An Agent-based Framework for Distributed Collaborative Model Evolution

Hoa Khanh Dam and Aditya Ghose
School of Computer Science and Software Engineering
University of Wollongong
New South Wales 2522, Australia
{hoa,aditya}@uow.edu.au

ABSTRACT

In recent years, an increasingly large number of software systems have been developed at different geographical regions. As a result, the maintenance and evolution of those systems have shifted from being conducted at a single site to being geographically distributed at multiple locations around the world. In these collaborative development environments, it is a critical challenge to maintain consistency within a software model during its evolution since changes are rapidly and concurrently made to the model without the awareness of team members at various locations. Most of existing software modelling applications however primarily support single-user settings whereas some other recent approaches which rely on version control tools fail to provide effective, real-time support in a collaborative modelling setting requiring frequent interactions and short feedback cycles. In this paper, we present a framework that supports designers in evolving software models in a collaborative modelling setting. This framework is built upon the well-known Belief Desire Intention agent architecture to utilise its robustness and flexibility in maintaining consistency within a design model and resolving conflicts in real time when changes are concurrently made to it by different designers.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Theory

Keywords

Software evolution, collaborative software engineering, consistency maintenance, change propagation

1. INTRODUCTION

Software engineering projects are inherently cooperative in which they require the participation of many software engineers in producing large software systems. In recent years, the phenomenon of globalization and outsourcing has resulted in a large number of software systems that have been developed at different geographical regions across the globe. In order to continue remaining useful after delivery, such software systems undergo constant changes during their evolution to meet ever-changing user requirements and environment changes. As a result, the maintenance and evolution of software have shifted from being conducted at a single site to being geographically distributed at multiple sites around the world. In this distributed collaborative development environment, engineers may work on the same software artefacts independently and concurrently. Geographic separation may however lead to inadequate and ineffective communication [16, 17], which may in turn create redundant and/or conflicting work during software maintenance and evolution.

Previous approaches to maintain and evolve software in multi-site environments relied on traditional version management systems (e.g. CVS and Subversion). Such systems are able to provide automatic support for merging modifications and detecting conflicting changes made by different engineers to a software artefact (mostly source code). However, those merging and conflict detecting operations are not started until the engineers "check-in" their changes. At this point in time, automatic merging may not be successful, which results in conflicts that may be difficult and costly to resolve [1, 19]. Therefore, in order to detect conflicts earlier and avoid unnecessary effort expended, there has been an emerging need for collaborative development environments that detect conflicts in real-time, rather than waiting for a check-in action.

In the past few years, a number of tools and techniques have been proposed to support real-time distributed collaboration in software engineering. They aim to help software engineers be aware of development activities that are currently performed by their team members. Most of the work (e.g. [7, 15, 29]) mainly focus on detecting conflicting concurrent modifications on source code and notifying the engineers of those conflicts in real time. A majority of them (e.g. [7]) can detect direct conflicts (e.g. concurrent changes made to the same component in the code) whereas a small number of approaches (e.g. [15, 29]) are able to detect indirect conflicts (e.g. changes made to dependent components).

Much of the efforts have focused on collaborative coding whereas there has been very limited work on providing support for maintenance and evolution in a collaborative modelling setting. The recent emergence of model driven development has better recognised the importance of models in the software development process and thus it has become increasingly important to provide support for dealing with changes at the level of design models. Current modelling environments provide some support for fixing inconsistencies in a design model. For instance, IBM Rational Rose automatically detects inconsistencies between class diagrams and sequence diagrams, and suggests some potential resolutions for fixing such inconsistencies. However, those modelling tools do not capture completely possible changes and are only useful for trivial tasks. More importantly, those tools and some recent proposals (e.g. [9, 12, 14]) which aim to improve their capabilities in detecting and resolving inconsistencies primarily support single-user settings.

It is very important to provide real-time support for design and modelling that are distributed at multiple sites since these activities demand regular interactions among team members and short feedback cycles [6]. A number of recent work (e.g. [1, 19, 30]) have been proposed to address this issue to a certain extent. The work in [1] proposes a collaborative software modelling framework based on a client-server, event-based architecture where local model changes are considered as events which are sent to a server for processing. Their framework can be extensible to different types of design and architectural models and can detect conflicts and notify the designer. They however do not provide a mechanism in their framework to resolve conflicts. The work in [30] tackles the issue of inconsistency resolution using a distributed constraint maintenance approach. These existing work however fail to address the central issue of model evolution in a collaborative setting: a designer should proactively propagate changes within a model to implement a change request but should also respond (e.g. be aware of), in a timely fashion, to changes concurrently made to the model by other designers.

In our view, collaborative distributed modelling environments for software evolution tend to be *dynamic*, *unpredictable* and *unreliable*. These environments are dynamic in that they change rapidly, i.e. a designer cannot assume that the model will remain static (i.e. unchanged) while they evolve it since designers at other sites may also modify it at the same time. In addition, these environments are unpredictable in that it is not possible for a designer to predict future states of the model since it may be being modified (at the remote sites) without the designer's knowledge. Furthermore, these environments are unreliable in that the actions that a design can perform (to the model) may fail due to concurrent modifications. For example, the action of changing a method's name by a designer may fail because this method has just been deleted by another designer at a different site without the knowledge of the first designer.

The emerging intelligent agent technology provides solutions suitable to those challenging environments. A software agent [32] is an *autonomous* computational entity being *situated* in an environment and being able to operate independently in terms of making its own decisions about which activities to pursue. An agent have goals which it is able to pursue over time (i.e. *pro-activeness*) and at the same time it can respond in a timely fashion to changes that occur in the environment it operates (i.e. *reactivity*). An agent is also able to interact with other agents in order to accomplish its goals (i.e. *sociability*). Such useful notions of intelligent agents have made them a popular choice for developing software that needs to operate in dynamic, unpredictable and unreliable environments. In fact, the practical utility of agents has been demonstrated in a wide range of domains such as air traffic control, space exploration, weather alerting, business process management, holonic manufacturing, e-commerce and information management [22, 26].

