Cost-Based BDI Plan Selection for Change Propagation[*]

Khanh Hoa Dam
RMIT University
Melbourne, Australia
kdam@cs.rmit.edu.au

Michael Winikoff
RMIT University
Melbourne, Australia
michael.winikoff@rmit.edu.au

ABSTRACT

Software maintenance is responsible for as much as two thirds of the cost of any software, and is consequently an important research area. In this paper we focus on the *change propagation* problem: given a primary change that is made in order to meet a new or changed requirement, what additional, secondary, changes are needed? We build on previous work that has proposed to use a BDI (belief-desire-intention) agent framework to propagate changes by fixing violations of consistency constraints. One question that needs to be answered as part of this framework is how to select between different applicable (repair) plan instances to fix a given constraint violation? We address this issue by defining a suitable notion of *repair plan cost* that incorporates both conflict between plans, and synergies between plans. We then develop an algorithm, based on the notion of cost, that finds cheapest options and proposes them to the user.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Design

Keywords

Software Maintenance and Evolution, Change Propagation, Plan Selection, Belief Desire Intention, Plan Cost

1. INTRODUCTION

A large percentage — as much as two-thirds — of the cost of any software can be attributed to its *maintenance*: modifications to the software due to a range of causes[1], after the software has been written [18, page 449]. Consequently, software maintenance is a highly important area for research. In particular there has been very little work that we are aware of on software maintenance in agent-oriented software engineering, and this work aims to fill that gap, as well as apply agent technology to the problem of software maintenance in a broader context.

When software is modified, typically some primary changes are made, and then additional, secondary, changes are made as a result. For example, an agent type is added, and then consequently other agents need to be modified to communicate with the new agent type. Determining and making these secondary changes is termed *change propagation* [13, 14].

This paper builds on our previous work [4], which proposed an agent-based approach to change propagation. Given a model (i.e. a design) which has been subjected to primary changes, the system finds inconsistencies in the model (with respect to given constraints), and then invokes repair plans to fix these consistency violations. The change propagation engine proposed uses a Belief-Desire-Intention (BDI) platform to perform change propagation. The use of BDI-style, event-triggered, plans matches well with the cascading nature of change propagation where a change can cause other changes to be made. Further, there are usually many ways of fixing a given inconsistency, and this is naturally captured using multiple plans that respond to a given event. Although we do not use the full capabilities of BDI agents, these two properties of change propagation make the use of BDI plans natural and, we believe, well motivated.

Typically a given inconsistency will have a number of repair plans that could be used to restore consistency. In this paper we focus on the problem of *how to select amongst these repair plans*.

This problem is made harder because we need to handle infinite trees, due to the nature of cascading. On the other hand, plan execution for change propagation does not take place in a dynamic environment and thus given a number of relevant repair plans, the choice between them can be controlled by the system.

The remainder of this paper proposes a mechanism for automatically selecting between alternative repair plans based on a notion of *cost*. We define the cost of plans (section 3) in a way that takes into account cascades (where fixing the violation of a constraint breaks another constraint), and synergies between constraints (where fixing the violation of a constraint also fixes another violated constraint). An algorithm for calculating costs, and hence selecting between repair plans, is given (section 4), and its scalability is explored. Finally, we discuss related work (section 5) before concluding and outlining future work (section 6).

2. CHANGE PROPAGATION FRAMEWORK

This section briefly describes the agent-based change propagation framework of [4], and its components, including the plan representation and generation. The framework provides a "change prop-

[*]The primary author of the paper is a student.

[1]These are usually classified as being *corrective maintenance*, fixing bugs; *perfective maintenance*, adding new functionality; or *adaptive maintenance*, changing the system so it continues to work in a changed environment.

Cite as: Cost-Based BDI Plan Selection for Change Propagation, Khanh Hoa Dam and Michael Winikoff, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. XXX-XXX.

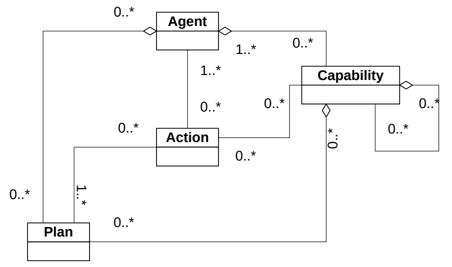


Figure 1: Prometheus Meta-Model (Excerpt)

agation assistant" that helps a designer by suggesting additional (secondary) changes once primary changes have been made. This framework is generic in that it can be applied to various software engineering methodologies, and we have applied it to both UML and Prometheus.

The key data items we deal with are a *meta-model*, a collection of well-formedness *constraints*, an application design *model*, and a collection of *repair plans*. The overall process is:

1. At design time the repair plans are automatically generated from the constraints and meta-model [3].
2. At runtime we check whether the constraints hold in the design model.
3. We use the repair plans to generate plan instances (i.e. repair options) for the violated constraints.
4. We calculate the cost of the different repair plan instances.
5. We select a repair plan instance (possibly by picking the single cheapest, if it exists, or by asking the user).
6. The selected repair plan instance is executed, and it updates the application design model.

Note that although it is possible for loops to exist, the cost calculation avoids them (if possible, i.e. if the constraints can be fixed) since they have infinite cost.

We now briefly describe each of the four key data items.

The **meta-model** specifies, in the usual manner, what entities exist in a design model, and their relationships. Figure 1 shows a small excerpt of the Prometheus meta-model, which depicts relationships between agents, plans, capabilities and actions. The meta-model is captured in UML, and is exported to XMI format for use by our implementation.

