Continuous Analysis of Collaborative Design

Jae young Bang
Kakao Corporation
Seongnam, Gyeonggi, Korea
jae.bang@kakaocorp.com

Yuriy Brun
University of Massachusetts
Amherst, MA, USA 01003-9264
brun@cs.umass.edu

Nenad Medvidovic
University of Southern California
Los Angeles, CA, USA 90089
neno@usc.edu

***Abstract*—In collaborative design, architects' individual design decisions may conflict and, when joined, may violate system consistency rules or non-functional requirements. These design conflicts can hinder collaboration and result in wasted effort. Proactive detection of *code-level* conflicts has been shown to improve collaborative productivity; however, the computational resource requirements for proactively computing *design* conflicts have hindered its applicability in practice. Our survey and interviews of 50 architects from six large software companies find that 60% of their projects involve collaborative design, that architects consider integration costly, and that design conflicts are frequent and lead to lost work. To aid collaborative design, we re-engineer FLAME, our prior design conflict detection technique, to use cloud resources and a novel prioritization algorithm that, together, achieve efficient and nonintrusive conflict detection, and guarantee a bound on the time before a conflict is discovered. Two controlled experiments with 90 students trained in software architecture in a professional graduate program, demonstrate that architects using FLAME design more efficiently, produce higher-quality designs, repair conflicts faster, and prefer using FLAME. An empirical performance evaluation demonstrates FLAME's scalability and verifies its time-bound guarantees.**

I. Introduction

Consider the following scenario, described to us by a division director in a multinational software services company. A team of software architects is designing a large system. While the team is distributed across three sites, a core group of senior architects physically collocates with the product manager for initial requirements analysis and architectural design. Once satisfied that the remaining design activities are appropriately divided, the core group rejoins their original subteams. Each subteam proceeds to refine the design of its portion of the system, while, in parallel, development teams proceed with the system's implementation. The architect teams capture the design using an in-house software modeling tool. All design changes are saved into a shared version control system (VCS) repository. The architects work on design tasks alone or in small local groups. Design consistency is encouraged both locally, through daily status meetings and regular communication, and team-wide, through weekly video-conferences. Despite the senior architects' best efforts initially and the subsequent regular discussions by the team, the division director reported two types of issues that arose regularly, requiring significant additional coordination among the architects and rework:

1) *Architects modify the design in a mutually-inconsistent way.* An example involved an architect making the type of an attribute in a utility component more general because many of the components in his portion of the system needed to use it. During the same time frame, a senior architect made the attribute type more specific because a development team alerted her to a security issue involving an off-the-shelf library. The architects discovered the conflict only when the VCS reported it and was unable to merge their changes.
2) *Architects make local modifications that, when merged, violate a critical non-functional property.* An example involved two teams trying to reduce message latency, respectively, via smart caching and via piggybacking multiple payloads onto a single message. An analysis of the merged design showed that, together, these solutions sometimes increased latency and introduced unacceptably high memory consumption.

Scenarios like these occur frequently in practice and directly motivate our work. Software systems are often collaboratively designed by multiple architects. During collaborative design, architects make decisions, reify those decisions into software models [54], and evolve the models as a team [32]. To support collaborative evolution of software models, architects have adapted to using traditional copy-edit-merge-style VCSs [2], resulting in individual workspaces that allow architects to design in parallel and synchronize their work on demand.

However, this loose synchronization exposes architects to the risk of introducing two types of *design conflicts*. *Synchronization* design conflicts (exemplified by scenario #1 above) are design decisions that are mutually inconsistent and hence cannot be merged automatically by the VCS. *High-order* design conflicts (exemplified by scenario #2) are decisions that can be merged automatically, but once merged, violate a consistency rule, non-functional requirement, or another system constraint. Unfortunately, VCSs help to discover synchronization design conflicts only when the architects synchronize, and high-order design conflicts after synchronization and after the architects elect to run their analyses. This can lead to design and implementation work that repeatedly needs to be corrected, even abandoned, which becomes more difficult when engineers forget the context of their initial decisions because of the time lag.