One of the most well-established and popular agent model is the Belief-Desire-Intention (BDI) architecture [4] which we will explain in detail in the next section. Previous work [10] has used agent technology to deal with change propagation[1] in model evolution using a BDI representation to model ways of repairing inconsistencies in design models. More specifically, repairing an inconsistency is considered as a goal/event while repair options are represented as BDI plans achieving/handling such a goal/event. The use of BDI-style, event-triggered plans, naturally reflects the cascading nature of change propagation, where a change can cause other changes to be made. In addition, the BDI structure of having multiple plans that respond to a given event elegantly captures the fact that there are usually many ways of fixing a given inconsistency. Such *abstract repair plans* are a way to practically capture the large number of concrete ways of fixing inconsistencies [23]. In addition, new (or alternative) ways to resolve an inconsistency can be added via additional plans without changing the existing structure. Previous work ([8, 9]) has also demonstrated that such an agent-based framework for model evolution is applicable to a range of design models such as UML models (e.g. [9]) and enterprise architecture models (e.g. [8]).

These previous work however do not use the full capabilities of a BDI agent since they only deal with evolution in a static single-user environment in that changes are made by a single designer. In this paper, we propose to extend the existing framework to deal with model evolution in a collaborative distributed modelling environment. In particular, we exploit the reactivity and the event-driven nature of planning in the BDI, including a configurable mix of sensing and replanning and the ability to interrupt plan execution when higher priority events occur. We will demonstrate how these properties of the BDI are particularly useful in a collaborative modelling environment.

The structure of this paper is as follows. In the next section, we briefly describe agent technology with a focus on the well-known and widely-used Belief-Desire-Intention agent architecture. We then present a motivating example using a simple UML design model in section 3. Section 4 serves to provide an overview of our proposed framework. We discuss how model changes and inconsistency resolution can be represented in our framework in section 5. A typical execution cycle of the underlying collaborative engine of our collaborative modelling framework is presented in section 6. Finally we briefly discuss related work (section 7), and conclude and outline some future work.

[1]Given a set of primary changes that have been made to a software model, change propagation is the process of determining what additional, secondary, changes are needed to maintain consistency within the model. Note that secondary changes also may introduce inconsistencies that then need to be resolved through additional secondary changes.

2. AGENT TECHNOLOGY

Since the 1980s, the field of agent technology has attracted a substantial amount of interest from both academia and industry. The literature of agent research has been very active with many agent theories, architectures, and languages proposed in the past decades. In the industry, there have also been a growing number of agent systems in various domains such as business process management [5], holonic manufacturing [21], e-commerce, and information management [22]. The number of agent-based applications continues to increase since there are compelling reasons to use intelligent agent technology. In fact, a recent study by a large company that used agent technology to develop complex software applications has shown an average productivity gain of over 350% [2].

One of the most well-established and widely-used agent models is the *Belief-Desire-Intention* (BDI) model. The BDI family of agent theories, languages and systems are inspired by the philosophical work of Bratman [4] about how humans do resource bounded practical reasoning, i.e. figure out what to do and how to act under limited resource capacity. An agent's beliefs represent information about the environment, the agent itself, or other agents, from the agent's perspective. Desires represent the objectives to be accomplished in terms of states of the world that the agent wants to reach. Intentions represent the currently chosen courses of action to pursue a certain desire that the agent has committed.

Those initial ideas of the BDI model have been modified to suit a practical computational environment. In fact, while BDI theories focus on desires and goals, BDI implementations (e.g. [28]) deal with *events*. Events are significant occurrences that the agent should respond to in some way. Most BDI agent implementation platforms model the change associated with the adoption of new (sub)goals as events (e.g. achieving the goal of being at an airport). Furthermore, in BDI implementations intentions are viewed as the *plans* which are currently being executed by the agent. BDI agents have a collection of pre-defined plan recipes (or types), usually referred to as a *plan library*. Each plan consists of: (a) an invocation condition which defines the event that triggers this plan (i.e. the event that the plan is *relevant* for); (b) a context condition (usually referring to the agent's beliefs) which defines the situation in which the plan is *applicable*, i.e. it is sensible to use the plan in a particular situation; and (c) a plan body containing a sequence of primitive actions and subgoals (which can trigger further plans) that are performed for plan execution to be successful.

Let us consider an example (adapted from [25]) of an agent having the following plans *A–D*.

Plan A
Triggering event: *achieve goal* be-at-airport
Context condition: raining
Plan body:

1. Call to order a taxi
2. Taxi to airport

Plan B
Triggering event: *achieve goal* be-at-airport
Context condition: not raining
Plan body:

1. *Subgoal:* Catch train

Plan C
Triggering event: *achieve goal* be-at-airport
Context condition: not raining
Plan body:

1. *Subgoal:* Catch bus

Plan D
Triggering event: Catch X
Context condition: true (always applicable)
Plan body:

1. Walk to station for X
2. Check X running on time
3. Board X to airport.

We now use this example to describe a typical reasoning cycle that implements the decision-making of an agent following an implementation of the BDI architecture (e.g. AgentSpeak [28]). The cycle can be viewed as consisting of the following steps. First, an event is received from the environment (e.g. achieving the goal of *be-at-airport*), or is generated internally by belief changes or plan execution (e.g. subgoals *Catch Bus* or *Catch Train* in plans B and C respectively). The agent responds to this event by selecting from its plan library a set of plans that are relevant (i.e. match the invocation condition) for handling the event (by looking at the plans' definition). In this example, Plans *A, B,* and *C* are relevant to handle the event of achieving the goal *be-at-airport* (but not Plan *D*).