The **constraints** specify conditions that a well-formed design should satisfy. We use the Object Constraint Language [12] to specify constraints. OCL 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, capabilities and actions. In the OCL notation "self" denotes the context node (in this case an Agent) to which the constraints have been attached and an access pattern such as "self.action" indicates the result of following the association between an agent and an action (in the meta-model), which is, in this case, a collection of actions which are performed by the agent. 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 [12]. For example, the following constraint, which could be expressed in more traditional form as $\forall\, ac \in self.\text{action}\, \exists\, pl \in self.\text{plan} : \text{ac} \in pl.\text{action} \vee \exists\, cap \in self.\text{capability} : \text{ac} \in cap.\text{action}$, states that: considering the set of actions that are performed by the agent (*self.action*), for each of the actions (*ac*) if we consider the plans of that agent (*self.plan*) then one of these plans (*pl*) must include the current action (*ac*) in its list of actions (*pl.action*) or if we consider the capabilities of that agent (*self.capability*) then one of these capability (*cap*) must contain the current action (*ac*) in its list of actions (*cap.action*).

Constraint 1 *Any agent that performs an action should contain at least one plan or capability that performs that action.*
Context Agent inv*:*

```
self.action→forAll(ac : Action |
        self.plan→exists(pl : Plan | pl.action→includes(ac)) or
        self.capability→exists(cap : Capability |
                cap.action→includes(ac)))
```

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 [4].

The application design **model** is a design, in this case a Prometheus design. Abstractly, we can view a design as consisting of a set of entities E (with their types), a set of relationships R (e.g. the *action* attribute of *agent1* includes *act1*), and a value function V (e.g. the *name* attribute of the entity *agent1* has the value "*Monitor Agent*"). Formally, let E to be a set of entity-id and entity-type pairs; R be a set of triples: entity ID (source), attribute ID, and entity ID (destination); and V be a function from entity ID and attribute ID to a value (e.g. integer, string).

The four types of primitive actions that are used to update the model are creation of entities, adding and removing relationships between entities, and updating the values of attributes of entities. Formally $\mathbf{create(x, t)}$ has no precondition, and has the postcondition $E' = E \cup \{\langle x, t\rangle\}$ (where E' denotes the value of E after the operation); $\mathbf{add(e_2, e_1, a)}$ has the precondition $\{e_1, e_2\} \subseteq \text{dom}\, E$ (where dom X is the domain of X) and postcondition $R' = R \cup \{\langle e_1, a, e_2\rangle\}$; $\mathbf{remove(e_2, e_1, a)}$ has true precondition and postcondition $R' = R \setminus \{\langle e_1, a, e_2\rangle\}$; and $\mathbf{change(e, a, v)}$ has true precondition and the postcondition $V' = V \oplus \{\langle e, a\rangle \mapsto v\}$ where $A \oplus B = \{\langle x, y\rangle \mid \langle x, y\rangle \in A \wedge x \notin \text{dom}\, B\} \cup B$.

An important observation is that the preconditions of these primitive actions are quite weak. This allows us to arbitrarily reorder a sequence of actions subject to the following constraints: (1) creation of entities must remain before addition of relationships between the entities; (2) if the sequence of actions has redundant pairs — an action that undoes the effects of an earlier action — then the pair cannot be swapped, but it can be simplified by deleting the earlier action. For example, adding a relationship followed by deleting it can be replaced by simply deleting the relationship. Condition (2) is not needed if we assume that the sequence of actions being reordered is *non-redundant*, i.e. does not contain any redundant pairs.

Lemma 1 (Action sequence reordering) *A non-redundant sequence of actions S can be arbitrarily reordered, so long as creation of entities precedes relating these entities, without affecting the overall effect of S.*

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

Figure 2: Repair plan abstract syntax

The syntax for **repair plans**[2] (see figure 2) is based on AgentSpeak(L) [15], 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) that specifies when the plan should be applicable[3]; 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 ($!E$).

The repair plans are generated automatically from the constraints using a repair plan generator that takes the OCL constraints and the UML meta-model as inputs, and returns a parameterized set of event-triggered repair plan types. *Each OCL constraint (or sub-constraint) has a corresponding goal (or sub-goal) and we repair the constraint by posting the goal and using the plans to achieve the goal.* Thus in the remainder of this paper we will talk about repairing constraints and achieving sub-goals as being the same thing.

For example, the constraint given earlier is translated (by the $\mathcal{R}$ operator of [3]) to the following repair plans[4], where we define $c \equiv \forall\, ac \in \mathit{self.action} : c1$, and $c1 \equiv c2 \vee c4$, and $c2 \equiv \exists\, pl \in \mathit{self.plan} : c3$, and $c3 \equiv ac \in pl.action$, and $c4 \equiv \exists\, cap \in \mathit{self.capability} : ac \in cap.action$.

P1 $c_t(\mathit{self}) \leftarrow$ for each $ac \in \mathit{self.action}$
if $\neg\, c1(ac)$ then $!c'_t(\mathit{self}, ac)$

P2 $c'_t(\mathit{self}, ac) \leftarrow$ remove ac from $\mathit{self.action}$

P3 $c'_t(\mathit{self}, ac) \leftarrow !c1_t(\mathit{self}, ac)$

P4 $c1_t(\mathit{self}, ac) \leftarrow !c2_t(\mathit{self}, ac)$

P5 $c1_t(\mathit{self}, ac) \leftarrow !c4_t(\mathit{self}, ac)$

P6 $c2_t(\mathit{self}, ac) : pl \in \mathit{self.plan} \leftarrow !c3_t(\mathit{self}, ac, pl)$

P7 $c2_t(\mathit{self}, ac) : pl \in \mathit{Plans} \wedge pl \notin \mathit{self.plan} \leftarrow$
add pl to $\mathit{self.plan}$; $!c3_t(\mathit{self}, ac, pl)$

P8 $c2_t(\mathit{self}, ac) \leftarrow$ create $pl : \mathit{Plan}$; add pl to $\mathit{self.plan}$;
$!c3_t(\mathit{self}, ac, pl)$

P9 $c3_t(\mathit{self}, ac, pl) \leftarrow$ add ac to $pl.action$

Given a design model which has an action ac_1 assigned to agent a_1, where a_1 has plan p_1; these plans can produce a range of actions to repair the constraint including removing ac_1 from a_1 (**P2**), or assigning ac_1 to p_1 (**P3**, **P4**, and **P6**).

