CoDesign – A Highly Extensible Collaborative Software Modeling Framework

Jae young Bang, Daniel Popescu, George Edwards, and Nenad Medvidovic
University of Southern California
Los Angeles, CA 90089-0781, USA
{jaeyounb, dpopescu, gedwards, neno}@usc.edu

Naveen Kulkarni, Girish M. Rama, and Srinivas Padmanabhuni
Infosys Technologies Limited
Bangalore 560 100, India
{Naveen_Kulkarni, Girish_Rama, srinivas_p}@infosys.com

ABSTRACT

Large, multinational software development organizations face a number of issues in supporting software design and modeling by geographically distributed architects. To address these issues, we present CoDesign, an extensible, collaborative, event-based software modeling framework developed in a distributed, collaborative setting by our two organizations. CoDesign's core capabilities include real-time model synchronization between geographically distributed architects, as well as detection and resolution of a range of modeling conflicts via several off-the-shelf conflict detection engines.

1. INTRODUCTION

In recent years, many technology companies have transferred significant portions of their software development activities to emerging economies such as India and China [15]. At the same time, many stakeholders, such as customers and requirements engineers, remain in developed countries. As a result, companies have created global software development teams in which engineers are separated by large geographic distances. While the economic advantages of distributed software development are real, communication challenges must be overcome in order to fully realize these advantages. Convincing evidence shows that geographic separation can drastically reduce communication among coworkers [6, 7, 13]. Irregular and ineffective communication typically prevents shared understanding of problems and solutions, and incurs redundant work during software development.

In the past, global software teams relied on traditional IDEs that were developed for co-located development teams along with software configuration management (SCM) systems. SCM tools, such as CVS and Subversion, allow engineers to work on software artifacts independently and with reduced planning and coordination because they automatically merge modifications and detect conflicting changes. However, SCM systems do not detect conflicts until the engineers "check in" changes, at which point unnecessary or useless effort may have already been expended. Furthermore, conflicts may be more difficult and time-consuming to resolve at this stage.

To detect conflicts earlier and avoid costly conflict resolution, *collaborative IDEs* have become a popular mechanism to provide engineers with awareness of the concurrent development activities of their coworkers [1, 3, 5, 14]. Most collaborative IDEs detect conflicting concurrent modifications to the same artifact – such as the same file – and provide real-time notifications of these obvious, direct conflicts. A more limited number of collaborative IDEs also detect indirect conflicts that require more rigorous analysis [5, 14]. For example, if one engineer changes the implementation of a component while another engineer concurrently modifies the component's interface, an indirect conflict could result.

Current collaborative IDEs primarily focus on distributed *programming*. Other critical development tasks, particularly architecture design and modeling, are not readily supported, even though these activities require frequent interactions among team members and short feedback cycles [2]. As a result, geographically-distributed software architects still create and edit their models in traditional modeling environments and check-in their changes to a repository using an SCM system. Of course, this results in all the same problems noted above that collaborative IDEs help to solve. Like collaborative IDEs for programmers, software architects need collaborative modeling environments that detect conflicts in real-time, rather than waiting for a check-in action.

We present *CoDesign*, a collaborative software modeling environment that supports system design in geographically distributed work settings. A conceptual view of CoDesign is depicted in Figure 1. At its core, CoDesign relies on *CoWare*, a lightweight middleware platform that (1) provides the integration infrastructure, (2) synchronizes concurrent edits made in distributed CoDesign instances, and (3) notifies architects of conflicting modeling decisions.