Next, the agent then determines the subset of the relevant plans that are applicable to handle the particular event. This involves checking whether the plan's context condition holds in the current situation. Assume that it is not raining, the agent then determines that Plans *B* and *C* are applicable. The agent selects[2] Plan *B* and starts executing it, which leads the posting of an event of achieving *subgoal Catch Train.* Only Plan *D* is both relevant and applicable to handle this event. As a result, the agent executes this plan, walks to the train station, find out if the train is running on time, and then boards the train to airport.

A plan can be successfully executed, in which case the (sub)goal is regarded to have been accomplished (e.g. the agent being at airport). Execution of a plan, however, can fail in some situations, e.g. a subgoal may have no applicable plans or an action can fail. For example, once the agent arrives at the train station, it finds out that trains are significantly delayed. Step 2 of Plan *D* fails and if the agent is pursuing to achieve a goal (e.g. *be-at-airport*), a mechanism that handles failure is used. Typically, the agent tries an alternative applicable plan for responding to the triggering event of the failed plan. However, in this example since there is no other applicable plan, the subgoal *Catch Train* has failed and consequently the parent plan, i.e. Plan *B*, has failed. Therefore, the agent now has to consider alternative ways of achieving *be-at-airport.* Assume that while the agent walks to the train station, the weather has changed and it is now raining. The agent determines that only Plan *A* is

[2]The selection of applicable plans may be based on certain pre-determined priority (e.g. the first applicable plan is selected) although more sophisticated mechanisms can also be used, depending on the implementation.

applicable, and consequently executes it and takes a taxi to airport.

With the above characteristics, BDI agents offer two important qualities: *robustness* and *flexibility*. BDI agents are robust since they are able to pursue persistent goals over time (i.e. pro-activeness). In other words, agents will keep on trying to achieve a goal (e.g. *be-at-airport*) despite previously failed attempts. In order to be able to recover from such failures, agents have multiple ways of dealing with a given goal and such alternatives can be used in case any of them fail. This gives agents flexibility in terms of exercising choice over their actions. Flexibility and robustness are considered as useful qualities that a software system should possess, especially if it operates in complex, dynamic, open and failure-prone environments. With the emergence of distributed information systems, these types of environments are becoming more and more common. Therefore, it is increasingly required to have technologies that are able to cope well in such challenging environments.

3. A MOTIVATING EXAMPLE

The example we use is a design of a real, albeit simplified, video on demand (VOD) system [13] using Unified Modelling Language (UML) notation. Although we use UML models to demonstrate our framework, as will be seen, our approach is generic and applicable to different types of models. The VOD system allows a user to select a movie to play. The user is also able to play, pause and resume the movie. The class diagram (see Figure 1) represents the structure of the initial VOD system. The system consists of three classes: "Display" for visualising movie streams and receiving user inputs, "Streamer" for downloading and decoding movies, and "Server" for providing data. There are four operations in the "Display" class: "select()" for choosing a movie, "stream()" for playing and retrieving the movie streams, "draw()" for rendering the received movie stream, and "stop()" for halting the movie being played. There two operations in the "Streamer" class: "stream()" for streaming the movie data received from the server, and "wait()" for halting the streaming process. Finally, there are two operations in the "Server" class: "connect()", which is called by clients (e.g. the Streamer), and "handleRequest()" which deals with requests from clients.

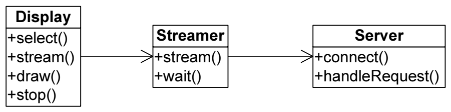

Figure 1: Class diagram for the VOD system (redrawn based on [13])

The sequence diagram (see Figure 2) describes a typical scenario of interactions between the user, a Display object ("disp") and a Streamer object ("st"). The user chooses a movie that they want to see (message 1). They then start playing the selected movie — in the sequence diagram the user sends a message "stream" to the Display (message 2). The Display then obtains the movie stream from the Streamer (message 3) and renders the movie (message 4). When the user wants to stop viewing the movie (message 5), the Display notifies the Streamer to stop streaming (message 6).

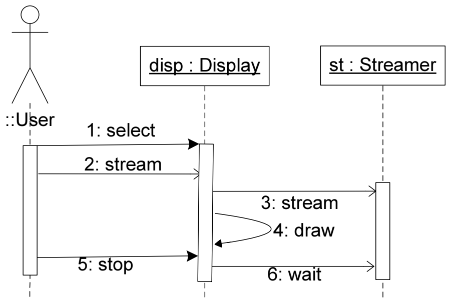

Figure 2: A sequence diagram for instances of classes Display and Streamer (redrawn based on [13])

In a collaborative modelling environment, there can be multiple designers who collaboratively evolve the existing VOD design. They can be a software architect making changes to the class diagram or a junior designer who is working on modifying the sequence diagram. As a result, changes can be concurrently made to the VOD model by a designer without knowledge of the others. For instance, in the current VOD design, the Display and Streamer classes have two different methods with the same name "stream". In order to avoid the confusing dual use of the term "stream", the software architect makes the following change: renaming the method "stream()" of class Display to "play()". This change makes the sequence diagram no longer be consistent with the class diagram: the message "2:stream" to the "disp:Display" object in the sequence diagram does not match with any method in the updated Display class. Therefore, the software architecture needs to make further changes to the sequence diagram and subsequently other diagrams (e.g. statechart diagrams) in the VOD design. Determining and making these secondary changes is termed change propagation [27] and is widely considered to be complicated, labour-intensive and expensive, especially in complex software systems.

