PROACTIVE DETECTION OF HIGHER-ORDER SOFTWARE DESIGN CONFLICTS

by

Jae young Bang

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

May 2015

Dedication

To my wife Youngmi, who has always been next to me through thick and thin during my PhD years, and without whom I would not have been able to complete this work.

Acknowledgments

It has been a long journey to become a scientist, an engineer, and a scholar ever since I began dreaming of becoming one. I would not have been able to reach this milestone without the tremendous help from the people around me, and I consider myself lucky for receiving all the support. The last seven years I spent as a grad student have been the most exciting time of my life. I am grateful to everyone who has been there for me.

I would especially like to thank my advisor Nenad Medvidović for his priceless guidance. I still remember the day he told me that I was admitted to the PhD program at USC. I was trying to look cool, and simply said "oh thanks", but I was screaming from inside with joy. Since that day, he has always been patient with me even when I make mistakes and given me numerous, invaluable advices so that I could grow as a scholar.

I have had an amazing dissertation committee, and I would also like to thank each of the members, Professor Chris Mattmann, Professor William G.J. Halfond, and Professor Viktor Prasanna, for their constructive comments they have given me on my research. This dissertation would not have been of this quality without their meticulous guidance.

I would also like to acknowledge my colleagues with whom I have worked during my PhD years. Yuriy Brun, who now has become a professor, was the first one with whom I worked in my early days after joining the Software Architecture Research Group. He

taught me the first steps of becoming a scholar by showing me how to conduct research projects like a "pro". He is indeed an amazing mentor. I co-authored my very first first-authored paper with Daniel Popescu and George Edwards, and I learned a lot from the ways they work. Daniel was highly detail-oriented (in a good way) and George was excellent at organizing thoughts. Ivo Krka was the one with whom I went on the unforgettable research trip to Infosys in India. He and I had countless discussions from which I gained the insights that eventually formed my dissertation. Joshua Garcia showed me how enthusiastic and focused a scholar should be. During my PhD, I have enjoyed collaborating with many other students in the research group (in the order I first met them); Dave, Farshad, Daniel (Link), Reza, Youn, Lau, Eric, Duc, Pooyan, Arman, Ali, Yixue, Roohy, and Michael. Su and Tip from Prof. Boehm's group have been huge inspirations as well. It has been a joy to work with you! Also, I cannot even imagine how my time at USC would have been like without Lizsl, the greatest PhD program advisor.

I should not forget to mention the USC Trojans, the LA Dodgers, and the LA Lakers players for continuously performing at league-topping level. I will never be able to forget those days I was writing papers while watching their spectacular, dramatic games on TV.

Lastly, but most importantly, I would like to thank my family members for their unconditional love. I thank my beautiful wife, Youngmi, for her support and patience through my PhD years. My parents, Jinsoo Bang and Ok Ja Lee, supported me for my entire life giving me everything they had so that I could have as many opportunities as I wanted. My grandfather Byungran to whom I promised to bring a PhD degree unfortunately passed away a few years ago. I wish he would know that I finally made it.

Table of Contents

List Of Tables

List of Figures

Abstract

A team of software architects who collaboratively evolve a software model often rely on a copy-edit-merge style version control system (VCS) via which they exchange and merge the individual changes they perform to the model. However, because the current generation of software model VCSs detect conflicts only when architects synchronize their models, the architects remain unaware of newly arising conflicts until the next synchronization, raising the risk that delayed conflict resolution will be much harder.

Collaborative software *implementation* faces an analogous risk, and there are existing techniques and tools that proactively detect conflicts at the level of source code in order to minimize the conflict unawareness. However, it is challenging to directly apply them as they are to collaborative software design because those are constructed to manage code-level rather than model-level changes. Furthermore, no empirical data is currently available regarding the impact of proactive conflict detection on collaborative design.

In order to address the risk of design conflicts, this dissertation applies proactive conflict detection to collaborative software design. Specifically, this dissertation focuses on higher-order conflicts that do not prevent merging but do violate a system's consistency rules, because higher-order conflicts are generally harder to detect and resolve than synchronization conflicts that are caused by incompatible changes and prevent merging.

This dissertation presents *FLAME*, an extensible collaborative software design framework that detects the higher-order design conflicts in a proactive way, i.e., before an architect synchronizes her model and finally becomes aware of them. FLAME has an extensible architecture that provides facilities via which the modeling tools and consistency checkers appropriate for the target system's domain can be integrated. FLAME captures modeling changes as they are made, performs a trial merging and conflict detection in the background in order to immediately detect newly arising conflicts, and presents the results to the architects. Also, FLAME explicitly deals with the potentially resource-intensive computations necessary for higher-order conflict detection by parallelizing and offloading the burden to remote nodes. Moreover, by implementing its novel algorithm that prioritizes instances of conflict detection, FLAME guarantees that the outstanding conflicts at a given moment can be detected in a reasonable amount of time even when the available computation resources for conflict detection are scarce.

This dissertation presents the results from two user studies and three systematic experiments on FLAME. The two user studies were conducted involving 90 participants, and the results indicated that the participants who used FLAME were able to create higher quality models in the same amount of time, and to detect and resolve higher-order conflicts earlier and more quickly. The results from the three systematic experiments provided evidence that FLAME minimizes delay in conflict detection, adds only negligible amount of overhead as the number of architects increases, and delivers conflict information early even when the computation resources for conflict detection are limited.

Chapter 1

Introduction

1.1 The Challenge: Software Design Conflicts

Modern software systems are often large and complex, requiring multiple engineering teams that consist of a number of members for their development. During the development, numerous decisions are made on various aspects of the system under development including the structure, the functionalities, as well as the non-functional properties of the system such as performance, security, etc. This essential development activity of making decisions is called *software design*. A key set of stakeholders who engage in software design, *software architects*, make design decisions that define the architecture of the system, reify those decisions into software models [61], and evolve the models as a team [39].

The team of software architects, when they design a large software system, often divide the system into modular subsystems, simultaneously design each of those, and merge them later [6]. A number of design environments have emerged to support this collaborative process of software model evolution. There are the group editors that provide a shared "whiteboard" [10,16,40] or synchronize the models in real time [8], but the major research

effort has been toward the asynchronous, copy-edit-merge style software model version control systems (VCSs). Those VCSs provide each architect her individual workspace by loosely synchronizing the models in an on-demand fashion to parallelize the architects' work and maximize their productivity [2].

However, the loose synchronization of the VCSs exposes software architects to the risk of making design decisions that conflict with each other, called *design conflicts* [8]. In general, design conflicts can be categorized into two different types: *synchronization* and *higher-order* conflicts [9]. A synchronization conflict is a set of contradictory modeling changes by multiple architects made to the same artifact or to closely related artifacts, which means they cannot be merged together. A higher-order conflict is a set of modeling changes by multiple architects that can be merged but together violate the system's consistency rules (e.g., cardinality defined by the metamodel). While both types pose similar risks, they differ in that they require different sets of detection techniques.

Design conflicts are a major challenge in collaborative software design. Today's VCSs detect conflicts only when software architects synchronize their models. As a result, the architects often make changes to the model without fully understanding what issues may arise when they merge their own changes with the others' changes. It is also possible that new changes made after a conflict has been introduced need to be reversed in the process of resolving the conflict, which results in wasted time and effort [6]. Moreover, the current trend of global software engineering, in which software engineering teams tend to be widely distributed geographically due to economic advantages [57], only aggravates the challenge by reducing the opportunities for direct communication among the architects [36, 50].

What if those conflicts could be detected earlier, in a proactive fashion, that is, before an architect synchronizes her model and finally becomes aware of them? This dissertation will present a technique that alleviates the risk of having design conflicts by proactively detecting them and informing the architects of the conflict information early. Specifically, of the two types of design conflicts, this dissertation will focus on the higher-order conflicts because they are generally more difficult to detect and resolve [9, 14, 53]. The remainder of this chapter outlines (1) the existing techniques and their limitations, (2) the list of hypotheses that will be tested by this dissertation, (3) the proposed approach and the contributions of the dissertation, and lastly, (4) the structure of the remaining chapters.

1.2 Solution to An Analogous Problem in Implementation

Collaborative software *implementation* faces a similar challenge to design conflicts. Software developers who use an asynchronous VCS are exposed to the risk of causing conflicts at the level of source code, and the state-of-the-art techniques and tools are indeed capable of proactively providing the code-level conflict information [54] by continuously performing trial merging and conflict detection in the background. However, while the previous, code-level proactive conflict detection research sheds light on how to deal with the conflicts in general, it is challenging to directly apply that to the conflicts at the level of software models due to the following two reasons. First, the existing proactive conflict detection tools are not designed to manage changes made to graphical software models and are often limited to specific development environments into which they are integrated. Tools that are designed to manage textual changes made to source code are known not

to work well with graphical software models [1, 9, 38, 44, 48], and may not be configurable to deal with various kinds of design conflicts that differ per design environment. Second, to our best knowledge, no empirical study has been reported yet on whether or to what extent proactive conflict detection may impact the cost of collaborative software design.

Furthermore, the existing proactive conflict detection techniques are limited with regards to the *version* of the artifacts under version control (e.g., source code or software models) on which they perform proactive conflict detection. To detect conflicts early, those techniques perform trial merging and conflict detection activities in the background. Different versions of the artifacts may be derived at the time of merging depending on (1) how frequently one wishes to detect conflicts and (2) which and whose changes one needs to include in each instance of the detection. Techniques that derive different versions would differ in how early they detect the conflicts and would suffer from different numbers of false positives. For example, Palantír [55], a code-level proactive conflict detection tool, detects conflicts at every file saving or timer expiration while another similar tool, Crystal [15], detects conflicts at every synchronization. Between the two, Palantír would detect conflicts earlier, but because it also marks some "benign" conflicts that could have been resolved before they are synchronized as actual conflicts, it is likely to suffer from larger numbers of false positives [14]. While the appropriate versions of the artifacts to derive on which to perform proactive conflict detection may differ for different development activities (e.g., design and implementation) or different target systems' domains, the existing tools are limited to certain versions on which they perform conflict detection.

An added challenge is that the previous proactive conflict detection research has not explicitly addressed the *delayed conflict detection* problem, which occurs when the computation resources necessary to perform proactive conflict detection surpass what is available. In that case, the conflict detection and the delivery of the conflict information to software architects would consume a lot of time, and that may hamper the actual benefits of proactively detecting the conflicts. Many well-known model analysis techniques that can be adopted for design conflict detection (e.g., discrete-event simulation [56], Markov-chain-based reliability analysis [65], or queueing-network-based performance analysis [4]) are indeed highly computation-intensive and time-consuming. Unfortunately, the existing techniques implicitly assume that each instance of conflict detection will be done in nominal time, and it is not yet clearly known how the delayed conflict detection problem will impact the benefits of proactive detection of higher-order design conflicts.

1.3 Insights and Hypotheses

The limitations in the existing conflict detection techniques and the challenges in applying them to collaborative software design, as introduced in Section 1.2, suggest that several research questions regarding design conflicts and proactive conflict detection remain open. The rest of this section presents the hypotheses that this dissertation tests in order to overcome those limitations and challenges.

1.3.1 Hypothesis 1

Insight 1A: In a VCS, software architects initially duplicate the model in the repository to create a local *working copy* of the model. They perform *commits* to merge the changes

that have been made in the working copy to the repository and *updates* to merge the changes in the repository that have been made by others to the working copy.

Insight 1B: An extensible collaborative software design environment that proactively detects higher-order conflicts can be built such that it delivers conflict information to architects earlier than a similar environment that only detects conflicts on-demand.

Insight 1C: A proactive conflict detection technique that derives and performs conflict detection on the version of a software model that consists of *all* changes made up to a given moment, whether or not committed, would provide architects the awareness of what conflicts may arise if every architect performs a *commit* at that moment.

Insight 1D: A proactive conflict detection technique that, for each architect, derives and performs conflict detection on the version of a software model that is based on the newest version in the repository and merges the uncommitted changes made in her working copy up to a given moment would provide the architect the awareness of what conflicts may arise if she performs an *update* at that moment.

Hypothesis 1: In a collaborative software design environment that proactively detects higher-order conflicts, the quality of the resulting model is *better* after a modeling session of the same length conducted by engineers of the same experience level than in a similar environment that only detects conflicts on-demand.

1.3.2 Hypothesis 2

Insight 2A: The time to complete performing an instance of higher-order design conflict detection may take *longer* and the conflict information delivery to software architects may

be *delayed* when the machine on which the instance is processed also has other instances to perform than when the instance is the only one to be processed on that machine.

Insight 2B: An instance of higher-order design conflict detection consists of a version of a software model as the input and a consistency checker to be run on that version. Different instances do not depend on the results of the other instances, hence they can be processed independently and simultaneously on multiple machines in parallel.

Insight 2C: When M_n represents a version of a software model in which a higher-order design conflict c initially appeared, and $M_{n+\alpha}$ represents a version of the same model that is chronologically behind M_n, the conflict c can be detected by performing a conflict detection on $M_{n+\alpha}$, if c has not been resolved by the time $M_{n+\alpha}$ was created.

Hypothesis 2: For cases where the number of higher-order software design conflict detection instances to be processed exceeds the number of available machines on which those instances can be processed simultaneously, an algorithm that assigns higher priority to the instances with chronologically newer versions of the model can be devised such that, when t is the longest time required to process a single detection instance with no delay, any *outstanding* higher-order design conflict that had not already been resolved at a given moment can be detected, at most, in the amount of time $2 \cdot t$ from its creation.

1.4 The Proposed Solution and The Contributions

This dissertation presents the research that applies proactive conflict detection to collaborative software design. The list of the contributions of this dissertation is as following:

1. The first application of proactive conflict detection on collaborative software design.

2. The extensible collaborative software design environment that can integrate modeling tools and model analysis techniques appropriate for the target system's domain.

3. The distributed conflict detection architecture that moves the burden to remote nodes with minimal overhead in order to not disturb collaborative design process.

4. The conflict detection prioritization algorithm that ensures early delivery of conflict information even when computation resources are scarce.

5. The first reported empirical evidence that proactive conflict detection positively impacts collaborative software design.

It is important to note that the focus of this dissertation is *not* on the manner in which higher-order design conflicts are detected, which has already been widely studied [2]. It focuses on the ways to *proactively* detect the higher-order conflicts by exploiting the existing techniques, and further, on the impact of doing so on collaborative design.

FLAME has been developed as an attempt to alleviate the risk that higher-order software design conflicts pose. It does so by performing the conflict detection activities in the background without getting software architects' attention and by notifying the architects of the conflict information early in order to minimize the risk of having unknown higher-order design conflicts. FLAME is unique as it is the first reported proactive conflict detection framework that is specifically targeting collaborative software *design*. It has the following characteristics that distinguish it from the existing tools. First, it is *extensible* to allow the architects to plug-in the most appropriate conflict detection tools for their modeling environment. Second, it can *scale* by exploiting cloud technology to parallelize the potentially computation-intensive and time-consuming higher-order design

conflict detection. Third, it ensures *early delivery of conflict information* by prioritizing the detection instances and setting a worst-case bound on the time of detecting newly arising higher-order design conflicts. Fourth, it implements *operation-based* model construction [12, 38] to be able to perform conflict detection at the rate of each individual modeling change if necessary, which in turn, enables adopting different techniques that derive and perform proactive conflict detection on various versions of the model.

Different types of analytical and empirical evidence on how the proactive conflict detection that FLAME provides positively impacts collaborative software design will be presented in the dissertation. As one example, the dissertation reports the results of two user studies conducted with a total of 90 participants using FLAME to assess whether and to what extent providing proactive conflict detection impacts the cost of collaborative software design. In each of the studies, the participants were divided into two groups and performed collaborative software design tasks using FLAME in its two modes: one that provides proactive conflict detection and the other that does not. All modeling changes and design conflicts were tracked and recorded by FLAME, and the collected data was used for the comparisons between the two modes. The result showed that the participants who were provided with proactive conflict detection had more opportunities to communicate and were able to create higher-quality models in the same amount of time and to detect and resolve higher-order conflicts earlier and more quickly.

Using the data collected from the user studies, three systematic experiments were conducted on how the framework scales by exploiting remote nodes and prioritizes the conflict detection instances to ensure early delivery of conflict information to the architects. The first experiment evaluated whether the delay in conflict detection actually decreases as

FLAME parallelizes the detection and offloads the burden to remote nodes. The second experiment focused on the overhead in conflict detection time that FLAME may add in collaborative design scenarios involving many (e.g., 24) architects where FLAME has to process a large number of simultaneous instances of conflict detection. The third experiment compared the two FLAME configurations, one that does and the other that does not implement FLAME's algorithm that prioritizes the conflict detection instances. The results from the experiments showed that FLAME (1) alleviated the delay in conflict detection, exploiting remote nodes, (2) added only a negligible amount of overhead even when it processes a large volume of simultaneous conflict detection, and (3) detected outstanding higher-order conflicts at a given moment within the amount of time $2 \cdot t$ at most (where t is the longest time required to process a single detection instance with no delay) when the available computation resources for conflict detection were limited.

1.5 Structure of This Dissertation

The remainder of this dissertation is structured as follows: Chapter 2 defines the problem this dissertation aims to solve. The architecture and implementation of FLAME, the proposed solution, is in Chapter 3, and the evaluation is presented in Chapter 4. Chapter 5 provides the details on related work, and lastly, the dissertation concludes in Chapter 6.

Chapter 2

The Problem Space

This dissertation targets questions such as "What are the *higher-order* design conflicts?", "Why are they undesirable?", and "In which ways can they properly be handled?" In order to drive this discussion, in this chapter, the problem space of the dissertation will be defined in detail. The rest of the chapter is organized as follows. It begins with describing the background on which this dissertation is founded in Section 2.1. Then in Section 2.2, the two types of design conflicts, *synchronization* and *higher-order*, are defined using a realistic collaborative software design scenario. Section 2.3 describes the ways architects detect and resolve those conflicts in practice. Lastly, in Section 2.4, the proactive conflict detection technique and the limitations of the existing applications of it are revisited.

2.1 Background

2.1.1 Software Design Concepts

The definition of *software architecture* that will be used throughout this dissertation is "a set of principle decisions about a software system" [61]. Talyer et al. [61] define *software*

design as "an activity that creates part of a system's architecture, which typically defines a system's structure, identification of its primary components, and their interconnections". Software design is the activity on which this dissertation focuses. Engineers who perform software design, a key activity in developing software systems, make *design decisions*, "the decisions encompassing every aspect of the system under development" [61]. Those decisions are regarding the structure of the system, the functional behavior, the interactions, the non-functional properties, and even the implementation. *Software architects* are the engineers who make those design decisions, and they produce software models as the output. A *software model* is defined as "an artifact documenting some or all of the architectural design decisions about a system" [61]. Lastly, the activity of reifying and documenting a software model is called *software modeling*.

2.1.2 Global Software Engineering

In recent years, due to economic advantages, many technology companies have transferred significant portions of their software development activities to emerging economies such as India and China [57]. 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 [59]. The engineers coordinate with each other from distributed sites in such collaboration environments. While the economic advantages of distributed software development are real, communication and coordination challenges must be overcome in

order to fully realize these advantages. There is significant evidence that geographic separation can drastically reduce communication among coworkers [36, 50] and slow down collaborative development activities [35] including collaborative software design [6].

2.1.3 Version Control Systems Overview

During software development, engineers produce many artifacts such as requirements documents, software models, and source code. Those artifacts continue to evolve during the development, incorporating numerous changes made by the engineers. To track the changes and organize the artifacts, engineers have adopted *version control systems (VCSs)* that "track incremental versions (or revisions) of files and directories over time" [18]. *Version control* is "the task of keeping software systems consisting of many versions and configurations well organized" [62]. Since the very first stint of VCS, which was Source Code Control System (SCCS) by Rochkind [52], many VCSs have arisen to support collaborating software engineers [19]. Concurrent Versions System (CVS) [31], Subversion (SVN) [17], and Git [27] are among the variants that have widely been adopted by the public. Unfortunately, those VCSs that are designed to manage textual artifacts are known not to work well with graphical software models (recall Section 1.2), hence a number of VCSs that are specifically designed to manage software models have emerged to cope with the challenges of collaborative software *design* [2]. For example, those *software model VCSs* are (1) capable of computing differences between independently modified copies of a software model [44, 48], (2) customized for a modeling tool [26] or a notation [42, 46], or (3) extensible to adapt to a given modeling environment [1].

However, since the current generation of software model VCSs only detect or allow detecting conflicts when software architects synchronize their models, the architects are often exposed to the risk of making changes to the models without fully knowing what may happen when they merge each other's changes. Existing research has not yet satisfactorily explored potential mitigations of that risk, and the problem still remains open.

2.1.4 Software Model Version Control Terminology

This section introduces the common version control terms that will be used in the rest of this dissertation. The terms introduced here are used by many VCSs that are widely used in practice (e.g., Concurrent Versions System [31] and Subversion [17]), or if they use different terms, the corresponding terms are directly interchangeable.

A copy-edit-merge style software model VCS maintains a centralized copy of the model called the *repository*. Software architects who work as a team "copy" the model in the repository to their local machines. Those copies of the model located in the local machines of the architects are called the *working copies*. That version control activity–copying the entire repository to create a working copy–happens at the beginning of the design session, and the activity is called the *check-out*. During the design session, the architects make design decisions and make modifications to their working copies. When each of those architects has completed a segment of work and is confident with the outcome, she performs the *commit* activity to push the modeling changes she has locally made to the repository. The architects also periodically perform the *update* activity to pull the modeling changes that have been made by others and have been committed. The sets of

modeling changes made by the architects are automatically merged by the VCS as the architects perform those version control activities.

The copies of the model—the repository and the working copies—evolve over time, and therefore versions of the model are created along the way. A *version* (or a revision) of the model is a state of the model that consists of a list of modeling changes that have been made up to a particular point in time. A *base version* is a version of a model on which an architect makes new modeling changes. The *head version* is the newest version of the model in the repository, and a new head version is created when an architect successfully commits new modeling changes made in her working copy to the central repository.

A working copy can be in two states in terms of its relationship with the repository. First, a working copy can be in the *same* or *current* state when its base version is the same as the head version. Second, a working copy can be in the *behind* or *trailing* state when its base version is older than the head version. These states show whether there are new modeling changes that have been committed to the repository but are missing from the working copy. For example, if a working copy is in the *same* state with the repository, that means the working copy has all the modeling changes that the head version has, hence performing an update at the moment is unnecessary. On the other hand, if the working copy is in the *behind* state, that means an update will be necessary since there are new modeling changes in the head version that are missing from the working copy.

