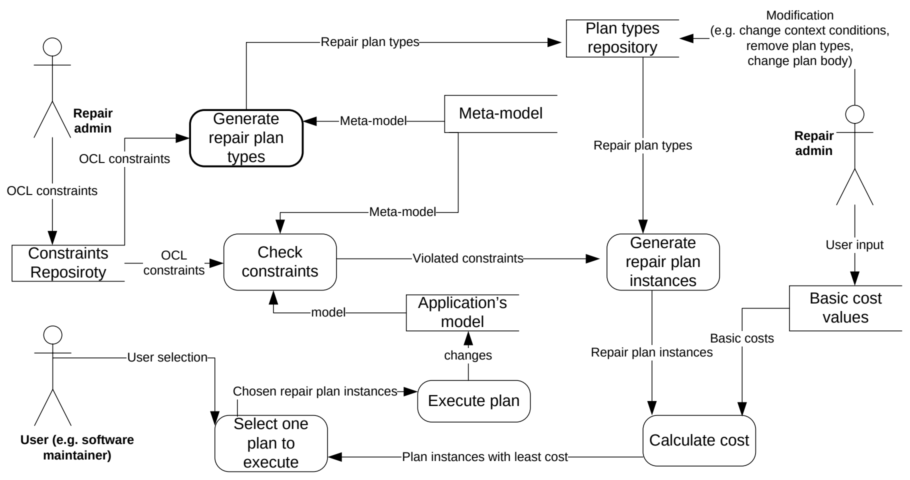


Fig. 2. Change propagation framework

The initial requirements for the STMS, however, only deal with buying stocks and adding funds. Now suppose that the clients have asked to add a new functionality to the system: *STMS should also allow the users to place selling stock orders*. Assume that the software designer begins making some primary changes by adding a new percept *"Selling Stocks Request"* and assigning the "*GUI Agent*" to handle it. At this point the designer may start wondering what are the next changes that they need to make. The following sections present our change propagation framework and show how it supports the software engineer in propagating changes.

3 Architectural overview

This section serves to introduce the architecture of the change propagation framework. Figure 2 shows an overview of our architecture as a data flow diagram. At design time, consistency constraints that are created by the repair administrator are input to the change propagation framework along with a meta-model. The meta-model and constraints can be developed by extracting relationships and dependencies from the methodology that we want to apply the framework to. For instance, a Prometheus meta-model and a set of related constraints have been developed in [6]. Normally the meta-model and constraints are developed once for a given methodology (e.g. Prometheus) and then reused: the user (software maintainer) is not required to develop a meta-model or OCL constraints, although in some cases they may desire to add additional domain or application specific constraints. Figure 3 shows an excerpt of the Prometheus meta-model (refer to [6] for a full version) represented as a UML class diagram capturing the relationships between an agent and other entities. The meta-model shows that an agent can contain one or more plans. A plan can send and/or receive messages as well as perform some actions which may include accessing data to handle a percept or to achieve some goals. As a result, agents also have these associations with goals, percepts, actions, messages, and data[1].

The exact relationships between agents, plans and other entities are expressed using a set of Object Constraint Language (OCL) constraints. OCL [8] is part of the UML standards which is used to specify invariants, pre-conditions, post-conditions and other kinds of constraints imposed on elements in UML models. Below is an example of an OCL constraint that defines the semantics of relationships between agents, plans and percepts. In the OCL notation "self" denotes the context node (in this case a Percept) to which the constraints have been attached and an access pattern such as "self.agent" indicates the result of following the association between a percept and an agent, which is, in this case, a collection of agents which handle the percept. OCL also denotes operations on collections such as "$SE \rightarrow \text{includes}(x)$" stating that a collection *SE* must contain an entity *x*, or "$SE \rightarrow \text{exists}(c)$" specifying that a certain condition *c* must hold for at least one element of *SE*, or "$SE \rightarrow \text{forAll}(c)$" specifying that *c* must hold for all elements of *SE*. For detailed information on OCL see [8]. For example, the following constraint, which could be expressed in more traditional form as $\forall a \in \text{self.agent}\, \exists pl \in a.\text{plan} : \text{self} \in pl.\text{percept}$, states that: considering the set of agents that handle the percept (*self.agent*), for each of the agents (*a*) if we consider the plans of that agent (*a.plan*) then one of these plans (*pl*) must include the current percept (*self*) in its list of percepts (*pl.percept*).

Constraint 1 *Any agent that handles a percept should contain at least one plan that is triggered by the percept.*
Context Percept inv*:*
self.agent→forAll(a : Agent | a.plan→exists(pl : Plan | pl.percept→includes(self)))

The repair plan generator takes the constraints and the meta-model as inputs, and returns a parameterized set of event-triggered repair plan types[2] that are able to repair violations of the constraint. Our translation schema guarantees completeness and correctness, i.e. there are no repair plans to fix a violation of a constraint other than those produced by the generator; and any of the repair plans produced by the generator can fix a violation. However, we also allow the repair administrator to use their domain knowledge and expertise to modify generated repair plans or remove plans that should not be executed. In section 4 we discuss this in more detail. The set of repair plan types is created ahead of time and forms the library of plans that the change propagation engine uses to fix constraint violations. Since the library of plans is derived before runtime, the efficiency of deriving it is not crucial.

