1-1-2006

An agent-oriented approach to change propagation in software evolution

Khanh Hoa Dam
Royal Melbourne Institute of Technology, hoa@uow.edu.au

Michael Winikoff
Royal Melbourne Institute of Technology

Lin Padgham
Royal Melbourne Institute of Technology

An agent-oriented approach to change propagation in software evolution

Abstract

Software maintenance and evolution are inevitable activities since almost all software that is useful and successful stimulates user-generated requests for change and improvements. One of the most critical problems in software maintenance and evolution is to maintain consistency between software artefacts by propagating changes correctly. Although many approaches have been proposed, automated change propagation is still a significant technical challenge in software engineering. In this paper we present a novel, agent-oriented approach to deal with change propagation in evolving software systems that are developed using the Prometheus methodology. A metamodel with a set of the Object Constraint Language (OCL) rules forms the basis of the proposed framework. The underlying change propagation mechanism of our framework is based on the well-known Belief-Desire-Intention (BDI) agent architecture. Traceability information and design heuristics are also incorporated into the framework to facilitate the change propagation process.

Keywords

agent, approach, software, evolution, propagation, change, oriented

Disciplines

Engineering | Science and Technology Studies

Publication Details

Dam, K. Hoa., Winikoff, M. & Padgham, L. (2006). An agent-oriented approach to change propagation in software evolution. 2006 Australian Software Engineering Conference, ASWEC 2006 (pp. 309-318). Australia: IEEE.

An agent-oriented approach to change propagation in software evolution

Khanh Hoa Dam, Michael Winikoff, and Lin Padgham
School of Computer Science and Information Technology
RMIT University
GPO Box 2476V, Melbourne, 3010, Australia
{kdam,winikoff,linpa}@cs.rmit.edu.au

Abstract

Software maintenance and evolution are inevitable activities since almost all software that is useful and successful stimulates user-generated requests for change and improvements. One of the most critical problems in software maintenance and evolution is to maintain consistency between software artefacts by propagating changes correctly. Although many approaches have been proposed, automated change propagation is still a significant technical challenge in software engineering. In this paper we present a novel, agent-oriented approach to deal with change propagation in evolving software systems that are developed using the Prometheus methodology. A metamodel with a set of the Object Constraint Language (OCL) rules forms the basis of the proposed framework. The underlying change propagation mechanism of our framework is based on the well-known Belief-Desire-Intention (BDI) agent architecture. Traceability information and design heuristics are also incorporated into the framework to facilitate the change propagation process.

1. Introduction

Software maintenance and evolution is an important and lengthy phase in the software life-cycle which can account for as much as two-thirds of the total software development costs [34, page 449]. Software maintenance activities are usually classified as *adaptive* maintenance (changing the system in response to changes in its environment so it continues to function), *corrective* maintenance (fixing errors), and *perfective* maintenance (changing the system's functionality to meet changing needs). An early study [17] suggested that a significant proportion of the maintenance effort is concerned with perfective maintenance. The importance of software maintenance and evolution has been further emphasised with the emerging popularity of incremental development and release approaches to software development.

A critical issue in software evolution is how to propagate changes so that consistency is maintained between different artefacts. When some part of the software is altered, other parts of the system may need to change as well. The identification of these parts and the changes that need to be made to them is a very difficult task. Although impact analysis techniques [1] address this problem to a certain extent, these approaches are still labour-intensive manual solutions, and managing inconsistencies in software models remains a challenging problem [22, 31]. In particular, there is a need for *automated* support for change propagation.

Much of the work that has been done in change propagation has been addressing the issue at the code level [11, 26, 27]. Recently, however, 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 [15, 16, 31]. The work we present here deals with propagating changes through *agent oriented* design models. As far as we are aware, no work has yet been done in this area. The models we use are those based on the Prometheus design methodology [23], which is a detailed, full life-cycle methodology, providing a range of different models for designing agent based systems.

We also use an *agent-oriented* approach to deal with the change propagation issue. The framework we present is based on the well-known Belief Desire Intention (BDI) agent architecture [28] in conjunction with the use of Object Constraint Language (OCL) [29] and traceability. A software agent [36] is a piece of software which is *situated* in an environment, *autonomous* (i.e. acts on its own), *social* (interacts with other similar entities), and which appropriately balances being *reactive* (responding to changes in its environment) with being *proactive* (working to achieve its goals).

The particular agent framework that we use is a *Belief-Desire-Intention* (BDI) framework. The BDI family of agent theories, languages and systems are inspired by the philosophical work of Bratman [5] about how humans do

Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)
1530-0803/06 $20.00 © 2006 IEEE

resource bounded practical reasoning. These systems use the concepts of beliefs, desires, intentions, and predefined hierarchical plans or "recipes". The BDI architecture is realised in a number of agent platforms (e.g. see [4]), and provides a flexible, robust approach for our change propagation system.

