Evaluating an Agent-Oriented approach for Change Propagation[*]

Khanh Hoa Dam and Michael Winikoff

RMIT University, Australia
kdam@cs.rmit.edu.au, michael.winikoff@rmit.edu.au

Abstract. A central problem in software maintenance is *change propagation*: given a set of primary changes that have been made to software, what additional *secondary* changes are needed? Although many approaches have been proposed, automated change propagation is still a significant technical challenge in software engineering. In this paper we report on an evaluation of an agent-based approach for change propagation that works by repairing violations of well-formedness consistency rules in a design model. The results have shown that given a reasonable amount of primary changes, the approach is able to assist the designer by recommending feasible secondary change options that match the designer's intentions.

1 Introduction

A large percentage of the cost of software can be attributed to its maintenance and evolution [1]. The essence of software maintenance is change: in order to adapt a system to desired requirements (be they new, modified, or an environmental change), the designer makes changes to the system. In practice, those changes form a sequence of actions (addition, removal and modification) that contains some primary changes followed by additional, secondary, changes. Primary changes are usually identified based on the characteristics of the change requests and the designer's knowledge and expertise. After that, the designer ensures that other entities in the software system are updated to be consistent with these primary changes. As a result, secondary changes are then determined and performed, mostly by identifying and fixing inconsistencies in the design previously modified by primary changes. This process is known as *change propagation* [2] and is complicated, labour-intensive and expensive, especially in complex software systems that consist of many artefacts and dependencies [3].

Therefore, it would be desirable to have a tool that automates change propagation. However, we do not believe that a tool can fully automate change propagation because a tool cannot make decisions involving trade-offs and design styles where human intervention is required. However, a tool *can* be an assistant that helps the designer by providing feasible change propagation options.

[*] This work has been funded by the Australian Research Council under grant LP0453486, in collaboration with Agent Oriented Software. We would also like to thank Lin Padgham, Sebastian Sardina and other members of the RMIT Agent Group for discussion relating to this work.

Although a substantial amount of work has looked at the issue of change propagation, most of it has focused on source code (e.g. [3, 2]). Recently, as the importance of models in the software development process has been better recognised, more work has aimed at dealing with changes at the *model* level (e.g. [4–6]) However, most existing work either fails to advocate effective automation or fails to explicitly reflect the cascading nature of change propagation, where each change (primary or secondary) can require further changes to be made.

We have developed an agent-based framework to deal with change propagation by fixing inconsistencies in a design. In other words, we identify change propagation options by finding places in the design where desired consistency constraints are violated by primary changes, and we then fix them. This approach represents options for repairing inconsistencies ("repair plans") using event-triggered plans, as is done in Belief-Desire-Intention (BDI) architectures.

The main focus of this paper is on an evaluation of the effectiveness of this approach in assisting with change propagation.

In the sections ahead, we first review this approach to change propagation including repair plan generation and execution (section 2). We then describe the Change Propagation Assistant (CPA) tool, which we have implemented and integrated with an existing modeling tool (section 3). Section 4 is the main focus of this paper in which an evaluation of the framework and our prototype tool is reported. We then discuss some related work in section 5 before concluding and outlining some future work (section 6).

2 An overview of the approach

Our approach to change propagation is to define consistency conditions using a (UML) **meta-model** and (OCL [7]) **consistency constraints**, and then use a library of **repair plans** to fix inconsistencies in the **design model**. Consistency constraints define conditions that all models must satisfy for them to be considered valid. These conditions may include syntactic well-formedness, coherence between different diagrams, and even best practices. Figure 1 depicts a very small excerpt of a UML metamodel [8] and below is an example consistency condition (in both OCL and logic) between class diagrams and sequence diagrams.