It can be observed that the above example of change propagation and inconsistency resolution can take place in the usual single-user setting. These issues become even more challenging in a collaborative setting since change propagation can take place concurrently with normal changes made by different designers. For instance, in the above example while the software architect is propagating the change to the sequence diagram by renaming the message "2:stream", a designer at a different site might remove the message from the sequence diagram. Inconsistencies may arise not only during change propagation but also when single changes are made to the model. Let us further assume that the software architect now decides to remove class Server from the class diagram (for example, to move these functionalities to the Streamer class). At the same time, a junior designer at a different site adds an object of the Server class to the sequence diagram since he/she decides that it should also involve in the interaction. These two concurrent modifications cause an inconsistency in the VOD design, between the class diagram and the sequence diagram.

This simple example indicates some significant challenges in collaborative modelling settings. Those issues would manifest themselves in complex software models, which are concurrently evolved by many distributed designers. In the next section, we will present our approach to these issues in a form of an agent-based framework.

4. AN OVERVIEW OF THE FRAMEWORK

Those properties of the BDI model discussed in section 2 offer a suitable solution to deal with model evolution in collaborative distributed environments. Firstly, BDI agents operate in an event-triggered manner, where events trigger plans, which in turn can create new events resulting in further plans being triggered. This hierarchical relationship between plans is very suitable for representing the cascading nature of change propagation and inconsistency repairing (where fixing an inconsistency by performing an action can cause further inconsistencies requiring further action) in model evolution. In addition, 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. The BDI architecture also offers flexibility as new or alternative ways of resolving an inconsistency can readily be added via additional plans, without changing the previous structure.

Furthermore, the setting of collaborative modelling represents a dynamic environment where the model may evolve rapidly due to changes constantly and concurrently made by multiple designers at different sites. In this environment, a designer cannot predict the future states of the model since changes can be made to the model (from the remote sites) without the designer's awareness. As a result, changes made to an element of the model by a designer may fail since this element may no longer exist in the model (because it has been deleted by a different designer and the first designer has not been aware of this change). Such a dynamic, unpredictable, and unreliable environment is well suited for a BDI framework.

We exploit those useful properties of the BDI model to develop an agent-based collaborative software modelling framework that supports system design in geographically distributed maintenance and evolution settings. Figure 3 describes a conceptual view of this framework. The framework employs a client-server architecture in which the server keeps a centralized version of a design model, and the designer works on a local version of the model at the client side using a modelling tool (e.g. UML modelling tool such as IBM Rational Software Architect). The client side consists of two layers. The top layer (i.e. `Modelling Tool`) provides conventional modelling capabilities in a form of traditional CASE tools such as ArgoUML and Rational Software Architect (for UML modelling). The bottom layer (`Collaborative Controller`) acts as a plugin to the `Modelling Tool` and is responsible for communication with the server. Each change (e.g. removing a class or changing a method's name) made locally by the designer to the model is immediately captured by the `Collaborative Controller` as a model edit event, which is then sent to the server.

At the server side, such model edit events are received and handled by the `Collaborative Engine`. It uses the `Constraint Validation` component to detect if an edit event causes inconsistencies in the model. Such inconsistencies are represented as BDI goals/events while options for resolving them are presented as BDI plans (i.e. *repair plans*). It then synchronizes concurrent edits made by distributed clients and may notify designers of conflicting modelling decisions. The `Plan Generator` component of our framework relies on previous work [10] to automatically generate repair plans for consistency conditions expressed as constraints in Object Constraint Language and UML. The automatic generation of repair plans, as opposed to being developed manually by the user, guarantees completeness(i.e. all possible inconsistency resolutions are generated) and correctness (i.e. that the resolutions actually fix a corresponding inconsistency). In addition, it eliminates the significant effort which is required to manually hard-code such repair plans when the number of consistency constraints increases or changes.

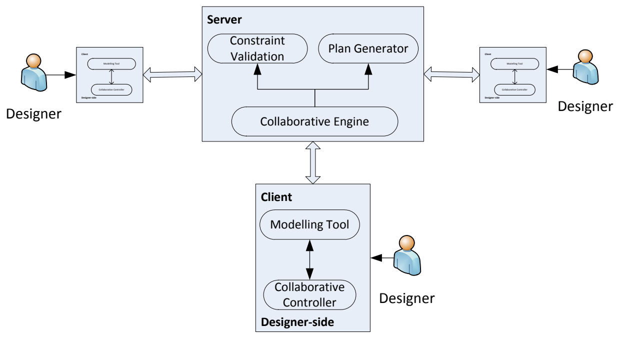

Figure 3: An overview of the collaborative modelling framework

The most important component of our framework is the `Collaborative Engine` which is represented and implemented based on notions and ideas inspired by BDI agents. The BDI architecture provides a flexible, robust approach to deal with conflicts and inconsistencies in a real time distributed collaborative environment. Details of the `Collaborative Engine` are presented in the next sections.

5. EVENTS AND PLANS

In our framework, evolution actions are classified into five types: *addition* of entities to the model; *removal* of entities from the model; *connection* of entities with relationships (i.e. adding relationships); *disconnection* i.e. removal of relationships between model elements; and *modification* of an entity's attributes. These action are performed locally by a designer at the client side using a modelling tool. In our framework, the occurrence of each evolution action is captured by a `Collaborative Controller` which is attached with the modelling tool at the client side. The `Collaborative Controller` then generates a corresponding event and sends it to the server. Such an event is referred to as a *remote model edit event* which carries some information related to the entity being modified such as the entity type and its ID, and the ID of the client that sends the event. For instance, in the VOD example in section 3, when the software architect renames a method of the "Display" class, an event is generated and sent to the server.

Those events are handled by simple *Evolution Plans.* Basically, this type of plan first identifies the entity being modified based on information carried with the event and performing the actual modification. It then delegates to the `Constraint Validation` to perform checking of all the constraints associated with this entity. If any of the constraints is violated, the plan will generate a *Constraint Violation Event*. For example, the following plan is to renaming a method. Note that step 1 of this plan (i.e. change the value of attribute "name" of method M to N) may fail since when the plan is executed, method M may be deleted by another designer.