At runtime, the change propagation engine checks the current design models against the OCL constraints, and any violations of these constraints are fixed using the repair plans. The engine is represented and implemented using the BDI agent architecture.

[1] In figure 3, associations marked with an 'A*' are between agent and plan, and goal, percept, action, data, and message. We group and represent them as a single association for readability.

[2] The use of plans which are triggered by events is taken from the well-known and studied Belief Desire Intention (BDI) agent architecture [9], which has been widely implemented within the agents community (e.g. see [10]).

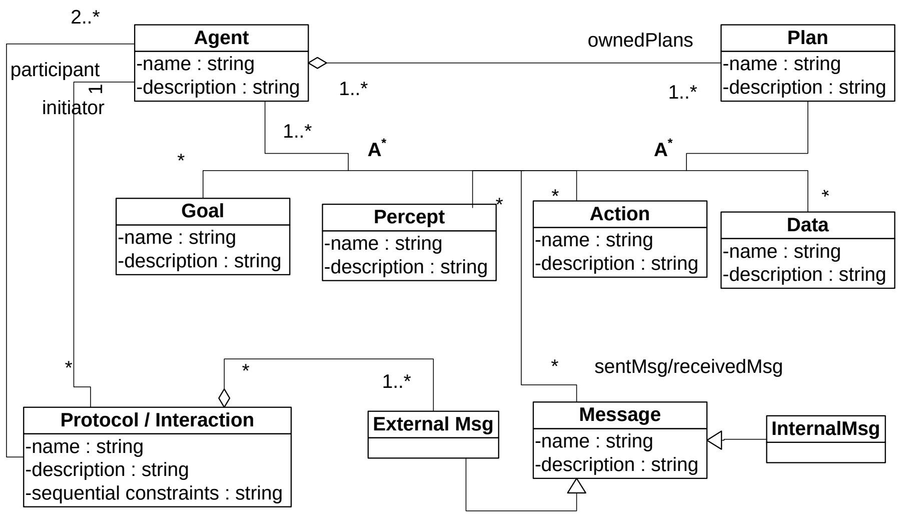


Fig. 3. An excerpt of the Prometheus meta-model

Constraint checking may result in the detection of a number of violated constraints. Each violated constraint is then posted as an event ("ViolationEvent"). A given *ViolationEvent* may trigger a number[3] of possible repair plan instances. In order to help select which repair plan instance to use we calculate the cost of each repair plan instance. We recognize that fixing one violated constraint may also repair or violate others as a side effect, and so the cost calculation algorithm computes the cost of a given repair plan instance as including the cost of its actions (using basic costs assigned by the repair administrator), the cost of any other plans that it invokes directly, and also the cost of fixing any constraints that are made false by executing the repair plan. If there are several equal least cost plans, they are presented to the user, otherwise the cheapest plan is selected. Once a plan is selected, it is then executed to fix the violation, and hence propagate changes. We allow the repair administrator to specify the repair cost for each basic repair action. The repair administrator may use this mechanism to adjust the change propagation process. For example, if he/she wishes to bias the change propagation process towards adding more information then he/she may assign lower costs to actions that create new entities or add entities, and higher costs to actions that delete entities.

The above change propagation framework is currently being implemented. The cost calculation and plan execution components were implemented and used to test some small case studies. At the time of writing this paper, we are at the final stage of completing the implementation of the plan generator component. Due to space limitations, in the remainder of this paper we focus on describing the repair plan generator compo-

[3] Which will always be greater than zero, due to the way in which plans are generated (see section 4).

$$\begin{array}{lcl} P & ::= & E[:\, C] \leftarrow B \\ C & ::= & C \vee C \mid C \wedge C \mid \neg\, C \mid \forall x \bullet C \mid \exists x \bullet C \mid \mathit{Prop} \\ B & ::= & \mathit{true} \mid \mathit{Add\ Entity\ To\ SE} \ \mid \mathit{Remove\ Entity\ From\ SE} \mid \mathit{Create\ Entity} : \mathit{Type} \mid \\ & & \mathit{Change\ Property\ to\ Property} \mid \mathit{if}\ C\ \mathit{then}\ B\ \mid !E\ \mid\ B_1;\ B_2 \mid \mathit{for\ each}\ x\ \mathit{in}\ SE\ B \end{array}$$

Fig. 4. Repair plan abstract syntax

nent of the framework. However, we will also briefly illustrate how the cost calculation component works based on the STMS example.

