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 its 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
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 removes an instance $a_1$ of a class $A$ with an instance $b_1$ of class $B$ and, before the connection addition event arrives, another architect connects $a_1$ and a new instance $b_2$ 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 conflicts 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 $C_1$ to make direct requests of component $C_2$ in the system; the implication of this is that $C_1$ is a client and $C_2$ is a server. Another architect may, however, model component $C_2$ to make direct requests of component $C_1$; the implication of this interaction dependency is that $C_1$ is, in fact, a server and $C_2$ 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...
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 the conflict by itself and (2) alerts those architects involved in the conflict by generating 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 the University of Southern California.

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 another 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
Conflict Detector Connector distributes the event to each forwards each event to the Conflict Detector Connector. The does not itself check whether an event causes a conflict but syntax and the semantic constraints of the edited models, it sign Clients. Since the Conflict Detector is unaware of the each event that the CoWare Server receives from the CoDe-
(tions of the conflict engine's API and to tie the results re-
turned by the conflict engine back to the conflicting events
Figure 2). The Conflict Detector component checks 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.

4. CONCLUSION AND FUTURE WORK

CoDesign is a software modeling infrastructure, developed collaboratively at our two institutions, that supports real-time model synchronization between geographically distributed software architects, as well as detection and resolution of syntactic and semantic modeling conflicts. While CoDesign is a mature prototype, the work on CoDesign is on-going. We are investigating the root causes of design-time conflicts, the relationships between conflict types and modeling activities, and conflicts caused by complex event sequences, such as those caused by many parallel events. 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.

5. REFERENCES


Figure 3: Synchronization Conflict Detection Rule

<table>
<thead>
<tr>
<th>rule</th>
<th>&quot;Object was edited after it had already been removed&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>when</td>
<td>$e1 : Event(name == &quot;remove&quot;) $e2 : Event($e1.objectID == objectID, $e1.timestamp &gt; $e2.timestamp)</td>
</tr>
<tr>
<td>then</td>
<td>out.send(new ConflictEvent($e1, $e2));</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
The CoDesign research demonstration will begin after a short presentation highlighting the main points of the accompanying research paper. The short presentation will explain:

- **The motivation of the work** – why a collaborative software modeling infrastructure is needed. We will give examples of software development off-shoring, explain the development of collaborative software design systems, and introduce the overview of the project.

- **The theoretical foundations of our approach** – how we categorized and defined the potential issues created by geographically distributed software modeling. We have grouped the issues, which we refer as conflicts, into three categories: (1) synchronization conflicts, which are directly conflicting design decisions that occur due to the latency between distributed architects, (2) syntactic conflicts, which are design decisions that cannot be simultaneously accommodated by the metamodel of the modeling tool or language being used, and (3) semantic conflicts, which are design decisions that break the intended rules of the system being modeled. We define the terms using examples since the definitions of these terms vary based on context.

- **The CoDesign architecture** – how we detect and resolve conflicts. We discuss how extensibility is achieved and the roles of components in the high-level CoDesign architecture. We show the implementation of the architecture and the list of research off-the-shelf software used in the system.

We will then begin our demonstration. In this demonstration, we will use (1) GME, a software modeling tool created by Vanderbilt University, (2) Drools, a business logic integration platform developed by the JBoss Community, and (3) Prism-MW, a lightweight middleware created by the software architecture research group at USC.

**Start up:**

1. An instance of the CoWare Server is initialized and waits for connections from CoWare Clients.
2. The plug-in conflict detection engines within the CoWare Server are instantiated automatically as CoWare Server initializes.

![Figure 1: CoWare Server Initialization](image)

3. Two or more architects (depending on the projection equipment in the demonstration room) denoted $A_1,...,A_n$ connect CoWare Clients to the CoWare Server. Each architect logs in with a unique user ID and password.
After logging in, all architects receive identical copies of a design model from the CoWare Server. A diagram showing a portion of the design model is displayed on each architect’s screen.

Each of the architects makes a modification to a design element in the model, such as changing its attributes, relations, or location in the model hierarchy. All other architects see the changes in near real-time.

Examples of Conflict Detection:

6. Synchronization Conflict – Architect $A_1$ removes a modeling object, and Architect $A_2$ changes the geometric location of it before the remote removal event has arrived and been applied to the second architect’s model data. We will (1) introduce an artificial delay in the event distribution and (2) color the relevant object(s) so that the audience may appreciate what is occurring. (See Figure 4)

The action of $A_2$ creates a conflict through modification of an object that no longer exists. CoWare detects the conflict and notifies the architects. The notification includes information about the conflict, including (1) the identities of the architects involved, (2) the modeling elements involved, and (3) the actions that created the conflict. (See Figure 5)

7. Syntactic Conflict – We show the metamodel for a client/server system in which each server can only support $n$ clients. In the modeling environment, $A_1$ connects $n$ clients to a server and $A_2$ connects an additional client to the server. CoWare detects the conflict and notifies the architects.

8. Semantic Conflict – We show a set of OCL constraints for a component-based system. In this system, a component that makes requests of other components is a client, and a component that receives requests is a server. A component may not be both a client and a server. $A_1$ creates a design element indicating that component $C_1$ makes a request of component $C_2$. $A_2$ creates a design element indicating that component $C_2$ makes a request of component $C_1$. CoWare detects the conflict and notifies the architects.

We conclude the demonstration by mentioning the potential of CoWare/CoDesign as an infrastructure for conducting additional research on collaborative modeling environments.
CoDesign Screenshots

Jae young Bang, Daniel Popescu, George Edwards, Nenad Medvidovic

Computer Science Department
University of Southern California
{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

This screenshots document depicts an example conflict that two architects are collaboratively modeling at remote locations.

As the first step, CoWare Server is turned on. Two architects then connect from CoWare Client to CoWare Server. The architects are requested to log in to CoWare.

Figure 1: CoWare Server Start-up

Figure 2: A1 initializes CoWare Client

Figure 3: A2 Initializes CoWare Client
The architects turn on GME, and check out the current model data to join the remote modeling session.

Architect 1: A1

Architect 2: A2

Figure 6: CoWare Server Status after the Logging-in

Figure 7: A1's Screen after Checking out the Model

Figure 8: A2's Screen after Checking out the Model
The architect A1 moves a design element, and A2 sees the change on her/his screen after synchronization.

**Figure 9:** A1’s Screen after Moving an Element

**Figure 10:** A2’s Screen after Moving an Element

The architect A1 removes a design element, and the element appears in red on A2’s screen.

**Figure 11:** A1’s Screen after the Removal

**Figure 12:** A2’s Screen after the Removal
The architect A2 changes the geometric location of removed object, which creates a synchronization conflict since A2 deletes an element that no longer exists in the model. CoDesign captures the conflict, and notifies both of them.

Architect 1: A1

Architect 2: A2

Figure 13: A1 Receives a Conflict Notification

Figure 14: A2 Receives a Conflict Notification

Figure 15: CoWare Server Status after It Detects a Conflict between A1 and A2

The log of Conflict Detector shows that A1 and A2 have a conflict, and it sends conflict notification messages to both A1 and A2.