The work we describe in this paper is closely related to work on rule-based engines to detect and resolve inconsistencies [12, 30, 35]. In these approaches, rules are defined in terms of constraints and actions in such a way that if a constraint is violated, one or more actions will be performed. In [30] such rules then form the knowledge base of an expert system which gives advice to users to repair inconsistencies in their working models. In [12, 35], the event-driven consistency check approaches make an improvement in terms of efficiency by incrementally re-validating only the context of the last changes and not the whole model.

Our framework also uses an event-driven mechanism, where events trigger the appropriate BDI plans to resolve the inconsistency. The BDI architecture allows for more flexibility than the rule based approaches as new or alternative ways of resolving an inconsistency can readily be added via additional plans, without changing the previous structure. Additionally, the hierarchical relationship between plans which consist of actions to repair inconsistencies allows for a natural representation of rules that can cascade, i.e. where fixing an inconsistency by performing an action can cause further inconsistencies requiring further action (see 4.2 for details). Finally, unlike the expert system approach, our framework takes advantage of the situatedness of agents to directly perform changes to the model rather than just giving advice to users.

In the next section we briefly introduce the Prometheus methodology. We then describe a meta-model for Prometheus and some example well-formedness constraints. Section 4 then describes our change propagation framework, followed by an example in section 5. We conclude with some comments on related work, and discussion of future directions.

2. The Prometheus methodology

Prometheus[1] is a prominent agent-oriented software engineering methodology which has been used and developed over a number of years. The methodology is complete, described in considerable detail, and has tool support[2]. The description in this paper is necessarily extremely brief, and for further details we refer the reader to [23]

[1] Prometheus was the wisest Titan. His name means "forethought" and he was able to foretell the future. Prometheus is known as the protector and benefactor of man. He gave mankind a number of gifts including fire. (www.greekmythology.com)

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

The Prometheus methodology consists of three phases: *system specification*, *architectural design* and *detailed design*. In this section, we describe them briefly with the main focus being on the artefacts that are produced at each stage.

2.1. System Specification

The system specification phase involves: identifying *actors* and their interaction with the system; developing *scenarios* illustrating the system's operation; identifying *system goals* and sub-goals; and grouping goals into the basic *roles* of the system.

The concept of actors in Prometheus is similar to that of object-oriented analysis. Actors are any stakeholders who will interact with the system to achieve some goals, and can be humans or other software systems. For each actor, *percepts* which are inputs from the actor to the agent system are identified. In addition, outputs from the system to actors (*actions*) are identified. A *stakeholder diagram*[3] is used in Prometheus to describe the relationship between actors, percepts, actions and scenarios.

Similar to identifying use cases in the object-oriented approach, the interaction between each actor and the system is described using scenarios in Prometheus. Each interaction scenario is described in a structured form which includes a sequence of steps, where each step can be an action being performed by a role, a percept being received by a role, a goal being achieved by a role, or a sub-scenario[4].

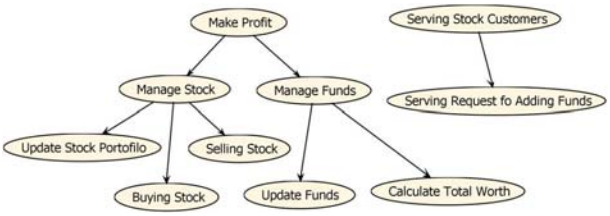

Figure 1. Example of a goal overview diagram for a stock trading management system

The system goals are identified on the basis of the initial scenarios as described above. Further goals are then elicited using abstraction and refinement techniques, as well as by developing scenario steps. This results in a goal hierarchy which is represented in a *goal overview diagram*. Figure 1 shows an example goal overview diagram for a stock management system where the goal *Make Profit* has two subgoals: *Manage Stock* and *Manage Funds*.

[3] This is being changed to an analysis overview diagram for greater flexibility.

[4] There is also an "other" step type which can be used to represent miscellaneous things such as waiting for a response.

The final step of the system specification phase involves identifying *roles*. Roles are obtained by grouping similar goals, and also including the percepts and actions associated with the included goals. A *role diagram* is used to capture the roles, and their percepts, actions and goals.

2.2. Architectural Design

The major purpose of the architectural design phase in Prometheus is to identify the agent types within the agent system and the interactions between these agent types. The main steps of this phase are: determining what agent types will be implemented; developing the interaction diagrams and interaction protocols that describe the dynamic behaviour of the system; and developing the system overview diagram which captures the system's overall (static) structure.

Agent types are derived as groups of one or more roles. The choice of grouping is guided by considerations of coupling and cohesion which are identified with the aid of the *data coupling diagram* and *agent acquaintance diagram*.

Once the agent types have been determined it is possible to start defining the interactions between them using *interaction protocols*. These protocols capture the dynamic behaviour of the system by defining the intended valid sequences of messages between agents. The interaction protocols are developed from *interaction diagrams* which in turn are based on the scenarios developed in the system specification phase. Interaction protocols can be captured using a range of possible notations. The Prometheus methodology does not prescribe a particular notation, but the Agent UML (AUML[5]) notation is often used.