Plan: Changing a method name
Triggering event: Renaming method M to N
Context condition: true
Plan body:

1. Change the value of attribute "name" of method M to N.
2. Check consistency constraints associated with method M.
3. If a constraint is violated, generate a "Constraint Violation Event" event

A model usually has multiple views, for example a UML model having a set of diagrams (e.g. class diagrams, sequence diagrams, etc.) which provide multiple perspectives of the system under development. In a collaborative environment, a designer may focus on making changes to some of the views (e.g. a class diagram) while their remote team members may, at the same time, making changes to the other views (e.g. a sequence diagram). It is very important that each view of a model must be both syntactically and semantically consistent [31]. Syntactic consistency ensures that a model's view conforms to the model's abstract syntax, i.e. a metamodel, which guarantees that the overall model is well-formed. Semantic consistency requires different views of a model to be semantically compatible (i.e. coherence). For instance, the name of a message in a sequence diagram must match a method in its receiver's class in a class diagram (*Constraint 1*), or an object in the sequence diagram must be an instance of a class in the class diagram (*Constraint 2*). Such consistency requirements upon a model are often expressed using its metamodel and a set of constraints (e.g. using the Object Constraint Language [24]) that specify conditions that a well-formed and consistent model should satisfy.

On the one hand, model modification in a single-user environment may result in inconsistencies in the form of constraint violations. For example, changing the name of a method in the class diagram without changing the corresponding message in the sequence diagram, as in the VOD example, will violate *constraint 1* discussed earlier. On the other hand, concurrent model modifications in a collaborative environment may also create inconsistencies (alternatively referred to as conflicts). For example, a designer removes a class from the class diagram while another designer (at a different site) simultaneously adds an object of that class to a sequence diagram which violates *constraint 2*.

The `Constraint Validation` component is responsible for checking for those constraint violations. When either type of violations occurs, we generate a *Constraint Violation Event*. The plans (referred to as *Inconsistency Resolution Plans*) to handle such an event basically represent different ways of resolving the violation. For instance, if a consistency constraint c is of the form[3] $\exists\, x \in e \bullet c_1(x)$, its violation can be resolved in three different ways, each of which corresponds to a plan: selecting an existing item $y \in e$ and making $c_1(y)$ true (Plan 1); adding an element z to e and ensuring that $c_1(z)$ is true; in this case z may be an existing element in the model (Plan 2) or a newly created element (Plan 3). Note that making $c_1(y)$ true is an (*internal*) event (posted within Plan 1) which in turn triggers further plans.

Plan 1
Triggering event: c is not true (i.e. violated)
Context condition: $y \in e$
Plan body:

1. Subgoal: making $c_1(y)$ true.

Plan 2
Triggering event: c is not true (i.e. violated)
Context condition: z is an existing element and $z \notin e$
Plan body:

1. Add z to e
2. Subgoal: making $c_1(z)$ true.

Plan 3
Triggering event: c is not true (i.e. violated)
Context condition: true
Plan body:

1. Create a new element n.
2. Add n to e.
3. Subgoal: making $c_1(n)$ true.

After consistency constraints are written (either by tool developers or tool administrators), the plans to resolve their violation can be automatically generated by analysing the constraints' definition and systematically identifying the ways of fixing them. This generation is done once by the `Plan Generator` component, when the constraints are specified. It only needs to be re-done should the consistency constraints change. In addition, the tool developers or tool administrators are allowed to use their domain knowledge and expertise to modify generated repair plans or remove plans that should not be executed. Previous work [10] has defined and implemented a systematic process that translates Object Constraint Language (OCL) constraints on UML models into a set of BDI plans.

6. EXECUTION CYCLE

We now describe an execution cycle of the `Collaborative Engine`, which is adapted from the reasoning cycle of a typical BDI agents [3]. In each execution cycle (refer to figure 4), only one pending event will be dealt with. There can be various pending events because various model edits concurrently made at different client sides and sent to the server, and the `Collaborative Engine` has not gone through enough execution cycles to handle them all. As a result, it is necessary to select an event to be handled in a specific execution cycle. This is done by the event selection component S_E. This component can be customized to take into account any priorities that are specific to a collaborative environment. For

[3]In fact, *Constraint 1* above can be expressed in this form: there exists a method in the set of methods of a message's receiver's class, such that the name of the method is the same as that of the message.