4 Generating repair plan types

Since a large design can contain a substantial number of constraints, the number of repair plans may be very large. In these cases, hand-crafting repair plans for all constraints becomes a labour intensive task. In previous work [6], we encountered this issue when developing repair plans for Prometheus design models. In addition, it is difficult for the repair administrator to know if the set of repair plans which they create is complete and correct. Therefore, we have developed a translation schema that takes as input constraints, expressed as OCL invariants, and generates repair plans that can be used to correct constraint violations.

A similar approach has been proposed in [5] which takes xlinkit rules defining consistency between documents and returns a set of repair actions. One key difference between their work and ours is that we generate abstract, structured, repair plans that are instantiated at runtime.

Our syntax for repair plans (see figure 4) is based on AgentSpeak(L)[4] [11], but with some differences (most notably in specifying the actions, and in allowing for richer plan bodies). Each repair plan, P, is of the form $E : C \leftarrow B$ where E is the triggering event (conceptually, the name of the constraint P is fixing, subscripted with either t or f to indicate whether the constraint is being made true or false); C is an optional "context condition" (Boolean formula[5]) that specifies when the plan should be applicable[6]; and B is the plan body. The plan body can contain primitive actions such as adding and deleting entities and relationships, and changing properties. The plan body can also contain sequences ($B_1;\ B_2$), conditionals and loops, and events which will trigger further plans ($!E$).

For a given constraint form, for example $c_1 = \mathit{not}\ c_2$, we generate repair plans that repair c_1 (make it true). These plans are defined in terms of other plans that repair sub-constraints. In this example, the plan to repair c_1, i.e. make it true, is defined using

[4] We chose to use AgentSpeak(L) as a basis because it is a simple and compact notation that captures the essence of BDI-based agent oriented programming languages.

[5] "Prop" denotes a primitive condition such as checking whether $x > y$ or whether $x \in SE$.

[6] In fact when there are multiple solutions to the context condition, each solution generates a new plan instance. For example, if the context condition is $x \in \{1, 2\}$ then there will be two plan instances.

plans that make c_2 false. Formally, if we use $\mathcal{R}(c)$ to denote the complete set of repair plans for constraint c, and $\mathcal{P}(c)$ to denote the specific plans for the constraint form, then for a constraint c_1 with sub-constraint c_2 (as is the case for $c_1 = \mathit{not}\ c_2$) we have that: $\mathcal{R}(c_1) = \mathcal{P}(c_1) \cup \mathcal{R}(c_2)$. More generally, if constraint c has sub-constraints $s(c) = \{c_1, \ldots, c_n\}$ then

$$\mathcal{R}(c) = \mathcal{P}(c) \cup \bigcup_{c' \in s(c)} \mathcal{R}(c')$$

Figure 5 gives the definition of the $\mathcal{P}$ function: for each rule the generation of $\mathcal{R}(c)$ will include the given plans, as well as the plans obtained from the sub-constraints. The definition of $\mathcal{P}(c)$ considers, for each case, all the possible ways in which c can be false, and all the possible ways in which it can be repaired. For example, if $c = \mathrm{SE} \rightarrow$ includes(x) then c is false if and only if x is not an element of SE, and consequently c can be made true only by adding x to SE.

If a constraint has more than one sub-constraint then fixing a sub-constraint might conflict with the plan that repairs the other sub-constraint. For example, if $c = c_1$ and c_2 then fixing c_2 may, depending on the definition of c_2 and c_1, make c_1 false. Therefore, in order to guarantee that the generated repair plans always correctly fix the constraint, we include two top level plans into $\mathcal{P}(c)$. For making a constraint c true, in addition to the definition in figure 5, $\mathcal{P}(c)$ includes $\{\mathrm{fixC} : \mathrm{c} \leftarrow \mathrm{true}, \mathrm{fixC} : \neg\, \mathrm{c} \leftarrow c_t\ ;\ \mathrm{fixC}\}$.

The figure shows an excerpt of the translation rules from OCL constraints such as *includes*, *includesAll*, *excludes*, *excludesAll*, *isEmpty*, *notEmpty*, and logical connectives (*and*, *or*, *not*, *implies*, *forAll*, *exists*). Note that for each constraint c we generate rules that make that constraint true c_t and rules that make it false c_f, however, for space reasons, figure 5 only shows the c_t rules. Generally speaking the rules for generating c_f follow inverse patterns to those for c_t. For example, for $c = SE \rightarrow$ includes(x) the rule for c_f is $c_f \leftarrow$ Remove x from SE.

The term "SE" denotes a set expression which can be the name of a collection, or a derived collection, built from another collection using an operator. For example, the set $S_2 = S_1 \rightarrow$ select(c) includes elements from the set S_1 for which the condition c holds. Although adding an element to a basic collection (or removing it from the set) is a primitive action, adding or deleting elements from derived collections is not a primitive action. Instead, we model addition and deletion from derived collections using additional plans. For example, in order to add the element x to $SE = S \rightarrow$ select(c) we need to ensure both that x is in S, and that $c(x)$ is true; on the other hand if we want to *remove* x from SE then we can either remove x from S, *or* make $c(x)$ false. Similarly, if $SE = S \rightarrow$ intersection($S2$) then to add x to SE we need to ensure that x is added to both S and $S2$; and to remove x from SE we can either remove it from S or from $S2$. We have developed rules for operations including *select*, *reject*, *union*, *intersection*, *minus* and *symmetricDifference*. The complete set of rules to generate repair plans from OCL constraints can be found in [12].