Collaboration is both common and often causes conflicts [5], [12], [13], [29]. Prior research on proactive detection of *code-level* conflicts [12], [26], [51] using speculative analysis [10], [12], and on continuously making *code-level* analysis results available to developers [39], [40], [47], [57] has shown great benefits, reducing conflict lifetime and improving developers' ability to make well-informed decisions. Unfortunately, unlike *code-level* conflict detection, continuous proactive conflict detection (PCD) at the *design-level* may be prohibitively expensive. Many design analyses are highly computation in-

978-1-5090-5729-0/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSA.2017.45

tensive, including, e.g., discrete-event simulation [52], Markov-chain-based reliability analysis [59], queueing-network-based performance analysis [4], and symbolic model checking [15]. Running such analyses locally on an architect's machine may slow the architects' tools, and delayed conflict discovery may lead to decisions that must be reversed later. Slow analyses further exacerbate the problem by allowing more pending analysis instances to queue up. Thus, for PCD to be useful, it must prioritize and distribute the computational load (1) to deliver analysis results quickly and (2) to avoid detrimentally impacting the architect's use of her tools.

In this paper, we propose FLAME, a framework that interfaces with architects' modeling tools and design analyses to efficiently, continuously, and proactively detect design conflicts. We do not develop *new* architectural analyses that may identify new kinds of conflicts. Instead, we rely on existing architectural analyses (e.g., [4], [15], [21], [48], [52], [59]). Building on prior work, including our own [5], [6], this paper introduces four original contributions:

1) An algorithm for prioritizing merged designs that guarantees an upper bound on conflict detection time. (Section II)
2) A dynamic analysis and synchronization engine that distributes conflict detection onto cloud resources to avoid impacting the load on an architect's machine. (Section II)
3) Two controlled user studies involving 90 engineers studying software architecture in a professional graduate program, demonstrating that with PCD, architects design more efficiently, produce higher-quality designs, and repair conflicts faster. This evaluation significantly extends preliminary results from a small pilot study [6]. (Section III)
4) An empirical performance evaluation. (Section IV)

II. FLAME: Conflict Detection

As a solution to the challenges of collaborative design, we have designed and built FLAME, the Framework for Logging and Analyzing Modeling Events. We publicly release FLAME: http://flamedesign.org. FLAME integrates with architects' modeling and analysis tools, and provides a novel event-based VCS. While the ideas behind and design of FLAME are general, our implementation is built for the popular Generic Modeling Environment (GME) [30] and uses the XTEAM [19], [20] architecture modeling and analysis framework.

FLAME fundamentally re-architects and re-engineers an earlier version by the same name [6]. The earlier version did not support distribution (Section II-C) or guaranteed bounds on conflict prediction (II-D). FLAME also differs from prior tools in its *extensibility* and *operational granularity*. FLAME is extensible by providing explicit extension points through which it can interact with custom aspects of the architects' environments, including off-the-shelf modeling tools, languages, and analyses, such as consistency checkers. FLAME's operational granularity is that of *modeling operations*. FLAME's internal version control tracks every operation the architects enact (e.g., create, update, or remove modeling elements) and can detect conflicts after every operation. While traditional version control approaches rely on coarse-grained textual differences between model states, FLAME's finer granularity enables more precise conflict detection and allows identifying specific actions responsible for conflicts. FLAME tracks and synchronizes all modeling operations and makes the resulting synchronized models available for consistency analyses.

In a real collaborative design setting, FLAME will capture and process thousands of modeling operations, performed by varying numbers of architects. For the purpose of the discussion in this section, we offer a simplified, illustrative scenario involving nine such operations (O_1–O_9) collaboratively performed by three architects (A_1–A_3). For example, O_1 may be the addition of a component to a hardware host by A_1; O_2 may be the addition of an interface to the newly added component by A_1; O_3 may be the addition of a second component to the same host by A_2; O_4 may be the removal of the first component by A_3 from the system model in her local workspace; meanwhile, O_5 may be the modification of the first component's interface by A_2 in his local workspace; and so forth.

FLAME detects two kinds of conflicts: synchronization and high-order conflicts. Synchronization conflicts arise when multiple architects work on models in parallel and the sets of operations they perform cannot be synchronized, or merged, to form a single model. For example, operations O_4 and O_5 above cannot be synchronized at least in architect A_3's workspace: she removed the same component that architect A_2 had concurrently modified. High-order conflicts arise when the operations can be synchronized, but the resulting model violates a consistency rule, non-functional property requirement, or another system constraint. For example, the addition of two components in the above scenario (operations O_1–O_3) may violate a memory or energy usage constraint [21] on the host in question.