2.2 Software Design Conflicts

When sets of modeling changes are merged, two types of conflicts—*synchronization* and *higher-order*—could occur. To clearly illustrate the problem and to drive the discussion throughout this dissertation, an example system called Next-Generation Climate Architecture (NGCA, depicted in Figure 2.1) will be used. NGCA is created based on the design documents of NASA Computational Modeling Algorithms and Cyberinfrastructure (CMAC) [43], which is an infrastructure that supports computationally-heavy data comparisons between climate simulation models' output and the actual climate data collected via remote sensors. Because the climate simulation models and databases that compose NGCA belong to different organizations scattered around the world, designing NGCA naturally becomes collaborative, involving architects from those organizations.

Consider the following scenario with two architects participating in the NGCA design.

> The architects make design decisions and document the decisions into a software model. The modeling environment they use is semantically-rich and domain-specific, and is capable of estimating several critical runtime properties of NGCA, including memory usage, message latency, and energy consumption. The model is managed by a current generation, copy-edit-merge style software model VCS. In our scenario, each of the two architects makes changes to her respective working copy, runs estimations locally, and moves to another design task after she did not find any issue in the estimated property values. However, a while later, when they try to synchronize their working

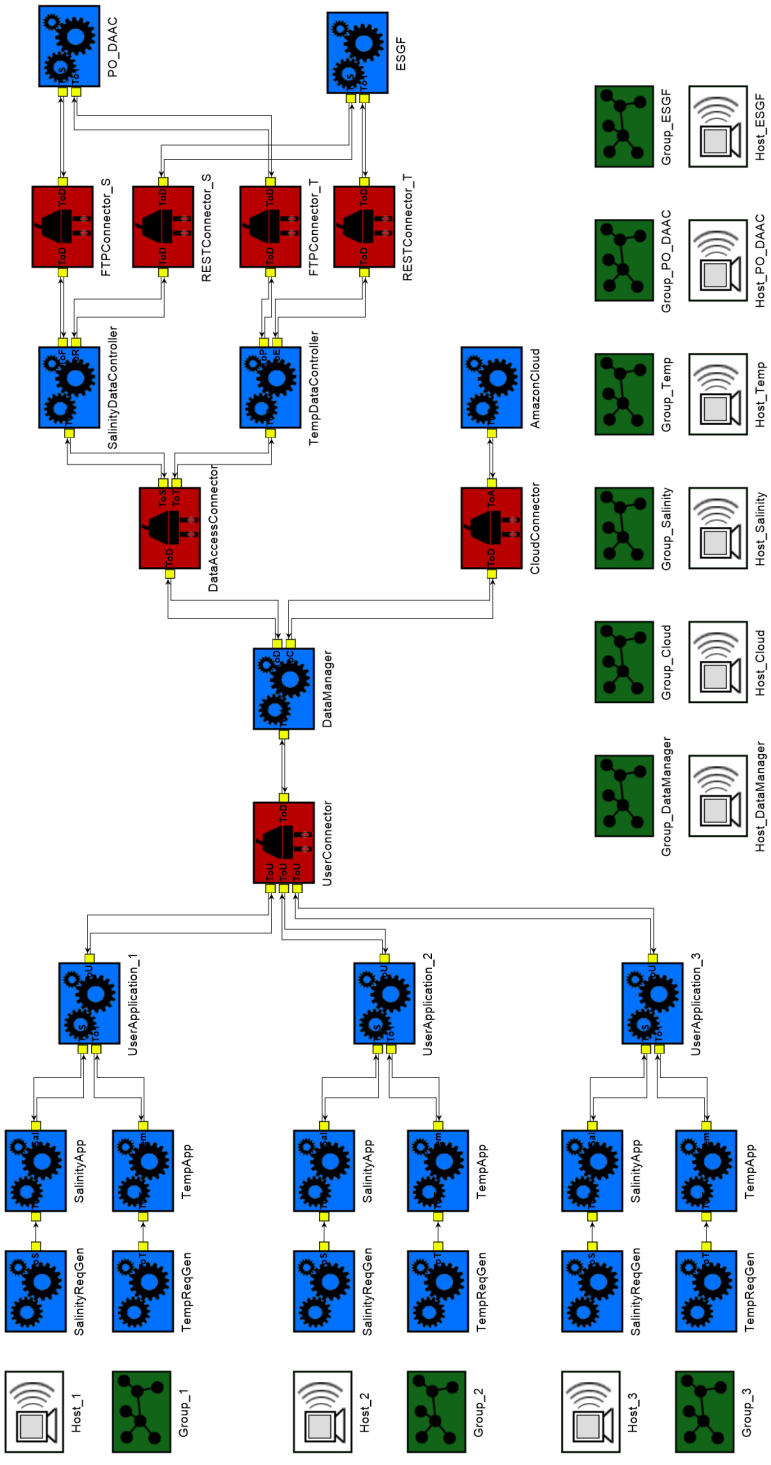


Figure 2.1: A high-level model of Next-Generation Climate Architecture.

copies by merging the changes, the two architects realize one or both of the following two situations:

1. Their changes were made to the same object and were incompatible. For example, one of them removed the object while the other one updated an attribute of the same object. As a result, their changes cannot be merged into a consolidated model.
2. They are able to merge their changes, but the merged model estimates that the memory usage of the system at runtime will surpass the threshold defined by the NGCA requirements.

The above scenario depicts examples of (1) a synchronization conflict and (2) a higher-order conflict respectively. Those conflicts are defined as following [9]:

- A **synchronization conflict** is a set of design decisions made by multiple architects that are not compatible and cannot be merged into a single, consolidated model. It occurs when multiple software architects make contradictory modeling changes on the same software modeling artifact or closely related artifacts. A synchronization conflict is also called a context-free conflict [64]. It is similar to a textual conflict [14] or a direct conflict [21, 32, 55, 66] discussed in literature.
- A **higher-order conflict** is a set of design decisions made by multiple architects that do not prevent synchronization but together violate a consistency rule or a semantic rule of the system. In other words, a higher-order conflict manifests itself as an *inconsistency* in the merged model. A higher-order conflict is also called a

context-sensitive conflict [64]. An analogous concept at the source code level [14] is known as an indirect conflict [21, 32, 55, 66].

A software model that is concurrently developed cannot realistically evolve without having inconsistencies [5, 28]. At the same time, having an inconsistency caused by a higher-order conflict that is undetected and unknown to the architects is a risk. For example, when the two architects in our scenario discover the higher-order conflict, they have to revisit their previous changes in order to understand and resolve the conflict, recalling the rationale and assumptions they made along the way. Moreover, their work performed after the conflict has been introduced may need to be reversed during the process, which leads to wasted effort and increased development cost.

2.3 Conflict Detection Practice

Software architects, during collaborative design, use VCSs to synchronize their modeling changes, and also to detect, comprehend, and resolve conflicts. This section describes in which ways those behaviors are supported in today's collaborative design environments.

2.3.1 Detecting Conflicts

Design conflicts are detected when modeling changes are merged. When an architect tries to merge her changes, if there are synchronization conflicts, the merge fails, and the VCS raises a flag. In case the merge succeeds (i.e., there are no synchronization conflicts), the architect (or the VCS) would analyze the merged model and check whether any consistency rule has been violated to detect higher-order conflicts.

While both commit and update activities trigger a merging of modeling changes, in modern VCSs, conflicts can be detected only when an update is performed. Modern VCSs often do not allow performing a commit when the base version of a working copy is behind the head version. The rationale behind that is to prevent potential higher-order conflicts to be transferred to the repository and propagated to the other architects. In that case, the architect must perform an update first in order to become able to perform a commit. After the update, and before eventually performing the commit, the architect is also encouraged to detect and resolve any higher-order conflicts in her working copy.

2.3.2 Comprehending Conflicts

In order to resolve a design conflict, an architect needs to understand the causes of the conflict first. Narrowing down which changes have caused the conflict and by which architect those changes were made is a key to resolving the conflict. Obtaining this knowledge, i.e., *comprehending* a conflict, is often not trivial since a large number of changes could have been made by the time the conflict is detected, and it is possible that most of those changes are not directly related to the cause of the conflict.

While comprehending a synchronization conflict is relatively straightforward, it is often not as easy to comprehend a higher-order conflict. When a synchronization conflict arises, most VCSs are indeed capable of automatically retrieving the list of modeling changes that have been made to the part of the model in which the conflict occurred and by which architects those changes are made to support the resolution process (e.g., as in [8, 13, 38]). On the other hand, comprehending a higher-order conflicts often requires manual inspection of the prior modeling changes. For example, in the scenario where

the joint changes made by the two architects violated the memory usage requirement (recall Section 2.2), the architects would need to revisit their previous work, recalling the decisions they made along the way in search for the set of modeling changes that pushed the memory usage over the threshold. The process of performing such a manual inspection and recollection is likely to be tedious, time-consuming, and error-prone. This is especially a problem when the higher-order conflict is detected after a large number of modeling changes have been made since the initial occurrence of the conflict.

2.4 Proactive Conflict Detection

To deal with the risk of having undetected higher-order conflicts (recall Section 2.2), one solution would be for the architects to synchronize and detect the conflicts with a high frequency, e.g., for every change that they make. However, the cost of conflict detection in that case is likely to overwhelm its benefits. In today's collaborative software design environments, that is a trade-off decision that the architects have to make. The burden of conflict detection grows further when the detection technique is computationally expensive. While a few performance-oriented model analysis techniques are lightweight [24,47], many other well-known techniques such as discrete-event simulation [56], Markov-chain-based reliability analysis [65], or queueing-network-based performance analysis [4] are often computation-intensive and time-consuming, rendering highly frequent conflict detection less affordable, especially as the size of the system model grows.

A number of tools for collaborative *implementation* that proactively detect conflicts have emerged [54] to minimize the analogous risk of having undetected conflicts at the

source code level. Those tools perform trial merging and conflict detection activities in the background to detect conflicts early. This dissertation hypothesizes that a similar approach, when it is applied to collaborative *design*, will reduce the risk that architects face. However, challenges exist in directly reusing the existing proactive conflict detection tools for software design because these tools are constructed to manage code-level rather than model-level changes and are often integrated into a specific development environment.

An added challenge is that different software modeling environments depend on their unique combinations of modeling tool and consistency checkers, and any proactive conflict detection solution should be able to cope with the differences. For example, revisiting our scenario, the proactive conflict detection tool for NGCA should be able to (1) orchestrate the NGCA-specific modeling tool and consistency checkers to automatically perform the higher-order conflict detection activities in the background, (2) present conflict information specific to the environment (e.g., violations of the three runtime properties of NGCA), as well as (3) synchronize the graphical modeling changes. Unfortunately, the existing proactive conflict detection tools do not fully satisfy the above requirements.

Chapter 3

Solution

To alleviate the risk of having undetected higher-order design conflicts, this dissertation proposes a novel collaborative software design framework, named *Framework for Logging and Analyzing Modeling Events* (FLAME). FLAME minimizes the duration of time during which the conflicts are present but unknown to software architects by proactively performing the conflict detection activity that includes a trial merging of modeling changes and execution of consistency checking tools in the background. FLAME subsequently presents the conflict information to the architects in case their attention is required.

The rest of this chapter describes FLAME. The architecture of FLAME, including how it is designed and how it differs from the existing proactive conflict detection tools, is presented in Section 3.1. Section 3.2 discusses FLAME's implementation.

3.1 The Architecture of FLAME

This section introduces the architecture of FLAME and the principal design decisions made in the process of constructing the architecture. Section 3.1.1 overviews the high-level architecture of FLAME—its primary components and the ways in which those components interact with each other—designed to proactively detect conflicts at the model-level. The remainder of this section describes how FLAME is designed in order to handle the potentially large computation resource needs for the detection of conflicts (recall Section 2.4). Section 3.1.2 is regarding how FLAME employs remote nodes to adopt computation resources as needed, and Section 3.1.3 is regarding how FLAME prioritizes instances of conflict detection to minimize delay in delivering conflict information to software architects that may be caused when computation resources are scarce.

3.1.1 Applying Proactive Conflict Detection to Software Design

FLAME is designed to overcome the challenges that arise when adopting the existing proactive conflict detection tools for collaborative software design (recall Section 1.2). FLAME has the following two characteristics that distinguish it from the existing tools. First, FLAME is *extensible*. Software modeling environments differ in their modeling tools, languages, and the suitable consistency checkers. FLAME utilizes an event-based architecture in which highly-decoupled components exchange messages via implicit invocation, allowing flexible system composition and adaptation. FLAME exploits this event-based architecture to provide explicit extension points for plugging a variety of off-the-shelf tools—namely, modeling tools and conflict detection engines—that are most

appropriate for the given modeling environment. Second, FLAME is *operation-based*. Conflict detection can become more fine-grained if the version control is done by tracking modeling operations (i.e., actions such as creation, update, or removal of a modeling element) rather than "diffs" between stored states of a model (e.g., saved files) [38]. The operation-based version control is advantageous because an architect can find out which particular operation that she performed has caused a conflict [12]. For each modeling operation made, FLAME automatically performs trial synchronization and conflict detection in the background in order to immediately detect newly arising conflicts. On the other hand, performing conflict detection for each operation could be a significant, even unacceptable tax on the system's performance. FLAME deals with this explicitly, by employing remote, cloud-based analysis nodes, as discussed later in this chapter.

FLAME adds higher-order proactive conflict detection on top of a conventional copy-edit-merge collaborative design environment. Figure 3.1 depicts the high-level architecture of FLAME. On the architect-side, FLAME attaches a modeling tool-specific adapter, *FLAME Adapter*, to the modeling tool to capture each operation as it is made via the modeling tool's APIs. A *FLAME Client* is installed at each architect's site to establish a channel between the architect-side and the server-side through which the captured operations can be sent. For proactive conflict detection, each captured operation is immediately forwarded from the *FLAME Adapter*, through the *FLAME Client*, to the server-side.

The server-side *Client Manager*, which manages the connections between the server-side and *FLAME Clients*, receives the operation and forwards it to *Detector Manager*, which subsequently replicates and broadcasts the operation to all connected *Detection Engines*. A *Detection Engine* is similar to a *FLAME Client* in the way that it is connected

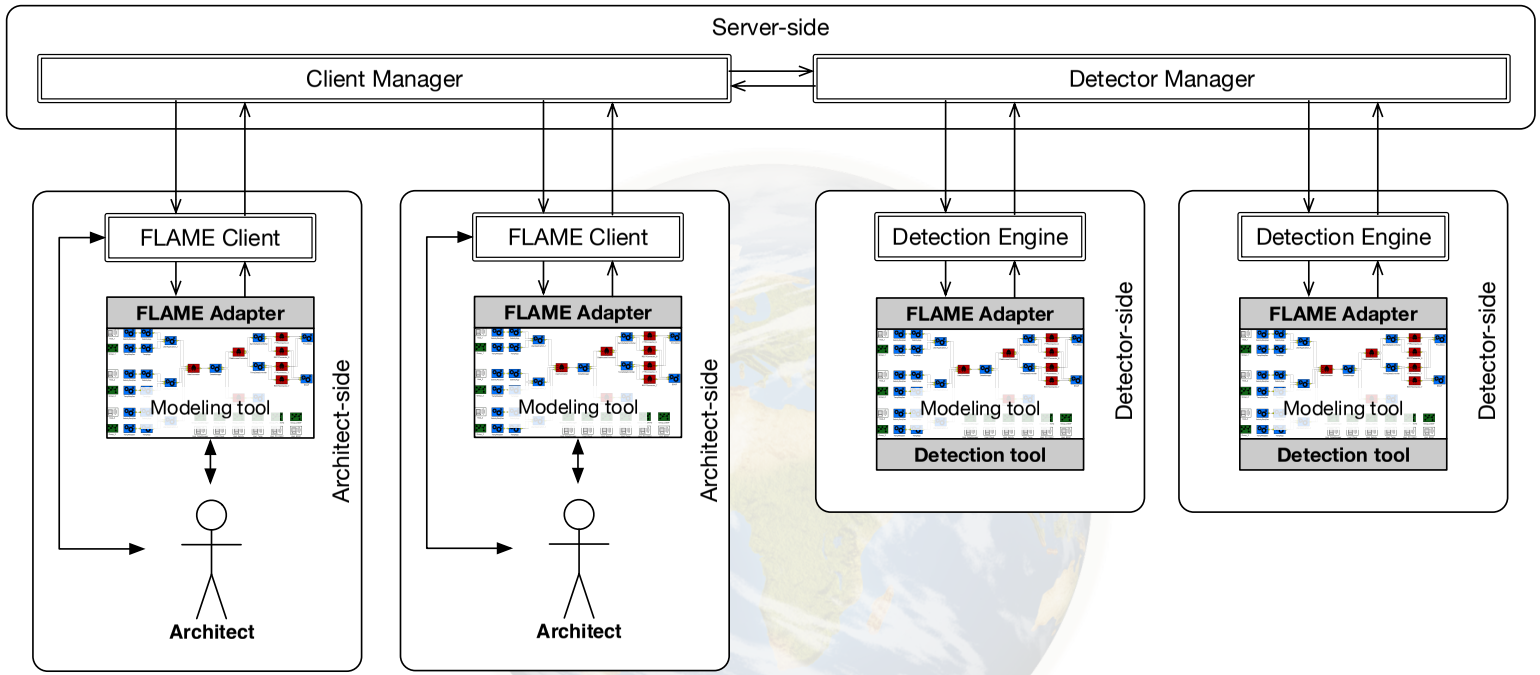


Figure 3.1: High-level architecture of FLAME with two architects and two *Detection Engines*. The gray polygons (*FLAME Adapters* and *Detection Tools*) as well as the modeling tool are design environment-specific components.

to an instance of the modeling tool with a local copy of the model internally, but a *Detection Engine* does not have an architect using the modeling tool initiating operations. Instead, it has an off-the-shelf conflict detection tool plugged into the modeling tool. An instantiation of FLAME may have multiple *Detection Engines*, each of which has a different conflict detection tool and may maintain a different version of the model.

When a *Detection Engine* receives an operation that has been broadcast by the *Detector Manager*, the *Detection Engine* applies the operation to its local copy of the model, automatically invokes the conflict detection tool, and analyzes the outcome as an architect would do. The result of the analysis is then consolidated and delivered back to the architects via FLAME in the reverse order to that described above, i.e., from the *Detection Engine*, via the server-side components *Detector Manager* and *Client Manager*, and eventually to the architect-side *FLAME Clients*.

3.1.2 Scaling Conflict Detection

In order to offload the potentially resource-intensive computations necessary for higher-order conflict detection from the architect-side or the server-side machines, FLAME employs remote nodes to perform the detection. If a computation-intensive type of conflict detection (recall Section 2.4) were to be performed on an architect's machine or on the server for every modeling operation, it could overwhelm the machine and hamper the design activity. FLAME therefore moves the burden to the *Detection Engine*.

FLAME can utilize more than one *Detection Engine* in concert to parallelize the higher-order conflict detection. An architect may need to perform multiple conflict detection activities using several tools that implement different techniques (e.g., a combination

of static and dynamic analysis) or different instances of the same technique (e.g., reliability and latency analysis via discrete event-based simulation [23]). FLAME distributes these conflict detection activities to multiple remote nodes on a cloud. It instantiates multiple *Detection Engines*, each of which is responsible for performing a single higher-order conflict detection activity using the corresponding conflict detection tool. This aspect of FLAME's architecture allows different *Detection Engines* to be instantiated as needed, possibly even at runtime. The network delay that is introduced by distributing the conflict detection activities to multiple nodes is likely to be minimal and negligible, especially if the conflict detection technique used employs time-consuming analysis.

While migrating the conflict detection to *Detection Engine* would prevent the architect-side or the server-side machines from being resource-starved, there is another risk that each *Detection Engine* may become a bottleneck in conflict detection when it is overwhelmed by a large number of of simultaneous conflict detection instances to perform. Suppose it takes a uniform amount of time t for a *Detection Engine* to complete processing a single conflict detection instance. If the *Detection Engine* had n (where $n > 1$) instances to process, whether it processes all of them simultaneously or one-by-one, at least one of the instances could take longer than t (possibly n times), or in other words, be *delayed*. Consequently, the delivery of the resulting conflict information to architects would also be delayed, and that might eliminate the benefits of proactively detecting conflicts. The risk of *Detection Engines* overwhelmed by a large amount of simultaneous conflict detection is real, and indeed, it was observed that many conflict detection instances were delayed in an empirical study conducted using FLAME, which will be

presented later in this dissertation. Moreover, that risk is likely to worsen in practice when the architect team's size is larger and the model is larger in size or more complex.

FLAME explicitly deals with the delay in conflict detection by employing remote, cloud-based nodes, or *slave nodes*, whose primary purpose is to perform conflict detection. During a collaborative design session, in general, a *Detection Engine* iterates through the following *six* steps: (1) receiving a modeling operation, (2) applying the received operation to the local copy of the model the *Detection Engine* internally maintains, (3) generating a representation (e.g., a saved model file) of the version of the model after the application, (4) invoking the off-the-shelf conflict detection tool that the *Detection Engine* integrates by feeding the representation into the conflict detection tool, (5) consolidating the result of the detection, and (6) forwarding the consolidated result to the server-side. The fourth step—invocation of the off-the-shelf conflict detection tool—may consume a significant amount of computation resources and time (the remaining steps should only require minimal resources and be completed in a nominal amount of time). The fourth step, however, does not depend on the outcome of the prior iterations, once the model representation that will be fed into the conflict detection tool has been generated. To minimize the potential delay in conflict detection, FLAME can parallelize the fourth step and further move the burden of conflict detection to the slave nodes.

Figure 3.2 depicts the FLAME instance that uses slave nodes for conflict detection. This instance differs from the one depicted in Figure 3.1, in terms of the machines on which it relies to perform the potentially resource-intensive conflict detection. The detector-side of this FLAME instance has a component called *Slave Manager*, which manages a pool of slave nodes. Slave nodes may join the pool on-the-fly by initiating a connection to the

Slave Manager, and the *Slave Manager* can asynchronously and simultaneously process several conflict detection instances by deploying the instances on the slave nodes.

The detector-side components, *Detection Engine* and *Slave Manager*, of the FLAME instance that uses slave nodes for conflict detection behave as depicted in Figure 3.3. During a collaborative design session, in general, the *Detection Engine* iterates through the following *four* steps instead: (1) receiving a modeling operation, (2) applying the operation to the local copy of the model the *Detection Engine* internally maintains, (3) generating a representation (e.g., a saved model file) of the post-application version of the model, and (4) forwarding the generated representation to the *Slave Manager*. The forwarded representation is then stored in *Model Representation Queue* in the *Slave Manager*. The *Slave Manager*, meanwhile, waits for a slave node to become available. When a slave node becomes available, the *Slave Manager* initiates a separate thread that retrieves the oldest model representation in the *Model Representation Queue*, transfers the representation to the recruited slave node, and has the slave node perform conflict detection on the representation. The thread waits until the slave node completes the detection and returns the result. The thread then releases the slave node, and consolidates and forwards the result to the *Detection Engine*. As the final step of this process, the *Detection Engine* forwards the consolidated conflict information to the server-side.