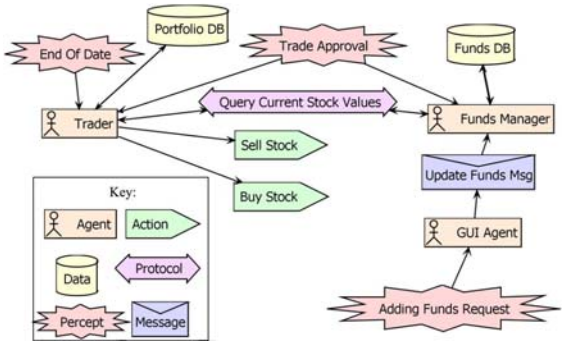

Figure 2. System overview diagram for a stock trading management system

The system's (static) structure is captured in a *system overview diagram* which gives the software engineers a general picture of how the system as a whole is structured. It shows the agent types, the communication links between them, and data. It also shows the system's boundary and its environment (actions, percepts, and external data). For example, figure 2 shows the system overview diagram for the stock management system. It shows, amongst other things, that the *Trader* and *Funds Manager* agent types communicate following the *Query Current Stock Values* protocol; that the *End Of Date* percept is handled by the *Trader* agent type; and that the *Trader* agent type performs the actions of *Buy Stock* and *Sell Stock*.

[5]http://www.auml.org

2.3. Detailed Design

The internal structure of each agent and how it will accomplish its goals within the overall system are addressed in this phase. Specifying agent internals in Prometheus is a process of progressive refinement, including: defining and developing capabilities (modules within agents) and their relationships; developing process diagrams depicting the internal processing of each agent related to the protocol specifications; and developing plans, events, and data and their relationship. Most of the Prometheus methodology does not assume any particular agent architecture. However, the lowest layer of plans and events does assume that the target agent architecture is plan-based (which is the case for BDI agent platforms). This could easily be exchanged for an alternative architecture if desired, but a detailed design which is close to code, must make some assumptions about the implementation architecture. *Agent overview diagrams* and *capability overview diagrams* capture the structure of the capabilities, sub-capabilities, plans, events and data within the agent.

3. The Prometheus meta-model and well-formedness constraints

A step towards the automation of change propagation and consistency maintenance between those Prometheus artefacts described earlier is formalising the relationships between all Prometheus entities (goals, agents, roles, percepts, actions, etc.). We do this by developing a meta-model for Prometheus and identifying well-formedness constraints for this model.

3.1. Prometheus meta-model

As described in section 2, following the Prometheus methodology produces a range of artefacts at different stages of the software development lifecycle. These artefacts form a semantically consistent abstraction of an agent system to be built. Each artefact represents a different aspect or abstraction level of the underlying system and can be seen as a "view" on the full underlying model. Each view depicts various associations between the Prometheus

concepts or entities, such as actors, goals, agents, roles, percepts, actions, etc. Many of the entities may appear in different views. For instance, the same percept entity can appear in the stakeholders diagram, a scenario descriptor, the role diagram, the system overview diagram, an agent overview diagram and a capability overview diagram. In each of these views, the percept has associations with other entities. This repetition of entities across different views induces dependencies among them.

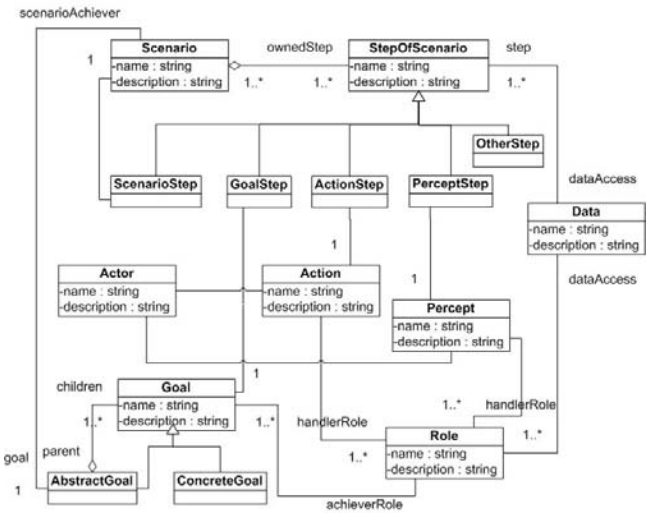

Figure 3. Prometheus meta-model (Part 1 - System Specification)

Figures 3 and 4 are a UML representation of the meta-model we have developed[6]. They summarize the structure of the underlying model and all the specified relations between entities. Figure 3 depicts entities described in the System Specification phase, while figure 4 depicts those described in the design phases. There are two types of goals: abstract goals and concrete goals. Concrete goals have no children (sub-goals) while abstract goals can have children. A scenario consists of a sequence of steps which are associated with corresponding entities such as goal steps and goals, action steps and actions, etc. An agent consists of several capabilities or plans and each capability contains some plans and/or sub-capabilities. Note that the relationship between agents and plans is constrained to be transitive: if an agent has a capability, and that capability has some plans, then that agent is deemed to also have these plans. A plan sends and receives messages as well as performs some actions which may include accessing data to handle a percept or to achieve some goals. As a result, agents and capabilities also have these associations with goals, percepts, actions, messages, and data[7]. There are two types of messages in Prometheus: internal messages posted within an agent to trigger other plans, and external messages exchanged between agents. Interaction protocols describe the patterns of messages between agents (i.e. external messages).