Synchronization conflicts are also called context-free conflicts [58] in design, and are analogous to textual conflicts [13] and direct conflicts [18], [26], [51], [60] at the level of source code. High-order conflicts are also called context-sensitive conflicts [58] in design, and are analogous to higher-order conflicts [13] and indirect conflicts [18], [26], [51], [60] at the code-level. By definition, all possible conflicts are either synchronization (version control can detect them) or high-order (version control cannot detect them). FLAME detects both these kinds of conflicts and can detect all possible high-order conflicts for which there exists an analysis implementation that computes whether a model satisfies the rule, requirement, or constraint potentially violated by the conflict.

Figure 1 depicts FLAME's architecture. To be used with a new modeling tool and analysis, one needs to write an *editor adapter* for the modeling tool and an *analysis adapter* for the analysis. The editor adapter plugs into the modeling tool and captures all model-editing operations performed by an architect; the analysis adapter applies the model-editing operations to the analysis tool's internal representation of the model. The rest is handled automatically by FLAME. We have implemented these adapters for the editors and analyses implemented within the XTEAM modeling framework [19], [20] built on top of GME [30], including energy consumption, memory usage, and

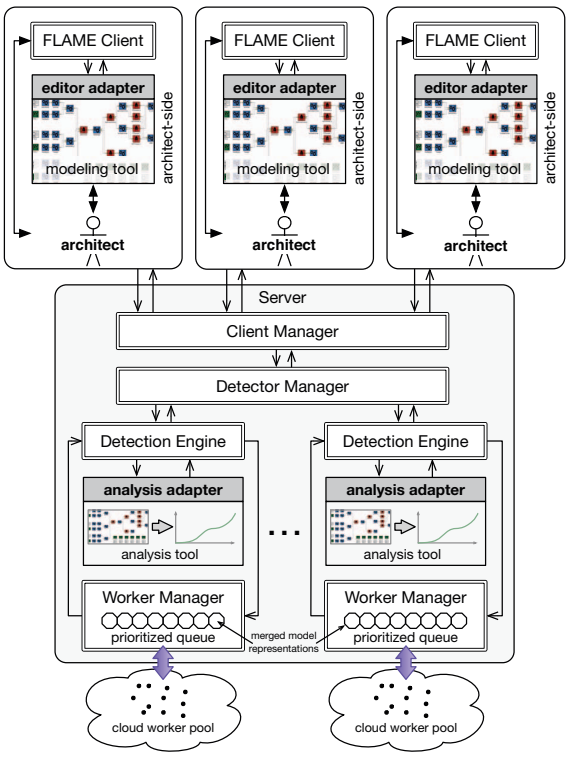


Fig. 1: FLAME architecture.

message latency analyses. This required a total of fewer than 6,500 lines of Java and C++ code. Creating new adapters is greatly simplified by the existence of frameworks that capture architects' operations and make them available via APIs (e.g., Solstice [39] for Eclipse). Aside from capturing and applying these operations, FLAME automatically manages operation version control, synchronization, and conflict detection: The editor adapters capture the operations, and the FLAME Clients communicate the operations to the Client Manager on a server. The Detector Manager gets the operations from the Client Manager and distributes them to conflict Detection Engines. Each Detection Engine has its own merging strategy (Section II-B) and can provide different conflict awareness to architects. For example, one Detection Engine may synchronize all modeling operations for a small set of architects working closely together, while another may synchronize all architects' modeling operations, but only once those architects have formally committed them. The Detection Engine synchronizes the architects' operations to detect synchronization conflicts and uses the analysis adapters to run the analyses on the successfully synchronized models, detecting high-order conflicts. Worker Managers may automatically offload the analysis computation onto worker nodes, such as clouds (Section II-C). Finally, FLAME's analysis adapters collect and consolidate the analysis results and report conflict information back to the architects.

Next, we describe the details of FLAME's version control (Section II-A), merging strategies (II-B), conflict detection distribution (II-C), and detection prioritization algorithm (II-D).