Constraint 1 *The name of a message (in sequence diagrams) must match an operation in a receiver's class (in class diagrams).*
Context Message inv c(self)*:*
self.receiver.base.operation→exists(op : Operation | op.name = self.name)
$\exists\, op \in self.receiver.base.operation : op.name = self.name$

There are two important properties of change propagation: (a) it is cascading, i.e. performing an action to fix an inconsistency can cause further inconsistencies which require further actions and (b) multiple choices, i.e. there are usually many ways of making the design consistent again. Those two properties are interestingly similar to the characteristics of the well-known and studied Belief-Desire-Intention architecture [9], where software agents have a library of plans ("recipes") which are triggered by events. Each plan specifies which event it is triggered by, under what conditions it should be

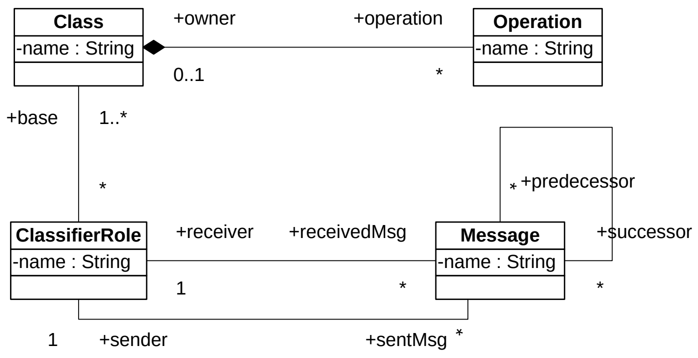

Fig. 1. A small excerpt of the UML metamodel

considered to be applicable (the "context condition"), and a plan "body": a sequence of steps that are what the plan does when it is executed. Steps in the plan body can create events, which result in further plans being triggered. A given event can have multiple plans that are triggered by it, and the plans' context conditions are used to select a plan to execute.

Based on that observation, repairing a violated constraint (an inconsistency) is represented as an event and the way to fix the violated constraint as (repair) plans. In previous work [10], the syntax for repair plans is defined based on AgentSpeak(L) [11]. Each repair plan, P, is of the form $E : C \leftarrow B$ where E is the triggering event; C is an optional "context condition" (Boolean formula) that specifies when the plan should be applicable[1]; and B is the plan body, which can contain sequences ($B_1;\ B_2$) and events which will trigger further plans (written as $!E$). We extend AgentSpeak(L) by allowing the plan body to contain primitive actions such as adding and deleting entities and relationships, and changing properties; and also to contain conditionals and loops.

Below is an example of repair plans for fixing constraint 1, i.e. $c(\mathit{self})$ defined above. We use $c_t(\mathit{self})$ to denote the event of making $c(\mathit{self})$ true (similarly for $c1$). We also define the following abbreviations: $\mathit{SE} \overset{\text{def}}{=} \mathit{self.receiver.base.operation}$ and $c1(\mathit{self}, \mathit{op}) \overset{\text{def}}{=} \mathit{op.name} = \mathit{self.name}$.

P1 $c_t(\mathit{self}) : \mathit{op} \in \mathit{SE} \leftarrow !c1_t(\mathit{self}, \mathit{op})$
P2 $c_t(\mathit{self}) : \mathit{op} \in \mathit{Set}(\mathit{Operation}) \wedge \mathit{op} \notin \mathit{SE} \leftarrow$ add op to SE ; $!c1_t(\mathit{self}, \mathit{op})$
P3 $c_t(\mathit{self}) \leftarrow$ create $\mathit{op} : \mathit{Operation}$; add op to SE ; $!c1_t(\mathit{self}, \mathit{op})$
P4 $c1_t(\mathit{self}, \mathit{op}) \leftarrow$ change $\mathit{op.name}$ to $\mathit{self.name}$
P5 $c1_t(\mathit{self}, \mathit{op}) \leftarrow$ change $\mathit{self.name}$ to $\mathit{op.name}$

[1] 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.