4.1 Completeness and correctness

The translation guarantees correctness and completeness since it is developed by considering all the possible ways in which a constraint can be false, and hence all the

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{SE{\rightarrow}includes(x)}) =$
$\{c_t \leftarrow \text{Add x to SE}\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{SE{\rightarrow}includesAll(SE')}) =$
$\{c_t \leftarrow \text{for each x in (SE' - SE)} \; !c'_t(\text{x}),$
$c'_t(\text{x}) \leftarrow \text{Remove x from SE'},$
$c'_t(\text{x}) \leftarrow \text{Add x to SE}\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{SE{\rightarrow}excludes(x)}) =$
$\{c_t \leftarrow \text{Remove x from SE}\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{SE{\rightarrow}excludesAll(SE')}) =$
$\{c_t \leftarrow \text{for each x in SE} \cap \text{SE'} \; !c'_t(\text{x})$
$c'_t(\text{x}) \leftarrow \text{Remove x from SE'},$
$c'_t(\text{x}) \leftarrow \text{Remove x from SE}\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{SE{\rightarrow}notEmpty}()) =$
$\{c_t : \text{x} \in \text{Type(SE)} \leftarrow \text{Add x to SE},$
$c_t \leftarrow \text{Create x : Type(SE) ; Add x to SE}\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{SE{\rightarrow}isEmpty}()) =$
$\{c_t \leftarrow \text{for each x in SE Remove x from SE}\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{SE{\rightarrow}forAll(c1)}) =$
$\{c_t \leftarrow \text{for each x in SE if} \neg \text{c1(x) then} \; !c'_t(\text{x}),$
$c'_t(\text{x}) \leftarrow \text{Delete x from SE},$
$c'_t(\text{x}) \leftarrow !c1_t(\text{x})\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{SE{\rightarrow}exists(c1)}) =$
$\{c_t : \text{x} \in \text{SE} \leftarrow !c1_t(\text{x}),$
$c_t : \text{x} \in \text{Type(SE)} \wedge \text{x} \notin \text{SE} \leftarrow \text{Add x to SE ;}$
$!c1_t(\text{x}),$
$c_t \leftarrow \text{Create x : Type(SE) ; Add x to SE ;} \; !c1_t(\text{x})$
$\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{c1 \text{ or } c2}) = \{c_t : \neg \text{c1} \leftarrow !c1_t,$
$c_t : \neg \text{c2} \leftarrow !c2_t\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{c1 \text{ and } c2}) =$
$\{c_t : \neg \text{c1} \wedge \text{c2} \leftarrow !c1_t,$
$c_t : \neg \text{c2} \wedge \text{c1} \leftarrow !c2_t,$
$c_t : \neg \text{c1} \wedge \neg \text{c2} \leftarrow !c1_t \; ; \; !c2_t\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{not \; c1}) = \{ c_t \leftarrow !c1_f\}$

$\mathcal{P}(\mathbf{c} \overset{\text{def}}{=} \mathbf{c1 \text{ implies } c2}) = \{ c_t \leftarrow !c1_f, c_t \leftarrow !c2_t\}$

Fig. 5. Plan generation rules (c_t)

possible ways in which it can be made true. In this section we provide an excerpt of the proof of correctness and completeness of the plan generator. The full proof can be found in [12].

In order to ease the discussion ahead, we provide here a definition of a *correct* repair plan and a *complete* set of repair plans.

Definition 1 (Correct plan) *A repair plan P correctly fixes a violated constraint c if and only if when P finishes its execution (i.e. all actions are performed and subgoals are achieved), c becomes valid.*

As mentioned earlier, the selection of applicable repair plans is based on the notion of costs. As a result, when repair plans are generated at compile time, we only focus on plans that have no redundant steps in fixing a particular constraint. Such plans are considered as *minimum plans* which are defined as below.

Definition 2 (Minimum plan) *A repair plan P for fixing constraint c is said to be a minimum plan if and only if all of its actions (obtained by running the plan) contribute towards fixing c, i.e. taking out any of the actions results in failing to fix c.*

Given the definitions of correct and minimum plans, a complete set of repair plans is defined as below.

Definition 3 (Complete set of repair plans) *A set of repair plans $\mathcal{R}(c)$ for a constraint c is said to be complete if and only if $\mathcal{R}(c)$ contains all minimum correct repair plans for c.*

Based on the above definitions, the correctness and completeness of our translation scheme are expressed by the following theorem.