[2]"Prop" denotes a primitive condition such as checking whether $x > y$ or whether $x \in SE$, and SE denotes a set-valued expression.

[3]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.

[4]For space reasons we have omitted the plans for $c4$, which are similar to those for $c2$.

One key consequence of generating plans from constraints, rather than writing them manually, is that by careful definition of the plan generation scheme (i.e. the $\mathcal{R}$ operator of [3]) it is possible to guarantee certain properties of the generated plans.

Theorem 1 ($\mathcal{R}$ complete and minimal) *The generated repair plans are* complete*, that is, given a model (i.e. design) M in which constraint C is violated, any minimal sequence of actions (that is, one that does not contain unnecessary actions) that leads to a model M' where C is not violated can be obtained by instantiating the plans in $\mathcal{R}(C)$.* **Proof:** *See theorem 1 of [3]*

At runtime, the application model is checked against the OCL constraints and any violations of these constraints are fixed using the repair plans. A given violation can be potentially fixed by a number of possible repair plan instances. In order to help select which repair plan instances to use we calculate the cost of each repair plan instance.

3. COST DEFINITION

In this section, we give equations that define the cost of fixing a given constraint and then explore some properties of the definitions. The notion of cost that we use is abstract: it can be viewed as counting the number of primitive actions (addition, removal, update, creation) involved in a given plan. For example, if repair plan P_1 involves 5 additions and repair plan P_2 involves 3 additions then we view P_2 as being cheaper. In order to compare "apples and oranges", e.g. if P_3 involves two additions and a creation, we assume that each primitive action type is assigned a numerical cost (its "basic cost"), for instance creation may have an assigned cost of 5 and addition a cost of 3. These numbers do not correspond to any real cost, and are simply used to compare different action types.

We begin with some preliminary concepts and terminology. A constraint that does not hold with regard to a model is said to be violated, and can be fixed by executing a repair plan. A repair plan instance contains repair actions (the set of which is denoted $\mathcal{A}(P)$) and subgoals (representing sub-constraints) the set of which is denoted $\mathcal{G}(P)$. Repairing a constraint is done in the context of a *repair scope*: a set of constraints that need to be considered. The constraints in the repair scope are checked when the repair plan finishes executing, and any violated constraints are then repaired. We denote the repair scope of a plan P as $\mathcal{S}(P)$. A global repair scope involves all constraints whilst a local one contains constraints related to certain entities in the model. Normally the repair scope is set initially (typically to be global) and then is not changed. We define the repair scope explicitly, rather than automatically considering all constraints, in order to allow a user to limit the propagation to certain constraints or model entities.

We now define the cost of a repair plan in terms of the costs of its basic actions (basicCost), the cost of its subgoals (subGoalCost), and the cost of fixing violated constraints in its repair scope (scopeCost). Note that $cost$ is defined for actions ($cost(A)$), plans ($cost(P)$), constraints ($cost(C)$), and (sub)goals ($cost(G)$).

Definition 1 (Action cost) *The cost of an action A, denoted $cost(A)$, is the user-defined basic cost associated with the action type (i.e. addition, removal, update, or creation).*

Definition 2 (Plan cost) *The cost of a plan P (denoted cost(P)) is equal to the sum of its main cost and its repair scope cost. The main cost of a plan is the sum of the plan's basic cost and its subgoal cost. The scope cost is the cost of repairing all (violated)* *constraints in the plan's repair scope*[5].

$$\begin{aligned}
\mathit{cost}(P) &= \mathit{mainCost}(P) + \mathit{scopeCost}(P) \\
\mathit{mainCost}(P) &= \mathit{basicCost}(P) + \mathit{subGoalCost}(P) \\
&= \sum_{A \in \mathcal{A}(P)} \mathit{cost}(A) \ + \sum_{G \in \mathcal{G}(P)} \mathit{cost}(G) \\
\mathit{scopeCost}(P) &= \sum_{C \in \mathcal{S}(P)} \mathit{cost}(C)
\end{aligned}$$

There are usually several applicable plan instances to repair a constraint violation. The best plan, which is selected for execution, is the one with minimum cost. Hence the cost of repairing a constraint is the cost of the cheapest repair plan instance.

Definition 3 (Constraint cost) *The cost of fixing constraint C is equal to the cost of the best applicable repair plan instance with regard to C. If there are no applicable repair plans, the cost of C is undetermined. The cost of fixing an unviolated constraint is 0. We formalise this as follows, where $\mathcal{P}(C)$ is the set of all repair plan instances that can be used to fix constraint C.*

$$\mathit{cost}(C) = \begin{cases} 0 & \text{if } C \text{ unviolated} \\ \mathit{min}\ \{\mathit{cost}(P) \mid P \in \mathcal{P}(C)\} & \text{otherwise} \end{cases}$$

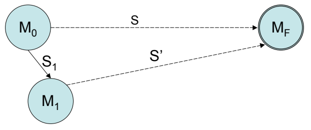
Definition 4 (Goal cost) *The cost of achieving a goal is the cost of the cheapest available repair plan. Similarly to constraints, we use $\mathcal{P}(G)$ to denote the set of all repair plans that can be used to achieve the goal G.*

$$\mathit{cost}(G) = \mathit{min}\ \{\mathit{cost}(P) \mid P \in \mathcal{P}(G)\}$$

We now briefly note some properties of these definitions. We say that a sequence of actions S repairs constraint C in model M iff (a) C is violated in M; and (b) performing S on M yields a new model M'; and (c) C holds in M'. We say that the sequence S is *minimal* if removing any action from it results in a sequence that no longer repairs C in M. This generalises to a set of constraints in the obvious way.