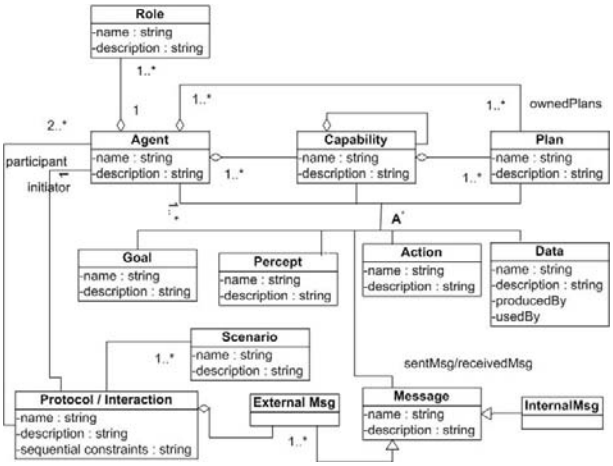

Figure 4. Prometheus meta-model (Part 2 - Design)

3.2. Model well-formedness constraints

The meta-model is not expressive enough to formally describe all constraints and relationships between Prometheus entities. For instance, it is difficult for the meta-model to express the constraint that the plan being triggered by a percept should belong to the agent that is responsible for handling the percept. UML models have limited expressiveness, which is also a common issue for object-oriented development using UML. A solution that has been widely used is to extend the UML meta-model with the Object Constraints Language (OCL) [29]. OCL is used to specify invariants, pre-conditions, post-conditions, and other kinds of constraints imposed on objects in UML models. As a declarative language, OCL's expressions do not specify any action that changes the state of the model. One of the strengths of OCL is that it is carefully designed to be both formal and simple.

We have also adopted OCL to specify additional constraints on the Prometheus meta-model. Each node in the

[6]The union of the two figures is the meta-model. We break it into two figures for readability. For association ends that do not have a multiplicity, it should be interpreted that their multiplicity is *zero or many* (0..*).

[7]In figure 4, associations marked with an 'A*' are between agent, capability, and plan and goal, percept, action, data, and message. We group and represent them as a single association for readability.

meta-model is annotated with a set of OCL constraints. For example, figure 5 shows constraints that are applied to the *ConcreteGoal* node (constraints 1-6) and constraints that are applied to the *Plan* node (constraints 7 and 8).

In the OCL notation "self" denotes the context node to which the constraints have been attached and an access pattern such as "self.achieverRole" indicates the result of following the association labelled "achieverRole", which is, in this case – rule (1), a collection of roles to which the concrete goal (context node) is allocated. OCL also denotes operations on collections such as "size" returning the number of elements in the collection, and "forAll" specifying that a certain condition must hold for all elements of a collection. For detailed information on OCL see [29].

context ConcreteGoal **inv**:

self.achieverRole→size ≥ 1 (1)

self.agent→size ≥ 1 (2)

self.plan→size ≥ 1 (3)

self.agent.role→includesAll(self.achieverRole) (4)

self.plan→forAll(pl:Plan |
self.agent→exists(a:Agent |
pl.agent→includes(a))) (5)

self.capability→size ≥ 1 *implies*
self.capability→forAll(c:Capability |
self.agent→exists(a:Agent |
a.capability→includes(c)))
and self.plan.capability→includesAll(
self.capability) (6)

context Plan **inv**:

self.agent→size ≥ 1 (7)

self.capability→size ≥ 1 *implies*
self.agent→includesAll(self.capability.agent) (8)

Figure 5. Constraints on concrete goals (1-6) and on plans (7-8)

Rule (1) means that each concrete goal should be allocated to at least one role. Similarly, rules (2) and (3) specify that each concrete goal should be assigned to at least one agent and one plan respectively. Rule (4) says if a role is assigned to achieve a goal then there exists at least one agent which achieves the goal and plays that role. Rule (5) means any plan that achieves a goal should belong to at least one of the agents that aim to accomplish that goal. Finally, rule (6) means if a goal is allocated to some capabilities then some agents achieving that goal should contain these capabilities and there should exist some plans in these capabilities which handle that goal. Constraints similar to these have been developed for each of the entities within the meta-model, and provide a mechanism for checking well-formedness of a full model.

4. A change propagation framework

In this section we first explain why Prometheus is a multi-view modelling methodology which can be based on the single model principle. We also present a classification of evolution actions. We then describe the major component of our framework which is the underlying mechanism to detect consistency violation and to resolve inconsistencies by propagating changes.