Theorem 1. *For any given OCL constraint c the set of repair plans $\mathcal{R}(c)$ produced by the plan generator is correct and complete i.e. it contains all the possible correct minimum plans. In other words, there are no minimum plans to fix c that do not belong to $\mathcal{R}(c)$; and any of the repair plans in $\mathcal{R}(c)$ can fix c and are minimal.*

We will prove, by induction, that the above theorem holds with respect to the translation schema for c_t (figure 5).

First of all, we will prove that theorem 1 holds for all the *basic* OCL constraints that we cover. This is relatively easy since repair plans are generated by considering all the possible ways in which a constraint can be false. In [12], we provide a proof of the theorem for all basic OCL constraints that we consider in figure 5. Due to space limitation, we provide here a detailed proof for a typical example: c $\stackrel{\text{def}}{=}$ SE→includesAll(SE').

The above constraint can be written as:

$$\begin{aligned} c &\stackrel{\text{def}}{=} \forall x \bullet x \in SE' \Rightarrow x \in SE \\ &\stackrel{\text{def}}{=} \forall x \bullet (\neg\, x \in SE') \vee (x \in SE) \\ &\stackrel{\text{def}}{=} \forall x \bullet x \notin SE' \vee x \in SE \end{aligned}$$

Assume that c is violated, i.e. $\neg\, c$ is true, expressed as follows:

$$\begin{aligned} \neg\, c &\stackrel{\text{def}}{=} \neg\, \forall x \bullet x \notin SE' \vee x \in SE \\ &\stackrel{\text{def}}{=} \exists x \bullet \neg\, (x \notin SE') \wedge \neg\, (x \in SE) \\ &\stackrel{\text{def}}{=} \exists x \bullet x \in SE' \wedge x \notin SE \end{aligned}$$

Therefore, to prevent $\neg\, c$ from being true (or c from being false) we either delete x from SE' (to make $x \in SE'$ false) or add x to SE (to make $x \notin SE$ false). In other words, there is *exactly* one way to fix c when it is violated: for each of the elements in SE' but not in SE, either delete it from SE' or add it to SE. This is also the minimum way of fixing c, i.e. it does not involve removing or adding any redundant elements. As can be seen in figure 5, the repair plan set $\mathcal{R}(c)$ is $\{c_t \leftarrow$ for each x in (SE' - SE) !c'_t(x), c'_t(x) $\leftarrow$ Remove x from SE', c'_t(x) $\leftarrow$ Add x to SE$\}$. The three repair plans exactly address the fixing approach for c that we mentioned earlier. Therefore, we can conclude that $\mathcal{R}(c)$ contains minimum correct plans and that it is complete.

An OCL constraint is ultimately a combination (*and*, *or*, *not*, *xor* and *implies*) of basic constraints. We have proved that theorem 1 holds for all basic constraints. We now use that to prove, by induction, that theorem 1 holds for the basic connectives: *and*, *or*, and *not*. The other connectives (*xor* and *implies*) can be derived from the basic ones. Below is a proof for the *or* connective. For the others, please refer to [12].

For c $\stackrel{\text{def}}{=}$ c1 or c2, assume that theorem 1 holds for $\mathcal{R}(c1)$ and $\mathcal{R}(c2)$, i.e. both of them are correct and complete sets. Now we need to prove that it also holds for $\mathcal{R}(c)$. According to figure 5, we have:

$$\mathcal{P}(c) = \{c_t : \neg\ c1 \leftarrow !c1_t, c_t : \neg\ c2 \leftarrow !c2_t\}$$

and we also have:

$$\mathcal{R}(c) = \mathcal{P}(c) \cup \mathcal{R}(c1) \cup \mathcal{R}(c2)$$

Because of our induction assumption, $c1_t$ and $c2_t$ can fix c1 and c2 respectively. Therefore, plan $c_t : \neg$ c1 $\leftarrow !c1_t$ is able to repair c1 and plan $c_t : \neg$ c2 $\leftarrow !c2_t$ is able to repair c2. Since the constraint c holds if either of c1 or c2 holds, any plan that is able to fix c1 or c2 can fix c. As a result, we can conclude that $\mathcal{R}(c)$ contains plans that correctly fix c. These plans are also minimum because they do not contain redundant repair actions. For instance, plan $c_t : \neg$ c1 $\leftarrow !c1_t$ fixes only c1 when c1 is false, which is just sufficient to repair c without the need to fix c2.

We have proved that $\mathcal{R}(c)$ contains correct and minimum repair plans for c. Now we prove the completeness of the set $\mathcal{R}(c)$. Assume that there is a minimum plan P that fixes c and does not belong to $\mathcal{R}(c)$. Plan P should aim to fix either c1 or c2 and without loss of generality we assume that P aims to fix c1. Therefore, plan P is also the minimum plan for fixing c1, which results in, due to the induction assumption, that P belongs to $\mathcal{R}(c1)$. Since $\mathcal{R}(c)$ contains $\mathcal{R}(c1)$, P also belongs to $\mathcal{R}(c)$, which contradicts our previous assumption. Hence, there does not exist any minimum plan P that fixes c and does not belong to $\mathcal{R}(c)$, i.e. the set $\mathcal{R}(c)$ is complete.