Lemma 2 *Let M_0 be a model in which the constraints C_i are violated. Let S be a minimal sequence of actions for repairing all the constraints C_i in M_0. Then for a given constraint, say (without loss of generality) C_1, there exists at least one sequence of actions S' which is obtained by removing some number (possibly zero) of actions from S such that S' repairs C_1 in M_0 and is minimal.*
Proof: *S repairs C_1 in M_0, but may contain actions that are unnecessary for repairing C_1. We construct S' by simply removing these unnecessary actions, resulting in a minimal S'.* ■

Theorem 2 *Let M_0 be a model where some number of constraints C_i are violated and let S be a minimal (and hence non-redundant) sequence of actions that repairs the C_i in M_0, yielding model M_F:*



[5]Since we will define $\mathit{cost}(C) = 0$ if the constraint C is not violated we simply sum over the cost of all constraints in $\mathcal{S}(P)$.

Then for any of the given constraints, say (without loss of generality) C_1, there exists a minimal action sequence S_1 that repairs C_1 in M_0 yielding M_1. Furthermore, there then exists a (non-redundant) action sequence S' that takes us from M_1 to M_F where $\mathit{cost}(S) = \mathit{cost}(S_1) + \mathit{cost}(S')$.
Proof: *We construct S' and S_1 from S as follows. We form S_1 by removing actions from S to yield a minimal S_1 for repairing C_1 in M_0 (using lemma 2). The actions that are not removed from S are the remainder, S'. We can view the sequence S_1 followed by S' as being a reordering of S, and by lemma 1 it has the same effect as S, i.e. results in M_F. Since S_1 followed by S' has the same actions as S it must have the same cost.* ■

By applying this theorem repeatedly, on C_1, then C_2, etc. we can show that in order to repair a set of violated constraints we can consider a single constraint at a time, in an arbitrary order, with no loss of generality. Furthermore, since the repair plans are complete (theorem 1), the action sequence S_1 can be generated by instantiating the repair plan set.

This strong result is only possible because the actions we consider have limited preconditions, allowing them to be reordered fairly freely. A specific corollary is that, considered as a planning domain, our actions do not allow for a Sussman anomaly situation to exist.

4. A COST CALCULATION ALGORITHM

In the previous section, we have defined how a repair plan's cost is calculated. We now give algorithms that calculate this cost. The algorithms operate with plan-goal trees, where a goal has as children the plans that can be used to achieve it ($\mathcal{P}(G)$ in definition 4) and a plan has as children its sub-goals ($\mathcal{G}(P)$). Each plan node stores the plan's basic cost (*basicCost*, initially the basic cost of the plan), other costs (*dynamicCost*, initially 0), a boolean value indicating whether the node is a leaf (*isLeaf*, initially false), and a queue of its sub-goals (*subGoalQueue*, initially empty). Each goal node stores a list of best (i.e. least cost) plan(s) (*bestPlans*, initially empty) that achieve the goal.

Before we present the algorithm, we discuss a tree transformation that the algorithm uses. When considering the alternative ways of dealing with a given (sub)goal the algorithm considers the available plans and selects the cheapest. In doing so, it needs to consider the future: what will happen after the goal is handled. We do this by transforming the tree so that the "future" is pushed down into the tree beneath the current goal. Specifically, when we consider a goal that has a future (i.e. a parent plan with non-empty sub-goals) we copy the sub-goals of the parent plan to the sub-goals of the children plans (see figure 3).

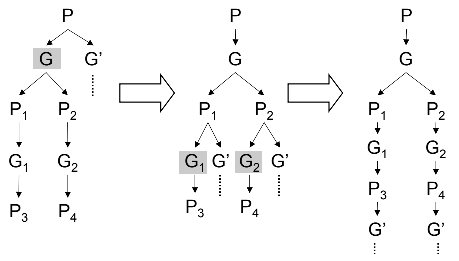


Figure 3: Tree Transformation

The algorithm presented in figure 4 computes the cost of a plan according to the equations in the previous section. Since we assume that *basicCost* is already computed (by simply summing the costs of primitive actions in a plan), the algorithm only needs to work out the plan's subgoal costs and repair scope costs (see definition 2). These costs are stored in *dynamicCost* which is initially set to 0, and is progressively incremented with the costs of sub-goals and of violated constraints in the repair scope.

The algorithm selects each sub-goal in turn (lines 2 and 3) and adds the cost of any violated constraints onto the dynamic cost (line 6). If the plan node has children (i.e. violated constraints[6]) then we are done, since the scope cost will be calculated in those children. On the other hand, if this plan node has no children ($isLeaf = true$, line 9) then we check for violated constraints in the repair scope (lines 10 and 11), and if there are any, we select one of the violated constraints (line 12), add it to the queue (line 13), and recursively call $cost(P)$ to compute its cost (line 14).

The algorithm in figure 5 calculates the cost of a goal node (see definition 4) by considering the possible plans and looking for the cheapest one. We first retrieve a list of applicable plans for the goal (line 2). We then iterate through the list of plans (line 4) and calculate the cost for each of them (line 9). When a plan that is cheaper than the previous best is found, the previous best plan(s) are replaced with the new plan (lines 10-13). When a plan is found that is as good as the current best, it is added to the list of best plan(s) (lines 14-15).