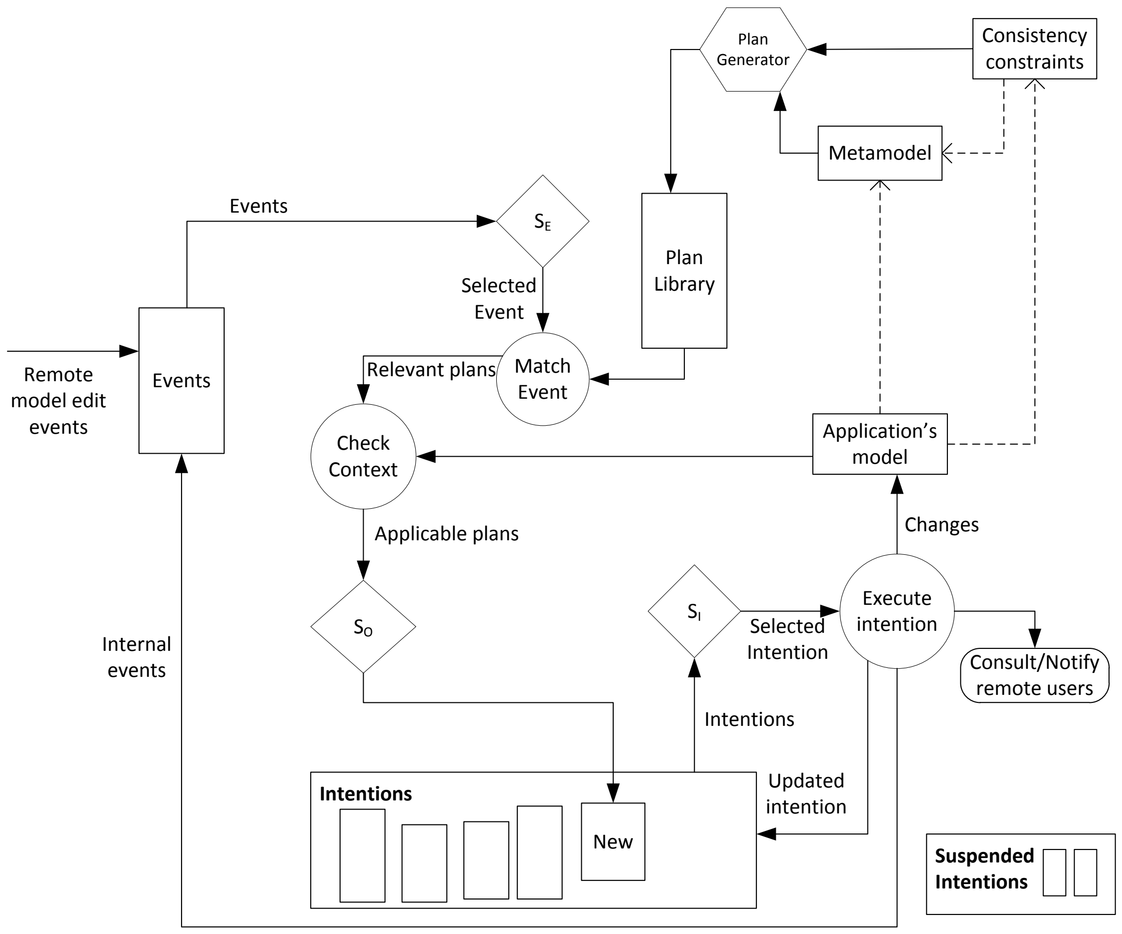

Figure 4: An execution cycle of the `Collaborative Engine` (adapted from the reasoning cycle of a typical BDI agents in [3])

instance, changes by a software architect may be regarded more important than those made by a junior designer, and thus events associated with the former have a higher priority than those related to the latter. Another example is that the addition/deletion of an entity could be regarded as more important than changing its attributes. A default implementation would assume all events are equally importance and the Events component can be regarded as a queue of events. In this situation, the event which is currently at the top of the queue is chosen in the next execution cycle and newly received events are added to the back of the queue.

Once an event is selected, the next step is to find a plan to handle that event. This is done by the Match Event component which retrieves all plans from the Plan Library that have a triggering event that can be matched with the selected event. At the end of this step, a set of relevant plans for the selected event are determined. For instance, plans 1–3 above are selected for handling the event of constraint c being violated.

As mentioned earlier, a plan's context condition determines whether a plan can be used at a particular moment in time, given the current status of the design model. Therefore, in the next step the `Collaborative Engine` needs to select, from the set of relevant plans, all those which are currently applicable (i.e. their context condition holds). In practice, there can be multiple possible plans for handling a given event which represent equally valid *options*. A domain-specific *option selection* (i.e. S_O in figure 4) machinery can be plugged in to make choice amongst multiple competing applicable plans. One of the criteria for option selection that is of interest in our context is *cost* (reflecting the number of changes made to a model) – we might prefer to choose plans that offer the least-cost approach to implementing/propagating a given change request (as proposed in previous work [10]).

Another criterion of interest is model quality - given multiple options for implementing a given change request, it would be useful to pick the option that would lead to best quality model, under some agreed upon measure of model quality. There is an existing body of work that has explored conceptual model quality (see for instance, [18]). The additive combination of a given set of these metrics represents an *objective function* (in the sense of optimization). Such an objective function can be leveraged to make optimal choices during option selection. An extension to the BDI agent programming language, CASO [11], permits users to specify objectives dynamically (in the agent's event queue). These are maintained in an *objective store* (which in our instance would be the collection of model quality metrics of interest) and are brought to bear in the option selection mechanism.

Option selection involves look-ahead search over a tree obtained by enumerating the alternative ways of realizing each option, and recursively enumerating alternative ways of realizing each of the alternatives, and so on. These trees (often referred to as *goal-plan trees*) may in general be infinitely deep. The look-ahead search is therefore parameterized by the number of steps we might want to look ahead. In more time-constrained settings, this parameter is set to a smaller number, while in less time-constrained settings, we might be willing to explore deeper into the goal-plan tree. Given a goal-plan tree obtained by cut-off at the depth specified by the parameter, we explore each distinct path from the root to each (pseudo-)leaf node. Each such path represents one way in which change might be propagated, and thus provide a (possibly partial) specification of the model that would be obtained at the end of the change propagation process. We assess each of the these alternative and possibly partially-specified models against the current set of model quality metrics, and pick an option that offers the likelihood of leading to the highest quality model.

Once an applicable plan is selected, it is pushed into the set of intentions waiting to be executed in the next execution cycle. There can be multiple inconsistencies and change requests at one time, and thus the `Collaborative Engine` may have multiple intentions, each of which corresponds to either making a change or resolving an inconsistency and represents different focus of attention. The engine can potentially execute any of those intentions in the next step of the execution cycle. However, at most one of the intentions is executed in one reasoning cycle. Therefore, the engine needs to choose one particular intention among those currently ready for execution. This selection (i.e. S_I in figure 4) can be customized in such a way that preferences can be given to an intention of handling some urgent or higher priority events (e.g. fixing the violation of a critical consistency constraint). This mechanism allows the engine to keep a balance between propagating changes to resolve inconsistencies and performing changes made by designers to implement a change request. If all the intentions get attention equally, a simple round robin selection can be used in which each of the intentions is selected in turn.