CoDesign's main contribution is an *extensible conflict detection framework for collaborative modeling*. CoDesign utilizes an event-based architecture [16] in which highly-decoupled components—different instances of CoDesign—exchange messages via implicit invocation, allowing flexible system composition and adaptation. CoDesign couples this event-based

ICSE '10, May 2-8 2010, Cape Town, South Africa

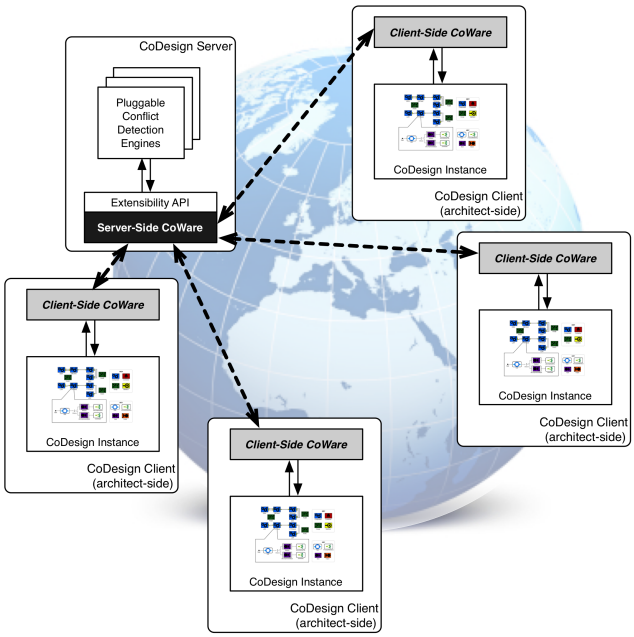

Figure 1: High-Level Architecture of CoDesign.

architecture with an API that provides explicit extension points for plugging in conflict detection engines. This allows different CoDesign clients (e.g., a UML modeling tool or a finite state machine modeling tool) to be paired with the most appropriate consistency checkers. It also allows multiple consistency checkers to be used in concert, in order to handle different types of modeling inconsistencies (see Section 2). To demonstrate this capability of CoDesign, we have integrated three off-the-shelf conflict detection engines: Drools [8], Jess [9], and a metamodel checker [4].

In Section 2, we classify the types of conflicts that can occur during collaborative distributed architectural modeling. Section 3 then explains the architecture and implementation of CoDesign, with a particular focus on CoDesign's extensibility mechanism. The paper concludes with a summary of lessons learned and a discussion of planned future work.

2. DESIGN-TIME CONFLICTS

When designing distributed collaborative systems, it is necessary to understand the potential issues and conflicts caused by modeling events that are generated simultaneously in remote locations. Two categories of issues may occur in collaborative software modeling over the network: parallel modification and modeling conflicts.

Parallel modification represents a situation when multiple architects modify the same modeling object or multiple objects that are very close in a model, e.g., an object and its parent. Parallel modification need not manifest itself as a conflict. However, detecting it and notifying the architects may be crucial as a warning to exercise caution and avoid future conflicts. For example, even though two simultaneous modifications to an object and its parent may be consistent with one another, each of the architects making one of those modifications may be unaware of the other architect's actions and may be more likely to make subsequent changes that will, in fact, result in a conflict.

A *conflict*, as we define it, is an issue that is engendered by synchronization latency, that is, when one architect makes a design decision that cannot be reconciled with another design decision that was made previously but that has not yet been synchronized with the architect's local model data (i.e., with the local *CoDesign Instance* in Figure 1). Because of their nature, decentralized systems cannot always be perfectly synchronized, inducing the architects to make such potentially erroneous decisions.

We categorize modeling-level conflicts into three classes based on the rules that the system modeling events violate: (1) synchronization, (2) syntactic, and (3) semantic conflicts. Each category is briefly elaborated next.

Synchronization conflicts can be resolved with little or no human intervention. For example, if an architect removes a class from a system model and another architect decides to add an attribute to the same class before the removal event arrives, those two events would result in inconsistent states in the two CoDesign instances. This type of conflict would not happen if the two architects were in the same workspace, since the removal event would be instantly "recorded" and the class would no longer be there for the second architect to modify. Although synchronization conflicts are the simplest of the three conflict types, they are the most common and thus must be detected and resolved efficiently and scalably.