In the above example, there are three plans that make $c(\mathit{self})$ true (P1, P2, and P3). Plan P1, for instance, posts an event $c1_t(\mathit{self}, op)$ which in turn can trigger either plan P4 and P5. The context condition of plan P1, $op \in SE$, indicates that at run time it can generate several plan instances, each for an operation belonging to the message's receiver's class. Informally, plan P1 fixes $c(\mathit{self})$ by either changing the name of an existing operation contained in the message's receiver's class to the name of the message (plan P4) or vice versa (plan P5).

In this change propagation framework, the repair plans are generated automatically (at design time) from the constraints and metamodel [10], and form a plan library which is used at run time. One key consequence of generating plans from constraints, rather than writing them manually, is that, by careful definition of the plan generation scheme, it is possible to guarantee that the plans generated are correct, complete, and minimal [10].

At run time, after primary changes are made to the design model, all of the constraints are evaluated, and violated constraints are repaired. Each violated constraint will usually have several repair plan instances for fixing it. While one repair plan may also fix other violated constraints, another may break constraints. As a result, selecting a repair plan needs to take into account its effect on other constraints. We have therefore defined an algorithm [12] which calculates a cost for each repair plan instance, taking into account its consequences, i.e. which constraints it may break or fix. The designer is presented with a set of the cheapest repair options, and they select one to apply to the design.

3 Implementation

We have recently implemented the Change Propagation Assistant (CPA), a prototype tool that demonstrates how the above approach works in practice. We have developed a PDT Interface Communicator component which provides an API that is used to integrate the change propagation tool with the Prometheus Design Tool (PDT[2]), a freely-available tool supporting designers using the *Prometheus* methodology [13] for building agent-based systems. The CPA uses the Dresden OCL Toolkit[3] to parse OCL constraints.

Our tool includes a Plan Creator component which generates (at design time) a repair plan library from the constraints and metamodel.

When the designer requests the CPA for help the PDT design model is converted by a model transformer component to a MOF[4]-compliant model which is stored in a NetBean MDR[5] repository. The Constraint Checker component identifies which constraints are violated and then instructs the Constraint Repairer component to find plans for fixing them, using the repair plans library. The Constraint Repairer performs the

[2] http://www.cs.rmit.edu.au/agents/pdt

[3] An open source project providing various tools for OCL http://dresden-ocl.sourceforge.net

[4] Meta Object Facility(MOF) is an OMG standard [14] for defining metamodels and metadata repositories.

[5] http://mdr.netbeans.org/

cost calculation and returns to the designer a set of cheapest repair options. If the designer accepts one of the options proposed then the Constraint Repairer instructs the PDT Interface Communicator to apply those changes to the current PDT design model.

4 Evaluation

Having implemented the above approach for a change propagation assistant we would now like to perform an empirical evaluation of the effectiveness of the approach and tool. The key question is how well this approach works in practice and, specifically, how useful is it likely to be to a practising software designer who is maintaining and evolving a system?

Unfortunately, an evaluation to answer this question raises a number of challenges and questions such as: which methodology should be used? which application(s) should be used? what changes to the system should be done? and, how do we select primary changes to perform?

Our original plan was to use the UML design models, but the effort involved in implementing all of the constraints in the UML standard was beyond our resources, and so instead we have chosen to use the *Prometheus* [13] methodology for the design of agent systems. The Prometheus notation is simpler than UML, and in addition to local expertise, we had easy access to the source code of the Prometheus Design Tool (PDT), allowing the Change Propagation Assistant to be integrated with PDT. We used the Prometheus metamodel described by [15], and defined consistency constraints by examining the well-formedness conditions of Prometheus models, the coherence requirements between Prometheus diagrams, and best practices proposed by [13].

Our choice of application was the Bureau of Meteorology's multi-agent system for weather alerting (MAS-WA) [16]. This application was chosen because the prototype system developed by the Australian Bureau of Meteorology had been extended in a number of ways, and these extensions gave us well-motivated and realistic change scenarios to evaluate.