The extensible architecture of FLAME provides the elasticity necessary to adapt to the varying amounts of computation resources needed for conflict detection. During a collaborative design session, the number of simultaneous conflict detection instances that a *Slave Manager* needs to handle may continuously change depending on factors such as the number of software architects who are participating in the session, the rate

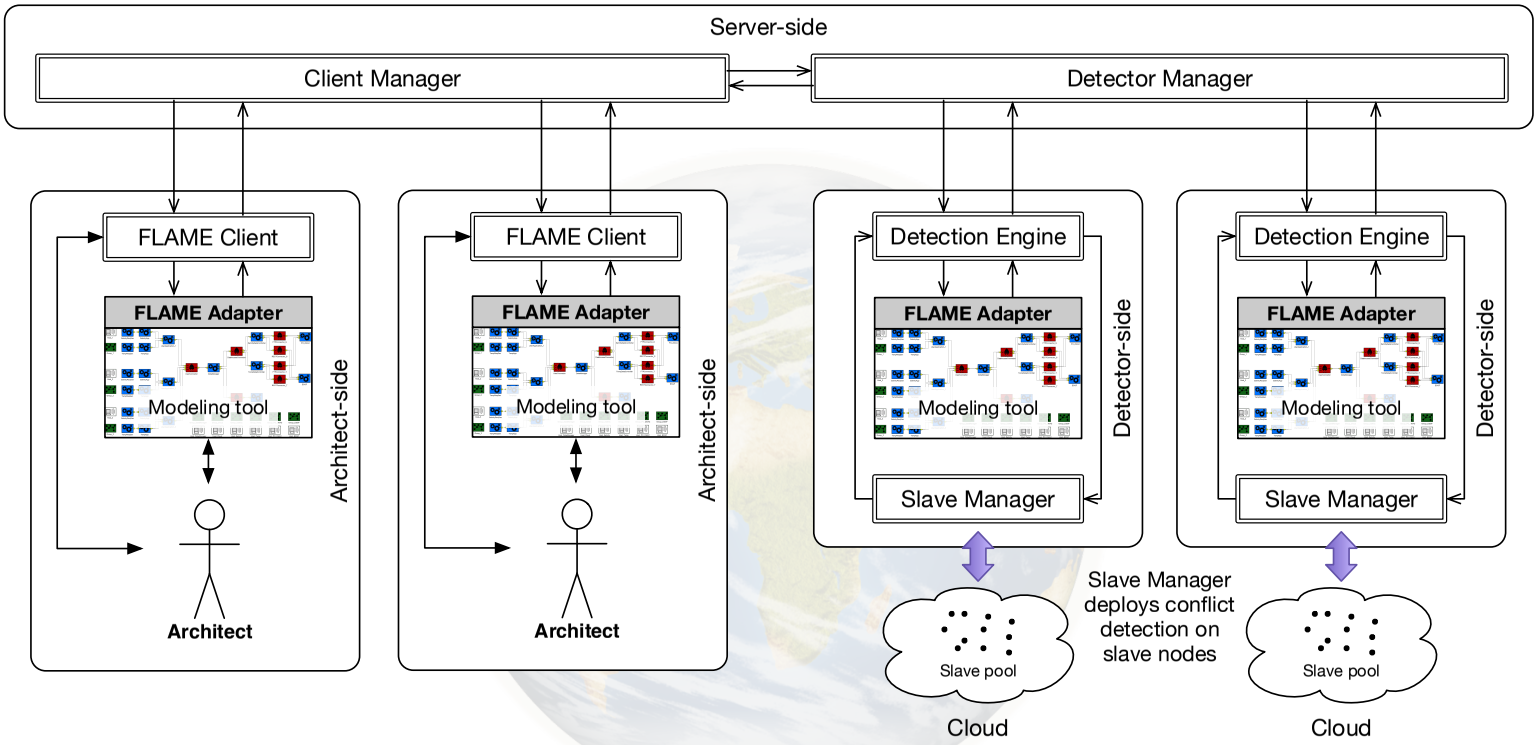


Figure 3.2: High-level architecture of FLAME that uses slave nodes for conflict detection.

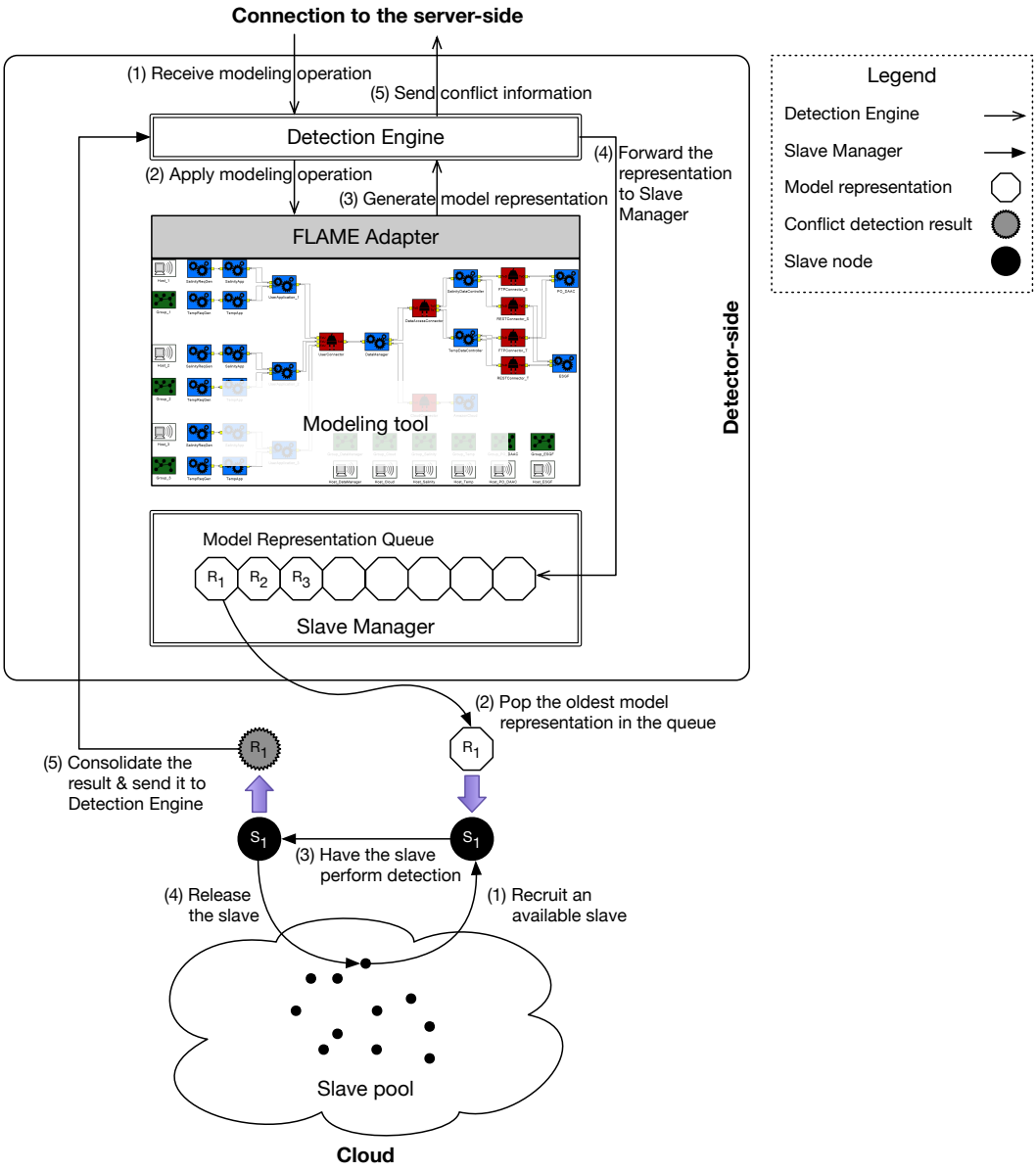


Figure 3.3: A model of detector-side that uses slave nodes for conflict detection.

at which those architects are generating modeling operations, the amount of time for a slave node to complete processing each conflict detection instance, etc. Meanwhile, at the *Slave Manager*, it is possible that some of the conflict detection instances are delayed while waiting for a slave node to become available if the number of simultaneous conflict detection instances exceeds the number of slave nodes in the pool. FLAME allows increasing and decreasing the size of the slave node pool on-the-fly to address such change in the number of slave nodes needed. Furthermore, integration of a cloud into FLAME as the platform on which the slave nodes run would make the resizing of the slave node pool easier since a cloud can provide varying amounts of computation resources on-demand [3].

3.1.3 Prioritizing Conflict Detection

In practice, the available computation resources for conflict detection are likely to be limited, and the *Slave Manager* may have fewer than the desired number of slave nodes in the pool. Figure 3.4 depicts an example scenario with five conflict detection instances. O_1 through O_5 are modeling operations that arrive in that chronological order in a *Detection Engine*. The *Detection Engine*, for each of those modeling operations, generates a model representation R_1 through R_5 respectively after applying the operation to its local model. Suppose that the available computation resources for conflict detection are limited, and the *Slave Manager* can only have a single slave node at its disposal. The *Detection Engine* would generate R_1 as O_1 arrives, then the *Slave Manager* would begin processing an instance of conflict detection on R_1 using its sole slave node. However, delay may occur if the operations O_2 through O_5 arrive in the *Detection Engine* before the conflict detection on R_1 is complete. The corresponding model representations R_2 through R_5 would have

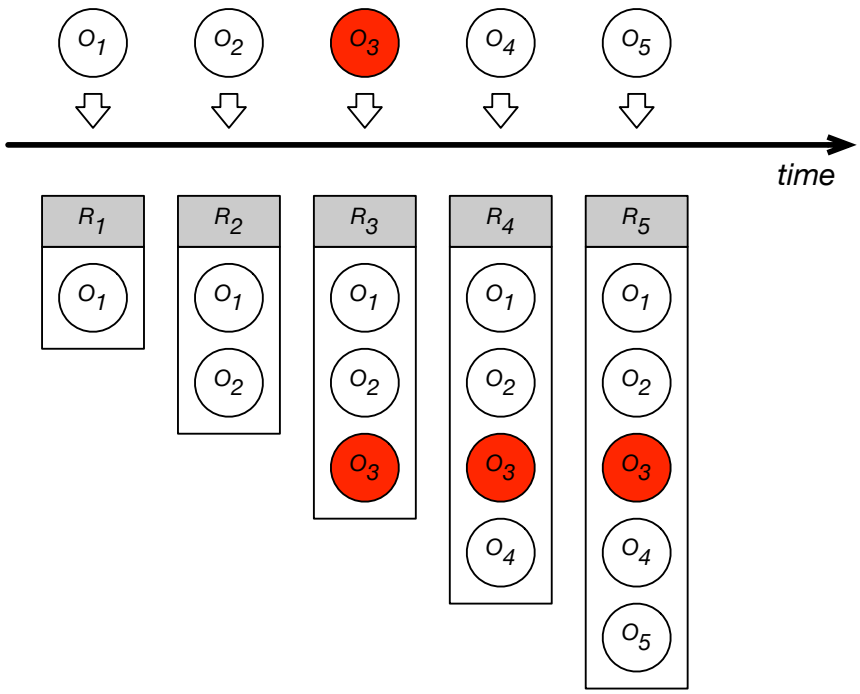


Figure 3.4: O_1 through O_5 are modeling operations a *Detection Engine* receives, and R_1 through R_5 are their corresponding representations of the versions of the model that consist of those operations over time. The modeling operation in red (darker highlighting in grayscale), O_3, causes a higher-order design conflict with a previous modeling operation.

to wait in the *Model Representation Queue* until the slave node completes the on-going detection and becomes available again. Moreover, for some of those representations, the wait could be longer than just one conflict detection instance since there are several of them in the queue while there is only one slave node.

In order to minimize the delay that may occur when the computation resources for conflict detection are scarce, FLAME implements an algorithm that assigns higher priority to the conflict detection instances with chronologically newer versions of the model such that, when t is the longest time required to process a single detection instance with no delay, any *outstanding* higher-order design conflict that had not already been resolved at a given moment can be detected, at most, in the amount of time $2 \cdot t$ from its creation.

The insight behind the algorithm is that a higher-order design conflict can also be detected by performing conflict detection on a version of a model that is chronologically later than the version of the model in which the conflict first appeared, unless the conflict had already been resolved. A *Detection Engine* generates a model representation R_i for each modeling operation O_i it receives. A model representation R_{n+1} would then represent a version of the model that has merged all modeling operations that the version of the model that R_n represents had merged (i.e., O_1 through O_n) as well as an additional modeling operation, O_{n+1}. For any modeling operation O_i, a consistency rule violation that was caused by O_i and that has not been resolved by the time a chronologically newer modeling operation O_{i+j} (where $j > 0$) was introduced can be detected by performing conflict detection on R_{i+j}. For example, in the scenario depicted in Figure 3.4, a modeling operation O_3, together with a previous modeling operation, violates a consistency rule, causing a higher-order design conflict. While the versions of the model that R_1 and R_2 represent do not violate the consistency rule, the version that R_3 represents, after merging O_3, does violate the rule and causes the conflict. The versions of the model that have been generated after R_3 (i.e., R_4 and R_5), can be in one of the following two states:

1. **The model does *not* violate the consistency rule**: For example, the modeling operation that was applied after O_3 had been applied, O_4, counteracted to O_3, and the versions of the model that have merged O_4 on top of O_3 no longer violate the consistency rule. In this case, it would not be as urgent to notify the architects because the conflict has already been resolved.

2. **The model does violate the consistency rule**: Even after merging the modeling operations that have been applied after O_3 was applied, the consistency rule violation persists. In this case, performing conflict detection on R_4 or R_5 would detect the same consistency rule violation. Also, it will be necessary to notify the architects so that they become aware of this outstanding conflict.

Figure 3.5 depicts the behavior of the detector-side components that implement the algorithm. In case there is no available slave node at a given moment, a newly created model representation will have to wait until a slave node completes performing its on-going conflict detection and eventually becomes available. When a slave node becomes available, the *Slave Manager* retrieves a model representation R_{newest} that is chronologically the newest among the ones in its *Model Representation Queue* and has the slave node perform conflict detection on it. In this process, it will take, at most, the amount of time $2 \cdot t$ to detect any outstanding conflicts that exist in the version of the model that R_{newest} represents. That is because the wait time for R_{newest} until a slave node becomes available would be the amount of time t at most, and it would take another t at most for the slave machine to complete performing the conflict detection on R_{newest}.

It should be noted that this algorithm guarantees the upper bound in the number of conflict detection instances that need to be performed until a higher-order design conflict to be detected, but it does not guarantee the absolute amount of time that it may take for the detection. The amount of time t, defined as the longest time required to process a single detection instance with no delay, is the upper bound in chronological time for a slave node to complete processing an instance of conflict detection on a version of the model.

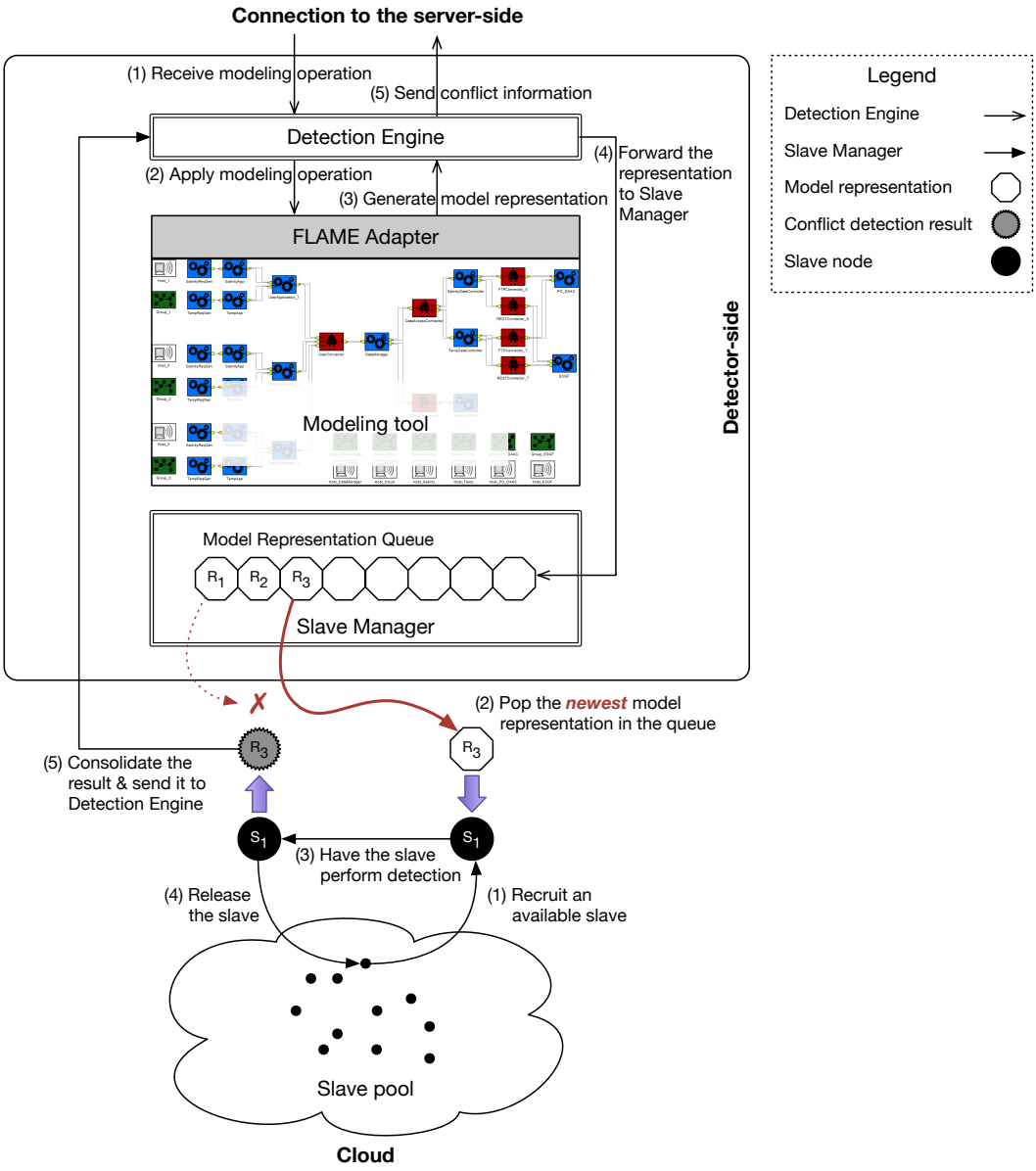


Figure 3.5: A model of detector-side that retrieves newest model representation first.

It is a variable that is affected by several factors such as (1) the size and complexity of the model fed into the conflict detection tool and (2) the amount of available computation resources on the slave node on which the conflict detection is processed.

This dissertation makes the assumption that the time duration t would not significantly fluctuate at a given point during a collaborative design session, hence having $2 \cdot t$ as the upper bound time for an outstanding higher-order design conflict to be detected will be reasonable in practice. The reason for this assumption is two-fold. First, two model representations generated by the same *Detection Engine* that are consecutive or close in their generation order are likely not to significantly differ in their size and complexity. A *Detection Engine* generates a model representation for each modeling operation it receives. The difference between two consecutive model representations is only a single additional modeling operation that the later representation has merged on top of the earlier one. Second, the available computation resources for conflict detection on the slave nodes are likely not to significantly differ. The cloud-based slave nodes can readily be initialized to have similar or even identical amounts of computation resources.

3.2 The Implementation of FLAME

This section introduces the implemented FLAME instance that has been developed in order to evaluate whether and to what extent proactive conflict detection may impact the cost of collaborative software design. The rest of the section is organized as follows. It begins with describing the primary components in the implementation and how those components interact with each other in Section 3.2.1. In Section 3.2.2, FLAME's unique

way of operation-based version control that enables proactive detection of design conflicts is described. Lastly, in Section 3.2.3, the two *Detection Engines* developed for this FLAME instance that derive different versions of a software model for proactive conflict detection are introduced and compared with one another.

3.2.1 Primary Components

Figure 3.6 depicts the detailed, as-implemented architecture of FLAME. The implemented FLAME instance integrates the following three off-the-shelf software tools,

1. **GME** [37]: A configurable tool for domain-specific software modeling,
2. **XTEAM** [23]: A model-driven design, analysis, and synthesis tool-chain, and
3. **Prism-MW** [41]: An event-based middleware platform.

FLAME integrates those off-the-shelf tools to provide proactive conflict detection to software architects. GME allows architects to create a domain-specific modeling notation (e.g., for the Next-Generation Climate Architecture in the scenario from Section 2.2) in which the architects can specify different aspects of the target system. An architect modeling in FLAME uses GME to specify the structure of the system by creating a set of components, connectors, and the connections between them. She then specifies, for each component and connector, (1) how it stores data, (2) how it behaves and reacts to different events, (3) on which physical host it is deployed, and (4) other characteristics in the form of property lists. Each modeling operation the architect makes along the way is captured by the *FLAME Adapter* that immediately transfers the operation, through the server-side FLAME components, to *Detection Engines*, via Prism-MW. Prism-MW establishes event-based interaction channels for the transfer of the operations in FLAME.