Syntactic conflicts violate a modeling tool's or language's meta-model constraints. Suppose, e.g., that an architect connects an instance *a1* of class *A* with an instance *b1* of class *B* and, before the connection addition event arrives, another architect connects *a1* and a new instance *b2* of class *B*. If the cardinality constraint of the meta-model allows class *A* to have an association to only one instance of class *B*, this becomes a conflict that would likely not have occurred if the two architects were co-located. When the modeling tool such as CoDesign receives the second event, the tool's meta-model constraint checker will detect an error. Alternatively, the tool could experience an unexpected crash if it does not support syntactic conflict detection. Either way, unlike the synchronization conflicts, the resolution of syntactic conflicts will typically require human intervention.

Unlike the synchronization and syntactic conflicts, **semantic conflicts** reflect violations in the *intended*, implicit rules by which a system's model should abide. For example, a collaboratively completed design in a given architecture description language (ADL) [16] may have no irreconcilable events on the same model elements (i.e., no synchronization conflicts) and no violations of the ADL's grammar (i.e., no syntactic conflicts). However, the model may be modified in a way that violates, e.g., the rules of the underlying design style. As a simple example, let us assume that the intended style is client-server. An architect may model component *C1* to make direct requests of component *C2* in the system; the implication of this is that *C1* is a client and *C2* is a server. Another architect may, however, model component *C2* to make direct requests of component *C1*; the implication of this interaction dependency is that *C1* is, in fact, a server and *C2* a client. Hence, the same component is erroneously modeled both as a client and a server. Again, the language in which the model is specified (e.g., UML) may not consider this a conflict. In order to be properly checked for, this semantic rule would have to be specified externally (e.g., in the Object Constraint Language, or OCL). As with syntactic conflicts, semantic conflicts such as the one illustrated

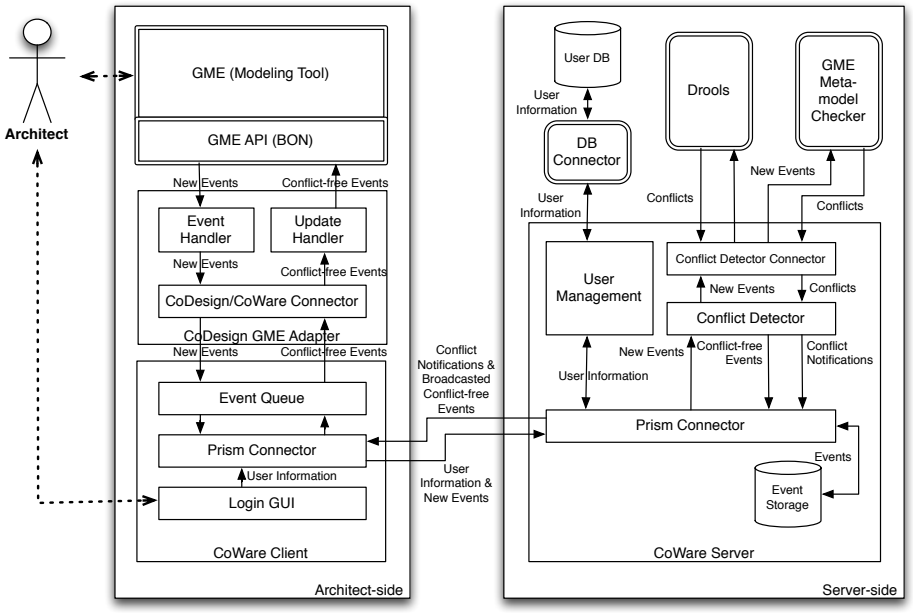

Figure 2: CoDesign's Architecture. The double-lined polygons represent off-the-shelf software.

above can only be highlighted by a tool such as CoDesign, but cannot be resolved without human intervention.