An executing intention might be suspended (and put into the set of suspended intentions) due to waiting for feedback from a remote designer at a client side. For example, during the semi-automated process of propagating changes in the model, the engine may consult a designer for input (e.g. name of a newly created method). Before another execution cycle begins, the `Collaborative Engine` checks whether any such feedback are now available, and if so the relevant intentions are updated and pushed back in the set of intentions so that their execution can be resumed in the next execution cycle. In some other cases, an executing plan may be suspended since a higher-priority event has just occurred and the `Collaborative Engine` needs to deal with it urgently. For instance, while the engine is propagating changes to fix a given inconsistency, a higher priority event (e.g. a change request made by a senior software architect) occurs. The engine needs to suspend the current plan and executes another plan to deal with this higher priority event.

The execution of a selected intention in each cycle involves performing steps in the corresponding plan, including the three main things: make changes to the model, generate further internal events (i.e. subgoals), and consult and/or notify designers. The execution of a plan may fail due to two main reasons: lack of relevant or applicable plans for an event, and an action failure. The former can be dealt with by having inconsistency resolution plans automatically generated in such a way that they cover all possible inconsistency resolutions (as in previous work [10] for UML models and OCL constraints). The second cause of plan failure (i.e. action failure) indicates a typical situation in which a change involves a model element which no longer exists (due to concurrent modifications in collaborative settings). For instance, while executing the first step of Plan 3 (i.e. "Create a new element n") in section 5, the engine needs to suspend the execution to consult a designer for the name of this new element. While waiting for the designer's feedback, it executes another plan (i.e. intention) which leads to the deletion of set e. As a result, when the engine continues executing Plan 3, step 2 (i.e. "Add n to e") fails because set e no longer exists in the model.

As discussed in section 2, the most common approach to deal with a plan failure is to select an alternative applicable plan (e.g. Plans 1 and 2 in our example). In order to do so, the engine recalculates the set of applicable plans (ignoring those that have already been attempted). This is to make sure that the applicable plans are determined with respect to the current state of the model. In the case when there are no remaining applicable plans, an event is considered to have failed and the relevant designers is notified.

7. RELATED WORK

General version management tools (e.g. CVS and Subversion) have been dominantly used in projects that involve teams of software engineers. Those tools can compare different versions to detect and/or resolve conflicts, and if necessary merge these versions together. They are however more effective for comparing and merging text documents (e.g. source code) than for diagrams and models. Recently, increasing efforts (e.g. [20]) focus on providing support for differencing and merging design models (e.g. UML models). Unlike our framework, these approaches can only support non-real-time collaboration since conflicts only detected until the engineers "check-in" their changes.

Much of the efforts ([7, 15, 29]) in supporting real-time distributed collaboration primarily focus on detecting conflicting concurrent modifications on source code. Recently, due to the emergence of the model-driven paradigm, providing support for collaborative modelling environments has started attracting attentions. The work in [19] proposes to convert Rational Software Architect (RSA), a UML modelling tool, into a real-time collaborative modelling application using the Transparent Adaptation approach [33]. Their approach basically adapts the single-user modelling interface (e.g. the API provided by RSA) to the data and operational models of the underlying collaboration supporting technique. There are however still many challenging issues in their approach such as maintaining semantic consistency of a UML model. In contrast, our framework explicitly addresses the maintenance of both syntactic and semantic consistencies.

More recently, the *CoDesign* framework proposed in [1] specifically supports system modelling in geographically distributed work environments. Similarly to our work, this framework has also been developed using a client-server, event-based architecture where local model changes are considered as events and sent to a server for processing. Their framework allows for off-the-shelf conflict detection engines to be integrated, and relies on these engines to detect inconsistencies caused by concurrent modifications. The *CoDesign* framework can only notify the relevant users when an inconsistency arises, and leaves the issue of conflict resolution to the users. In contrast, the work in [30] proposes an approach to maintain consistency of a model that is evolved across all distributed sites. Their approach keeps the effects of all concurrent modelling change actions (even those that cause constraint violations), and generate different consistent versions of the model, each of which is the outcome of a number of change actions in a certain order. These versions of the model are prompted to the users for selection. These existing work however do not explicitly take into account the priority of changes as in our work. In addition, unlike our approach they do not address the critical issues of dealing with failures in both making changes to a model and propagating changes to maintain consistency within the model.

8. CONCLUSIONS AND FUTURE WORK

Due to an increasing number of software systems that have been developed across geographically distributed teams, collaborative modelling has become an important and challenging technical issue. Since such systems are becoming larger and more complex, software modelling demands collective and collaborative contributions from many designers that are dispersed by large geographic distances. On the one hand, geographical separation can significantly reduce communication among team members, and consequently may lead to redundant and conflicting work. On the other hand, the nature of design and modelling requires frequent interactions among team members and short feedback cycles. Most of existing software modelling applications however primarily support single-user settings, and do not offer much support for collaborative modelling – which allows multiple software designers to concurrently work on the same software models. Therefore, the current practice of software modelling relies on the use of traditional version management tools to automatically merge modifications and detect conflicting changes. This approach of non-real-time collaboration may however be error-prone, ineffective and costly since it may lead to conflicts and redundant work that are discovered very late in the process.