A. Version Control in FLAME

Modern version control (e.g., Git, Mercurial, and Subversion) stores differences in textual representations of files (e.g., source code or architectural models) and requires manual checkpointing, producing a coarse-grained history [41]. Automated, fine-grained version control records every developer action, but still uses textual differences, albeit with metadata [14], [34], [41], [45], [61]. FLAME introduces a real-time version control that automatically records every architect operation and enables immediate synchronization. This real-time synchronization enables (1) continuous analysis execution even when an architect is the only one working and (2) continuous proactive detection of synchronization and high-order conflicts.

FLAME encapsulates both modeling and manual synchronization operations (e.g., importing another architect's changes) in Design Event objects, which capture the operation, the performing architect, and a unique, sequentially ordered event ID. For example, in our scenario above, two Design Event objects would be $\{O_4 : RemoveComponent(C_1),\ A_3,\ E_4\}$ and $\{O_5 : ModifyInterface(C_1.I_1),\ A_2,\ E_5\}$. In real-time, FLAME shares the Design Events with all FLAME Clients and Detection Engines, each of which stores the Design Events in Event Queues, one queue per architect. That enables the architects to import others' operations and the Detection Engines to synchronize the relevant architects' operations to detect potential conflicts. FLAME's version control can export the per-operation history of the architects' modeling operations, and these operations can be used to examine, replay, and manipulate the history, e.g., by selectively undoing operations [14], [41], [61].

FLAME's version control, in effect, combines two approaches to collaborative editing: real-time operation-based group editing [23], [53] (used by, e.g., Google Docs [25]) that continuously handles fine-grained conflicts; and traditional version control (e.g., Git, Mercurial, and Subversion) that supports individual editing workspaces and subsequent, coarse-grained merging.

B. Merging Strategies

FLAME's Detection Engines can employ a variety of merging strategies for their proactive conflict detection (PCD). For example, an architect may be interested if her design conflicts with specific colleagues, if merging all operations by all the architects in her group leads to a conflict, or if merging formally *committed* versions will lead to a conflict. Different merging strategies aid different kinds of collaborative awareness and are appropriate for different collaborative scenarios.

As exemplars, we developed two merging strategies, captured in two Detection Engines: Global Engine and Head-and-Local Engine. The Global Engine merges all operations performed by all architects locally, whether formally committed or not. For example, in our scenario above, this would mean all nine operations (O_1–O_9). The result is the most current design that may give PCD the most predictive power, albeit at the risk of detecting false-positive conflicts that do not materialize because the architects may revert uncommitted operations. The Head-and-Local Engine merges an architect's latest operations with

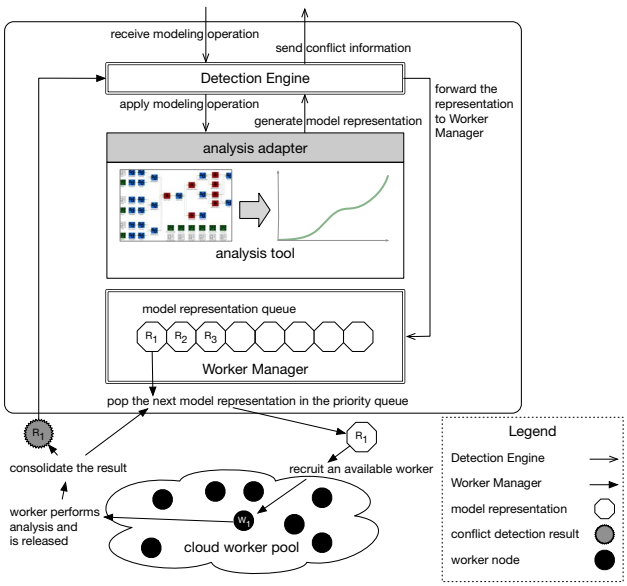


Fig. 2: The Detection Engine is responsible for synchronizing the model, employing the Worker Manager to deploy the analysis on worker nodes, and reporting the analysis result.

all the other architects' *committed* operations. For example, in our scenario, this may include only the operations performed by architects A_1 and A_2, if A_3 has not yet committed her operations (e.g., O_4). This strategy reduces false positives, but delays the detection of some conflicts (e.g., the synchronization conflict between O_4 and O_5 discussed above).

We use these two merging strategies in our evaluation (Sections III and IV), but FLAME also allows many other strategies.

C. Conflict Detection Offloading

Because continuously executing model analyses can be expensive, FLAME may need to use external resources (e.g., the cloud). An overtaxed Detection Engine may delay processing analyses and reporting conflict detection results back to the architects, especially when the team size or system complexity grow. We observed this in our user experiments (Section III).