The induction proof above shows that the generated repair plans of a constraint correctly fix the constraint. However, there are special cases in which repair plans for fixing sub-constraints conflict with each other. For instance, for c $\stackrel{\text{def}}{=}$ c1 and c2 the generated repair plans are:

$$\begin{aligned}\mathcal{P}(c) = \{&\mathit{fixC} : c \leftarrow \mathit{true}, \mathit{fixC} : \neg\ c \leftarrow c_t;\ \mathit{fixC},\\ &c_t : \neg\ c1 \wedge c2 \leftarrow !c1_t, c_t : \neg\ c2 \wedge c1 \leftarrow !c2_t, c_t : \neg\ c1 \wedge \neg\ c2 \leftarrow !c1_t;\ !c2_t\}\end{aligned}$$

Assume that c is false because c1 is true and c2 is false, then *fixC* calls the plan aiming to fix c2, i.e. $c_t : \neg$ c2 $\wedge$ c1 $\leftarrow !c2_t$. However, this plan may make c1 become false, which results in c still being false. Since *fixC* is called recursively until c becomes true, the plan aiming to fix c1 is called, i.e. $c_t : \neg$ c1 $\wedge$ c2 $\leftarrow !c1_t$. However, this plan may also make c2 false, in which case the plan aiming to fix c2 is called and this may continue as a loop. In general, if it is not possible to make both c1 and c2 true at the same time, i.e. every plan that fixes c1 violates c2 and vice versa, then c is not satisfiable. If c is satisfiable, then there exists a repair plan that is able to fix c. In this case, our cost algorithm (discussed in section 3) will favor that repair plan over any other repair plans that causes an infinite loop.

Overall, we can conclude that our generated repair plans for a constraint correctly fix it if the constraint is satisfiable. Our cost algorithm is able to detect infinite loops caused by conflict between repair plans fixing sub-constraints.

4.2 Example

Now let us consider a simple example of how repair plans are generated for the constraint previously presented in section 3.
Context Percept inv:
self.agent→forAll(a : Agent | a.plan→exists(pl : Plan | pl.percept→includes(self)))

We denote the above constraint as c(self), and c_t(self) is the event of making c(self) true. We also define the following abbreviations:

c1(self, a) $\overset{\text{def}}{=}$ a.plan→exists(pl : Plan | pl.percept→includes(self))
c2(self, pl) $\overset{\text{def}}{=}$ pl.percept→includes(self)

Our repair plan generator produces the following repair plans for constraint c, since it has the form $SE \rightarrow \text{forAll}(c)$.

c_t(self) ← for each a in self.agent if ¬ $c1_t$(self, a) then !c'_t(self, a) **(P1)**
c'_t(self, a) ← Delete a from self.agent **(P2)**
c'_t(self, a) ← !$c1_t$(self, a) **(P3)**

For constraint $c1$ we generate the following plans, since the constraint is of the form $SE \rightarrow \text{exists}(c)$. In the rules of figure 5 "Type(SE)" denotes the type of SE's elements, in this case SE (which is *a.plan*) contains plans, and therefore in P5 the context condition requires that *pl* be an element of the set of all plans, denoted Set(Plan).

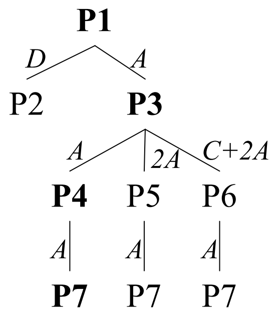
$c1_t$(self, a) : pl ∈ a.plan ← !$c2_t$(self, pl) **(P4)**
$c1_t$(self, a) : pl ∈ Set(Plan) ∧ pl ∉ a.plan ← Add pl to a.plan ; !$c2_t$(self, pl) **(P5)**
$c1_t$(self, a) ← Create pl : Plan ; Add pl to a.plan ; !$c2_t$(self, pl) **(P6)**

Similarly, for constraint $c2$ we generate the following plan.

$c2_t$(self, pl) ← Add self to pl.percept **(P7)**

The above repair plan types are instantiated with actual variable bindings at run-time to produce different plan instances. For instance, in our STMS example previously described in section 2, after the software engineer performs the primary changes, the constraint c(self) is violated where *self* is the "*Selling Stock Request*" percept because there is only one agent handling it (*self.agent* = {"*GUI Agent*"}), and that agent contains two plans (*a.plan* = {"*Handling adding-funds request*","*Handling buying-stocks request*"}) but neither of the plans is triggered by the percept. The plans to repair the constraint are either not assigning "*GUI Agent*" to handle the "*Selling Stock Request*" percept (P2), or having one of the agent's plans be triggered by the percept (P3). Plan P3 then produces three alternatives: choosing an existing plan of the agent (which is either "*Handling adding-funds request*" or "*Handling buying-stocks request*) and make the "*Selling Stock Request*" percept be one of its triggers (P4 and P7); or choosing an existing plan in the design other than the two already in the agent and make the percept be one of its triggers (P5 and P7); or creating a new plan, adding it to "*GUI Agent*", and make the percept be one of its triggers (P6 and P7).