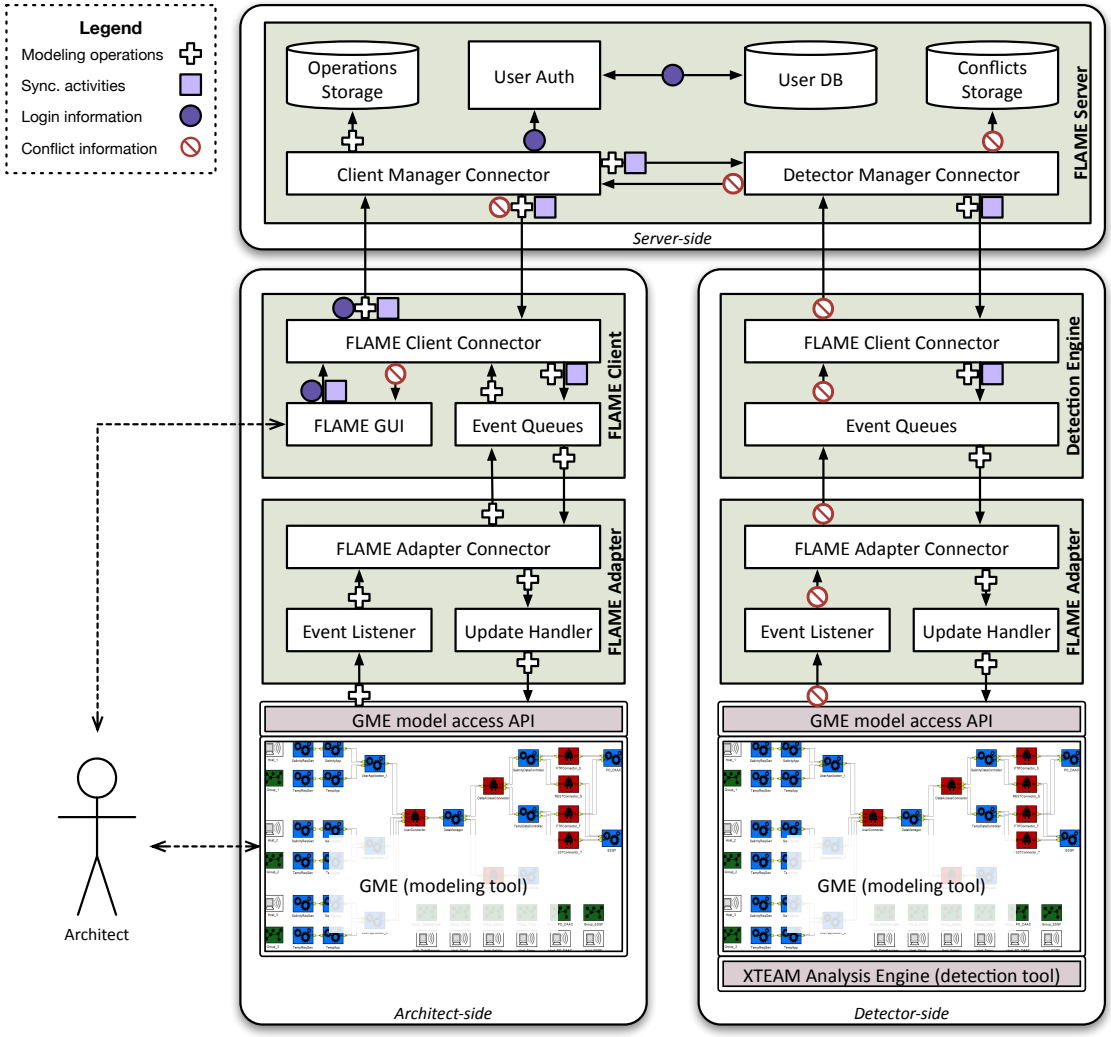
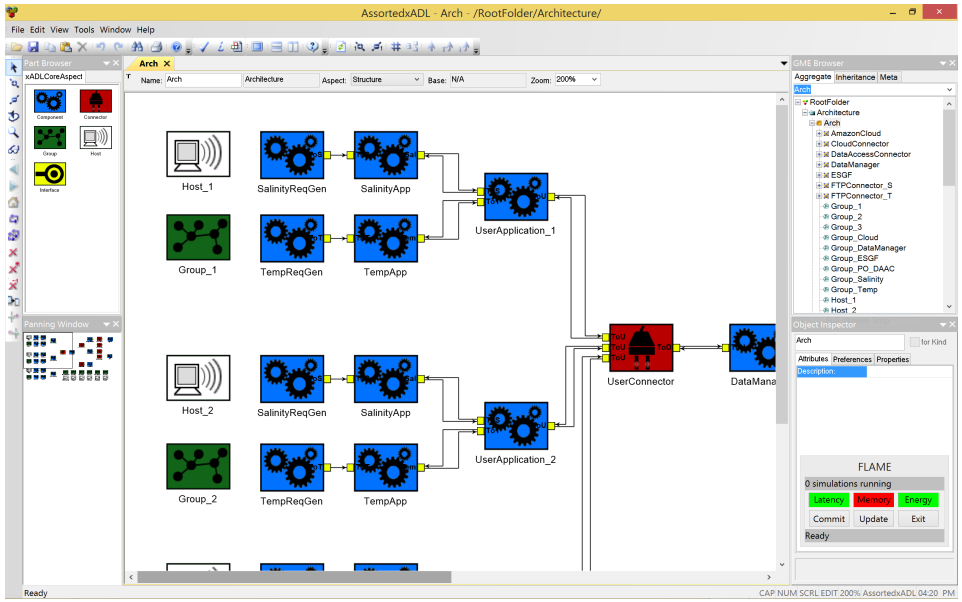


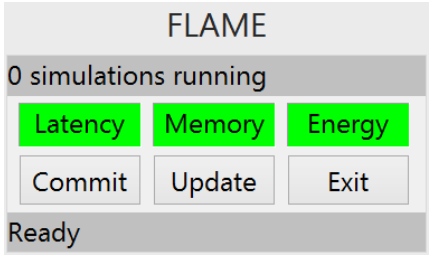
Figure 3.6: Detailed architecture of FLAME. The double-lined polygons are the off-the-shelf, domain-specific software systems integrated into FLAME.

When a *Detection Engine* receives an operation, it applies the operation to its local model and invokes XTEAM to analyze the model and estimate one or more runtime properties of the modeled system such as memory usage (as in [23]), energy consumption (as in [58]), and message latency (as in [67]). The model analysis result produced by XTEAM is raw and needs to be consolidated since it could distract the architects if provided as-is. For example, the memory usage estimation outputs a simple but large memory usage log for each component and connector of the target system during the runtime simulation execution. The *Detection Engine* responsible for the memory usage estimation consolidates the result by computing the statistics necessary to determine whether a consistency rule regarding the memory usage has been violated, and forwards it to *FLAME Clients*, where the result is processed further before being eventually presented to an architect, as described below.

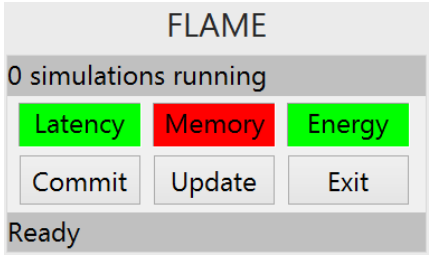
FLAME provides an extension point at which a customized GUI capable of presenting the domain-specific conflict information can be plugged-in. To be used with the current FLAME instances, as a proof-of-concept, a small GUI has been implemented in order to minimize the obtrusiveness while continuously delivering proactive conflict information to the architects. The GUI uses color coding to indicate the presence of a higher-order design conflict. For example, recalling the NGCA scenario from Section 2.2, the GUI in Figure 3.7 changes the color of the "Memory" indicator from green to red (darker highlighting in grayscale) if the estimation surpasses the stated threshold.



(a) An architect's screen with the FLAME GUI.



(b) All property requirements are satisfied.



(c) The memory requirement is *not* satisfied.

Figure 3.7: The simple FLAME GUI for NGCA.

3.2.2 Version Control in FLAME

FLAME synchronizes modeling operations that software architects perform in *real time*, i.e., as the operations are performed. This real time synchronization of operations is beneficial since it enables (1) proactively performing higher-order design conflict detection per-operation and also (2) integrating *Detection Engines* that derive various versions of the model without any modification made to the server-side of FLAME. The rest of this section describes how FLAME implements this operation-based version control.

When FLAME transfers modeling operations between its components, it uses *Design Events*, each of which encapsulates a single operation. A *Design Event*, in addition to the operation, carries the identification of the architect who performed the operation and a counter that represents how many prior *Design Events* had been created by the same architect. FLAME also treats version control activities (i.e., commit and update) in a similar way. It creates, for each version control activity performed, a *Design Event* that encapsulates the activity with the same two properties. Those *Design Events* are transferred from *FLAME Clients* to the *FLAME Server*, and when the *FLAME Server* receives the *Design Events*, it automatically broadcasts the *Design Events* to the rest of the *FLAME Clients* and *Detection Engines*. In this manner, all *FLAME Clients* and *Detection Engines* will eventually have the same set of *Design Events* locally.

FLAME implements version control by adjusting when the *FLAME Clients* and *Detection Engines* apply the received *Design Events* to their local models. When a *FLAME Client* or a *Detection Engine* receives a *Design Event*, the *Design Event* is not always immediately applied to the local model. Instead, the *Design Event* is first put into the

Event Queues (recall Figure 3.6) of the component. The *Event Queues* is a group of queues, each of which stores the list of *Design Events* from an architect. Later, the *Design Events* in the *Event Queues* are applied to the local models but in different ways depending on whether the component is a *FLAME Client* or a *Detection Engine*, and in case the component is a *Detection Engine*, depending on which version of the model the *Detection Engine* derives for proactive conflict detection. A *FLAME Client* applies the *Design Events* in the *Event Queues* to its local model when the architect interacting with the *FLAME Client* requests to perform an "update" activity. Upon the request, the *FLAME Client* retrieves, from each queue in the *Event Queues*, the list of *Design Events* that had been made prior to the latest "commit" *Design Event* in that queue. The *FLAME Client* subsequently applies the retrieved lists of *Design Events* to its local model. On the other hand, since *Detection Engines* do not interact with an architect like a *FLAME Client* does, each *Detection Engine* has its own *policy* that decides when and which *Design Events* to apply to its local model, as introduced in the following section.

3.2.3 Detection Engines

In order to evaluate FLAME, two *Detection Engines* that derive different versions of a software model have been developed. Because they perform proactive conflict detection on different versions of the model, software architects can gain different kinds of awareness from the conflict information each of those *Detection Engines* provides. The details of the two implemented *Detection Engines* are as follows:

- The *Global Engine* derives and performs proactive conflict detection on the version of the model that consists of all modeling operations that have been made up to a

given moment in time regardless whether the operations have been committed or not. It provides the awareness of what conflicts may arise if every architect performs a *commit* at that moment. Because the *Global Engine* merges all operations across architects, only a single instance of it is necessary for a team of software architects.

- The *Head-and-Local Engine*, for each architect, derives and performs conflict detection on the version of the model that is based on the head version and merges the modeling operations in her working copy that have not been committed up to a given moment. It provides the awareness of what conflicts may arise if the architect performs an *update* at that moment. Because every architect has her own working copy, an instance of *Head-and-Local Engine* for each software architect is necessary.

The implementations of the two *Detection Engines* do not differ significantly other than their policies in the application of *Design Events* to local models. Figure 3.8 shows the simplified policies of the *Detection Engines*. When a *Detection Engine* receives a *Design Event* via the channel that Prism-MW establishes, Prism-MW automatically invokes the handle() function and passes the incoming *Design Event* as the argument (Line 2). The *Detection Engine* subsequently puts (adds) the *Design Event* into the *Event Queues* (Line 4), and invokes an appropriate function that retrieves *Design Events* from the *Event Queues* depending on whether it is a *Global Engine* or a *Head-and-Local Engine*. The *Detection Engine* then applies the retrieved *Design Events* to its local model (Lines 9 and 13). In case the *Detection Engine* is a *Global Engine*, the *Design Event* retrieval function returns all *Design Events* in the *Event Queues* (Lines 19–27). In case the *Detection Engine* is a *Head-and-Local Engine*, the retrieval function returns (1) all *Design Events*

from the queue that stores *Design Events* from the architect to which the *Head-and-Local Engine* corresponds, and (2) from each of the rest of the queues, the *Design Events* that had been created prior to the latest "commit" *Design Event* in that queue (Lines 30–44). No additional modifications to the other parts of FLAME are necessary other than these *Design Events* application policies to implement the two *Detection Engines*, or any other potential *Detection Engines* that derive a different version of the model, because of the way FLAME synchronizes modeling operations in which all *Detection Engines* maintain the same set of modeling operations, as described in Section 3.2.2.

The *Global Engine* and *Head-and-Local Engine* introduce certain trade-offs. In general, the *Global Engine* detects conflicts earlier than the *Head-and-Local Engine* while the *Head-and-Local Engine* reports fewer benign conflicts (these can be thought of as false positives) than the *Global Engine*. Consider a scenario with two architects designing a system as a team. They each perform a modeling operation that jointly cause a higher-order design conflict. Those operations are then immediately transferred from the *FLAME Clients* to the *Detection Engine*. If the architects were using FLAME with the *Global Engine*, the conflict would be detected as soon as the operations are transferred to the *Global Engine*. On the other hand, if the *Head-and-Local Engines* were used, the conflict would not be detected until one of the operations is committed, and the conflict might be unknown to the architects for a longer period of time. Another possibility is that the conflict is benign, i.e., it has already been resolved before the operations that caused it were committed. In that case, if the *Global Engine* were used, it would report the conflict that would eventually be resolved even without the architects' attention, and

the design activity may unnecessarily be disturbed as a result. The conflict information from the two *Detection Engines* will, therefore, ultimately complement each other.

```
// handles an incoming Design Event
void handle (Event event) {
   // adds the Design Event to the Event Queues
   eventQueues.add(event);

   // applies the Design Events differently
   switch(engineMode) {
      case "Global":
         apply_events_to_local_model(retrieve_events_global());
         break;

      case "HeadAndLocal":
         apply_events_to_local_model(retrieve_events_headlocal());
         break;
   }
}

// Retrieves all Design Events, whether or not committed
EventArray retrieve_events_global() {
   EventArray eventsToApply;
   for(Queue queue : eventQueues) {
      for(Event event : queue) {
         eventsToApply.add(event);
      }
   }
   return sort_chronological(eventsToApply);
}

// Retrieves all committed Design Events and uncommitted Design
   Events from the corresponding architect
EventArray retrieve_events_headlocal() {
   EventArray eventsToApply;
   for(Queue queue : eventQueues) {
      if(queue.get_username().equals(correspondingUsername)) {
         for(Event event : queue) {
            eventsToApply.add(event);
         }
      } else {
         for(int i=0; i < queue.find_latest_commit(); i++) {
            eventsToApply.add(queue.get(i));
         }
      }
   }
   return sort_chronological(eventsToApply);
}
```

Figure 3.8: *Design Events* retrieval from *Event Queues* in *Detection Engines*.

Chapter 4

Evaluation

The *Framework for Logging and Analyzing Modeling Events*, FLAME, has been designed and developed according to the two hypotheses introduced in Section 1.3. This chapter presents the empirical and systematic evaluation of FLAME as a way of testing the two hypotheses. The rest of the chapter is organized as follows. In Section 4.1, the results from the two user studies that were conducted with total of 90 participants are presented. The focus of that section is to test Hypothesis 1, regarding whether and to what extent the proactive conflict detection that FLAME provides would impact the cost of collaborative software design. Then in Section 4.2, the results from the systematic evaluations of FLAME with regards to its scalability and performance are presented. In particular, Section 4.2 tests Hypothesis 2, which deals with prioritization of conflict detection instances in order to guarantee the worst-case time to detect an outstanding conflict when available computation resources are scarce (recall Section 3.1.3).

4.1 Empirical Evaluation

This section presents the designs, the executions, and the results of two user studies conducted using FLAME, with the primary goals to assess (1) how much earlier higher-order design conflicts can be detected and resolved by implementing proactive conflict detection as opposed to solutions that rely on the traditional on-demand merging of models, and (2) whether and to what extent providing proactive conflict detection impacts the quality of the software model. Each of the two studies evaluated one of the two *Detection Engines* introduced in Section 3.2.3, i.e., the *Global Engine* and the *Head-and-Local Engine*. The rest of this section is organized as follows. Section 4.1.1 details how the two user studies were designed and discusses the differences between the two. After that, Section 4.1.2 and Section 4.1.3 present the findings, insights, and implications from the *Global Engine* user study and the *Head-and-Local Engine* user study, respectively. Lastly, in Section 4.1.4, the known threats to validity of the user studies are discussed.

4.1.1 Study Setup

Sharing the same primary goals, the two user studies were designed in a similar way but with some differences (listed in Table 4.1). In both studies, the participants were divided into two groups, each of which performed collaborative design tasks with and without proactive conflict detection. Then the results from the groups were compared in order to determine in which ways providing proactive conflict detection affected the collaborative design. By using FLAME for both groups—those that did and those that did not present

Table 4.1: FLAME User Studies Comparison.

User Study	***Global***	***Head-and-Local***
Target system	NGCA (Section 2.2)	BOINC [63]
Number of participants	42	48
Number of teams	21 teams of 2	24 teams of 2
User study period	Span of 18 days	Span of 12 days
Detection Engine of choice	*Global Engine*	*Head-and-Local Engine*

the proactive conflict detection results—it was possible to track the participants' collaborative design activities at the level of each modeling operation as well as the higher-order conflicts from their creation, to detection, and eventually to resolution for both groups. To compare the collaborative design cost with and without proactive conflict detection, in both studies, the cost was estimated by controlling the extent of time the architects spent performing collaborative design activities and measuring the resulting model's quality upon the completion of the design task. Granular variables were derived to observe participants' collaborative design behavior and to measure the resulting models' quality as shown in Table 4.2 and Table 4.3, which will be discussed in detail later in this chapter.

Both of the user studies were designed based on the design documents of real software systems. We selected open-source systems that have their design documents open to public. Based on those design documents, collaborative design scenarios where higher-order conflicts arise were recreated. The target system chosen for the *Global Engine* user study was the Next-Generation Climate Architecture (NGCA) that had the three essential, runtime system properties, i.e., memory usage, message latency, and energy consumption (recall Section 2.2). For the *Head-and-Local Engine* user study, the Berkeley

Open Infrastructure for Network Computing (BOINC, depicted in Figure 4.1) [63], an open-source system for volunteer-computing and grid-computing, was chosen as the target system. A comparable design scenario with the same three runtime system properties was created using BOINC, which in turn, enabled the use of the same model analysis tool (i.e., XTEAM [23]) as the conflict detection tool in both user studies.

The participants in the *Global Engine* and *Head-and-Local Engine* user studies were 42 and 48 students respectively, enrolled in the graduate-level Software Architecture class at the University of Southern California. No participant had prior experience with FLAME or their target systems (NGCA or BOINC). The participants spent four weeks performing two software design assignments using the NGCA- or BOINC-specific modeling environment and FLAME's constituent XTEAM subsystem in order to get familiar with its simulation-based analyses prior to the user studies. This resulted in comparable familiarity of each participant with the modeling environment and the target system domain [20].

The participants were grouped into 21 and 24 teams of two in the two respective studies. The teams were then divided into two groups by their team numbers (odd and even number teams; team numbers were randomly assigned): (1) the control group that used FLAME in the mode that does not present the proactive conflict detection results to simulate the behavior of a current generation software model VCS ("w/o PCD" in Table 4.2 and Table 4.3) and (2) the experimental group that used FLAME in the mode that does present the proactive conflict detection results ("w/ PCD" in Table 4.2 and Table 4.3). In each of the user studies, a survey was conducted regarding the participants' industry experience to assess their prior exposure to collaborative design, but no evidence was found that the prior industry experience of the two groups differed significantly.

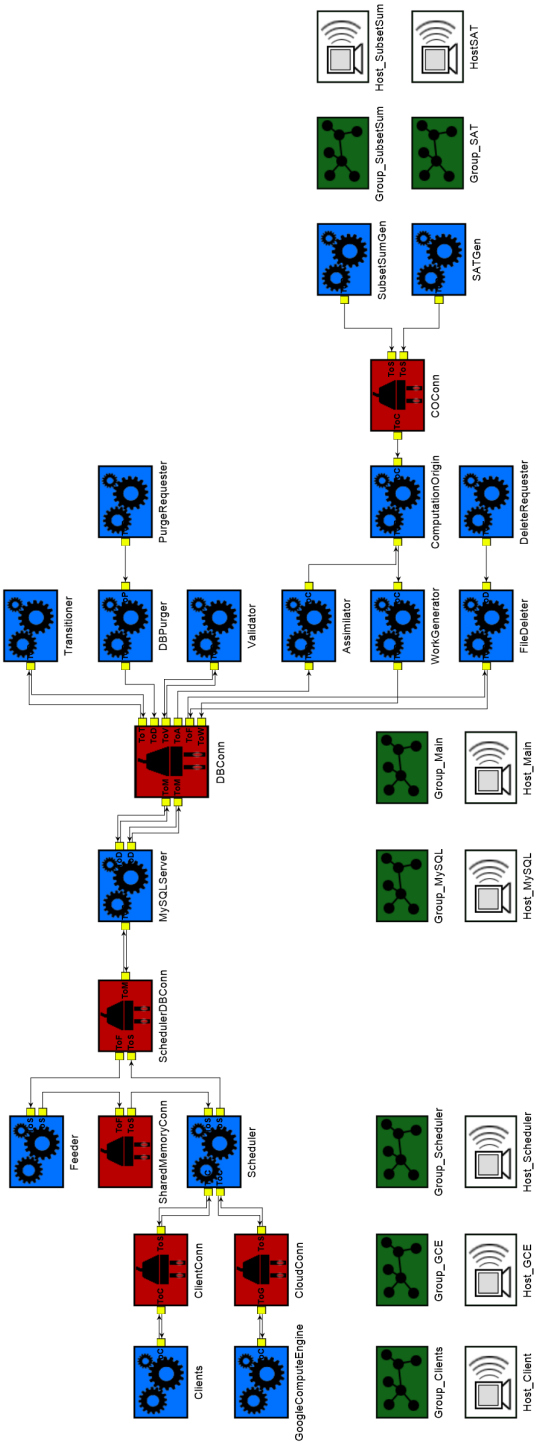
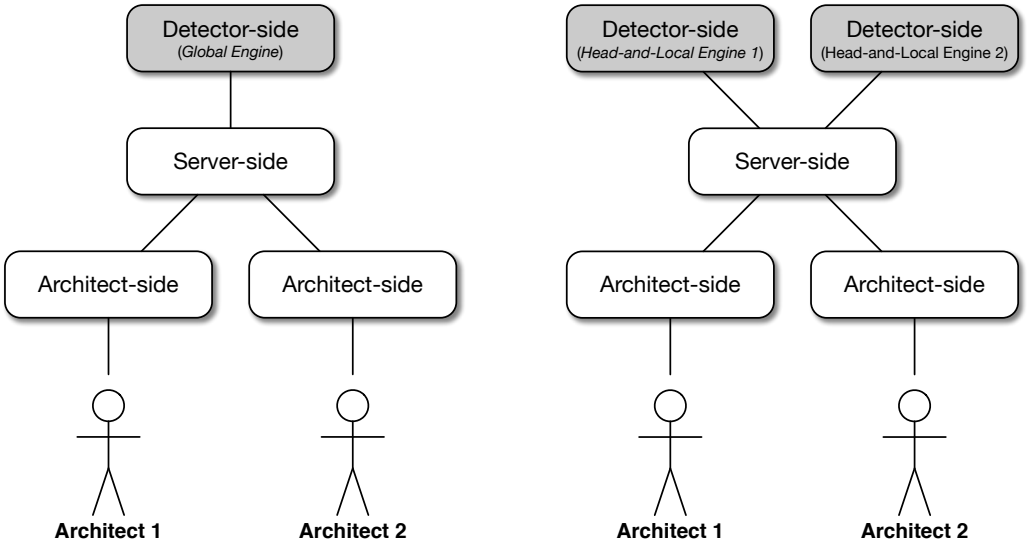


Figure 4.1: A high-level model of BOINC.