The purpose of the MAS-WA application is to monitor a range of meteorological data, and alert forecasters to situations such as extreme weather, inconsistencies between data sources, or changes to observed weather that contradict previously issued forecasts. We used a version of the system that simplified the application while retaining its key characteristics [17]. The simplified system monitored data from forecasts for airport areas (TAF) and from automated weather stations (AWS). TAF and AWS readings contain information about temperature, wind speed and pressure. The system issues alerts if there are significant differences between a prediction (TAF) and the actual weather (AWS). Figure 2 shows the system overview diagram for the application, as well as agent overview diagrams for the *Discrepancy* and *GUI* agent types.

Ideally, the evaluation would be done by giving the CPA to a group of selected users, who would be asked to work with the tool to implement requirement changes. However, due to time and resource limits, we were not able to conduct such a user study evaluation. In order to overcome this obstacle, we approached the evaluation by defining an abstract user behaviour in maintaining/evolving an existing design. We then

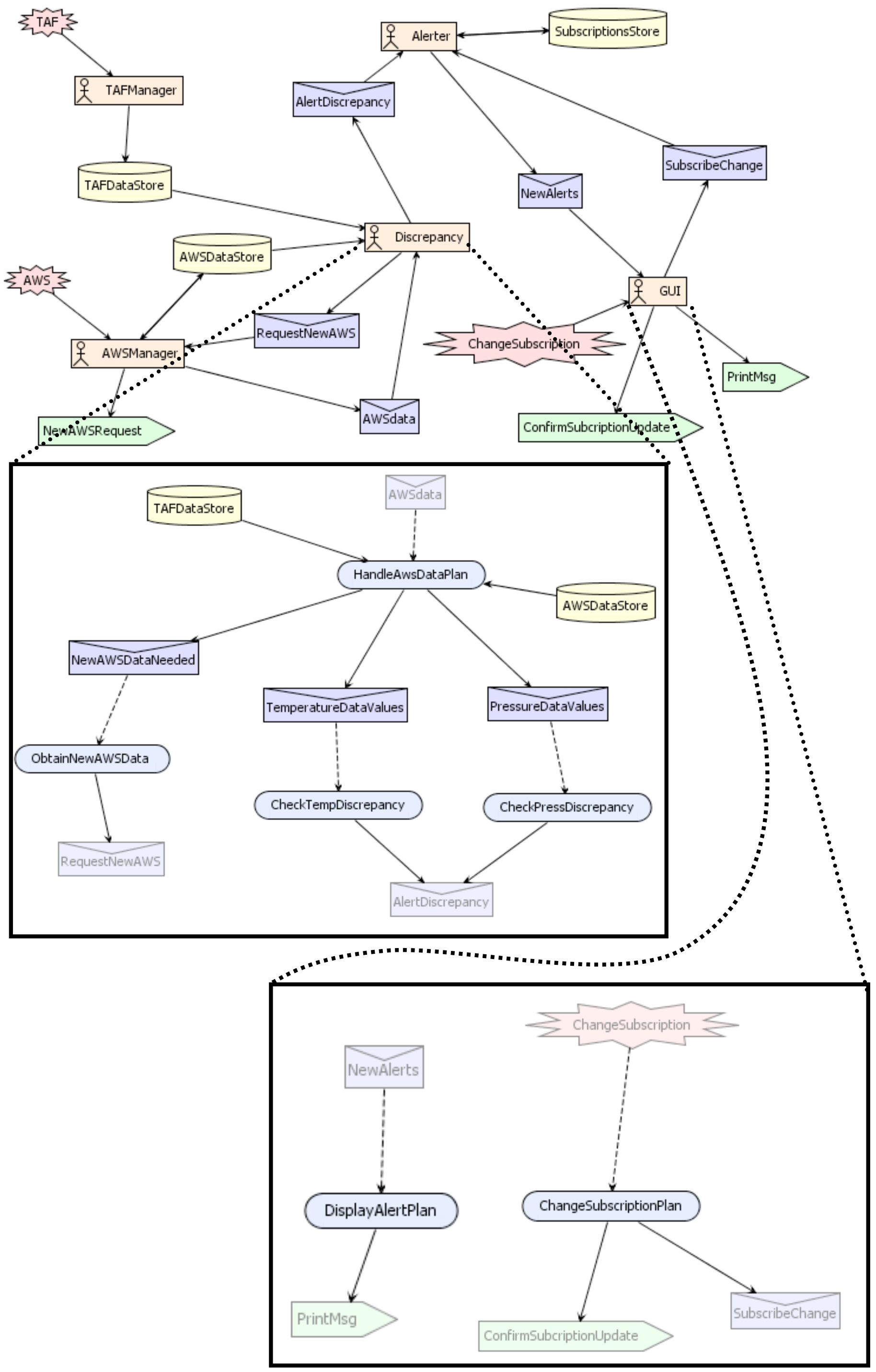

Fig. 2. MAS-WA Design

simulated a real user by following this behaviour: repeatedly making changes to the design, and invoking and assessing the responses from the CPA tool.

Our model of abstract user behaviour below is based on the model of change propagation process of Hassan and Holt [18]. In this model, the developer is guided by a change request to perform primary changes (i.e. determine initial entity to change and change entity) and some partial secondary changes (i.e. determine other entities to change). When the developer cannot locate other entities to change, she/he consults a Guru (which could be a person, a tool or a suite of test; in our case, it is the CPA tool[6]), and if the Guru indicates that an entity was missed, then it is changed and the change propagation process is repeated for that entity. This continues until all appropriate entities have been changed.