4.1. Multi-view development and evolution

As described in section 3.1, Prometheus promotes a multi-view development process by having views of different aspects and at different development phases. The different views and artefacts are based on a common and single model, as described by [24]. Consequently we rely on a Model View Controller model [9] to consistently update views when the model is changed. In particular, as registered views are updated, this causes changes to the underlying model. All views then get data from this model to update themselves to reflect the new model. Consequently, the main issue we address in this paper is to keep the Prometheus model well-formed as evolution actions are performed on the model. In doing so, changes should be propagated from one entity or relation to another. Our main goal is to assist the software engineer in making changes by automating the change propagation mechanism as much as possible.

The first step towards this goal is to understand the types of evolution and how they affect the Prometheus model. For this purpose, we have adapted an evolution model described in [18] which classifies evolution actions into four types: **addition** of entities (goal, role, agent, etc.) to the model; **removal** of entities from the model; **connection** of entities with relationships (i.e. adding relationships); and **disconnection**, i.e. removal of relationships between model elements.

The evolution actions modify model elements which result in a change from the current model to a new one. In the following section we describe the mechanism by which evolution actions are propagated from one model element to another model element.

4.2. A BDI change propagation engine

We assume that we start the change process with a Prometheus model which is well formed according to the meta-model, and the constraints as described in section 3. When a change (i.e. addition, removal, connection or disconnection) is made to the model, the constraints can be violated, resulting in inconsistencies in the model. The initial change is usually called the *primary change*. In practice,

software engineers have to make a lot of additional changes to reintroduce consistency into the model and preserve the well-formedness constraints. The process of making further changes is usually called *change propagation*. As this is a labour-intensive process, it is advantageous if we can automate this process as much as possible.

We have adopted the well-known Belief-Desire-Intention (BDI) architecture to represent and implement the underlying change propagation engine. A BDI agent has a collection of plans which are triggered by events or goals. Each plan defines what triggers it, under what conditions it is applicable, and a plan body: a sequence[8] of steps that are performed. The plan body can contain sub-goals which trigger further plans. Our BDI change propagation system has the main goal of maintaining the model consistency by resolving constraint violations as changes are made to the model. The belief component of the BDI system contains a representation of the Prometheus model. In addition, traceability information such as reasons for changes, and design decisions can be stored.

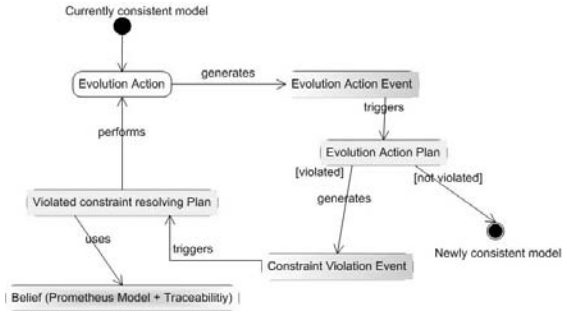

Figure 6. BDI change propagation model

Figure 6 depicts what happens when an evolution action takes place. Each modification made to the model will generate an event called *"Evolution Action Event"*. Currently, our framework deals with four basic types of event corresponding to the four evolution types described in section 4.1: "addition event", "removal event", "connection event" and "disconnection event". Each event carries some information such as the entity type, its ID, etc. For example, an "addition" event resulting from adding a goal would carry information about which entity type has been added (namely a goal), and which entity was added (namely the entity ID, e.g. "Print stock portfolio").

When an evolution action event is generated, a corresponding plan (called *"Evolution Action Plan"*) is triggered to handle that event. Basically, this plan first identifies the entity or relation being modified based on information carried with the event. After that, it checks all the constraints associated with this entity node one by one. As discussed in [12, 35] an incremental validation approach is more efficient than other similar rule-based approaches since we only revalidate the context of the last change, instead of the whole model. If any of the constraints is violated, the plan will generate a *"Constraint Violation Event"*. Below is an example of a plan which is triggered by the event of adding a goal. Note that there are dependencies between the constraints: for example, it does not make sense to check rule (4) (figure 5) – which is concerned with the agent and roles that are associated with the goal – before the goal is actually assigned to a role and to an agent (rules (1) and (2)). More generally, rules that require the existence of related entities should be checked (and enforced) before rules that check for conditions involving these entities.

Plan: Goal addition
Triggering event: Addition
Context condition: Addition type is goal
Plan body:

1. Check constraints associated with a goal from (1) to (6) (see figure 5 for details of these rules).

2. If a rule is violated, generate a "Goal violation" event (see the description of this type of event below) to address the violation, then go back to step (1).

3. If all constraints are valid, the new model is consistent. No further action should be taken.

Model modification may result in inconsistencies in the form of constraint violations. When such a violation occurs, we generate an event (called *"Constraint Violation Event"*) in our BDI system. For example, a "Goal violation" event is generated by the plan "Goal addition" when one of the constraints associated with the goal is invalid. This type of event may carry information that is needed to make changes so that the constraint becomes valid again. For example, a goal violation event would carry information regarding whether an association between the new goal and a role, an agent, or a plan is needed, depending on which rule violation resulted in the event.