The algorithm uses look-ahead and simulates the application of the plans. Line 5 executes the plan currently being considered by (a) updating the model with the effects of the plan's actions, and (b) adding the plan's sub-goals to the tree. In order to be able to consider alternative plans we need to undo the effects of the plan's execution on the model, and this is done by line 17. This is implemented by logging changes to the model, allowing these changes to be rolled back.

Lines 6-8 and 19-21 implement the tree transformation discussed earlier: the sub-goals of the parent plan (excluding the current sub-goal) are added to the end of the sub-goals of each plan P (lines 6-8). Once this has been done for all plans, we remove the sub-goals from the parent (lines 19-21).

```
function cost(P)
1   $P.isLeaf \leftarrow true$
2   while $P.subGoalQueue$ is NOT empty do
3       dequeue $subGoal$ from $P.subGoalQueue$
4       if the constraint associated with $subGoal$ is violated then
5           $P.isLeaf \leftarrow false$
6           $P.dynamicCost \leftarrow P.dynamicCost$ + cost($subGoal$, $P$)
7       end if
8   end while
9   if $P.isLeaf = true$ then
10      local $violatedSubGoals \leftarrow$ get-scope-violated-constraints()
11      if $violatedSubGoals$ is NOT empty then
12          get a random $violatedSubGoal$ from $violatedSubGoals$
13          enqueue $violatedSubGoal$ into $P.subGoalQueue$
14          return cost($P$)
15      end if
16  end if
17  return $P.dynamicCost + P.basicCost$
```

Figure 4: Calculating Plan Node Cost (No Pruning)

The algorithms given in figures 4 and 5 implement the definitions given in section 3, but they search the whole goal-plan tree. This is inefficient, and may lead to non-termination, since the tree may be infinite. We therefore modify the algorithms by adding loop checking, and a form of pruning. We add to each goal/plan node two

[6]Note that when we encounter a violated constraint we note that the plan node is not a leaf (line 5).

```
function cost($G$, $ParentPlan$)
1   local $bestCost \leftarrow +\infty$
2   local $planList \leftarrow$ get-repair-plans($G$)
3   $G.bestPlans \leftarrow empty$
4   for each plan $P$ in $planList$ do
5       execute plan $P$
6       if $ParentPlan$ is not $null$ then
7           copy all $ParentPlan.subgoals$ to the end of $P.subgoals$
8       end if
9       local c $\leftarrow$ cost(P)
10      if c $< bestCost$ then
11          $bestCost \leftarrow$ c
12          clear $G.bestPlans$
13          add $P$ to $G.bestPlans$
14      else if c $= bestCost$ then
15          add $P$ to $G.bestPlans$
16      end if
17      unexecute plan $P$
18  end for
19  if $ParentPlan$ is not $null$ then
20      $ParentPlan.subgoals \leftarrow empty$
21  and if
22  return $bestCost$
```

Figure 5: Calculating Goal Node Cost (No Pruning)

values[7]: β (initially $+\infty$) - the least cost of fixing all constraints in the repair scope, and σ (initially 0) - the (accumulative) cost of everything *above* the current node. In figures 6 and 7 lines that are new (relative to figures 4 and 5) are marked with "*".

Computing the cost of a plan is done by the algorithm in figure 6. We use a pruning mechanism, where we establish a threshold in order to avoid exploring alternatives that are more expensive than known solutions. The threshold is calculated (line 10 of figure 6) based on the current accumulative cost σ, the plan cost ($P.basicCost$ and $P.dynamicCost$) and the lower bound cost, which is an estimate of the minimum cost of achieving a (sub-)goal (lines 1-8 in the bottom of figure 6).

The algorithm in figure 6 also includes loop detection (lines 4-7). It keeps track of goals seen along a branch in a list named *history* (line 6-8 in figure 7). If the same goal is seen again, corresponding to the fact that a constraint has become violated and is being fixed again, then we have a loop and we terminate with infinite cost. This checking only needs to be done when β is at its initial value ($+\infty$): if β has a finite value, then an infinite branch will be pruned because its cost will (eventually) exceed β (because all plans do something, and hence have non-zero cost).

The two values β and σ are passed from the parent goal/plan nodes down to its child plan/goal ones (lines 14-15 in figure 6 and lines 10-11 in 7). Line 14 in figure 6 shows that σ is in fact an accumulative cost: we accumulate the cost of the current node in σ. When a plan cost is resolved, the total cost so far (i.e. the cost of the plan as well as σ, the cost of the path from the root of the tree to the current node) is compared against the current β to see if it needs to be updated (line 27 in figure 6). If at any point the total cost for a plan (*threshold*) exceeds β then we prune (lines 11-13 of figure 6). We also prune in the (admittedly unlikely) case that a plan's basic cost by itself exceeds β (line 4 of figure 7). Once a best plan for a goal is found, the goal's β is also updated with the plan's β (line 21 of figure 7). Line 5 of figure 7 implements a heuristic that considers plans with cheaper basic cost first.

The computational complexity of the algorithm depends on the cost of checking a single constraint (which, based on empirical evidence [6], we assume to be constant); and the degree to which prun-