3. CODESIGN

In this section, we describe CoDesign's architecture and mechanism for enabling the integration of off-the-shelf (OTS) conflict detection engines. As mentioned previously, CoDesign aims to support integration of a variety of modeling languages and environments. Since modeling languages differ in the way their syntax and semantics are defined, CoDesign allows distributed architecture teams to use their own specific conflict detection engines rather than attempting to provide a general-purpose conflict detection engine. Section 3.1 provides an example use case scenario of a collaborative conflict that helps to describe CoDesign's architecture. Section 3.2 describes CoDesign's conflict detection extension points and the integration and customization of two OTS components for conflict detection; other such components (e.g., Jess) have been integrated in the same manner.

3.1 CoDesign's Architecture

CoDesign uses a modeling tool-specific adapter (comprising a CoDesign Instance in Figure 1) to capture design decisions, in the form of model updates, from architecture modeling tools. Each model update is subsequently encapsulated within a CoWare design event and is transferred through the CoWare infrastructure. A CoWare Client is installed at each architect location to connect a CoDesign adapter to a CoWare Server, which is running a Conflict Detector module. The design events are forwarded from the CoDesign adapter, through the CoWare Client and CoWare Server, to the Conflict Detector. The Conflict Detector evaluates each event to determine whether it conflicts with any previous event(s) by requesting all plugged-in conflict detection engines to analyze the event. The CoWare Server broadcasts each event back to all CoWare Clients only if all plugged-in conflict detection modules affirm that the design event does not cause any conflict. However, if a conflict exists, CoWare (1) tries to resolve it by itself and (2) alerts the architects involved in the conflict using a notification message.

Figure 2 depicts our implementation. We use three off-the-shelf software components: (1) GME [4], a software modeling tool from Vanderbilt University, (2) Drools [8], a rule-based business logic integration platform developed by the JBoss community, and (3) Prism-MW [12], an event-based middleware platform created at USC.

As noted above, the use of GME with CoDesign requires a GME-CoDesign Adapter. The adapter captures design decisions made by architects using GME via GME's native API, packages them within Prism-MW events, and transfers them to the CoWare Client. The CoWare Client receives the events and utilizes Prism-MW's connector facilities to send them to the Conflict Detector in the CoWare Server. In this particular CoDesign configuration, we use Drools to detect synchronization conflicts, GME's native metamodel checker to detect syntactic conflicts and GME's OCL constraint checker to detect semantic conflicts.

As a simple scenario of conflict detection, suppose an architect *A1* deletes a design element *e1* from her model in GME. Once the Prism-MW event generated by this design decision arrives at the Conflict Detector, each plugged-in conflict detection engine will analyze it. The GME metamodel checker and Drools respond that the event does not cause a conflict. Both engines may also store the event temporarily or permanently, depending on the circumstances. The CoWare Server then broadcasts the event back to all CoWare Clients except the original sender, *A1*.

Now suppose architect *A2* changes the geometric location of *e1* before the remote deletion event is applied to her local model data. The event is sent to the Conflict Detector via the same route, and this time the Drools engine detects that the model update applies to an object that no longer exists.

CoWare does not broadcast the location event, and since the intentions of the two architects differ, CoWare notifies the architects to ensure that they are aware of the situation.

3.2 Extending CoDesign

We illustrate CoDesign's support for integrating and customizing conflict detection engines using two example engines: Drools and GME's metamodel checker.

Detecting synchronization conflicts using Drools: Drools is a production rule system that can be used to detect complex events [11]. Drools evaluates whether a production rule triggers based on the facts it receives and computes. A production rule follows a simple pattern: **when** <condition> **then** <action>. A complex event(e.g., a synchronization conflict) is a pattern-based abstraction of other events and can also be evaluated using production rule systems [10]. Whenever CoDesign's Drools customization receives a CoDesign event to evaluate, it adds the event to its working memory and evaluates all synchronization conflict rules. Figure 3 shows a simplified example of a Drools rule that detects when one CoDesign client changes a model element that had already been deleted by another CoDesign client. CoDesign is able to detect modifications to the same model elements because all distributed instances of a model element have a single objectID in every CoDesign client.