When a *"Constraint Violation Event"* occurs, a plan type called *"Violated Constraint Resolving Plan"* handles it with the aim of "repairing" the violation. To "repair" a violation, this plan will perform further evolution actions. In most cases, there are several options to resolve a constraint violation. For example, to make rule (1) valid, we need to associate the newly added goal with a role. This can involve either creating a new role or using one of the existing roles. The decision regarding which action should be taken is based on constraint rules, traceability information, heuristics and, in some cases, human intervention.

[8] Some agent platforms generalise this to allow more control structures such as loops, conditionals etc.

In our BDI system, when an event is generated, there are possibly several plans that are able to handle it. This naturally corresponds to the fact that there are different options to repair a violation. For instance, in our example below there are several different plans (such as "Associating an existing role with goal" plan, "Associating a new role with goal" plan, "Associating an existing agent with goal" plan, etc.) which are triggered by a "Goal violation" event. However only one of them will be executed to handle a particular event. The determination as to whether a plan is applicable to handle a specific generated event is expressed in its *context condition*. For example, if the "Goal violation" event indicates that a role-goal association is needed then the applicable plan should have this as part of its context condition (e.g. "Associating an existing role with goal" plan or "Associating a new role with goal" plan). Additional criteria can be incorporated in the context condition based on design heuristics, allowing for a single plan to be chosen. If multiple plans are applicable, they are tried in order of pre-determined priority (although more sophisticated mechanisms can also be used).

In our framework, automated change propagation is achieved not only based on explicit traceability links (as specified in the meta-model and constraints) but based also on design heuristics. The ability of BDI systems to have multiple plans to handle an event in different situations makes it easy to incorporate design heuristics. Such heuristics are usually expressed in the context condition of a plan. For instance, one heuristic is that if a new goal has just been added, and all of its sibling goals are assigned to the same role, then assign the new goal to the same role as its siblings. This heuristic is expressed as one of the criteria in the context condition of the "Associating an existing role with goal" plan.

There is also a default plan which involves the intervention of the software engineer. This corresponds to the situations where traceability information and heuristics are insufficient to make a design decision. This also ensures that there is always at least one applicable plan.

Plan: Associating an existing role with goal
Triggering event: Goal violation
Context condition: An association between the goal G (related to the event) and a role is needed; *and* all its sibling goals are implemented by an existing role $R_{existing}$
Plan body:
1. A connection is made from goal G to role $R_{existing}$

Plan: Associating a new role with goal
Triggering event: Goal violation
Context condition: An association between the goal G (related to the event) and a role is needed; *and not* all its sibling goals are implemented by an existing role
Plan body:
1. A new role R_{new} is added
2. A connection is made from goal G to role R_{new}

Plan: Associating an existing agent with goal
Triggering event: Goal violation
Context condition: An association between the goal G (related to the event) and an agent is needed; *and* there exists an agent $A_{existing}$ which plays an existing role to which the goal is allocated
Plan body:
1. A connection is made from goal G to agent $A_{existing}$

Plan: Associating a new agent with goal
Triggering event: Goal violation
Context condition: An association between the goal G (related to the event) and an agent is needed; *and* there does *not* exist an agent which plays an existing role to which the goal is allocated
Plan body:
1. A new agent A_{new} is added
2. A connection is made from goal G to agent A_{new}

Plan: Associating plan with goal
Triggering event: Goal violation
Context condition: An association between the goal G (related to the event) and a plan is needed
Plan body:
1. Retrieve the agent $A_{existing}$ that implements this goal
2. A new plan P_{new} is added
3. A connection is made from plan P to agent $A_{existing}$
4. A connection is made from goal G to plan P_{new}

In this section, we have explained how BDI concepts such as plans, events, and context conditions fit naturally with the process of resolving constraint violations and propagating changes. In the next section, we describe an example to illustrate how our framework works in practice.

5. Example

In order to illustrate how our framework works, we are developing a case study, an excerpt of which is presented in this section. The full case study is a stock trading management simulation (STMS) which is specified and designed, along with a number of additional requirements. These additional requirements give examples of software evolution (perfective maintenance) that are used to assess our proposed framework.

The stock trading management simulation has three major goals: *Manage Stocks*, *Manage Funds* and *Serving Stock Customers* (see figure 1 for more details). The system's design has five roles: *Purchase Management* (achieving *Buying Stock* goal), *Sale Management* (achieving *Selling Stock* goal), *Funds Management* (achieving *Manage Funds* goal), *Stock Tracker* (achieving *Update Stock Portfolio* goal) and *Customer Interaction* (achieving *Serving Request for Adding Funds* goal). Figure 2 shows the system

overview diagram of the initial version of STMS which has three agent types: *Trader* agent (playing the roles *Purchase Management*, *Sale Management*, and *Stock Tracker*), *Funds Manager* agent (playing the role *Funds Management*) and *GUI Agent* (playing the role *Customer Interaction*).

Now we assume that the clients have asked to add a new requirement: the ability to print a list of current stocks held and their market value.