[7]We use "β" since we do the β part of a classical $\alpha - \beta$ pruning. We do not do the α part because we have a min-sum tree, rather than a min-max tree.

```
    function cost(P)
1   $P.isLeaf \leftarrow true$
2   while $P.subGoalQueue$ is NOT empty do
3       dequeue $subGoal$ from $P.subGoalQueue$
*4      if $P.\beta = +\infty$ and $subGoal$ is in $history$ then
*5          clear $history$
*6          return $P.\beta$
*7      end if
8       if the constraint associated with $subGoal$ is violated then
9           $P.isLeaf \leftarrow false$
*10         local $threshold = P.\sigma$ + lowerBoundCost(subGoal) +
                            $P.basicCost + P.dynamicCost$
*11         if $threshold > P.\beta$ then
*12             return $threshold$
*13         end if
*14         $subGoal.\sigma \leftarrow P.\sigma + P.dynamicCost + P.basicCost$
*15         $subGoal.\beta \leftarrow P.\beta$
16          $P.dynamicCost \leftarrow P.dynamicCost$ + cost($subGoal$, $P$)
17      end if
18  end while
19  if $P.isLeaf = true$ then
20      $violatedSubGoals \leftarrow$ get-scope-violated-constraints()
21      if $violatedSubGoals$ is NOT empty then
22          get a random $violatedSubGoal$ from $violatedSubGoals$
23          enqueue $violatedSubGoal$ into $P.subGoalQueue$
24          return cost($P$)
25      end if
26  end if
*27 $P.\beta \leftarrow \min(P.\beta, P.\sigma + P.dynamicCost + P.basicCost)$
28  return $P.dynamicCost + P.basicCost$

    function lower-bound-cost(G)
*1  local $planList \leftarrow$ get-repair-plans(G)
*2  local $lowerBound \leftarrow +\infty$
*3  for each plan $P$ in $planList$ do
*4      if $P.basicCost < lowerBound$ then
*5          $lowerBound \leftarrow P.basicCost$
*6      end if
*7  end for
*8  return $lowerBound$
```

Figure 6: Calculating Plan Node Cost (Pruning)

```
    function cost($G$, $ParentPlan$)
1   local $bestCost \leftarrow +\infty$
2   local $planList \leftarrow$ get-repair-plans($G$)
3   $G.bestPlans \leftarrow empty$
*4  remove plans in $planList$ that have basic cost greater than $G.\beta$
*5  sort plans in $planList$ based on their basic action costs
*6  if $G.\beta = +\infty$ then
*7      add $G$ into $history$
*8  end if
9   for each plan $P$ in $planList$ do
*10     $P.\beta \leftarrow G.\beta$
*11     $P.\sigma \leftarrow G.\sigma$
12      execute plan $P$
13      if $ParentPlan$ is not $null$ then
14          copy all $ParentPlan.subgoals$ to the end of $P.subgoals$
15      end if
16      c $\leftarrow$ cost(P)
17      if c $< bestCost$ then
18          $bestCost \leftarrow$ c
19          clear $G.bestPlans$
20          add $P$ to $G.bestPlans$
*21         $G.\beta \leftarrow P.\beta$
22      else if c $= bestCost$ then
23          add $P$ to $G.bestPlans$
24      end if
25      unexecute plan $P$
26  end for
27  if $ParentPlan$ is not $null$ then
28      $ParentPlan.subgoals \leftarrow empty$
29  end if
30  return $bestCost$
```

Figure 7: Calculating Goal Node Cost (Pruning)

ing reduces the search space (see below); as well as the number of child nodes each (non leaf) node has (N), the depth of the plan-goal tree (D), and the size of the application design model (M). Space limitations preclude a detailed derivation, so we merely note that the work to be done for each plan node is $O(N + M)$, and that the work to be done for each goal node is roughly $O(N \log N + D)$. Since the number of nodes is roughly $O(N^D)$ this gives an overall computational complexity of $O(N^D \times (N \log N + D + M))$.

Without pruning, the algorithms in figures 4 and 5 are not guaranteed to terminate since looping may occur when the repair plan of a constraint breaks the other constraint and vice versa. In contrast, the algorithms equipped with pruning capabilities in figures 6 and 7 are guaranteed to terminate due to two reasons. Firstly, when a solution has been found and the best cost β has been determined, branches that contains cycles (and potentially leads to an infinite tree) are eventually pruned because of having a higher cost. Secondly, in case when looping occurs before β is determined, we also have a loop detection to prune the search tree.

4.1 Evaluation

One key question is how practical the algorithm is, specifically, how well does it scale to larger problems?

In order to investigate this question we perform a number of experiments where we "stress test" the algorithm in an artificial setting. Two key parameters that we vary are the number of repair plan instances (for one constraint), which corresponds to the *width* of the plan-goal tree; and the overall size of the tree, which we do by varying the number of constraints, and hence the *depth* of the tree. We measure the running time, and how many nodes did the algorithm avoid having to explore through pruning. In addition to considering an artificial setting, we also perform some experiments with a non-artificial application.

Our simple artificial setting involves a design that has some number of roles, and some number of agents. All of the Prometheus' 46 well-formedness constraints are used, with the exception of the constraint that states that roles need to be associated with at least one goal. However, the only constraint that will be violated in this artificial setting is the one that states that all roles should be associated with an agent: **Context Role inv c** : self.agent→size()≥ 1. This constraint is translated to the following repair plans[8], where sa is short for $self.agent$

P1 $c_t(self) \leftarrow$ for each $i \in \{1 \ldots (1 - size(sa))\}$ $!c'_t(self)$
P2 $c'_t(self) : x \in$ Type(sa) $\wedge\ x \notin sa \leftarrow$ Add x to sa
P3 $c'_t(self) \leftarrow$ Create x : Type(sa) ; Add x to sa

In order to explore how the algorithm performs as the number of repair plan instances is increased we have a design with a single role and N agents. This gives a single violated constraint to fix, and by increasing N we increase the number of repair options (since there is always a single instance of **P3**, but there are N instances of **P2**, one for each agent).

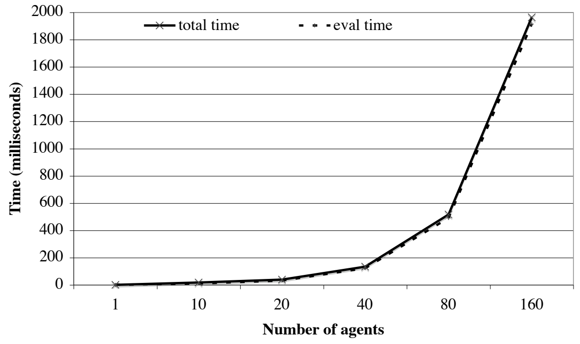
The graph below shows the runtime (in milliseconds) for the first experiment[9]. In this experiment pruning made no significant difference, since there is nothing to distinguish between the agents (the

[8]The translation is not optimal because it also caters for constraints of the form $size() \geq n$.