Each team participated in a 2-hour-long session during a span of 18 days and 12 days in the *Global Engine* and *Head-and-Local Engine* user studies, respectively. The author of this dissertation administered all of the sessions (21 and 24 sessions in the two studies, respectively) during which he was physically present close to the participants, monitoring and observing them. Each session was divided into three smaller sessions: (1) the 1-hour-long FLAME tutorial, (2) the main, 30-minute-long design session during which the participants' design activities were recorded, and (3) the subsequent 30-minute-long design session for the participants to experience the alternative mode of FLAME. For example, if a participant used FLAME in the mode that presents the proactive conflict detection results in her main design session, she would use FLAME in the mode that does not present the results in her alternative-mode session, and vice versa. The rationale behind conducting the two sessions was for the participants to experience both modes of FLAME, with and without proactive conflict detection, and to collect their preferences and assessment of the differences they experienced between the two modes. The design activity during the alternative-mode session was not recorded since the participants' behavior in that session may have been influenced by having undergone the main session.

During the design sessions, each team was given a partially complete model of the target system—NGCA or BOINC—and assigned with design tasks to replace a set of components in the model, as well as a set of three system requirements regarding the three runtime system properties to satisfy (design task samples are in Appendix A). The two participants in each team were directed to make trade-off design decisions corresponding to two different non-overlapping parts of the model in order to avoid synchronization conflicts. The given tasks were designed in a way that the participants, in the course of



(a) The *Global Engine* user study. (b) The *Head-and-Local Engine* user study.

Figure 4.2: Detector-side configurations in the two user studies.

decision making, could violate two of the system requirements: energy consumption and memory usage. The third requirement, message latency, was not designed to be violated.

The biggest difference between the two user studies was that the participants used different *Detection Engines* in each study, as depicted in Figure 4.2. Recalling Section 3.2.3, when the *Global Engine* is to be used in FLAME, only one instance of the *Global Engine* is necessary that maintains the version of the model that consists of modeling operations from all architects. On the other hand, when the *Head-and-Local Engine* is to be used, one instance of the *Head-and-Local Engine* for each architect is necessary that maintains the version of the model that is based on the head version and merges the local changes from that architect. In the *Head-and-Local Engine* user study, each team used FLAME with two instances of *Head-and-Local Engines*, one per architect in the team.

Also, in both studies, the communication between the two participants in each team was restricted to online communication media during the sessions to reproduce the communication challenges of geographically distributed collaborative software design. For example, the two participants were not allowed to speak with each other but had to initiate an email thread or use an instant messenger in order to discuss their conflict resolution strategy.

4.1.2 The Global Engine User Study Result

This section presents the analyses on the *Global Engine* user study data by addressing three research questions derived from the primary goals defined earlier in this chapter.

Q1: Did FLAME with *Global Engine* affect the amount of time architects spend in design activities? The top portion of Table 4.2 shows the frequency of design activities (DV01-DV03) performed during the main design sessions. During the same length of time (CV01; 30 minutes), the frequency of design activities differed significantly between the two groups of participants. Specifically, the group with proactive conflict detection (1) performed a higher number of modeling operations (t-test; p-value of 0.078) and (2) communicated more frequently (t-test; p-value of 0.064), and while not significantly, (3) performed synchronization activities more often. The increase in the number of performed operations can be explained by the increase in the confidence of participants in making new changes. Fear of conflicts [14] could make an architect take additional care when she performs new operations. The following quote is from a participant, which aligns with this reasoning: *"our confidence that the combined design would meet the requirements was much higher when using proactive conflict detection."*

The increase in communication frequency aligns with a previous empirical study in which a similar phenomenon was observed for detection and resolution of higher-order conflicts in collaborative software implementation [53]. The group of participants with proactive conflict detection was immediately notified by FLAME as each new conflict arose, hence it was natural for them to initiate communication between the team members more often. One of the participants responded about how the increased amount of communication helped her team by saying: *"I prefer the proactive environment because my teammate and I completed the task on time as we had enough communication."*

Q2: Did FLAME with *Global Engine* facilitate the way architects detect and resolve higher-order design conflicts? The middle portion of Table 4.2 shows the number of conflicts that occurred and how long they lasted during the design sessions (DV04-DV07). While the group of participants with proactive conflict detection dealt with a higher number of conflicts on average, no conflict was left at the last commit nor at the end of session (DV05-DV06), and the average lifetime of the conflicts, from when they are introduced to when they are resolved, was significantly shorter (DV07). These results also corroborate those reported in the previous research conducted at the code-level [32, 53]. The following quotes from the participants further explain our observation: *"It was quicker and easier to detect conflicts and fix them immediately."* and *"[FLAME was] making it easier to identify errors and fix them before further changes are made."*

Some conflicts took longer than the others to detect and resolve, and the group of participants with proactive conflict detection resolved those *hard* conflicts significantly earlier than the group without. Some conflicts may be easier to resolve, possibly because they are already known to the architects before they are explicitly detected. Indeed, the

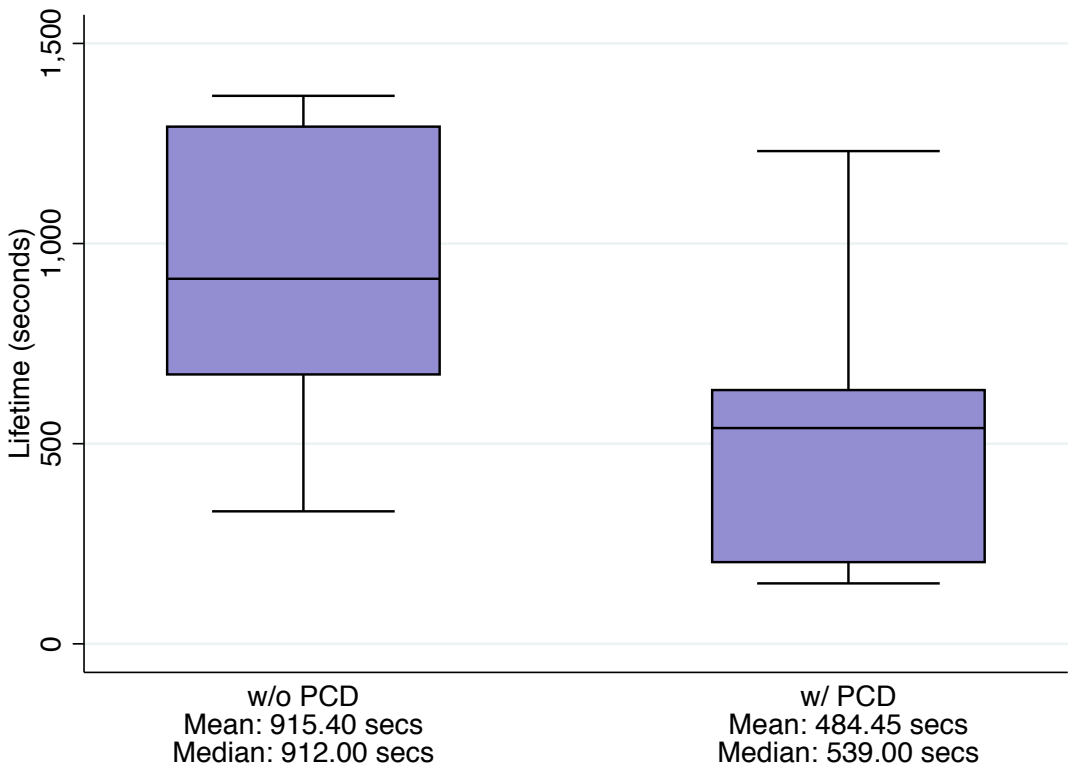


Figure 4.3: Lifetime of higher-order conflicts in the *Global Engine* user study.

lifetimes of 27% of the conflicts were shorter than 100 seconds while the mean lifetime of the rest of the conflicts was much higher at 619.13 seconds. Figure 4.3 presents box plots of the lifetimes of the conflicts that took longer than 100 seconds to resolve, with and without proactive conflict detection (w/o PCD and w/ PCD respectively in the figure). The median in the w/o PCD mode, 912 seconds, was significantly higher than the median in the w/ PCD mode, 539 seconds (Mann-Whitney-Wilcoxon test; p-value of 0.036).

Q3: Did FLAME with *Global Engine* affect the quality of the resulting model? The bottom portion of Table 4.2 shows the two factors (DV08-DV09) that was used to estimate the quality of the resulting model in addition to the number of unresolved conflicts (DV05-DV06). The participants were assigned with design tasks to modify the

system in a way that would maximize the *throughput* of the system, which subsequently results in higher energy consumption and memory usage. The variations of those two factors from the beginning to the end of each collaborative design session were tracked, and the maximum values of the factors that each team reached during the session were recorded. It was observed that the group of participants with proactive conflict detection was able to design, on average, NGCA systems with higher throughput while leaving fewer unresolved conflicts than the group without proactive conflict detection, in the same amount of time (CV01). The increased number of operations (as shown in Q1) can be seen as evidence of participants' higher productivity. The following is a quote from a participant that can show the link between the number of operations and the participants' productivity: *"[FLAME] increased productivity as we were able to try more combinations [of modeling operations] in same amount of time."* The reduced effort in higher-order conflict resolution (as shown in Q2) could also have contributed in the higher productivity. The following quotes from the participants corroborate this conclusion: *"... proactive conflict detection drastically minimizes the integration effort."* and *"[FLAME] shows my partner and me any conflicts that we have without running the simulation as much as we did with the one without proactive [conflict detection]."* In fact, the teams in this user study that performed a higher number of modeling operations during their sessions did achieve higher throughput of the system (univariate linear regression; energy consumption, p-value of 0.060, R^2 of 0.174 / memory usage, p-value of 0.088, R^2 of 0.146).

Table 4.2: Global Engine User Study: Variables.

Collaborative Design Activities and Conflicts		w/o PCD	w/ PCD
Q1	CV01: Duration (mins) of modeling session per team	30.00	30.00
	DV01: Number of modeling operations made per team	48.18	60.80
	DV02: Number of communication activities per team	11.00	19.50
	DV03: Number of synchronizations per team	6.18	8.00
Q2	DV04: Detected conflicts at synchronizations per team	1.27	2.40
	DV05: Teams with unresolved conflicts at last commit	3 of 11	0 of 10
	DV06: Teams with unresolved conflicts at session end	3 of 11	0 of 10
	DV07: Lifetime (secs) of a higher-order conflict	671.00	363.40
Resulting Model Quality; the Higher is Better		w/o PCD	w/ PCD
Q3	DV08: Throughput factor: energy consumption (J)	8.18 M	8.55 M
	DV09: Throughput factor: memory usage (MB)	729.09	747.60

CV is a control variable, and DV is a dependent variable. PCD stands for proactive conflict detection. All values were rounded off at the third decimal.

4.1.3 The Head-and-Local Engine User Study Result

This section presents the *Head-and-Local Engine* user study data by addressing the same three research questions derived and addressed for the *Global Engine* user study data.

Q1: Did FLAME with *Head-and-Local Engines* affect the amount of time architects spend in design activities? The top portion of Table 4.3 shows the frequency of design activities (DV01-DV03) performed during the main design sessions. While different *Detection Engines* were used, the result generally corroborated that from the *Global Engine* user study. The group of participants with proactive conflict detection (1) performed higher numbers of modeling operations (t-test; p-value of 0.031), and

while not significantly, (2) communicated more frequently, and (3) performed synchronization activities more often than the group without proactive conflict detection, in the same amount of time. The increase in the number of operations performed can be explained with the increase in the participants' confidence in performing the operations, as previously discussed in the *Global Engine* user study. The following quotes from the participants support that reasoning: *"[FLAME provided] the ease of committing the model without much effort as I was assured that the color flags would show if my model, on update, was not going to violate any of the system requirements."* and *"[Proactive conflict detection] helps me focus more on the design and modeling rather than worrying about the effect of a single change."* The following quote from another participant further explains the fear of conflicts [14] (recall Section 3.2.3): *"In the FLAME without proactive conflict detection mode, deciding when to commit local changes to the VCS was a challenge. In that mode, we were unaware of whether our changes made any conflict with the other teammates' change. So none of us wanted to commit a change which would cause the merged model to violate requirements."*

Interestingly, the difference in the amount of communication (DV02) between the groups of participants with and without proactive conflict detection was not as dramatic as that of the *Global Engine* user study. This might be a result of the differences between the two *Detection Engines* that provide different kinds of conflict awareness, and further investigation will be required in order to understand the root cause of this phenomenon.

Q2: Did FLAME with *Head-and-Local Engines* facilitate the way architects detect and resolve the higher-order design conflicts? The middle portion of Table 4.3 shows the observed variables related to conflicts (DV04-DV08). A noticeable

observation was that the group of participants with proactive conflict detection resolved conflicts without synchronizing them (DV04-DV05) more often than the group without proactive conflict detection. This is likely due to the kind of conflict awareness that the *Head-and-Local Engine* provides. An architect who uses a *Head-and-Local Engine* is provided with the awareness of the outstanding conflicts in the version of model that is based on the head version and merges the uncommitted modeling operations in her working copy (recall Section 3.2.3). That means the architect can foresee whether her working copy will have a certain conflict if she performs an update at a given moment. In case there is a potential conflict, with that awareness provided, the architect may choose to resolve the conflict even without performing an update. Another observation was that, similar to that observed in the *Global Engine* user study, the group of participants with proactive conflict detection (1) left fewer unresolved conflicts at session ends (DV06-DV07) and (2) detected and resolved conflicts earlier (DV08). The following two quotes from the participants further explain this observation: *"With help of proactive conflict detection, there will still be a need to revert [to resolve a conflict], but that will be to revert only that change which caused higher order conflict."* and *"We can see whether an individual's atomic step can produce a high-order conflict and quickly rollback that operation."*

Also, the average lifetime of the *hard* conflicts, which lasted for longer than 100 seconds (recall Section 4.1.2), of the group of participants with proactive conflict detection was significantly shorter, similar to the observation in the *Global Engine* user study. In the *Head-and-Local Engine* user study, about 47% of the conflicts took less than 100 seconds to resolve while the rest of the conflicts had the average lifetime of 348.56 seconds. Figure 4.4 presents box plots of the hard conflicts' lifetimes, without and with proactive

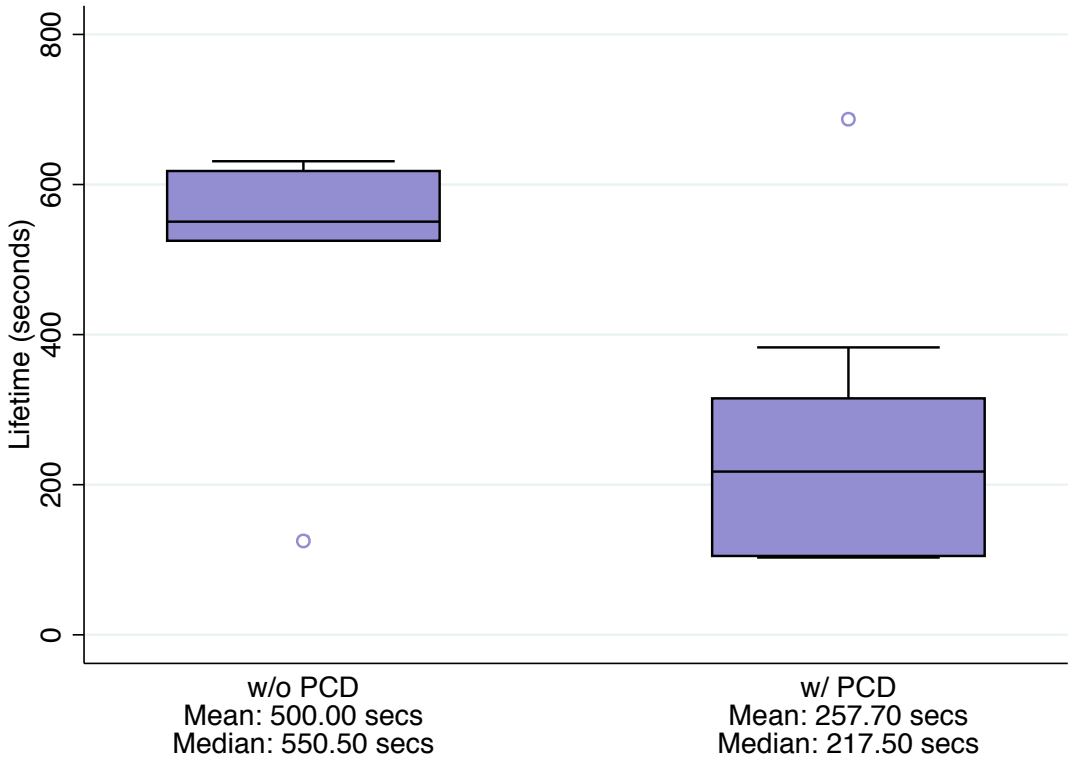


Figure 4.4: Lifetime of higher-order conflicts in the *Head-and-Local Engine* user study.

conflict detection (w/o PCD and w/ PCD respectively in the figure). The median in the w/o PCD mode, 550.50 seconds, was significantly higher than the median in the w/ PCD mode, 217.50 seconds (Mann-Whitney-Wilcoxon test; p-value of 0.051).

Q3: Did FLAME with *Head-and-Local Engines* affect the quality of the resulting model? The bottom portion of Table 4.3 shows the factor that was used to estimate the quality of the resulting model (DV09) in addition to the number of unresolved conflicts (DV06-DV07). The target system used in this user study, BOINC, is a system that divides a large computation task into smaller subcomputations and performs them remotely. The participants of this user study were assigned with design tasks to modify the BOINC system model in a way that maximizes the number of the subcomputations

of identical complexity that BOINC can perform in the same amount of execution time, which can be considered as achieving higher system performance. The result showed that the group of participants with proactive conflict detection, in the same amount of design time (CV01), was able to design better performing BOINC systems on average (DV09) while leaving fewer unresolved conflicts in the models (DV06-DV07). This result corroborates that of the *Global Engine* user study and has likely been affected by the improved productivity of the participants since (1) the participants were able to focus more on design rather than caring about conflict detection (as shown in Q1) and (2) the conflict detection became easier as the conflicts were detected and resolved earlier (as shown in Q2). The following response from one of the participants aligns with this conclusion: *"By comparison, [in the mode with proactive conflict detection] we have tried more combinations of the options (modeling operations) than in the one without proactive conflicts detection, and I think it means that our working efficiency has been improved."* The teams in this user study that performed a higher number of modeling operations during their sessions indeed achieved higher performance of the target system (univariate linear regression; subcomputations completed, p-value of 0.038, R^2 of 0.181).

The participants also responded that they preferred their experience in the mode with proactive conflict detection and that the way FLAME presented the conflict information was not distracting, in the post-session survey with five survey questions (SQ01-SQ05). Table 4.4 shows the result of the survey, which was conducted only for the *Head-and-Local Engine* user study. It is essential for a proactive conflict detection tool that its user interface does not distract the on-going design activity or overwhelm the architects with large amount of information. That is because it is possible that the conflict information

Table 4.3: Head-and-Local Engine User Study: Variables.

	Collaborative Design Activities and Conflicts	w/o PCD	w/ PCD
Q1	CV01: Duration (mins) of modeling session per team	30.00	30.00
	DV01: Number of modeling operations made per team	29.58	38.50
	DV02: Number of communication activities per team	14.29	15.04
	DV03: Number of synchronizations per team	5.33	7.25
Q2	DV04: Detected conflicts at synchronizations per team	2.50	0.92
	DV05: Proportion of conflicts never "updated" to local	32%	56%
	DV06: Proportion of unresolved conflicts at session end	52%	28%
	DV07: Teams with unresolved conflicts at session end	4 of 12	2 of 12
	DV08: Lifetime (secs) of a higher-order conflict	255.83	149.72
	Resulting Model Quality; the Higher is Better	w/o PCD	w/ PCD
Q3	DV09: Performance factor: subcomputations completed	598.17	605.92

CV is a control variable, and DV is a dependent variable. PCD stands for proactive conflict detection. All values were rounded off at the third decimal.

presented can be ignored by the architects in case the presentation is regarded as a distraction, which hinders the benefits of proactive conflict detection.

In both of the user studies, the participants with proactive conflict detection were able to produce higher quality models in the same amount of time, while the participants in each study were provided with different kinds of conflict awareness by different *Detection Engines*. This suggests that there may be other *Detection Engines* than the two *Detection Engines* introduced in this dissertation that maintain a different version of the model and that can also contribute in detecting and resolving higher-order design conflicts.

Table 4.4: Head-and-Local Engine User Study: Post-Session Survey.

Question	Mean	S.D.
SQ01: I preferred the FLAME mode with proactive conflict detection more than without.	6.22	1.25
SQ02: FLAME's proactive conflict detection helped me dealing with design conflicts.	6.15	1.03
SQ03: It was difficult to understand the conflict detection information that FLAME provided.	2.80	1.60
SQ04: Early detection of conflicts made the resolution easier.	6.07	0.88
SQ05: The FLAME GUI was distracting.	2.48	1.44

All values are on a 7-point Likert scale, where 1 is "strongly disagree" and 7 is "strongly agree". The values have been rounded off at the third decimal. S.D. denotes standard deviation.

4.1.4 Threats to Validity

As is commonly the case with controlled experiments, the two user studies have threats to validity due to their design. First, the user studies were conducted with students. While all students were graduate-level, their design behavior may not be identical to that of a real-world practitioner. Second, the design tasks assigned to the participants were not from actual projects. Instead, in order to recreate realistic collaborative design scenarios, the tasks were based on the actual design documents of the target systems (NGCA and BOINC). Third, the duration of observed design session per team was short (30 minutes), which could have caused bias from low familiarity with the target system domain or the model analysis framework (XTEAM). The bias was minimized by having the participants perform multiple design assignments using the target system model and XTEAM over the span of four weeks prior to their sessions. We also note that the higher-order conflicts may persist even longer in design sessions of longer duration, which would only increase the

benefit of detecting them early. Last, the team size was small (two per team). In a bigger team, it may become ambiguous which architects are directly involved in a higher-order design conflict. This is a hard problem in general, and has not yet fully been answered.

Some aspects of the study execution were challenged by threats to validity. First, the amount of time it takes a conflict detection tool to complete its consistency checking was not varied, while in a real setting, it may influence the architects' reaction to proactive conflict detection. In the two studies, that conflict detection times were kept relatively constant (38 and 37 seconds on average in the *Global Engine* and the *Head-and-Local Engine* user studies respectively) in order to avoid introducing bias. Second, only a single kind of analysis tools (i.e., XTEAM) was used in the user studies. In a real-world setting, architects may work in a design environment using several kinds of consistency checking tools. We tried to recreate a more realistic design environment by integrating three XTEAM model analysis tools in FLAME.

4.2 Systematic Evaluation

This section presents the designs, the executions, and the results of the analytical evaluations of FLAME with regards to (1) its scalability and performance and (2) its algorithm that prioritizes conflict detection instances in order to minimize the delay that may occur when the computation resources used for conflict detection are scarce. The rest of the section is organized as follows. Section 4.2.1 reports the results of the two scalability- and performance-related experiments conducted, each of which focuses on (1) the variations in the times required to process conflict detection instances when a *Detection Engine*

has different numbers of slave nodes at its disposal and (2) the overhead of FLAME in handling highly frequent conflict detection instances in a collaborative software design scenario with more than two architects. Then, in Section 4.2.2, we present the results of the experiment that evaluates FLAME's conflict detection prioritization algorithm by testing whether it guarantees delivering conflict information in the amount of time that is at most $2 \cdot t$ when the available computation resources for conflict detection are limited.