The following scenario may take place:

- The software engineer creates a new concrete goal *Print stock portfolio* in the Goal Diagram
- The software engineer then adds this goal as a sub-goal to the existing goal *Serving Stock Customers*

At this point, the software engineer may wish to ask our system what other artefacts he/she should alter to correctly propagate the new change. Below we outline the sequence of events that takes place, showing how our change propagation framework helps in propagating the changes.

The reader should refer to sections 3.2 and 4.2 for the description of constraint rules and plans which are used in this example.

1. As the goal *Print stock portfolio* has been added, the event "addition" is fired. This event triggers the "Goal addition" plan because it matches this plan's context condition.
2. The"Goal addition" plan is executed and checks constraints (rules 1-6 in section 3.2) against the newly added *Print stock portfolio* goal:
   (a) Rule (1) fails as this goal is not allocated to any role yet
      i. This will result in a "Goal violation" event being generated. This event also carries information indicating that an association between the goal and a role is needed. This type of event can potentially trigger several plans. However, in this case only the "Associating an existing role with goal" plan is applicable because its context condition holds (the sibling goal *Serving Request for Adding Funds* is allocated to the existing role *Customer Interaction*)
      ii. As this plan is executed, a connection is made from goal *Print stock portfolio* to role *Customer Interaction*
      iii. Rule (1) now holds
   (b) Rule (2) fails since there is no agent assigned to achieve this goal yet.
      i. This will result in a "Goal violation (agent associated)" event generated which can also potentially trigger several plans. The plans' context conditions again determine that only one plan (which in this case is "Associating existing agent with goal") is applicable
      ii. As this plan is executed, a connection is made from goal *Print stock portfolio* and agent *GUI Agent*
      iii. Rule (2) now holds
   (c) Rule (3) fails as there is no plan specified to achieve this goal
      i. This will result in a "Goal violation (plan associated)" event generated which triggers the plan "Associating plan with goal".
      ii. As this plan is executed, the following actions are performed: retrieving the agent implementing the new goal (which is the *GUI Agent*), adding a new plan called *Print-stock-portfolio-defaultplan*, making a connection from this plan to agent *GUI Agent*, and making a connection from goal *Print stock portfolio* to plan *Print-stock-portfolio-defaultplan*
      iii. Rule (3) now holds
   (d) Rules (4) and (5) hold due to changes made in previous steps.
   (e) Rule (6) holds because there is no capability associated with this goal.

As can be seen from the above example, the change propagation engine has helped by automatically adding several new entities and relations to the existing model.

6. Related Work

Since maintenance is widely regarded as a critical phase and evolution is an inevitable activity in software development, there has been a significant amount of research in this area. Some approaches include program comprehension and reverse engineering [21], restructuring [10, 19], re-engineering [20], impact analysis [1], and management processes [25, 32]. However, dealing with software evolution remains one of the most challenging issues in mainstream software engineering [2].

Software change is the basic operation of software maintenance and evolution. The two central aspects of software change are planning for changes and implementing changes [7]. Change impact analysis [1] is a planning activity by which the software engineer assesses 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, in which changes are propagated from one artefact to others in order to maintain consistency between artefacts as the software evolves.

Various techniques and methods have been proposed in the literature to address different activities of the consistency management process including: detecting overlaps between software models, detecting inconsistencies, identifying the source, the cause and the impact of inconsistencies, and resolving inconsistencies [31]. As UML has become the de facto notation for object-oriented software development, most research work in consistency management has focused on problems relating to consistency between UML diagrams and models [15, 16]. Several approaches strive to define fully formal semantics for UML by extending its current meta-model and applying well-formedness constraints to the model [3, 6]. Our framework also involves developing a meta-model and specifying well-formedness constraints. However, our framework also addresses the mechanism of resolving constraint violations when changes are made to the model.

Other approaches applying formal methods to UML, transform UML specifications to some mathematical formalism such as Petri-Nets [8], Description Logic [33], or graph grammars [13]. Such approaches have been advocated with the recent emergence of the Model Driven Architecture (MDA) paradigm [14]. The consistency checking capabilities of such approaches rely on the well-specified consistency checking mechanism of the underlying mathematical formalisms. However, it is not clear to what extent these approaches suffer from the traceability problem: to what extent can a reported inconsistency be traced back to the original model. Furthermore, the identification of transformations that preserve and enforce consistency still remains a critical issue at this stage [8]. Our framework does not involve any separate model translation step. We have a single, common model for all diagrams or views. Change actions are performed directly to the model and consistency checks are done within this model only.

7. Conclusions and Future work

In this paper we have presented a flexible and incremental change propagation approach to maintain consistency between software development artefacts described in the Prometheus methodology. As part of our framework, we have introduced a UML meta-model which represents all Prometheus entities and their relations. In addition, we have developed a set of OCL well-formedness and consistency constraints imposed on the meta-model.

The novel aspect of our framework is the underlying change propagation mechanism which uses agent technology. Specifically, the change propagation process in our framework is represented using the well-known Belief-Desire-Intention (BDI) agent architecture. Within this system, evolution and constraint violation are represented as BDI events. The strategies of propagating change to resolve model inconsistencies caused by constraint violation are represented as BDI plans. Heuristics and traceability information are also incorporated into the plan and belief components of the system to help the reasoning about which plan is more appropriate to resolve an inconsistency.