[9]All experiments reported in this paper were performed on a laptop running Windows XP and Java v1.5.0_06, with a Intel Centrino 1.73Ghz CPU and 1GB RAM. Times (reported in milliseconds) are an average of 30 runs (we ignored the first run, since it was inconsistent due to JVM startup). For each run we collected the number of goal and plan nodes explored, the total time (broken down into the constraint evaluation time, time taken to update models, and other time), and the number of constraint instances.

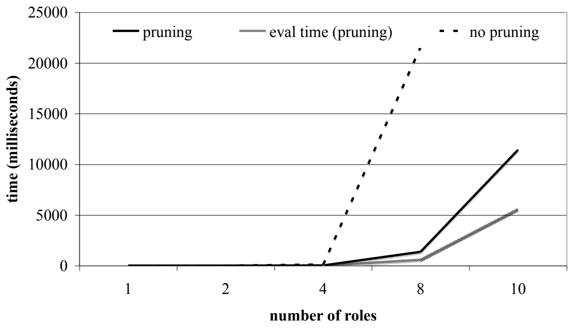
results in the graph are from the no-pruning run). Most of the time was taken up with checking for violated constraints in the repair scope; for instance, for 160 agents, the total execution time was 1,964ms, of which 1,915ms was taken in constraint evaluation.

One technique (proposed by [6]) which we have not applied, but which we expect to make a big difference to execution time, is to track which entities are used to evaluate each constraint, and then use this information to work out which constraints might be affected by a change to the design, and only re-evaluate these constraints. However, even without this, the algorithm is able to deal with a reasonable number of repair plan instances quite rapidly (just under two seconds for 161 design entities and 1,606 constraints).



We now consider the algorithm's performance as the number of constraints, and consequently the depth of the tree, is increased. We create an artificial situation with N constraints by having N roles and one agent. Since each role has a single violated constraint, this gives N constraints, and consequently a tree of depth N.

In this case pruning made a significant difference: for $N = 8$ without pruning the algorithm considered 10,590 goal nodes and 31,736 plan nodes taking a total of 21,594 milliseconds, whereas with pruning the algorithm considered 1,673 goal nodes and 3,753 plan nodes taking 1,360 milliseconds. On the other hand, the heuristic of sorting plans by their basic cost made no difference. As the graph below shows[10], the algorithm (with pruning) performs well for $N = 8$. In this experiment the evaluation time was a smaller component of the total time.



Finally, in order to assess the performance of the algorithm in a non-artificial situation, we conducted experiments on the design of a weather alerting system [9]. The initial system helps the airport weather personnel in identifying discrepancies between current weather readings and previously issued forecasts for pressure and temperature. We introduced several new requirements, and for each requirement made primary changes, and then ran the algorithm to recommend secondary changes. We report here one typical case where a new requirement is that the system shall also show Volcanic Ash alerts where volcanic activities are reported on special mailing list. The existing design model contains 93 elements and 46 constraints are considered.

With regard to the no-pruning case, the tool does not terminate as the algorithm fell into a cycle. Without the plan sorting heuristic the algorithm took 7,671 milliseconds (of which 5,887 was constraint evaluation). With the plan sorting heuristic total time was 7,921ms (with 6,090ms being constraint evaluation). This shows that, despite a worse case exponential complexity, the algorithm is practical for small to medium designs. Note that there are still a number of techniques for improving the algorithm's efficiency which we have not yet implemented.

5. RELATED WORK

A range of approaches have been proposed to deal with change propagation and inconsistency management in mainstream software engineering. A large amount of this work such as [1, 11, 13, 16] uses rule-based engines to detect and resolve inconsistencies and propagate changes. Our work uses the BDI architecture which allows for more flexibility than the rule based approach since the hierarchical relationship between plans allows for a natural representation of rules that can cascade, i.e. where fixing an inconsistency can cause further inconsistencies. Also, an event can have multiple plans that it can trigger, with plan selection being made at run-time. This allows us to represent multiple ways of resolving a given inconsistency as separate plans, with the choice between them corresponding to available traceability information, design heuristics and (possibly) human intervention.

Recently, Egyed [7] proposed an approach based on fixing inconsistencies in UML models, using model profiling to locate choices of starting points for fixing an inconsistency in a UML model. By means of model profiling, he also tried to predict the side-effects of fixing an inconsistency. However, there are several significant differences between his work and ours. Firstly, his work treats a constraint as a black box whilst we analyse the constraints to generate repair plans. Secondly, his approach does not provide options to repair inconsistencies, but only suggests starting points (entities in the model) for fixing the inconsistency.

The cost calculation algorithm can be seen as a form of reasoning about an agent's plans, albeit in a special setting. There has been previous work on investigating the interaction between plans either within a single agent or between different agents in a multi-agent system (e.g. [2, 17]). There are some similarities between this work and ours, for example, a plan's cost can be viewed as its resource consumption and the fact that fixing one constraint can partially/totally repair other constraints can be seen as positive interaction between plans. However, there are several major differences between their work and ours. First of all, the selection between applicable plans is not controllable. Secondly, the algorithms of [17] rely on a finite plan-goal tree, whereas our algorithm does not require a complete tree, rather, the search tree is pruned as soon as cheaper plans are identified.