4.2.1 Scalability and Performance

Previously, Section 3.1.2 presented how a *Detection Engine* elastically adapts to the varying needs for computation resources for proactive conflict detection by offloading the detection to remote, slave nodes. FLAME's *Detection Engine*, regardless of the version of the model it maintains, is designed to be able to utilize slave nodes to reduce the potential delay in processing conflict detection instances that may occur when there are an overwhelming number of simultaneous instances to process. In order to evaluate whether the delay actually reduces as the number of slave nodes increases, an experiment was conducted that varies the number of slave nodes that a *Detection Engine* has at its disposal and measures the amount time it takes to process a conflict detection instance.

In this experiment, the participants' collaborative software design behavior that was recorded during the *Head-and-Local Engine* user study—the modeling operation logs of the 24 teams—was reused in order to collect more realistic data. For the purpose of measuring the *optimal* conflict detection time with no delay, the time gaps between each pair of modeling operations in those 24 logs were stretched to a length that would not cause any delay, i.e., a length that guaranteed no new conflict detection instance would be

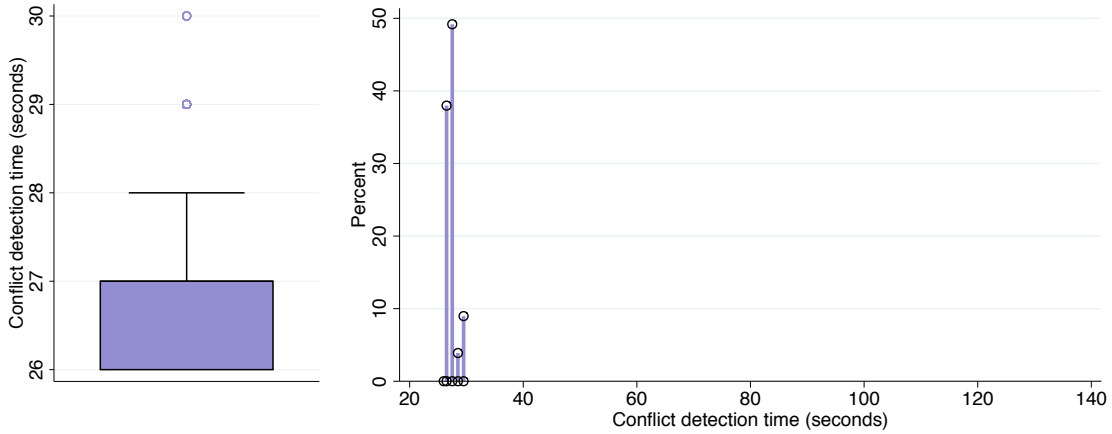


Figure 4.5: Box plot and histogram of conflict detection time of the no-delay scenario. Mean: 26.83 seconds. Median: 27 seconds. Maximum: 30 seconds.

initiated while another instance is being processed. The modified logs were then replayed in (i.e., fed into) FLAME, configured with a *Global Engine* with one slave node. By doing so, it was possible to measure the time it would have taken to process each of the conflict detection instances from the 24 logs with no delay (depicted in Figure 4.5).

The "sufficient" number of slave nodes necessary to prevent causing delay in conflict detection, n, was estimated by performing a retrospective analysis on the operation logs from the *Head-and-Local Engine* user study. Given t as the longest conflict detection time with no delay (e.g., 30 seconds in the *Head-and-Local Engine* user study; refer to Figure 4.5), the number n can be determined by calculating the maximum number of operations performed during the time spans of length t, each of which starts from when its corresponding operation is performed. Figure 4.6 depicts an example scenario with 7 operations. In this scenario, t_4 is the time span during which the maximum number of operations is performed and also the largest number of slave nodes is necessary. The estimated n for the *Head-and-Local Engine* user study by performing this analysis was

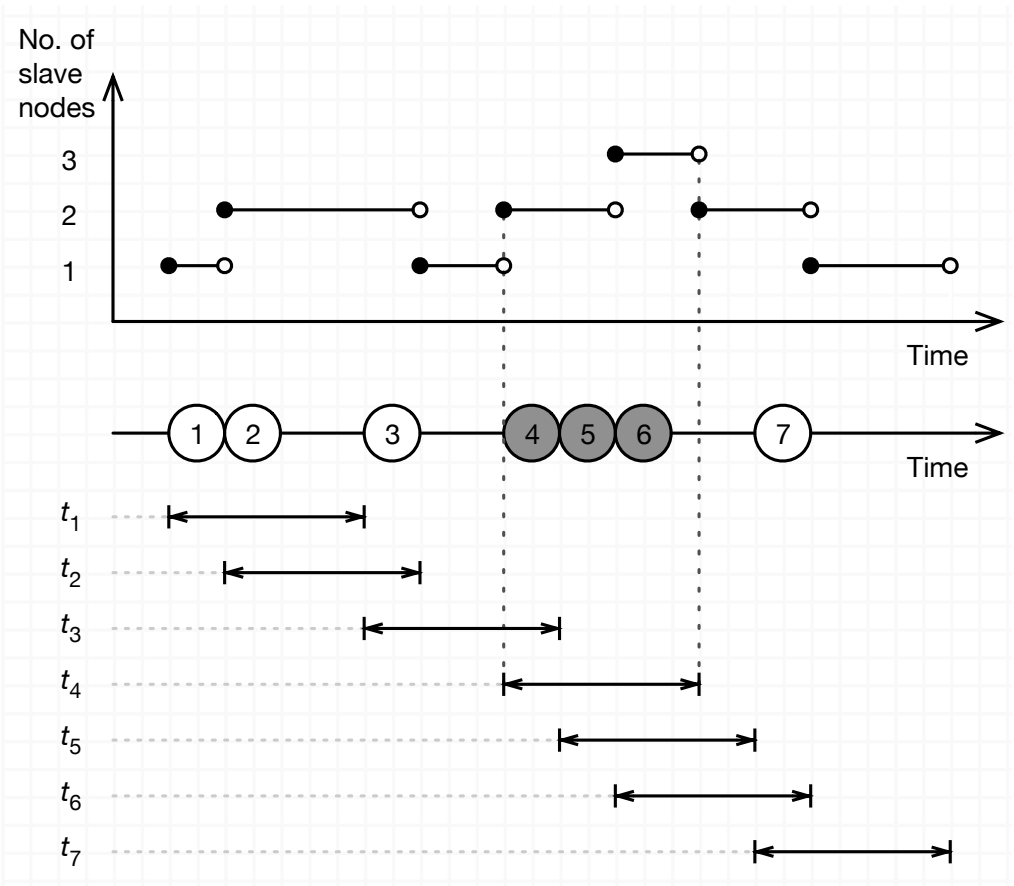


Figure 4.6: Estimating the "sufficient" number of slave nodes. Each circle on the time line represents a modeling operation performed at that time. In this scenario, at least three slave nodes would be necessary not to cause delay in conflict detection, during t_4.

9. Note that this analysis is likely to overestimate n when the longest conflict detection time is used for t. The estimated n was 8 when the median detection time (27 seconds; refer to Figure 4.5) was used for t instead.

After having measured the optimal conflict detection time and estimated the sufficient number of slave nodes, four configurations of FLAME with varying number of slave nodes were created, and the resulting conflict detection times were compared with each other. FLAME was configured to have a *Global Engine* with 2, 4, 8, or 12 slave nodes each, instantiated with identical computation resources on Google Compute Engine [30]. In

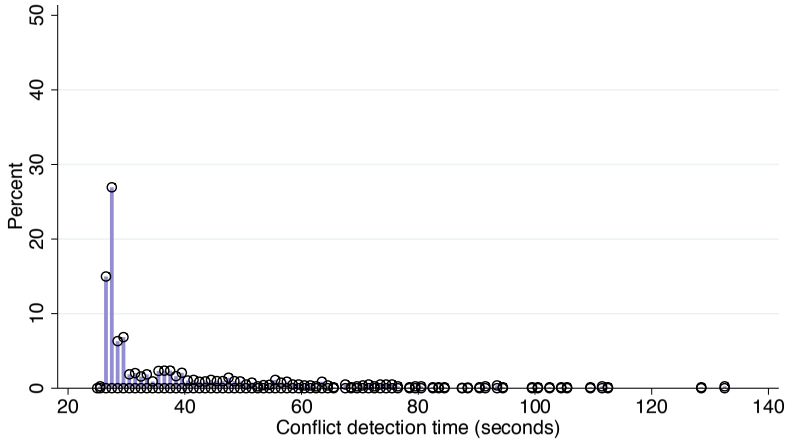
each of the configurations, the original modeling operation logs of the 24 teams in the *Head-and-Local Engine* user study were replayed. That enabled computing the actual time that would have been needed to process each conflict detection instance if those teams used FLAME with the *Global Engine* that utilizes the given number of slave nodes.

Figure 4.7 presents the experiment results showing that the maximum conflict detection time noticeably drops as the number of slave nodes involved in the detection increases from 2 (133 seconds) to 4 (54 seconds), and then to 8 (32 seconds). However, the conflict detection time did not noticeably change in the 12 slave node configuration compared to the 8 slave node configuration. It is important to note that, by parallelizing processing conflict detection instances, FLAME only minimizes the delay in conflict detection, i.e., the amount of time a conflict detection instance may have to wait in the *Detection Engine* until a slave node becomes available to process the instance. FLAME does not reduce the execution time of the consistency checking tool it invokes to detect higher-order conflicts. That means involving a larger number of slave nodes in conflict detection will decrease the conflict detection time only as long as there are delays in conflict detection. In other words, involving more than a "sufficient" number of slave nodes would not further decrease the conflict detection time. The maximum conflict detection time in the 8 slave node configuration (32 seconds) was already close to the optimal time (30 seconds, recall Figure 4.5), so adding more nodes (from 8 to 12) did not further decrease that time. Figure 4.7 confirms this reasoning. While Figure 4.7a and Figure 4.7b have long tails due to the delayed conflict detection instances, Figure 4.7c and Figure 4.7d do not have the tail. In order to fully realize the benefits of proactive conflict detection, it is essential for the *Detection Engine* to have a sufficient number of slave nodes as its disposal to keep

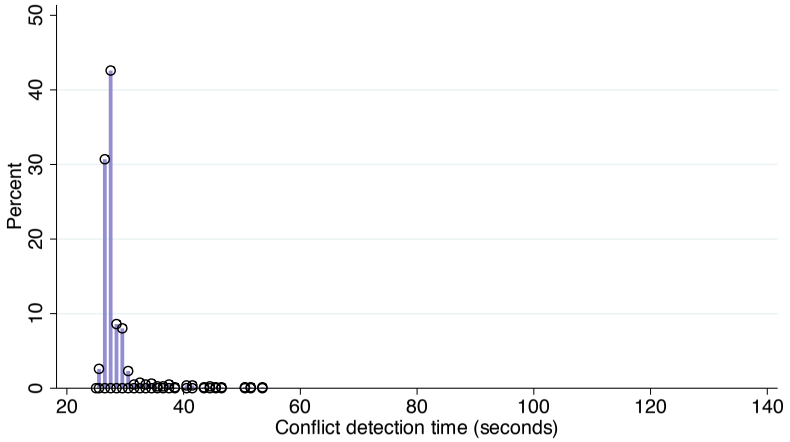
the maximum conflict detection time low. However, since it is hard to foresee the rate at which the architects may perform modeling operations at a given moment, ultimately, the capability to automatically adjust the number of slave nodes on-the-fly is desirable.

Additionally, in order to evaluate whether FLAME can handle collaborative software design scenarios with more than two architects, an experiment was conducted to measure the potential overhead in processing conflict detection instances that may occur when a *Detection Engine* processes a large number of simultaneous instances. As the size of the architect team grows, it is likely that the number of modeling operations performed by the team of architects in the same amount of time would also increase. In other words, the rate at which the team performs modeling operations would increase. Since a *Detection Engine* generates an instance of conflict detection for each modeling operation it applies to the local model it maintains, that higher rate would result in an added number of conflict detection instances for the *Detection Engine* needs to process in the same amount of time. In that case, involving a higher number of slave nodes would be desirable to lessen the delay in conflict detection that may occur. However, by having to manage a higher number of slave nodes to process the increased number of conflict detection instances, the *Detection Engine* may cause an additional amount of overhead in processing those detection instances. Such overhead would lead to delay in conflict detection and result in longer conflict detection times, hindering the benefits of proactively detecting conflicts.

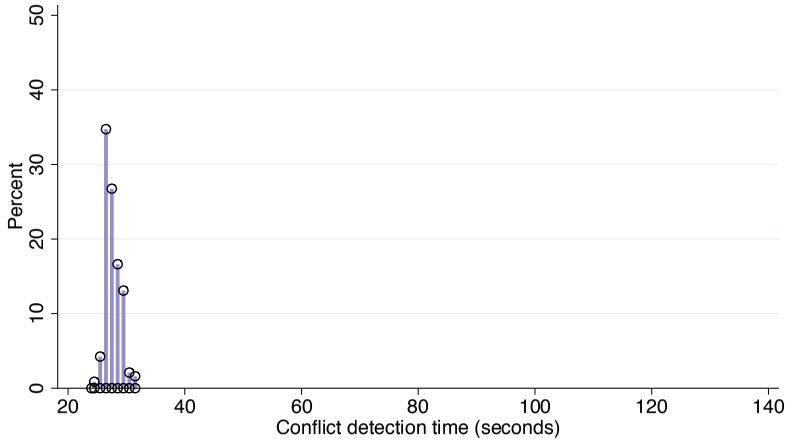
In this experiment, the rate at which a large group of architects simultaneously perform modeling operations was simulated. 10 logs of modeling operations that represent scenarios with 24 architects collaboratively updating the BOINC system model (recall



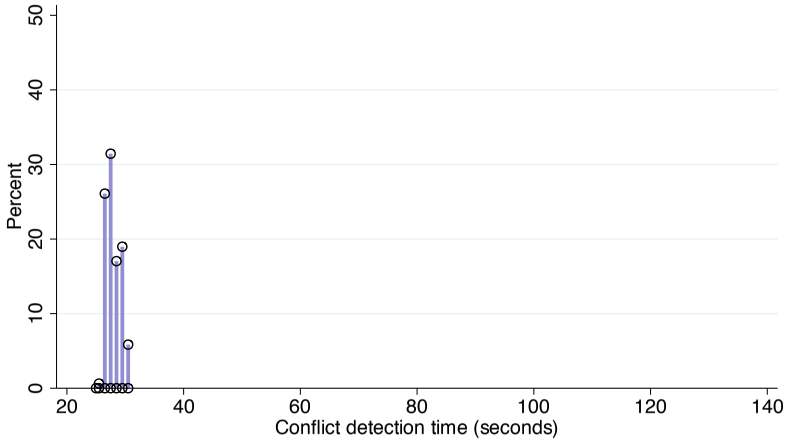
(a) Using 2 slave nodes. Mean: 36.89 secs. Max.: 133 secs.



(b) Using 4 slave nodes. Mean: 27.49 secs. Max.: 54 secs.



(c) Using 8 slave nodes. Mean: 27.10 secs. Max.: 32 secs.



(d) Using 12 slave nodes. Mean: 27.46 secs. Max.: 31 secs.

Figure 4.7: Histograms of conflict detection time using various numbers of slave nodes.

Figure 4.1) were generated by merging the modeling operation logs of 12 randomly selected participant teams of the *Head-and-Local Engine* user study. It was not possible to merge 12 teams' logs into a single log as-is because the teams performed identical design tasks on the same given model, hence the logs included modeling operations that were overlapping and incompatible with each other. To address that issue, after the logs were merged in a way that preserves the rate at which the 12 teams simultaneously perform modeling operations, all modeling operations in the merged log were replaced with a dummy modeling operation that does not overlap with each other but for which the *Detection Engine* would create a conflict detection instance, based on the assumption that the complexity or the size of the model would not likely change significantly during a short span of time (30 minutes, the duration of each design session in the *Head-and-Local Engine* user study). Those logs were then replayed in FLAME that had a *Global Engine* with 48 slave nodes, initiated to have identical computation resources on Google Compute Engine. The resulting conflict detection times were measured and compared with those of the scenarios with two architects, presented earlier in this section.

The 10 logs of modeling operations were generated in a way to best reflect the rate at which a large team of software architects would perform modeling operations. First, rather than artificially manipulated, the modeling operation logs were generated reusing the collected collaborative design behavioral data from the *Head-and-Local Engine* user study. It is important to note that the rate at which an architect performs modeling operations would likely fluctuate. Because of that, it is possible that a higher number of conflict detection instances may be initiated at a particular moment, and the available

computation resources for conflict detection may temporarily become insufficient, introducing delay in the detection. Figure 4.8 depicts at which points modeling operations were performed during the design sessions in the *Head-and-Local Engine* user study. This shows the natural variation in the rate at which modeling operations were performed. The way the modeling operation logs for this experiment were generated preserved that variation. Second, for each of the generated modeling operation logs, 12 teams' logs were merged to simulate the rate at which a "large" architect team would perform modeling operations. While it is likely to differ across projects in practice, the proportion of architects in a large software development team is one out of ten [39]. Each of the generated logs of modeling operations mimicked the rate at which a team of 24 architects simultaneously perform modeling operations, in a development team of 240. Figure 4.9 depicts the generated logs. Each log in Figure 4.9 is much denser than the logs in Figure 4.8 and preserves the variation in the rate at which modeling operations were performed.

Figure 4.10 presents the histogram of the conflict detection times from the replays of the 10 generated modeling operation logs. While the number of conflict detection instances processed and the number of slave nodes used by the *Detection Engine* were much higher than those of the 2-architect scenarios presented earlier, the mean and the maximum conflict detection times did not significantly increase from the 2-architect scenarios (recall Figure 4.7c and 4.7d). This provides evidence that the additional overhead that FLAME adds when the size of the architect team increases is *negligible* as long as the *Detection Engine* has a sufficient number of slave nodes at its disposal. In reality, however, the architect team's size may impact the rate at which software architects perform modeling operations as well as the size and the complexity of the model under design in

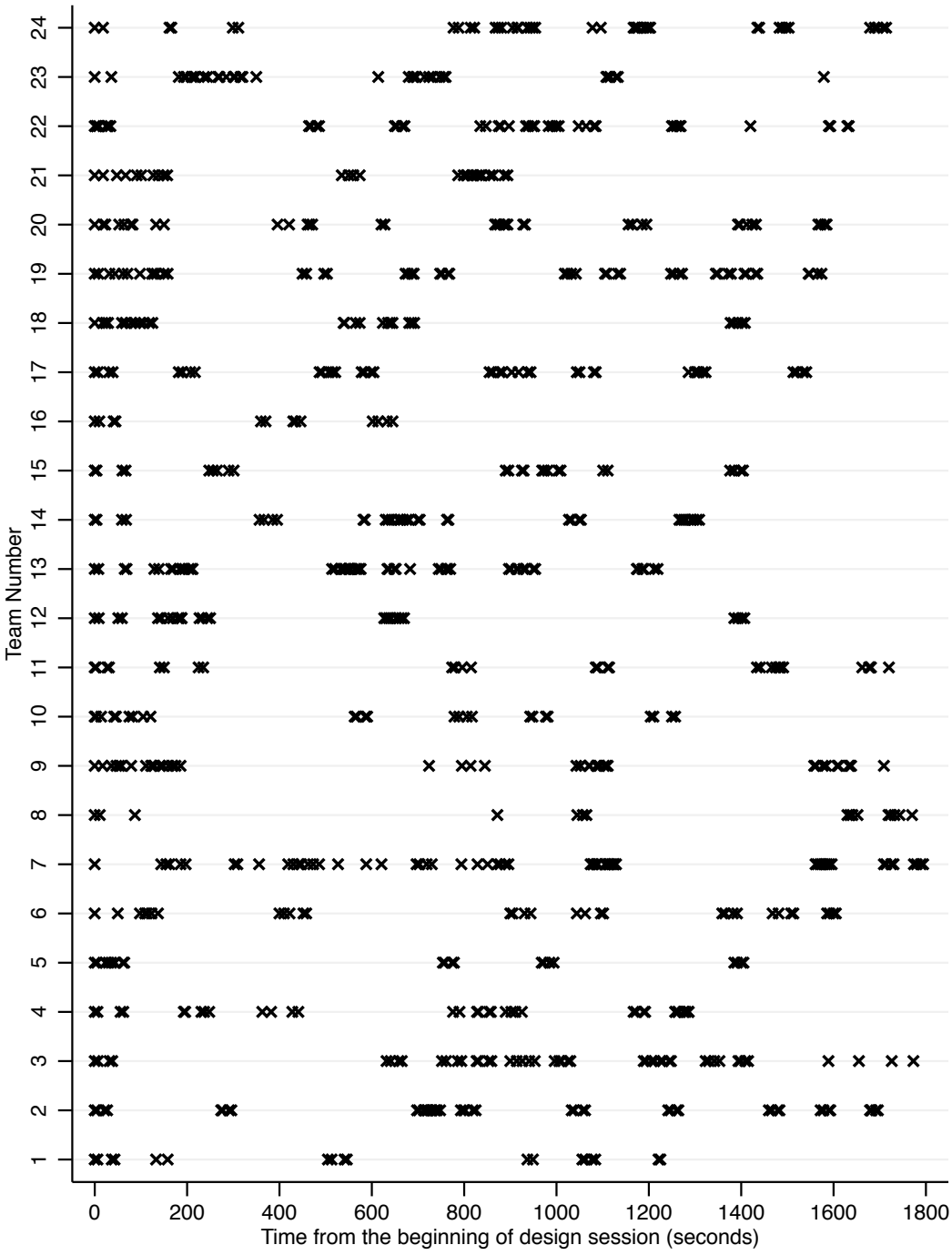


Figure 4.8: Modeling operation logs from the *Head-and-Local Engine* user study. Each × represents one modeling operation performed.