In order to evaluate how useful the CPA is, we consider a given change to the system's design that was done in order to meet a new requirement. We view such a change (denoted by D, and not to be confused with the repair plans) as being a sequence of actions, with each action being a primitive change to the model such as adding or removing a link between entities. We then ask what proportion of the actions in the change was done by the user, and what proportion was done by the CPA.

However, this metric is not that simple to use, because it depends on the choice of change for a given requirement, and on the choice of primary change, i.e. how much of the change is done before invoking the tool. For each given change, D, we need to consider a range of possible primary changes P (with the actions in P being a subset of those in D). Now in fact, D is a sequence, and the abstract user behaviour below uses D sequentially, thus the possible primary changes are the initial segments of D.

We now define a process that captures (abstractly) the designer's behaviour. In the following process D denotes the designer's planned change: a sequence of actions that transforms the existing design model so that it meets the new requirement. The designer performs an initial segment P of D (step 2), updates D by removing the performed actions (step 3) and then invokes the tool, which returns a set $\mathcal{O}$ of repair options (step 4). Each repair option C_i is a sequence of actions. At this point (step 5) the user may select one of the C_i (step 6) and apply it to the model (step 7), or he/she may decide that none of the C_i is suitable. Deciding whether a C_i is suitable is done by comparing it with the designer's plan: C_i is *compatible* with D if all actions in C_i are in D (formally[7] $\forall\, a \in C_i : a \in D$). If all the changes in D have been performed the process ends, otherwise the user continues to perform more primary changes (step 9). We use $D := D - P$ to denote removing the actions in P from the sequence D, and we use $P \sqsubseteq D$ to denote that P is an initial segment of D (formally $\exists\, X : P + X = D$, where $+$ is sequence concatenation).

1. given a planned change D (sequence of actions)
2. select $P \sqsubseteq D$
3. do the actions in P and update D ($D := D - P$)
4. invoke the tool yielding $\mathcal{O} = \{C_1, \ldots, C_n\}$
5. if $\exists\, C_i \in \mathcal{O}$ where C_i is compatible with D then

[6] Unlike the Guru in their model, the CPA suggests not only entities to be changed, but also the specific changes to be made to them.