In this paper, we have proposed a generic, real-time collaborative modelling framework to maintain consistency within a design model as they evolve due to changes made by multiple designers. The framework is developed based on a client-server, event-based architecture. The novel aspect of our framework is the underlying mechanism of dealing with concurrent modifications and resolving inconsistencies which uses agent technology. Specifically, the collaborative engine in our framework is represented and implemented using the well-known Belief-Desire-Intention (BDI) agent architecture. Within our framework, evolution and constraint violation are represented as BDI events, and the options of making changes and propagating changes to resolve model inconsistencies are represented as BDI plans. Such options are computed on the fly so that they are relevant to the current state of the design model and the engine can react to the latest changes made to the model. In addition, the framework also has a mechanism that allows event selection based on a certain priority or other criteria. It also allows for failures caused by conflicting changes to be handled in an elegant way.

As a step towards validating our framework, we plan to apply it to a specific modelling language (e.g. UML) and environment (e.g. Rational Software Architect). We have implemented the Plan Generator component of the framework which can takes an OCL constraints as input and produces a set of BDI plans that can repair the constraint violations. A key area for future work is to implement the Collaborative Engine. We are planning to implement the Collaborative Controller as a plugin for Rational Software Architect. Apart from implementation, there are several other key issues that we plan to investigate. More specifically, we will further develop a notion of “change failure” and distinguish this notion from “change rejection” (driven by trade-off analysis). In addition, we plan to explore additional criteria for selecting events to be handled in each execution cycle and the use of model quality as a guide for option selection. Finally, an evaluation of the framework in a real collaborative environment is also an important agenda of our future work in this project.

9. REFERENCES

[1] J. y. Bang, D. Popescu, G. Edwards, N. Medvidovic, N. Kulkarni, G. M. Rama, and S. Padmanabhuni. Codesign: a highly extensible collaborative software modeling framework. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 243–246, New York, NY, USA, 2010. ACM.

[2] S. S. Benfield, J. Hendrickson, and D. Galanti. Making a strong business case for multiagent technology. In *AAMAS '06: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 10–15, New York, NY, USA, 2006. ACM.

[3] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, 2007. ISBN 0470029005.

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

[5] B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. BDI-Agents for agile goal-oriented business processes. In Padgham, Parkes, Müller, and Parsons, editors, *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 37–44, Estoril, Portugal, May 2008.

[6] M. Cataldo, C. Shelton, Y. Choi, Y.-Y. Huang, V. Ramesh, D. Saini, and L.-Y. Wang. Camel: A tool for collaborative distributed software design. In *Proceedings of the 2009 Fourth IEEE International Conference on Global Software Engineering*, ICGSE '09, pages 83–92, Washington, DC, USA, 2009. IEEE Computer Society.

[7] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up Eclipse with collaborative tools. In *Proceedings of the 2003 OOPSLA workshop on Eclipse technology eXchange*, eclipse '03, pages 45–49, New York, NY, USA, 2003. ACM.

[8] H. K. Dam, L.-S. Le, and A. Ghose. Supporting change propagation in the evolution of enterprise architectures. In *Proceedings of the 2010 14th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '10, pages 24–33, Washington, DC, USA, 2010. IEEE Computer Society.

[9] H. K. Dam and M. Winikoff. Supporting change propagation in UML models. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[10] H. K. Dam and M. Winikoff. An agent-oriented approach to change propagation in software maintenance. *Journal of Autonomous Agents and Multi-Agent Systems*, ?(?):?, To appear in print; published online 1/1/2011.

[11] A. Dasgupta and A. K. Ghose. Implementing reactive BDI agents with users' given constraints and objectives. *International Journal of Agent-Oriented Software Engineering*, 4:141–154, April 2010.

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

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

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

[15] R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 178–187, Washington, DC, USA, 2008. IEEE Computer Society.

[16] J. D. Herbsleb. Global software engineering: The future of socio-technical coordination. In *2007 Future of Software Engineering*, FOSE '07, pages 188–198, Washington, DC, USA, 2007. IEEE Computer Society.

[17] P. J. Hinds and D. E. Bailey. Out of sight, out of sync: Understanding conflict in distributed teams. *Organization Science*, 14:615–632, November 2003.

[18] C. Lange and M. Chaudron. Managing model quality in uml-based software development. In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice, 2005*, pages 7–16, 0-0 2005.

[19] S. Liu, Y. Zheng, H. Shen, S. Xia, and C. Sun. Real-time collaborative software modeling using UML with Rational Software Architect. In *Proceedings of International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 1–9, Los Alamitos, CA, USA, November 2006. IEEE Computer Society.

[20] A. Mehra, J. Grundy, and J. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 204–213, New York, NY, USA, 2005. ACM.

[21] L. Monostori, J. Váncza, and S. Kumara. Agent based systems for manufacturing. *CIRP Annals-Manufacturing Technology*, 55(2):697–720, 2006.

[22] S. Munroe, T. Miller, R. A. Belecheanu, M. Pěchouček, P. McBurney, and M. Luck. Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents. *Knowledge Engineering Review*, 21(4):345–392, 2006.

[23] 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.

[24] Object Management Group. Object Constraint Language (OCL) 2.0 Specification. `http://www.omg.org/docs/ptc/03-10-14.pdf`, 2006.

[25] L. Padgham and M. Winikoff. *Developing intelligent agent systems: A practical guide*. John Wiley & Sons, Chichester, 2004. ISBN 0-470-86120-7.

[26] M. Pěchouček and V. Mařík. Industrial deployment of multi-agent technologies: review and selected case studies. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 17:397–431, 2008.

[27] 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.

[28] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In V. R. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pages 312–319. The MIT Press, 1995.

[29] A. Sarma, G. Bortis, and A. van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 94–103, New York, NY, USA, 2007. ACM.

[30] H. Shen. Maintaining constraints of UML models in distributed collaborative environments. *Journal of System Architecture*, 55:396–408, July 2009.

[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] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons (Chichester, England), 2002. ISBN 0 47149691X.

[33] S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen. Leveraging single-user applications for multi-user collaboration: the coword approach. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 162–171, New York, NY, USA, 2004. ACM.