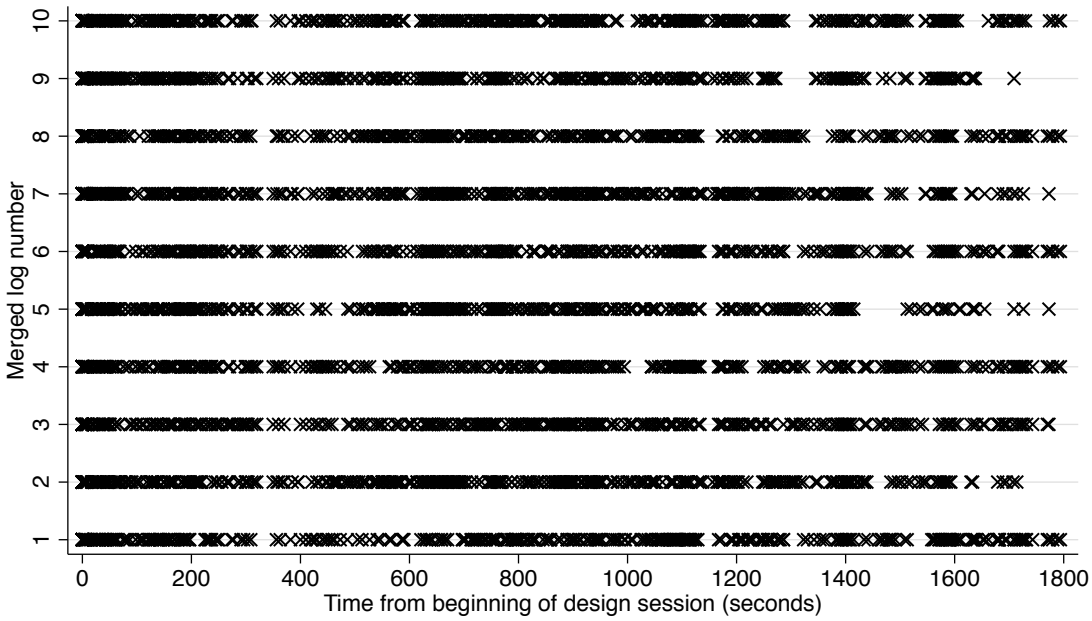


Figure 4.9: 24-architect modeling operation logs, generated for this experiment. Each × represents one modeling operation performed.

a way that leads to more needs in computation resources for detecting conflicts. Nevertheless, these threats can be mitigated by further increasing the number of slave nodes or implementing an algorithm that processes the chronologically newest conflict detection instance first in a *Detection Engine* (recall Section 3.1.3) as discussed below.

4.2.2 Setting a Bound on Conflict Detection Time

Section 3.1.3 presented FLAME's algorithm that prioritizes processing conflict detection instances in a way that gives higher priority to the chronologically newer instances to minimize the delay that may occur when the computation resources for conflict detection are scarce. The algorithm guarantees that any higher-order conflict outstanding at a given moment will be detected in the amount of time $2 \cdot t$ at most, where t is the longest processing time for a single detection instance with no delay. In order to empirically

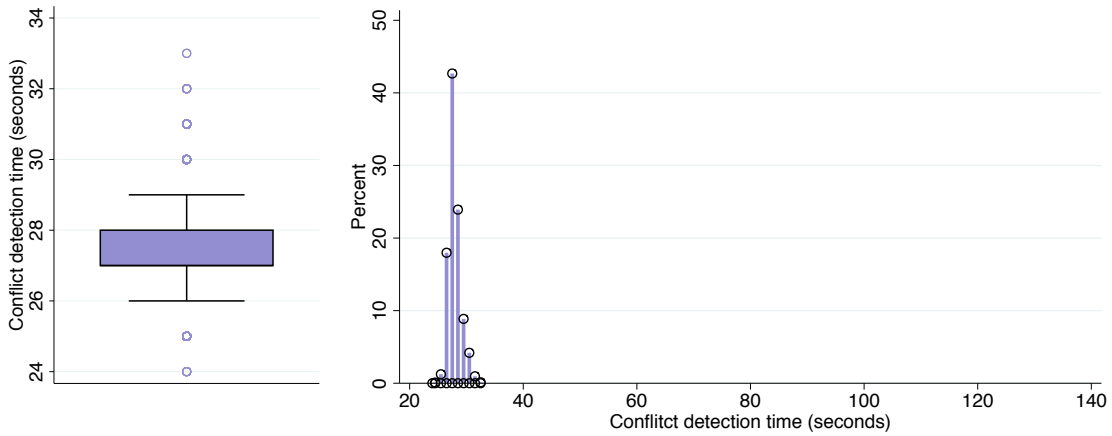


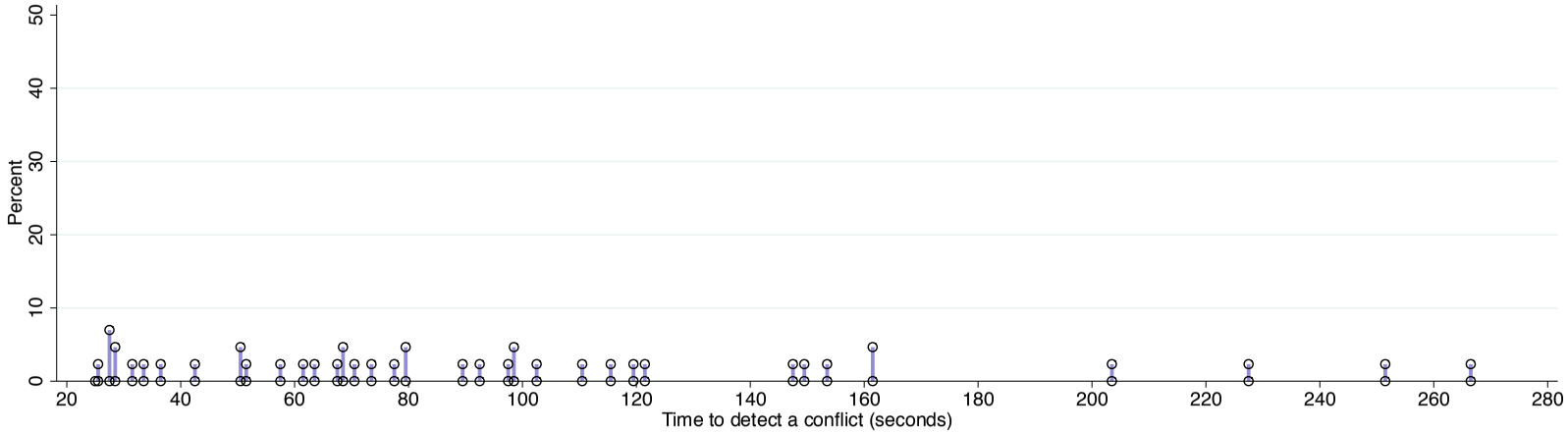
Figure 4.10: Box plot and histogram of conflict detection time of 24-architect scenarios. 10 scenarios combined. 48 slave nodes. Mean: 27.37 secs. Max.: 33 secs.

validate the algorithm to ensure it would not introduce any hidden costs when it is adopted in practice, a systematic experiment was conducted that compares a pair of identical FLAME configurations of which one does not (*oldest-first*) and the other does (*newest-first*) implement the algorithm. Two comparisons were made: (1) between a pair of configurations with one slave node and (2) between another pair with two slave nodes.

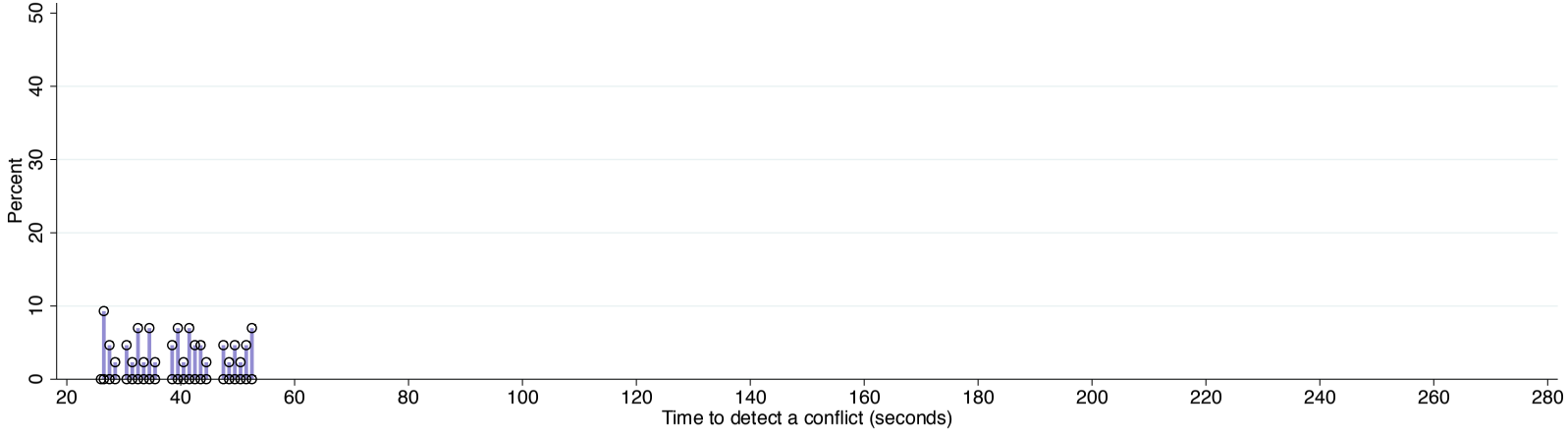
In each of the configurations, the modeling operation logs of the 24 teams of the *Head-and-Local Engine* user study were replayed. All configurations had one *Global Engine* with their respective number of slave nodes. It was previously shown in Section 4.2.1 that, if the *Head-and-Local Engine* user study participants used a *Global Engine* with two slave nodes, several conflict detection instances would have significantly been delayed. Intuitively, if the participants used a *Global Engine* with only one slave node, the delay would have only worsened. The delay, subsequently, could have increased the time-to-detection of the conflicts, which hampers the benefits of proactive conflict detection. In this experiment, each individual conflict was tracked from its creation to detection in

order to test whether implementing the prioritization algorithm would shorten the time-to-detect of the conflict when computation resources for conflict detection are limited.

Figure 4.11 (1 slave node) and Figure 4.12 (2 slave nodes) present the experiment results showing that, in the configurations that implement the prioritization algorithm (the *newest-first* configurations), (1) the mean time-to-detection times of the conflicts were significantly lower and (2) the maximum time-to-detection times were below the amount of time $2 \cdot t$ (where t is 30 seconds; recall Figure 4.5). The median time-to-detection times were 48.10% (p-value of 0.000; Mann-Whitney-Wilcoxon test) and 22.86% lower (p-value of 0.023; Mann-Whitney-Wilcoxon test) in the *newest-first* configurations with 1 slave node and 2 slave nodes respectively. More importantly, in the *newest-first* configurations, the maximum time-to-detection times fell below the amount of time $2 \cdot t$ (60 seconds); they were 53 and 43 seconds with 1 slave node and 2 slave nodes respectively. Also, in a *newest-first* configuration, a conflict may be detected when a conflict detection instance generated for a chronologically newer modeling operation than the operation that caused the conflict has been processed (recall Section 3.1.3). Overall, 60.47% and 32.56% of the conflicts were detected that way, in the *newest-first* configurations with 1 slave node and 2 slave nodes respectively. Those reversals happened more often with 1 slave node because a higher number of conflict detection instances were delayed due to the fewer computation resources available for conflict detection in that configuration.

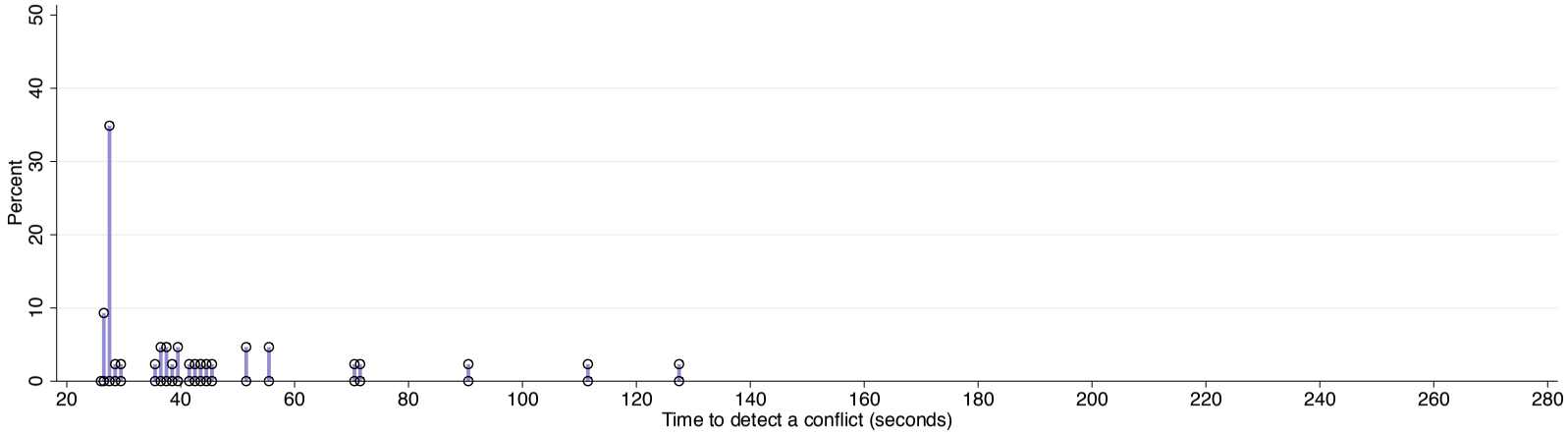


(a) Oldest-first prioritization. Mean: 92.49 secs. Median: 77 secs. Max.: 267 secs.

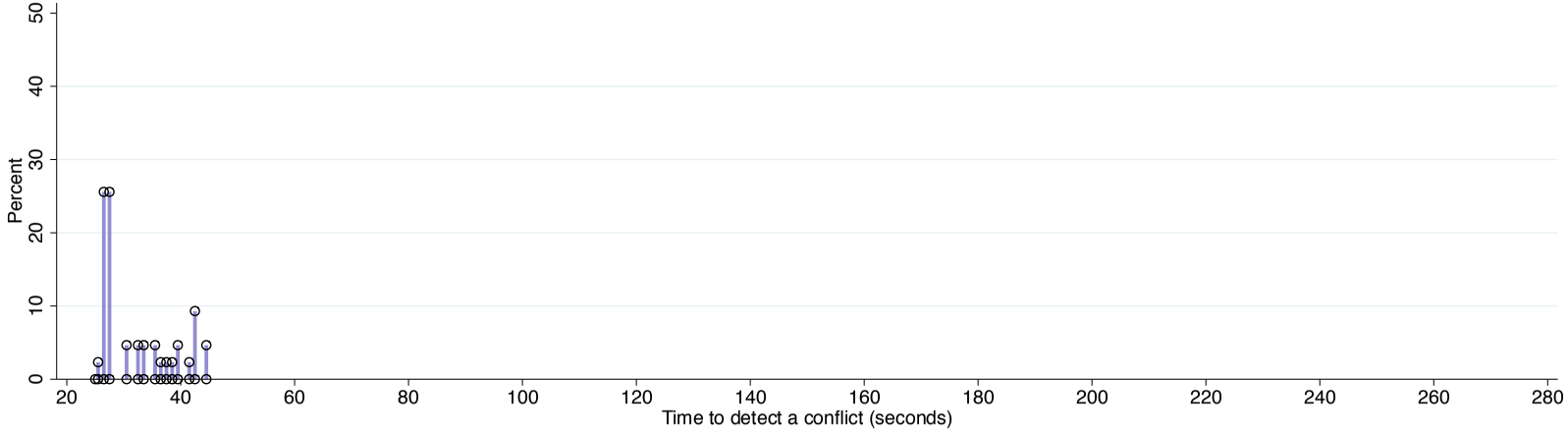


(b) Newest-first prioritization. Mean: 38.67 secs. Median: 39 secs. Max.: 53 secs.

Figure 4.11: Histograms of conflict time-to-detection with oldest- and newest-first prioritization: 1 slave node.



(a) Oldest-first prioritization. Mean: 40.93 secs. Median: 35 secs. Max.: 128 secs.



(b) Newest-first prioritization. Mean: 31.51 secs. Median: 27 secs. Max.: 45 secs.

Figure 4.12: Histograms of conflict time-to-detection with oldest- and newest-first prioritization: 2 slave nodes.

Chapter 5

Related Work

In general, this dissertation's related work can be divided into three groups, which will be discussed in detail in each of the three respective sections of this chapter. Section 5.1 is regarding the research in software model inconsistency detection, which is a key in dealing with higher-order design conflicts. Section 5.2 then introduces the prior work regarding software model version control systems (VCSs) that are specifically constructed to manage software models rather than source code. Lastly in Section 5.3, the existing proactive conflict detection techniques and tools are discussed and compared to FLAME.

5.1 Detection of Inconsistencies in Software Models

It is important to note that design decisions are *conceptual*, so as conflicts between those design decisions. Prior research has been directed to detecting those conflicts via *tangible* software models that reflect the design decisions. A class of those conflicts on which this dissertation focuses—the higher-order design conflicts—manifest themselves as *inconsistencies* in the models, which are violations of a consistency rule or a semantic rule of

the system. Hence, detecting the inconsistencies that exist in the software model would provide vital information in dealing with the higher-order design conflicts.

A substantial volume of research has been conducted in the software model inconsistency detection area. Many techniques and tools [24, 45] target detection of consistency rule violations in Unified Modeling Language (UML) [49], which is a widely used software modeling notation in practice. There are also techniques that analyze software models and provide crucial information that can be used to determine whether a system meets its requirements, such as discrete-event simulation [56], Markov-chain-based reliability analysis [65], or queueing-network-based performance analysis [4]. XTEAM [22] is one of those techniques, and it was adopted by the FLAME instance developed for this dissertation.

FLAME exploits these software inconsistency detection techniques and tools in order to proactively detect higher-order design conflicts. For example, the *Detection Engines* of the FLAME instance introduced in this dissertation integrated XTEAM in order to determine whether the model under design meets three different system properties: memory usage (as in [23]), energy consumption (as in [58]), and message latency (as in [67]).

A software model cannot evolve without having inconsistencies. *"Living with inconsistency"* is a concept to tolerate inconsistencies and exploit the information to push forward the software development [5, 28]. On the other hand, having an inconsistency that has been undetected and unknown to software architects is a risk since it is possible that the work done after the inconsistency has been introduced may need to be reversed in the process of resolving it. To minimize that risk, Argo/UML [51] has built-in design critics that provide unobtrusive review of design and suggestions for improvements continuously, i.e., as architects make modeling changes. While similar to FLAME's proactive conflict

detection in the sense that the feedback is continuously provided, Argo/UML only targets local inconsistencies. FLAME performs trial merging of modeling changes and invokes the model analysis tools in the background to detect higher-order design conflicts.

5.2 Software Model Version Control

This section focuses on the VCSs that are specifically designed to manage software models rather than source code. To cope with the challenges that arise when using a generic VCS geared toward managing textual artifacts to manage graphical software models (recall Section 2.1.3), a number of software model VCSs and the related research have appeared to support software architects to track and organize modeling changes [2]. Techniques that compute the "diffs" between independently modified copies of a software model [44, 48] have been proposed, which is an essential capability in versioning software models. Altmanninger et al. argued that the current generation of VCSs are inflexible (i.e., inextensible) and limited to certain design environments [1]. They developed an extensible software model VCS named AMOR [1, 13] as an attempt to solve that problem.

The software model VCSs also support design conflict detection [2]. While most of them detect conflicts by analyzing a set of states (e.g., saved files) of a model (called the *state-based* detection), some implement the *operation-based* detection [8, 12, 38] that detects conflicts from the sequence of modeling operations. The operation-based conflict detection techniques compliment the state-based techniques by adding an extra dimension of conflict detection. A noticeable benefit that an operation-based conflict detection technique can provide is the ability to find the operation that caused a conflict [12].

Free-form, synchronized group editors (e.g. a whiteboard shared electronically via the Internet) have also been proposed as a collaborative software design environment. The common goal they share is to promote communication between software architects. CAMEL [16] is a good example; it allows a team of architects simultaneously drawing various kinds of diagrams. The architects using CAMEL can instantiate "whiteboards" each of which is either a UML diagram or a free-form diagram. The changes the architects make to those whiteboards are shared across as they are made. More recent tools in this category have adopted multi-touch screens to further facilitate the collaboration [10, 40].

There are also synchronized group editors that implement more sophisticated conflict detection and resolution techniques. Our own tool CoDesign [8] is an extensible collaborative software modeling framework that synchronizes modeling operations in real time. It implements operation-based conflict detection, and notifies software architects with conflict information as a conflict arises. An interesting instance in that group is Google Docs [29], which supports real time group editing of documents including graphs. Google Docs implements an algorithm called *operational transformation* [25, 60] that guarantees automatic resolution of synchronization conflicts without getting users' attention.

Even though the synchronized group editors provide rich communication channels between software architects and may prevent causing conflicts in the first place, they may also distract the architects and lose parallelism in the case of rapid design. Each of the two collaborative design approaches, the software model VCSs and the synchronized group editors for software modeling, has its own pros and cons. There is an open question regarding whether a hybrid of those can be built and what benefits it may provide [9].

FLAME improves the software model VCSs by detecting conflict in a proactive way. While the existing software model VCSs are capable of managing software modeling changes, they expose software architects to the risk of having undetected conflicts because they only detect conflicts periodically (recall Section 2.2). FLAME mitigates that risk by detecting conflicts as soon as the modeling operation that causes a conflict is performed.

5.3 Proactive Conflict Detection

The risk of having conflicts from using a copy-edit-merge style, asynchronous VCS is not unique to collaborative software design. Collaborative software *implementation* also faces an analogous challenge at the level of source code. A number of techniques and tools have been reported, including those for proactive conflict detection [54].

Providing *workspace awareness* is an extensively studied aspect of conflict avoidance and detection. Workspace awareness is "the up-to-the-minute knowledge of other participants' interactions with the shared workspace" [33]. FASTDash prevents potential conflict situations (e.g., two developers editing the same file) by providing a visual presentation of the developers' activities on shared files [11]. Some workspace awareness tools analyze dependencies between program elements (files, types, or methods), and notify developers of conflicting changes made to elements that depend on each other [21]. Palantír shows which shared artifacts have been edited by whom, in a less obtrusive way by integrating the presentation into the development environment [55]. Syde [34] informs developers of concurrent changes by maintaining an abstract syntax tree of the target object-oriented system, interpreting code changes into tree operations, and using them

to filter conflict information. The tools in this group primarily detect synchronization conflicts and dependency-based higher-order conflicts.

Another group of proactive conflict detection tools perform deeper analyses such as compilation, unit testing, and so on. Safe-commit [66] proactively identifies "commit-table" changes that will not make test cases fail by running them in the background. Two tools in this group, Crystal [15] and WeCode [32], are closely related to FLAME. Both of them proactively perform merging, compilation, and testing of new changes developers make to source code in the background and notify the developers if any of the steps fails. Table 5.1 further compares those tools as well as Palantír to FLAME in detail.

FLAME differs from the existing tools in two ways: (1) exploiting its event-based architecture, FLAME integrates off-the-shelf higher-order conflict detection tools and offloads the potentially resource-intensive conflict detection, and (2) it synchronizes and is capable of performing conflict detection per modeling operation, which enables earlier conflict detection and pinpointing the specific operations that caused a conflict [12].