[7] Alternatively, if viewed as sets, $C_i \subseteq D$.

```
6.        select a compatible $C_i \in \mathcal{O}$ (if more than one)
7.        do actions in $C_i$ and update $D$ ($D := D - C_i$)
8.  end if
9.  goto step 2 if $D$ is not empty.
```

We thus have the following evaluation process: for each new requirement we develop (and justify!) a change plan D, and then apply the process above, considering a partial change P that expands by one step at a time. When the process terminates we count how many actions ended up being in C_is along the way, compared with the total number of actions in D so we calculate the metric $M = \mid C \mid / \mid D \mid$ (where C is the union of the C_is).

Note that the value of M depends on our choices for P. Clearly, if P is the whole of D then there is nothing left for the tool to do, and M will be 0. In some of the cases below we will see that there is a "tipping point": until enough of D is done the tool cannot help, but once enough is done, the tool performs the remaining steps in D.

In addition to measuring M, an important factor in the usefulness of the tool concerns the repair options, $\mathcal{O}$. Specifically, we are interested in how many of the C_i in $\mathcal{O}$ are compatible with D, and in the size of $\mathcal{O}$ (since it is clearly better if the designer is not being asked to select an option from a very large list). We thus, in addition to M, also measure the number of options and how many of the options are compatible with D.

Finally, we need to select values for the basic costs of action. In this evaluation we do not explore a range of costs, but instead select what we believe are reasonable values: addition, creation and modification are assigned the same cost (e.g. 1) and we consider that deletion is not normally a desirable action, and so give it a higher cost (e.g. 5).

4.1 Results and Analysis

Change requests can be classified into several categories, depending on the dimensions we are looking at. Swanson initially identified three categories of maintenance: corrective, adaptive, and perfective [19]. These have since been extended with perfective maintenance[8] and has become an ISO/IEC standard [20].

We also view maintenance in terms of the type of modifications made to the software system. More specifically, they can be: (a) adding a new functionality/feature; (b) removing an existing functionality/feature; and (c) modifying an existing functionality/feature.

We introduce four requirement changes to the MAS-WA application that cover most of the change types, according to the above classification of changes. They include: logging all alerts sent to the forecast personnel (preventative and functionality modification); adding wind speed alerting (adaptive and functionality addition); implementing a variable threshold alerting (perfective and functionality addition); and adding volcanic ash alert (perfective and functionality addition). Due to space limitations, we describe only one change in detail, although we do present results for all of the changes.

[8] modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

Change	1	2	3	4	5	6	7	8	9	10	11	12	13	M
Logging	24_1	T	T											67%
Wind Speed Alert	50_0	6_1	T	T	U									40%
Variable Threshold	4_0	16_0	12_1	T	T	26_3	T	T	11_2	T	U			45%
Volcanic Ash	24_0	24_0	1_1	52_1	T	T	T	T	78_0	28_1	T	T	T	54%

Fig. 3. Evaluation Results

Change: Implementing a Variable Threshold Alerting Currently, the alerting levels are fixed and hard-coded. However, the forecast personnel wants to be able to adjust the alerting levels, e.g. different regions will show alerts based on different discrepancies. Hence, a new requirement that the forecast personnel should be able to set a threshold for alerting is requested.

To meet this change request, the "GUI" and "Discrepancy" agents have to be changed. More specifically, the user's request for changing the thresholds will be represented by two percepts, each for temperature and pressure. A new plan is likely required to handle those percepts. A new data store that keeps the threshold information is introduced. Finally, the "Discrepancy" agent's plans for detecting discrepancies, i.e. "CheckTempDiscrepancy" and "CheckPressDiscrepancy", will need to use the new threshold data store. The designer's planned change D thus consists of the sequence[9]:

1. Create "SetNewTempThreshold" percept
2. Create "SetNewPressThreshold" percept
3. Create "ChangeThreshold" plan in "GUI" agent
4. Make "SetNewTempThreshold" percept to be a trigger of "ChangeThreshold".
5. Make "SetNewPressThreshold" percept to be a trigger of "ChangeThreshold".
6. Create a new "AlertingLevels" data
7. Link "GUI" agent to "AlertingLevels" data
8. Link "ChangeThreshold" plan to "AlertingLevels" data
9. Link "AlertingLevels" data to "Discrepancy" agent
10. Link "AlertingLevels" data to "CheckTempDiscrepancy" plan
11. Link "AlertingLevels" data to "CheckPressDiscrepancy" plan

Since the first three steps involves the creation of new entities and the later steps include actions related to those new entities, the tool does not return any compatible options until step 3 is performed (and the designer may well defer invoking the tool until s/he has created all three entities). One of the options proposed by the tool is making the two new percepts to be a trigger of the new "ChangeThreshold" plan. The tool does not suggest any further actions because the design is then consistent. The user then performs step 6 and the tool then recommends either steps 7 and 8 or steps 9 and 10 or steps 9 and 11. Assume[10] that the user chooses the option containing steps 7 and 8, and then performs step 9. The CPA recommends either step 10 or 11. The user chooses one of these and has to manually perform the other.

[9] Some variations in order are possible, but they do not affect the evaluation outcome

[10] If the user makes a different choice the overall M is still the same, just different actions are done by the tool.

Figure 3 shows the results of evaluation for the four changes. Each change has a row, where the entries marked with numbers show the situation for the *nth* step of the user's plan (D). An entry of the form n_m indicates that the tool returned n options (i.e. $\mathcal{O} = \{C_1 \dots C_n\}$), where m of the C_i were compatible with D. An entry "T" indicates that this step is done by the tool, that is, it is part of a selected repair plan from an earlier step. An entry "U" indicates that the user performs this change; this occurs in two places where D is non-empty, but the design is consistent, and in this situation the tool cannot assist the user. The final column gives the value of the metric $M = \mid C \mid / \mid D \mid$. Overall, for the four changes the average value of M is approximately 50%, that is, compared with maintenance without our tool, the user would have to perform roughly twice as many change actions.

5 Related work

There has been a lot of interest in addressing the issue of assisting software engineers to deal with software changes. In particular, change impact analysis has been extensively investigated but most of the work has focused on source code. Many of the impact analysis approaches are discussed in [3] and are typically used 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. Although these approaches are very powerful, they do not readily apply to design models [5]. In addition, our work focuses more on *implementing* changes, i.e. propagating changes between design artefacts in order to maintain consistency as the software evolves.

There have been several works that specifically target fixing inconsistencies in design models. The work by [6] provides a framework which automatically derives a set of repair actions from the constraint by analyzing consistency rules expressed in first-order logic and models expressed in xlinkit [21]. However, their work considers only a single inconsistency and consequently does not explicitly address dependencies among inconsistencies and potential consequences of repairing them, e.g. fixing one constraint can repair or violate others.

Recently, Egyed proposed an approach based on fixing inconsistencies in UML models [5]. The approach uses model profiling to locate possible starting points for fixing an inconsistency in a UML model. He also tried to use model profiling to predict the side-effects of fixing an inconsistency. His work, however, treats a constraint as a black box whilst we analyse the constraints to generate repair plans. Similarly the work of Briand et al. also looks at how to identify impacted entities during change propagation using UML models [4]. It defines specific change propagation rules (also expressed in OCL) for a taxonomy of changes. However, the major difference between both of these works and ours is that their approaches do not provide options to repair inconsistencies, but only suggest starting points (entities in the model) for fixing the inconsistency.

6 Conclusions and Future work

We have presented a novel agent-oriented approach to deal with change propagation by fixing inconsistencies in the design models. The approach has been implemented in a form of a prototype tool (i.e. the CPA tool) that assists the designer in propagating changes. We have also evaluated the effectiveness of the approach by applying it to Prometheus using the design of a real application, MAS-WA.