FLAME mitigates this risk by splitting the work onto multiple Detection Engines, which may be offloaded from the server, and allowing each engine to execute the analysis computation locally or remotely. FLAME does not parallelize each analysis execution (unless the analysis implementation is already parallelized), but can offload multiple analysis executions, one for each version of a model, onto multiple worker nodes.

Figure 2 summarizes the Detection Engine's architecture and conflict detection's control flow. The engine follows its merging strategy to apply architect operations to the model (e.g., operations O_1–O_5 in our scenario above), generates model representations for the Worker Manager to deploy analyses on, and, once it receives the result, communicates the conflict information back to the Detector Manager (recall Figure 1), which then communicates it with the architect via the Client Manager. The Worker Manager uses queue prioritization (Section II-D) to select a model representation, deploys its analysis onto a worker node, and relays the result to the Detection Engine.

D. Queue Prioritization

When architects work actively, the rate of new operations may be high and the queues of model representations may get long. Given an unlimited number of workers, these representations can be processed in parallel. However, practical limits on the worker pool are likely, and processing the representations in the order of the operations that produce them may introduce a significant delay before conflicts are discovered. If the frequency of operations is higher than the frequency with which analyses can be computed, this delay will grow with time.

FLAME implements a novel prioritization algorithm for selecting which representations to analyze first. This algorithm bounds the time required to detect the high-order design conflicts to twice the running time of the analysis that finds the conflict. The algorithm takes advantage of FLAME's operation-based granularity and processes the chronologically newest conflict detection instances first, without any loss of the collaboratively generated design information.

In our scenario above, let us assume that all nine operations (O_1–O_9) are relevant to a Detection Engine, and that O_7 causes a high-order conflict with an earlier operation. The Detection Engine will generate nine representations (R_1–R_9, where R_1 has only O_1 applied, R_2 has O_1 and O_2 applied, etc.), but if the Worker Manager only has one worker node at its disposal, its options are to simultaneously start the analysis of all nine representations, simultaneously start the analysis of a subset of the representations, or analyze one representation at a time, in some order. The former two options delay learning about the conflict result. Analyzing k representations in parallel may take kt time to complete, where t is the average time the analysis takes to compute (ignoring the context switching overhead). Moreover, for $k < 7$, this analysis may miss the conflict (if R_7, R_8, and R_9 are not in the chosen subset). On the other hand, the time to discover the conflict when analyzing the representations serially in some order depends on the order. The R_1, R_2, R_3, ... order will take $7t$ time to discover the conflict. Instead, FLAME reverses the order of the representations, and analyzes R_9 first. (After completing the R_9 analysis, FLAME analyzes R_8 if no new operations have taken place, or the newest representation, i.e., R_{9+n}, if there were n new operations.) Two outcomes are possible when analyzing R_9. Either FLAME discovers the conflict in total time t, or one of O_8 and O_9 (or O_8 and O_9 together) already resolved the conflict, in which case FLAME will avoid reporting the false positive conflict.

In this scenario, prioritizing the newest representation computed the conflict in the minimal time t necessary for just one analysis. Generally, because interrupting ongoing analyses could result in livelock, FLAME must finish an ongoing analysis computation before starting a new one. Thus, in the worst case, this queue prioritization scheme results in time $2t$ before a conflict is discovered. This scheme scales well with the available worker pool, always guaranteeing the $2t$ bound. Section IV-C evaluates this bound with real-world workloads.

task	measure	control	PCD
CMAC	energy consumption (MJ)	8.18	8.55
	memory usage (MB)	729	748
BOINC	subcomputations completed	593	600

Fig. 3: PCD's effect on design quality, as defined by throughput, and proxied by energy and memory usage, and subcomputations completed. (Higher measures represent higher quality.)

III. Utility Evaluation

We conducted two controlled user experiments to measure if proactive conflict detection (PCD) has a positive effect on (1) design quality and (2) design activity efficiency. All experimental data can be found at http://flamedesign.org.

The two experiments included 90 participants, all professional Masters students at the University of Southern California taking a Software Architectures course. In each experiment, students worked in pairs to perform a design task. All teams used FLAME, but a randomly selected half of the teams (experimental group) had PCD notifications activated, while the other half (control group) had them disabled. Having all teams use FLAME controlled the design environment variables that could affect the collaboration and allowed us to detect when conflicts occurred, even if the architects were not notified.