Detecting syntactic and semantic conflicts using GME: In the CoDesign configuration described thus far, GME is used as the system modeling environment. Hence, this CoDesign configuration's syntactic and semantic conflict detection engines need to understand the syntax and semantic constraints of GME models. To ensure that syntactic and semantic conflicts are detected early, we reused and integrated the relevant components of GME. GME's metamodel checker contains the logic that manages the data model and checks whether executing a received CoDesign event keeps the data model consistent with its meta-model.

Integrating conflict engines into CoDesign: To integrate an OTS conflict engine, we need to implement an adapter connector to translate CoWare events into invocations of the conflict engine's API and to tie the results returned by the conflict engine back to the conflicting events (see Figure 2). The Conflict Detector component checks each event that the CoWare Server receives from the CoDesign Clients. Since the Conflict Detector is unaware of the syntax and the semantic constraints of the edited models, it does not itself check whether an event causes a conflict but forwards each event to the Conflict Detector Connector. The Conflict Detector Connector distributes the event to each integrated conflict detection engine, which in turn evaluate the received event in parallel. The results are returned to the connector and evaluated by the Conflict Detector, which notifies the CoDesign Clients in the case of conflicts.

```
1 rule "Object was edited after
2       it had already been removed"
3   when
4     $e1 : Event(name == "remove")
5     $e2 : Event($e1.objectID == objectID,
6                 timestamp > $e1.timestamp)
7   then
8     out.send(new ConflictEvent($e1, $e2);
9 end
```

Figure 3: Synchronization Conflict Detection Rule

4. CONCLUSION AND FUTURE WORK

Since we are a geographically distributed team, CoDesign has presented a unique opportunity for "reflective" use in its own design and implementation. In turn, this has allowed us to test first-hand its scalability, efficiency, and extensibility. Our work is on-going. We are investigating the root causes of design-time conflicts, the relationships between conflict types and modeling activities, as well as conflicts caused by complex event sequences and/or large numbers of parallel events. CoDesign opens up another research area — *conflict resolution*. We have focused on conflict *identification* in our work to date and have assumed that human designers will use other means at their disposal to resolve the identified conflicts. However, we hypothesize that CoDesign will lend itself naturally to investigating automated conflict resolution solutions, and that its extensible architecture will be able to incorporate easily such solutions.

5. ACKNOWLEDGMENTS

This work has been supported by Infosys Technologies Ltd. It has also been supported by the NSF award 0820170.

6. REFERENCES

[1] J. T. Biehl et al. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proc. CHI 2007*.

[2] M. Cataldo et al. Camel: A tool for collaborative distributed software design. In *Proc. ICGSE 2009*.

[3] L.-T. Cheng et al. Jazzing up eclipse with collaborative tools. In *Proc. OOPSLA Workshop on Eclipse Technology eXchange*. ACM, 2003.

[4] GME. `http://isis.vanderbilt.edu/projects/gme/`.

[5] R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. In *Proc. ASE 2008*.

[6] J. Herbsleb. Global Software Engineering: The Future of Socio-technical Coordination. In *ICSE: 2007 Future of Software Engineering*. ACM, 2007.

[7] P. J. Hinds and D. E. Bailey. Out of sight, out of sync: Understanding conflict in distributed teams. *Organization Science*, 14(6):615–632, 2003.

[8] jBoss Drools. `http://jboss.org/drools/`.

[9] Jess. `http://www.jessrules.com/`.

[10] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proc. Middleware*, 2005.

[11] D. Luckham. *The power of events: an introduction to complex event processing in distributed enterprise systems*. Springer, 2002.

[12] S. Malek et al. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE TSE*, pages 256–272, 2005.

[13] G. M. Olson and J. S. Olson. Distance matters. *Human-Computer Interaction*, 15(2):139–178, 2000.

[14] A. Sarma et al. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proc. ASE 2007*. ACM.

[15] B. Sengupta et al. A research agenda for distributed software development. In *Proc. ICSE 2006*.

[16] R. N. Taylor et al. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2009.