As discussed in section 3, we calculate the cost for each repair plan and present a list of equal least cost plans to the user. Let A denote the cost of an addition, C the cost of creating a new entity, and D denote the cost of a deletion. Note that we allow the repair administrator to define these elementary costs. He/she may use this mechanism to adjust the change propagation process as discussed in section 3. Then, considering the tree of plans below, the cost of P7 is the cost of adding the percept to plan pl's percepts (i.e. A), the cost of P4 is just the cost of P7 (i.e. A), the cost of P5 is the cost of P7 plus the cost of adding plan pl to the plans of agent a (i.e. $2 \times A$), and the cost of P6 is that plus the cost of creating a new plan (i.e. $C + 2 \times A$). Since any of P4, P5 or P6 can be used to fix $c1_t$, and the system picks the cheapest, the cost of $P3$ is the cost of P4 (i.e. A). The cost of P2 is D, and if we assume that deletion is more expensive than addition, then the cost of P1 is A, and the repair plan selected involves P1, P3, P4 and P7 (indicated in bold).



All plan types generated by our repair system are stored in a repository. As we have noted earlier, the repair administrator is able to modify the generated repair plan types, including modifying the plans' context conditions, modifying the plans' body or even adding additional plans or removing generated repair plans. For example, the repair administrator may think that it does not make sense in practice to have a percept that is not handled by any agent. Therefore, he/she may add a context condition into plan P2 specifying that it is applicable only if the set *self.agent* contains at least two elements.

5 Related work

The issue of assisting software engineers to deal with software changes has received much attention in the areas of software evolution and maintenance. Change impact analysis has been extensively investigated, but is only loosely related to our work. Change impact analysis techniques [13] aim to assess the extent of the change, i.e. the artefacts, components, or modules that will be impacted by the change, and consequently how costly the change will be. Our work is more focused on *implementing* changes by propagating changes between design artefacts in order to maintain consistency as the software evolves.

There has been a range of work using a *rule-based* approach to detect and resolve inconsistencies (or constraint violations) both in the areas of databases and software engineering. In these approaches, rules are defined in terms of constraints and actions in such a way that if a constraint is violated, actions will be performed to repair the violation. The work in the area of databases focuses on integrity constraint maintenance [14], i.e. making changes to transactions or databases to recreate a state of integrity. There has been some work which addresses how repair actions can be automatically generated from constraints expressed in first order logic in relational, active and deductive databases [15–17]. One key difference between their work and ours is that we generate abstract, structured, repair plans that are instantiated at runtime. The approaches proposed in [15, 16] are similar to ours in which they involve user intervention in selecting repair actions. In contrast, in [17] a system is described that generates actions from closed, range-restricted first order logic formulae. Since the repair algorithm relies on the rules of the database and the closed-world assumption, it can automatically find repairs for violated existential formulae without user intervention.

In [5], constraints between distributed documents are expressed in xlinkit, a combination of first order logic with XPath expressions. The paper also presents a framework which automatically derives a set of repair actions from the constraints. In [4], such rules form the knowledge base of an expert system. However, these approaches tend to consider only a single change and consequently do not explicitly address the cascading nature of change propagation.

Consistency checking is an area clearly related to our work. Our framework can be built on top of existing (UML) consistency checking approaches. However, note that the iterative nature of cascading changes ideally requires incremental consistency checking as proposed in ArgoUML[7] and xlinkit [18]. Advanced event-driven consistency checking approaches such as Egyed's [19] can be integrated in our framework as the constraint checking component.

Several other approaches implement the change support mechanism based on some underlying mathematical formalism. For example, the formalism of graph rewriting has been used to deal with change propagation [3] and model synchronization [20]. In [21] and [22], they propose to transform (UML) specifications to Petri-Nets and Description Logic respectively. These approaches then exploit existing consistency checks that have been defined for the mathematical formalism. However, it is not clear to what extent these approaches suffer from the traceability problem: that is, 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 [21]. By contrast, our approach deals directly with UML models without any transformation and thus does not suffer from that issue.