Moreover, in spite of the promising results reported from empirical studies and actual use of proactive conflict detection for collaborative software implementation [11,14,21,53], due to the differences between software design and implementation, it had not been clearly known whether those techniques would positively impact collaborative software design when implemented. This dissertation provides the initial evidence that shows proactive conflict detection may benefit collaborative software design in practice.

Table 5.1: Proactive Conflict Detection Studies Comparison.

	Palantíŕ [55]	WeCode [32]	Crystal [14]	FLAME [7]
Published year	2012	2012	2013	2015
Target activity	Programming	Programming	Programming	Design
Target artifact	Program code	Program code	Program code	Software model
Merge implementation	SVN or CVS	Ecliper Equinox	Mercurial or Git	Itself
Version control mechanism	State-based	State-based	State-based	Operation-based
Merge granularity	A line of code	A structural element	A line of code	A modeling operation
Conflict definition	Direct, Indirect	Direct, Indirect	Texual, Higher-order	Synchronization, Higher-order
Detection frequency	At file savings, timer expirations	At file savings	At local commits	For each operation
Evaluation method	User study	User study	Public use	User study
Data collection	Conducted study with 40 users	Conducted study with 21 users	Released Crystal to public and gathered response	Conducted study with 90 users
User study team size	2	2	N/A	2

Chapter 6

Concluding Remarks

Higher-order software design conflicts are inevitable in collaborative software design. The asynchrony of today's copy-edit-merge style software model version control systems parallelizes individual design work and may achieve higher productivity, but at the same time, it exposes the architects to the risk of making modeling changes without being aware of the presence of a higher-order design conflict. That may lead to the reversal of those changes in the process of resolving the conflict, which results in wasted effort. For the analogous problem at the level of source code, the proactive conflict detection strategy has been proposed, and applications of it have shown promising results from empirical studies and actual use. However, those existing proactive conflict detection tools are constructed to deal with the code-level conflicts and do not work well with the design-level conflicts due to the inherent differences between software design and implementation.

This dissertation presented a solution, FLAME, an extensible, operation-based collaborative software design framework that proactively detects higher-order conflicts. FLAME minimizes the risk by performing a trial synchronization and conflict detection in the background, as frequent as for each modeling operation performed by the architects. FLAME's

novel architecture is specifically designed to support collaborative design, which enables integrating the appropriate modeling tool and consistency checkers for the target system's domain. Exploiting its event-based architecture, FLAME offloads the potentially computationally-expensive conflict detection activities to remote nodes called *Detection Engines*. Each *Detection Engine* is responsible for invoking an off-the-shelf conflict detection tool on a particular version of the model that the *Detection Engine* automatically derives by merging modeling operations from different architects. In order to prevent the *Detection Engines* from being overwhelmed by a large amount of conflict detection to perform, FLAME provides facilities to further distribute the burden of conflict detection to a set of cloud-based nodes called *slave nodes*. FLAME also implements an algorithm that prioritizes conflict detection instances a *Detection Engine* processes in order to minimize the delay in conflict detection that may occur when the available computation resources for conflict detection are scarce. That algorithm guarantees a reasonable worst-case time to detect each higher-order conflict outstanding at a given moment.

In this dissertation, FLAME has been evaluated both empirically and analytically. In order to validate whether or to what extent providing proactive conflict detection impacts the collaborative software design cost, two user studies were conducted with total of 90 participants, each of which targeted the two *Detection Engines* introduced in this dissertation: the *Global Engine* and the *Head-and-Local Engine*. In both of the user studies, it was observed that the participants who were provided with proactive conflict detection (1) had more opportunity to communicate with each other, (2) detected and resolved higher-order design conflicts earlier and more quickly, and (3) produced higher quality models in the same amount of time. Moreover, to our best knowledge, these

user studies provide the first reported empirical evidence showing that proactive conflict detection positively impacts the collaborative design cost. Also, the results from the three analytical studies showed that FLAME (1) minimized the delay in conflict detection especially when a *Detection Engine* had a sufficient number of slave nodes at its disposal and (2) added only a negligible amount of overhead in performing conflict detection even in collaborative design scenarios with a large team of architects. Most notably, with FLAME's prioritization algorithm implemented, all conflicts either had been resolved or were detected in the amount of time $2 \cdot t$ (where t is the longest time required to process one conflict detection instance with no delay), when the available computation resources for conflict detection were limited. Those results together indicate the potential in FLAME that it can handle the need in proactive conflict detection of a real-world collaborative design project that involves a large team of participating software architects.

FLAME is not limited to the form as presented in this dissertation, and variations of it may also benefit collaborating software architects and positively impact the collaborative design cost. FLAME can incorporate other *Detection Engines* beyond the two introduced in this dissertation that derive a different version of the model on which to perform conflict detection. *Detection Engines* may also integrate other detection techniques including the ones that are intended for synchronization conflicts. While not explicitly studied in this dissertation, variations in the interface via which FLAME presents conflict information to the architects may also influence the way the architects form conflict awareness.

Implementing *operation transformation* [25, 60]—an algorithm that automatically resolves synchronization conflicts—on top of FLAME would open up a variety of possible

future work that assist collaborating software architects. In fact, operational transformation can readily be implemented on top of FLAME because FLAME already implements operation-based version control on which the algorithm relies. Each architect in a team often designs different parts of a system, and different parts may need different kinds of design collaboration. FLAME can potentially be configured to support either the copy-edit-merge style version control or the synchronized group editing for different parts of the system by implementing operational transformation to fulfill such needs.

FLAME provides a foundation for exploring several other issues with design-level conflict detection. These include exploration of different ways of delivering feedback to architects, assessment of the effect of variations on the immediacy with which feedback is delivered, prioritization of the delivery of analysis results, and exploration of the utility of proactively analyzing software models for properties of whose importance the architects may be unaware. Conflict resolution support would be an instance of those. For example, possible actions that architects may take to resolve an existing conflict can be populated in a speculative fashion. FLAME may then run simultaneous analyses on the versions of the model in which each of those actions would result, and suggest a small set of resolution options from which the architects choose in order to expedite the resolution process.

Assessing the root cause of a higher-order conflict is another area in which FLAME may benefit collaborating architects. When a higher-order conflict arises, it is often ambiguous which specific set of modeling operations caused the conflict and who performed them. Knowing the list of those operations is the key to resolving the conflict, but the architects today have to manually inspect the model in order to find them. Exploiting FLAME's facilities that distribute conflict detection activities to remote nodes, different

combinations of the past modeling operations can be derived and checked in parallel to quickly find out which particular operations have contributed in causing the conflict.

References

[1] Kerstin Altmanninger, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. AMOR – Towards Adaptable Model Versioning. In *Proceedings of the International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS*, volume 8, pages 4–50, 2008.

[2] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A Survey on Model Versioning Approaches. *International Journal of Web Information Systems (IJWIS)*, 5(3):271–304, 2009.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[5] Robert Balzer. Tolerating Inconsistency. In *Proceedings of the 13th International Conference on Software Engineering (ICSE)*, pages 158–165. IEEE, 1991.

[6] Jae young Bang, Ivo Krka, Nenad Medvidovic, Naveen Kulkarni, and Srinivas Padmanabhuni. How Software Architects Collaborate: Insights from Collaborative Software Design in Practice. In *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 41–48. IEEE, May 2013.

[7] Jae young Bang and Nenad Medvidivoc. Proactive Detection of Higher-Order Software Design Conflicts. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, May 2015.

[8] Jae young Bang, Daniel Popescu, George Edwards, Nenad Medvidovic, Naveen Kulkarni, Girish M Rama, and Srinivas Padmanabhuni. CoDesign: A Highly Extensible Collaborative Software Modeling Framework. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, volume 2, pages 243–246. ACM, May 2010.

[9] Jae young Bang, Daniel Popescu, and Nenad Medvidovic. Enabling Workspace Awareness for Collaborative Software Modeling. *The Future of Collaborative Software Development at the ACM Conference on Computer Supported Cooperative Work (FutureCSD)*, February 2012.

[10] Mohammed Basheri and Liz Burd. Exploring the Significance of Multitouch Tables in Enhancing Collaborative Software Design Using UML. In *Proceedings of the Frontiers in Education Conference (FIE)*, pages 1–5. IEEE, 2012.

[11] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 1313–1322. ACM, 2007.

[12] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting Model Inconsistency through Operation-based Model Construction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 511–520. IEEE, 2008.

[13] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip Langer. We can work it out: Collaborative Conflict Resolution in Model Versioning. In *Proceedings of the 11th European Conference on Computer Supported Cooperative Work (ECSCW)*, pages 207–214. Springer, September 2009.

[14] Yuriy Brun, Reid Holmes, M. Ernst, and David Notkin. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering (TSE)*, 39:1358–1375, October 2013.

[15] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Crystal: Precise and Unobtrusive Conflict Warnings. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 444–447. ACM, 2011.

[16] Marcelo Cataldo, Charles Shelton, Yongjoon Choi, Yun-Yin Huang, Vytesh Ramesh, Darpan Saini, and Liang-Yun Wang. Camel: A Tool for Collaborative Distributed Software Design. In *Proceedings of the 4th International Conference on Global Software Engineering (ICGSE)*, pages 83–92. IEEE, 2009.

[17] Ben Collins-Sussman. The Subversion Project: Buiding a Better CVS. *Linux Journal*, 2002(94):3, 2002.

[18] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. *Version Control with Subversion*. O'Reilly Media, Inc., 2004.

[19] Catarina Costa and Leonardo Murta. Version Control in Distributed Software Development: A Systematic Mapping Study. In *Proceedings of the 8th International Conference on Global Software Engineering (ICGSE)*, pages 90–99. IEEE, 2013.

[20] Daniela Damian, Remko Helms, Irwin Kwan, Sabrina Marczak, and Benjamin Koelewijn. The Role of Domain Knowledge and Cross-Functional Communication in Socio-Technical Coordination. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 442–451. IEEE, 2013.

[21] Prasun Dewan and Rajesh Hegde. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In *Proceedings of the 10th European Conference on Computer Supported Cooperative Work (ECSCW)*, pages 159–178. Springer, 2007.

[22] George Edwards. The eXtensible Tool-chain for Evaluation of Architectural Models. `http://softarch.usc.edu/~gedwards/xteam.html`, 2014. [Online; accessed March 4, 2015].

[23] George Edwards, Chiyoung Seo, and Nenad Medvidovic. Model Interpreter Frameworks: A Foundation for the Analysis of Domain-Specific Software Architectures. *Journal of Universal Computer Science*, 14(8):1182–1210, 2008.

[24] Alexander Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions on Software Engineering (TSE)*, 37(2):188–204, 2011.

[25] Clarence A. Ellis and Simon J. Gibbs. Concurrency Control in Groupware Systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 18(2):399–407, 1989.

[26] EMFStore. `http://eclipse.org/emfstore/`. [Online; accessed March 4, 2015].

[27] Git. `http://git-scm.com`, 2014. [Online; accessed March 4, 2015].

[28] Michael Goedicke, Torsten Meyer, and Gabriele Taentzer. Viewpoint-Oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies. In *Proceedings of the International Symposium on Requirements Engineering (RE)*, pages 92–99. IEEE, 1999.

[29] Google. Google Docs. `https://docs.google.com`, 2014. [Online; accessed March 4, 2015].

[30] Google. Google Compute Engine. `https://cloud.google.com/compute/`, 2015. [Online; accessed March 4, 2015].

[31] Dick Grune. Concurrent Versions System, A Method for Independent Cooperation. *Report IR-114, Vrije University, Amsterdam*, 1986.

[32] Mário Luís Guimarães and António Rito Silva. Improving Early Detection of Software Merge Conflicts. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 342–352. IEEE, 2012.

[33] Carl Gutwin and Saul Greenberg. Workspace Awareness for Groupware. In *Conference Companion on Human Factors in Computing Systems (CHI)*, pages 208–209. ACM, 1996.

[34] Lile Hattori and Michele Lanza. Syde: A Tool for Collaborative Software Development. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, volume 2, pages 235–238. ACM, May 2010.

[35] James D. Herbsleb and Audris Mockus. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Transactions on Software Engineering*, 29(6):481–494, 2003.

[36] Pamela J. Hinds and Diane E. Bailey. Out of Sight, Out of Sync: Understanding Conflict in Distributed Teams. *Organization Science*, 14(6):615–632, 2003.

[37] Institute for Software Integrated Systems, Vanderbilt University. Generic Modeling Environment. `http://isis.vanderbilt.edu/projects/gme/`, 2014. [Online; accessed March 4, 2015].

[38] Maximilian Koegel, Jonas Helming, and Stephan Seyboth. Operation-based Conflict Detection and Resolution. In *Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models*, pages 43–48. IEEE Computer Society, 2009.

[39] Philippe Kruchten. The Software Architect. In *Software Architecture*, pages 565–583. Springer, 1999.

[40] Dastyni Loksa, Nicolas Mangano, Thomas D. LaToza, and André van der Hoek. Enabling a Classroom Design Studio with a Collaborative Sketch Design Tool. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1073–1082. IEEE Press, 2013.

[41] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. A Style-aware Architectural Middleware for Resource-constrained, Distributed Systems. *IEEE Transactions on Software Engineering (TSE)*, 31(3):256–272, 2005.

[42] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Proceedings of the 13th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 179–189. ACM, 2011.

[43] Chris A. Mattmann, Christopher S. Lynnes, Luca Cinquini, Paul M. Ramirez, Andrew F. Hart, Dean Williams, Duane Waliser, and Pamela Rinsland. Next Generation Cyberinfrastructure to Support Comparison of Satellite Observations with Climate Models. In *Proceedings of European Space Agency Conference on Big Data from Space (BiDS)*, November 2014.

[44] Akhil Mehra, John Grundy, and John Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE)*, pages 204–213. ACM, 2005.

[45] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *Proceedings of the 9th International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, pages 200–214. Springer, 2006.

[46] Leonardo Murta, Chessman Corrêa, João Gustavo Prudêncio, and Cláudia Werner. Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System. In *Proceedings of the International Workshop on Comparison and Versioning of Software Models (CVSM)*, pages 25–30. ACM, 2008.

[47] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology (TOIT)*, 2(2):151–185, 2002.

[48] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. An Infrastructure for Development of Object-Oriented, Multi-Level Configuration Management Services. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 215–224. ACM, 2005.

[49] Object Management Group (OMG). Unified Modeling Language. `http://www.omg.org`. [Online; accessed March 4, 2015].

[50] Gary M. Olson and Judith S. Olson. Distance Matters. *Human-Computer Interaction (HCI)*, 15(2):139–178, 2000.

[51] Jason E. Robbins and David F. Redmiles. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. *Information and Software Technology*, 42(2):79–89, 2000.

[52] Marc J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering (TSE)*, 1(4):364–370, 1975.

[53] Anita Sarma, David Redmiles, and André van der Hoek. Empirical Evidence of the Benefits of Workspace Awareness in Software Configuration Management. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 113–123. ACM, 2008.

[54] Anita Sarma, David Redmiles, and André van der Hoek. Categorizing the Spectrum of Coordination Technology. *IEEE Computer*, 43(6):61–67, 2010.

[55] Anita Sarma, David F. Redmiles, and André van der Hoek. Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Transactions on Software Engineering (TSE)*, 38(4):889–908, 2012.

[56] Thomas J. Schriber and Daniel T. Brunner. Inside Discrete-Event Simulation Software: How It Works and Why It Matters. In *Simulation Conference, 2005 Proceedings of the Winter*. IEEE, 2005.

[57] Bikram Sengupta, Satish Chandra, and Vibha Sinha. A Research Agenda for Distributed Software Development. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 731–740. ACM, 2006.

[58] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An Energy Consumption Framework for Distributed Java-Based Systems. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 421–424. ACM, 2007.

[59] Darja Šmite, Claes Wohlin, Tony Gorschek, and Robert Feldt. Empirical Evidence in Global Software Engineering: A Systematic Review. *Empirical Software Engineering*, 15(1):91–118, 2010.

[60] Chengzheng Sun and David Chen. A Multi-Version Approach to Conflict Resolution in Distributed Groupware Systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, pages 316–325. IEEE, 2000.

[61] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[62] Walter F. Tichy. RCS A System for Version Control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[63] University of California. BOINC. `http://boinc.berkeley.edu`. [Online; accessed March 4, 2015].

[64] Bernhard Westfechtel. Merging of EMF Models. *Software & Systems Modeling*, 13(2):757–788, 2014.

[65] James A. Whittaker and Michael Thomason. A Markov Chain Model for Statistical Software Testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.

[66] Jan Wloka, Barbara Ryder, Frank Tip, and Xiaoxia Ren. Safe-Commit Analysis to Facilitate Team Software Development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 507–517. IEEE Computer Society, 2009.

[67] Murray Woodside. Tutorial Introduction to Layered Modeling of Software Performance. Carleton University. `http://sce.carleton.ca/rads/`, 2005. [Online; accessed March 4, 2015].

Appendix A

Design Tasks from the User Studies

This appendix presents two samples of the design tasks given to the participants of the two user studies conducted for this dissertation (recall Section 4).

A.1 *Global Engine* User Study Design Tasks

Task 1

The NGCA system has these nonfunctional property (NFP) global requirements:

- The response rate (number of responses / number of requests) **must be greater than 95%**.
- The average latency at the monitored interfaces **must be less than 40**
- The overall energy consumption **must be less than 10,000,000**
- The maximum memory use **must be less than 800**

<table>
<tr><th>Task Set #</th><th>Task</th></tr>
<tr><td>Task 1-1</td><td>It has been 10 years since the client company began using the legacy SalinityReqGen component in their NGCA. The client company is not satisfied with the rate at which SalinityReqGen can create and send requests, and they want to replace the component with newer ones.
You did a thorough search with which newer components you could replace SalinityReqGen, and eventually found three with following characteristics:

<table>
<tr><td>Option</td><td>Execution time of the Task generate in Process generateRequest</td><td>Memory usage of the Task generate in Process generateRequest</td><td>SalinityReqGen’s corresponding Host energy coefficients</td></tr>
<tr><td>1</td><td>4 [↓↓]</td><td>250 [↑]</td><td>50/200/65/240 [↑]</td></tr>
<tr><td>2</td><td>7 [↓]</td><td>3 [as-is]</td><td>50/200/65/240 [↑]</td></tr>
<tr><td>3</td><td>6 [↓]</td><td>250 [↑]</td><td>12/45/18/48 [as-is]</td></tr>
</table>

You goal is to find which combination(s) of the three above for each of SalinityReqGen (e.g. option 2 for User1, option 3 for User2, and option 1 for User3) that the current NGCA has makes the most number of requests in the given amount of time (2,500 units of time).</td></tr>
<tr><td>Task 1-2</td><td>The client company was hit by a huge financial downturn last year, and they desperately need to cut cost down. Unfortunately, the budget for their NGCA has been drastically reduced as a result. The problem is that they have continuously been paying license for the use of the two FTP connectors that they implemented in the NGCA. They want to replace the connectors with cheaper ones while maintaining the minimum necessary throughput.
You did a thorough search with which cheaper connectors you could replace those connectors, and eventually found three with following characteristics:

<table>
<tr><td>Option</td><td>$/yr</td><td>Execution time of the Task forward in Process forwardResponse</td><td>Memory usage of the Task forward in Process forwardResponse</td><td>FTP connector’s corresponding Host energy coefficients</td></tr>
<tr><td>1</td><td>1.1k</td><td>7 [↑]</td><td>15 [↑]</td><td>12/40/20/70 [↑]</td></tr>
<tr><td>2</td><td>2.3k</td><td>0 [as-is]</td><td>1 [as-is]</td><td>12/40/20/70 [↑]</td></tr>
<tr><td>3</td><td>2.1k</td><td>0 [as-is]</td><td>15 [↑]</td><td>11/40/13/45 [as-is]</td></tr>
</table>

Your goal is to find which combination(s) of the three above for each of the FTP connectors (e.g. option 2 for FTP_S and option 3 for FTP_T) that the current NGCA has makes the cost the cheapest.</td></tr>
</table>

Figure A.1: A *Global Engine* user study design task sample.

A.2 *Head-and-Local Engine* Study Design Tasks

Task 1

System Requirements

The BOINC NPC system has the following three system requirements:

1. The average latency at the monitored interface (`ToBOINC`) must be less than **550**.
2. The overall energy consumption must be less than **4,000,000**.
3. The maximum memory use at any component/connector must be less than **30,000**.

Task for Student 1

You are the architect who is responsible for the server-side of BOINC NPC. Figure 1 depicts the BOINC NPC model. On the right side, there are two "task generator components" (`SubsetSumGen` and `SATGen`) that generate the `Task` events and forward them to `COConn`. To satisfy the client's request, you searched for with which alternative components you could replace those task generator components, and eventually found two, each of which has the following characteristics:

The Alternatives

Option	Execution time of the Task `generate` in Process `generateTask`	Memory usage of the Task `generate` in Process `generateTask`	The four energy attributes of the task generator component's corresponding Host (top to bottom)			
`Default`	8.0	3	55.24	82.45	101.28	64.99
1	7.5	5500	182.00	217.00	101.28	64.99
2	7.0	11000	330.00	429.00	101.28	64.99

The Goal

You goal is to find the best selection for each of the task generator components (e.g., option 1 for `SubsetSumGen` and option 2 for `SATGen`) from the three options above that will generate the most number of computation tasks. After you select an option for a component, *replace all six attributes* related to that component (column 2 to 7 in the table above) to those of the option you selected. You may choose not to replace a component in order to meet a system requirement. Try different selections to find the best one. Make sure your model satisfies all the system requirements above.

Task for Student 2

You are the architect who is responsible for the client-side of BOINC NPC. Figure 1 depicts the BOINC NPC model. On the left side, there are two "computation components" (`Clients` and `GoogleComputeEngine`) on which the NP-complete computation tasks are actually performed. To satisfy the client's request, you searched for with which alternative components you could replace those computation components, and eventually found two, each of which has the following characteristics:

The Alternatives

Option	Execution time of the Task `perform` in Process `performComputation`	Memory usage of the Task `perform` in Process `performComputation`	The four energy attributes of the computation component's corresponding Host (top to bottom)			
`Default`	500	27	105.11	129.87	150.01	101.86
1	350	39	312.00	341.00	150.01	101.86
2	200	92	514.00	755.00	150.01	101.86

The Goal

You goal is to find the best selection for each of the computation components (e.g., option 2 for `Clients` and option 1 for `GoogleComputeEngine`) from the three options above that will have the shortest computation time. After you select an option for a component, *replace all six attributes* related to that component (column 2 to 7 in the table above) to those of the option you selected. You may choose not to replace a component in order to meet a system requirement. Try different selections to find the best one. Make sure you also satisfy the system requirements given above.

Figure A.2: A *Head-and-Local Engine* user study design task sample.