The issue of calculating the cost of a plan or a goal in the context of existing plans has been previously addressed in [8]. The aim of their work is to determine whether an agent should adopt a new goal. They estimate the cost (with a range) rather than calculate the exact cost like our work. In addition, the plans which they consider contains only primitive actions, and they require complete plans. We also found that it is not easy to adopt their approach to deal with selecting between alternative plans, as opposed to deciding whether to adopt a goal.

Surprisingly, the specific problem of selecting between applicable plans in BDI agents has not received much attention. One particular work that tackles this issue is presented in [5]. They extend AgentSpeak(L) to deal with intention selection in BDI agents. They

[10]For $N = 10$ the no pruning case ran out of memory.

also use a lookahead technique to work out the potential cost of a plan and choose the best plan to execute, and their plan representation is also hierarchical. However, there are several differences between their work and ours. Firstly, they impose a limit on the plan-goal tree by giving the depth of the tree as an input to their algorithm. Secondly, they assume that the environment changes rapidly and expect the worst case scenarios when looking ahead. In the domain that we are interested in, the environment is static so we always choose the least cost plans. Finally, they do not consider costs in the context of existing plans.

Our process for computing cost — performing lookahead over an and-or tree — clearly resembles a planning problem, and it can be viewed as such, with a few rather specific requirements. Firstly, because we have repair plans we want to use an HTN (Hierarchical Task Network) planner. Secondly, we want to collect the set of all best (cheapest) plans, so we need a planner that supports a notion of plan cost, and is able to collect all cheapest plans. Finally, because we have a large, potentially infinite, search space, we want a planner that does pruning and loop detection. Unfortunately, we do not know of any planner that meets all three requirements. Perhaps the closest is SHOP2 [10] which is an HTN planner that supports collecting all best plans and that does branch and bound pruning. However, SHOP2 does not do loop detection, and although it provides iterative deepening, which can be used to avoid looping, this does not return the cheapest solution(s), as required. We encoded a UML design[11] and associated constraints and repair plans using SHOP2. Our experiments have shown that SHOP2 gives the same results as our cost calculation if it terminates, but that it is susceptible to looping, and that SHOP2 is slightly slower than our Java implementation (0.172 seconds vs. 0.157 seconds[12]).

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have briefly described a change propagation framework that has been implemented based on the BDI agent architecture. We then raised the issue of having multiple applicable repair plans and how to select amongst these repair plans. In order to deal with this problem, we have proposed a cost calculation mechanism for repair plans. This mechanism has been implemented, and we presented results of an empirical exploration of the scalability of the algorithm. The evaluation showed that checking for violated constraints takes up most of the execution time, that pruning does make a significant difference, and that the algorithms are practical for small to medium realistic examples.

A key area that we are currently working on is performing a case study in order to better ascertain the *effectiveness* of the approach as a "change propagation assistant". In order to ascertain this, we have integrated our implementation with the Prometheus Design Tool (http://www.cs.rmit.edu.au/agents/pdt/). Another area for future work is investigating the interaction between constraints in order to limit the number of plans to be explored and to allow for pruning more quickly.

Acknowledgements

This work has been funded by the Australian Research Council (grant LP0453486) in collaboration with Agent Oriented Software. We would like to thank Lin Padgham, Sebastian Sardina and other members of the RMIT agent group for discussions.

[11]The video-on-demand system [6], and see http://peace.snu.ac.kr/dhkim/java/MPEG/

[12]On a Windows XP PC with a 1.73Ghz CPU and 1GB RAM, using Java v.1.5.0_06 and SHOP2 v1.3 running with GNU CLISP v2.3 for Windows.

7. REFERENCES

[1] L. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of UML models. In *International Conference on Software Maintenance (ICSM)*, pages 256–265, 2003.

[2] B. J. Clement and E. H. Durfee. Top-down search for coordinating the hierarchical plans of multiple agents. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 252–259. ACM Press, 1999.

[3] K. H. Dam and M. Winikoff. Generation of repair plans for change propagation. In M. Luck and L. Padgham, editors, *Agent Oriented Software Engineering (AOSE)*, pages 30–44, Honolulu, Hawaii, May 2007.

[4] 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)*, pages 309–318. IEEE Computer Society, 2006.

[5] A. Dasgupta and A. K. Ghose. CASO: a framework for dealing with objectives in a constraint-based extension to AgentSpeak(L). In *Twenty-Ninth Australasian Computer Science Conference (ACSC 2006)*, pages 121–126. Australian Computer Society, Inc., 2006.

[6] A. Egyed. Instant consistency checking for the UML. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[7] A. Egyed. Fixing inconsistencies in UML models. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, May 2007.

[8] J. F. Horty and M. E. Pollack. Evaluating new options in the context of existing plans. *Artificial Intelligence*, 127(2):199–220, 2001.

[9] I. Mathieson, S. Dance, L. Padgham, M. Gorman, and M. Winikoff. An open meteorological alerting system: Issues and solutions. In V. Estivill-Castro, editor, *Proceedings of the 27th Australasian Computer Science Conference*, pages 351–358, Dunedin, New Zealand, 2004.

[10] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003.

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

[12] Object Management Group. Object Constraint Language (OCL) 2.0 Specification, 2006.

[13] V. Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 84–91. IEEE Computer Society, 1997.

[14] V. Rajlich. Changing the paradigm of software engineering. *Commun. ACM*, 49(8):67–70, 2006.

[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*, pages 42–55. Springer-Verlag, 1996.

[16] J. L. Sourrouille and G. Caplat. Checking UML model consistency. In *Workshop on Consistency Problems in UML-based Software Development at UML 2002*, Dresden, Germany, 2002.

[17] J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002*, pages 18–22. IOS Press, 2002.

[18] H. V. Vliet. *Software engineering: principles and practice*. John Wiley & Sons, Inc., 2nd edition, 2001. ISBN 0471975087.