6 Conclusions and future work

In this paper we have presented an approach for change propagation in design models. Our framework takes as input a meta-model and well-formedness constraints (in OCL), and makes use of repair plans to propagate changes by fixing inconsistencies. A key feature is that these repair plans are generated automatically from the OCL constraints, in a way that is sound and complete. We use a design of a stock trading management system developed using the Prometheus methodology as an illustrative example showing how the framework works. Our proposed framework is generic and can be applied to object-oriented methodologies as well, indeed, we have already applied it to the object-oriented UML design of an ATM system, which contains a class diagram (18 classes such as ATM, Bank, Transaction) and 10 sequence diagrams (corresponding to 10 use cases such as GetPIN, PrintReceipt). We then introduced several realistic requirement changes and applied our framework to propagate changes. Although the results have

[7] http://argouml.tigris.org/

shown the applicability and scalability of our framework, more complex case studies are needed to evaluate it.

Some specific areas for future work that we intend to investigate are: (1) Implementing the repair plan generator (2) Investigating how to extend our approach to deal with programming language code as well as design artefacts. (3) Dealing more efficiently with plan recipes with a context condition of the form $x \in Type(SE)$ by being lazy: instead of creating a plan instance for each possible value, defer the choice and use subsequent constraints to narrow down the range of possible values for x. (4) Performing a more extensive case study, in order to better ascertain the *scalability* of our approach; and also conducting more evaluation to better ascertain the *effectiveness* of the approach. There are two measurements that we should take into account when we evaluate the framework. We plan to choose an existing, reasonable-size, system and then define a classification of changes. We then input these to our framework and analyze the accuracy of what our framework recommends. In addition, we investigate how our framework deals with different system sizes to measure its efficiency.

Acknowledgments

This work is being supported by the Australian Research Council under grant LP0453486, in collaboration with Agent Oriented Software. The authors would also like to thank Lin Padgham for discussions relating to this work.

References

1. Swanson, E.B.: The dimensions of maintenance. In: ICSE '76: Proceedings of the 2nd international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1976) 492–497
2. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In Chang, K.S., ed.: Handbook of Software Engineering and Knowledge Engineering. World Scientific (2001) 24–29
3. Rajlich, V.: A model for change propagation based on graph rewriting. In: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE Computer Society (1997) 84–91
4. Sourrouille, J.L., Caplat, G.: Checking UML model consistency. In: Workshop on Consistency Problems in UML-based Software Development at UML 2002, Dresden, Germany (2002)
5. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society (2003) 455–464
6. Dam, K.H., Winikoff, M., Padgham, L.: An agent-oriented approach to change propagation in software evolution. In: Proceedings of the Australian Software Engineering Conference (ASWEC), IEEE Computer Society (2006) 309–318
7. Padgham, L., Winikoff, M.: Developing intelligent agent systems : a practical guide. John Wiley & Sons, Chichester (2004) ISBN 0-470-86120-7.
8. Object Management Group: Object Constraint Language (OCL) 2.0 Specification (2006)

9. Rao, A.S., Georgeff, M.P.: An abstract architecture for rational agents. In Rich, C., Swartout, W., Nebel, B., eds.: Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, San Mateo, CA, Morgan Kaufmann Publishers (1992) 439–449
10. Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A.E.F., eds.: Multi-Agent Programming: Languages, Platforms and Applications. Springer (2005)
11. Rao, A.S.: 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, Springer-Verlag (1996) 42–55
12. Dam, K.H., Winikoff, M.: An agent-based approach to change propagation. Technical Report TR-06-04, RMIT University (2006)
13. Arnold, R., Bohner, S.: Software Change Impact Analysis. IEEE Computer Society Press (1996)
14. Mayol, E., Teniente, E.: A survey of current methods for integrity constraint maintenance and view updating. In: Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling, London, UK, Springer-Verlag (1999) 62–73
15. Ceri, S., Fraternali, P., Paraboschi, S., Tanca, L.: Automatic generation of production rules for integrity maintenance. ACM Trans. Database Syst. **19**(3) (1994) 367–422
16. Gertz, M., Lipeck, U.W.: An extensible framework for repairing constraint violations. In: Proceedings of the IFIP TC11 Working Group 11.5, First Working Conference on Integrity and Internal Control in Information Systems, Chapman & Hall, Ltd. (1997) 89–111
17. Moerkotte, G., Lockemann, P.C.: Reactive consistency control in deductive databases. ACM Trans. Database Syst. **16**(4) (1991) 670–702
18. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: a consistency checking and smart link generation service. ACM Transactions on Internet Technology **2**(2) (2002) 151–185
19. Egyed, A.: Instant consistency checking for the UML. In: Proceedings of the 28th International Conference on Software Engineering (ICSE), Shanghai, China (May 2006)
20. Ivkovic, I., Kontogiannis, K.: Tracing evolution changes of software artifacts through model synchronization. In: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM), IEEE Computer Society (2004) 252–261
21. Engels, G., Kuster, J.M., Heckel, R., Groenewegen, L.: Towards consistency-preserving model evolution. In: Proceedings of the International Workshop on Principles of Software Evolution (IWPSE), ACM Press (2002) 129–132
22. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logics to maintain consistency between UML models. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 - The Unified Modeling Language, Springer-Verlag (2003) 326–340 LNCS 2863.