The evaluation demonstrated that the approach is effective given that a reasonable amount of primary changes are provided. However, there are several threats to the validity of our study. For instance, although the set of changes are motivated by real change requests, and cover most of the change types, they may not be representative of all changes. We also need to test the approach with different application types. Additionally, there is scope for evaluation with other methodologies and notations (e.g. UML), with a range of basic costs, and, of course, with human subjects.

One issue that arises in the proposed approach relates to the use of inconsistency as a driver for change. As seen in the evaluation, not all changes result in inconsistency, and in these cases the approach will not be able to completely identify the desired secondary changes. An opposite issue is that, as argued by [22], not all inconsistency should be fixed; this is easy to deal with by simply allowing certain constraint types or instances to be marked as "to be ignored".

In some cases there may be a large number of repair options returned by the tool, which makes it hard for the user to select which one to use. In practice this can be dealt with by ignoring the tool's list of options and performing further changes (which often provides the tool with information that enables it to return fewer options). A better approach which needs to be investigated is reducing the number of options by "staging" questions. Suppose we need to link a percept with a plan and with an agent, then instead of presenting a set of options, where each option specifies both a plan and an agent (which gives a cross product), specify first the choice of agent, and then based on that choice ask for a choice of (relevant) plan.

Overall, our conclusion is positive since the evaluation shows that the approach is able to (on average) perform approximately half of the actions in maintenance plans, across a number of changes motivated by experience with a real application.

References

1. Vliet, H.V.: Software engineering: principles and practice. 2nd edn. John Wiley & Sons, Inc. (2001) ISBN 0471975087.
2. 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
3. Arnold, R., Bohner, S.: Software Change Impact Analysis. IEEE Computer Society Press (1996)
4. Briand, L.C., Labiche, Y., O'Sullivan, L., Sowka, M.M.: Automated impact analysis of UML models. Journal of Systems and Software **79**(3) (March 2006) 339–352
5. Egyed, A.: Fixing inconsistencies in UML models. In: Proceedings of the 29th International Conference on Software Engineering (ICSE). (May 2007)

6. 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
7. Object Management Group: Object Constraint Language (OCL) 2.0 Specification (2006)
8. Object Management Group: Unified Modeling Langague Specification (UML 1.4.2, ISO/IEC 19501) (2005)
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. Dam, K.H., Winikoff, M.: Generation of repair plans for change propagation. In Luck, M., Padgham, L., eds.: Agent Oriented Software Engineering (AOSE), Honolulu, Hawaii (May 2007) 30–44
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.: Cost-based BDI plan selection for change propagation. In: Autonomous Agents and Multi-Agent Systems (AAMAS). (2008) (to appear).
13. Padgham, L., Winikoff, M.: Developing intelligent agent systems: A practical guide. John Wiley & Sons, Chichester (2004) ISBN 0-470-86120-7.
14. Object Management Group: Meta Object Facility Specification (MOF 1.4) (2002)
15. 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
16. Mathieson, I., Dance, S., Padgham, L., Gorman, M., Winikoff, M.: An open meteorological alerting system: Issues and solutions. In Estivill-Castro, V., ed.: Proceedings of the 27th Australasian Computer Science Conference, Dunedin, New Zealand (2004) 351–358
17. Jayatilleke, G.B., Padgham, L., Winikoff, M.: A model driven development toolkit for domain experts to modify agent based systems. In: Agent Oriented Software Engineering (AOSE). (2006)
18. Hassan, A.E., Holt, R.C.: Predicting change propagation in software systems. In: ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance, Washington, DC, USA, IEEE Computer Society (2004) 284–293
19. 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
20. ISO/IEC 14764: Information technology - software maintenance. ISO: Geneva, Switzerland (1999)
21. 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
22. Fickas, S., Feather, M., Kramer, J., eds.: Proceedings of the Workshop on Living with Inconsistency, Boston, USA (1997)