The focus of this paper is to set the foundations for a new approach to change propagation and to illustrate its capacity to deal with consistency management in evolving software systems. Although an agent-oriented methodology is chosen as the setting in which we applied our framework, we believe that our technique can be extended to other (non-agent) software engineering methodologies. We have started implementing the framework which we proposed here. As part of the future work, we will complete the Prometheus meta-model with additional well-formedness rules. We also plan to investigate how to integrate domain semantics into our framework and utilize them to give more support for automated change propagation. Currently the BDI plans are developed by hand based on the meta-model and the well-formedness constraints. We will investigate whether this process can be semi-automated, and how the resulting set of plans can be checked for completeness and consistency. In addition, more heuristics and traceability information such as design decisions, reasons for changes will be integrated with our framework. Furthermore, we plan to extend our framework to address the issue of change propagation and consistency management at the implementation level. More specifically, we will investigate how to (semi-) automatically propagate changes from the design to source code and vice versa.

Acknowledgments This work is being supported by the Australian Research Council under grant LP0453486, in collaboration with Agent Oriented Software.

References

[1] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[2] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 73–87. ACM Press, Limerick, Ireland, 2000.

[3] J.-P. Bodeveix, T. Millan, C. Percebois, C. L. Camus, P. Bazex, and L. Feraud. Extending OCL for verifying UML models consistency. In Kuzniarz et al. [15], pages 75–90.

[4] R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.

[5] M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[6] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the Unified Modeling Language. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 344–366. Springer, 1997.

[7] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, September-October 2005.

[8] G. Engels, J. M. Kuster, R. Heckel, and L. Groenewegen. Towards consistency-preserving model evolution. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 129–132, New York, NY, USA, 2002. ACM Press.

[9] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[10] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] S. Gwizdala, Y. Jiang, and V. Rajlich. JTracker - a tool for change propagation in Java. In *7th European Conference on Software Maintenance and Reengineering (CSMR 2003), 26-28 March 2003, Benevento, Italy, Proceedings*, pages 223–229. IEEE Computer Society, 2003.

[12] R. Haesen and M. Snoeck. Implementing consistency management techniques for conceptual modeling. In *Proceedings of the International Conference on the Unified Modeling Language 2004 (Workshop 7: Consistency Problems in UML-based Software Development)*, Lisbon, Portugal, October 10–15 2004.

[13] I. Ivkovic and K. Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[14] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, MA, 2003.

[15] L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, editors. *UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development. Workshop Materials*, 2002:06, Ronneby, 2002. Blekinge Institute of Technology.

[16] L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, editors. *UML 2003, Modeling Languages and Applications. Workshop on Consistency Problems in UML-based Software Development II. Workshop Materials*, 2003:06, Ronneby, 2003. Blekinge Institute of Technology.

[17] B. P. Lientz and B. Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, Nov. 1981.

[18] T. Mens and T. D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering*, 7(1):39–59, 2000.

[19] T. Mens and T. Tourw. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[20] H. W. Miller. *Reengineering legacy software systems*. Digital Press, Newton, MA, 1998.

[21] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE - Future of SE Track*, pages 47–60, 2000.

[22] H. Ossher, W. Harrison, and P. Tarr. Software engineering tools and environments: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, pages 261–277. ACM Press, Limerick, Ireland, 2000.

[23] L. Padgham and M. Winikoff. *Developing intelligent agent systems : a practical guide*. John Wiley & Sons, Chichester, 2004.

[24] R. F. Paige and J. S. Ostroff. The single model principle. *Journal of Object Technology*, 1(5):63–81, 2002.

[25] M. Polo, M. Piattini, and F. Ruiz, editors. *Advances in software maintenance management: Technologies and solutions*. Idea Group Publishing, U.S.A, 2003.

[26] V. Rajlich. A model for change propagation based on graph rewriting. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 84–91, Washington, DC, USA, 1997. IEEE Computer Society.

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

[28] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 439–449, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

[29] Rational Software Corporation. Object constraint language specification. In OMG Unified Modeling Language Specification, Version 1.4, September 2001.

[30] J. L. Sourrouille and G. Caplat. Checking UML model consistency. In Kuzniarz et al. [15], pages 1–15.

[31] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In K. S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, pages 24–29. World Scientific, 2001.

[32] A. A. Takang and P. A. Grubb. *Software maintenance : concepts and practice*. International Thomson Computer Press, London ; Boston, 2nd edition, 2003.

[33] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logics to maintain consistency between UML models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 326–340. Springer-Verlag, 2003.

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

[35] R. Wagner, H. Giese, and U. Nickel. A plug-in for flexible and incremental consistency management. In Kuzniarz et al. [16], pages 78–89.

[36] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons (Chichester, England), 2002. ISBN 0 47149691X, http://www.csc.liv.ac.uk/∼mjw/pubs/imas/.