The two experiments differed in design tasks and FLAME's merging strategies (recall Section II-B). Both design tasks involved a real-world, open-source system. The architects were asked to improve the throughput of a partial system model while being mindful of the system's energy consumption, memory usage, and message latency. The two participants in each team worked on two non-overlapping parts of the model (e.g., one architect worked on the server and the other on the client).

The first experiment had 42 participants, used the NASA CMAC system [35], [42], and relied on the Global merging strategy. CMAC is a climate analysis framework used to inform stakeholders who make policy decisions involving the weather, climate, tourism, water resources management, food management and security, etc. CMAC combines remotely sensed observations from space with climate model simulation. The second experiment had 48 participants, used the BOINC system [56], and relied on the Head-and-Local merging strategy. BOINC is a popular open-source system for volunteer-computing and grid-computing used for SETI@home, Folding@home, and similar projects [3]. Both CMAC and BOINC are well-known, popular systems with publicly available source code and design documents. The three system properties — energy consumption, memory usage, and latency — are integral to both systems. The XTEAM analysis framework [21] integrated into FLAME implements analyses for all three properties.

No participant had prior experience with FLAME, CMAC, or BOINC, but all participants spent four weeks leading up to the experiment performing two software design exercises using the CMAC- or BOINC-specific modeling environment and XTEAM to gain familiarity with its simulation-based analyses. This yielded comparable familiarity of participants with the modeling environment and target system domain [17]. A pre-experiment survey of participants' industrial experience showed no difference between the experimental and control groups.

task	measure (per team)	control	PCD
CMAC	modeling operations	48.2	60.8
	communication activities	11.0	19.5
	synchronizations	6.3	8.0
BOINC	modeling operations	29.6	38.5
	communication activities	14.3	15.0
	synchronizations	5.3	7.3

Fig. 4: PCD's effect on architect efficiency.

Each team participated in a 2-hour session: a 1-hour FLAME tutorial, a 30-minute design session (all design activities were recorded and all communication logged), and a 30-minute design session for the participants to experience the alternative FLAME mode (i.e., participants without PCD experienced such notification, and vice versa). The alternative-mode session was not recorded, but it enabled an exit survey to collect the participants' preferences and assess experience differences.

A. Results

We answered three research questions regarding the effect of PCD on design quality and design activity efficiency.

RQ1: How did PCD affect resulting design quality? Both experimental tasks used throughput as the measure of quality, but they required measuring throughput differently. The CMAC task's throughput was proxied by energy consumption and memory usage; higher measures indicated higher throughput. The BOINC task's throughput was proxied by the number of subcomputations completed in a fixed amount of time. Figure 3 shows that groups with PCD had uniformly higher-quality designs for all tasks. For each team and quality proxy, we selected the team's best design. The CMAC teams with PCD, as compared to the control teams, on average, had statistically significantly higher memory usage with a large[1] effect size (Welch test $p = 0.037$, Cohen's $\delta = 0.880$), as well as higher energy consumption, although the latter difference was not statistically significant. The BOINC teams with PCD, on average, completed a higher number of subcomputations. This difference was not statistically significant. However, one of the PCD teams neglected to use FLAME's conflict notifications (their design activity reflected no increase in modeling operations). Excluding that team from the data showed that BOINC teams with PCD had a statistically significantly higher model quality with a medium effect size (Welch test $p = 0.076$, Cohen's $\delta = 0.612$) than the control group.

RQ2: How did PCD affect architect efficiency? Figure 4 summarizes PCD's effect on architect efficiency. We measure efficiency in three ways, via numbers of modeling operations, communication activities, and model synchronizations in the 30-minute session. The PCD teams had uniformly higher efficiency for all tasks. The CMAC teams with PCD, on

[1]We use the standard Cohen's definition for what effect sizes are considered small (≥ 0.2), medium (≥ 0.5), and large (≥ 0.8). [16]

task	measure	control	PCD
CMAC	Detected conflicts at synchronizations per team	1.27	2.40
	Teams with unresolved conflicts at session end	3/11	0/10
	High-order conflict lifetime (sec.)	671	363
BOINC	Detected conflicts at synchronizations per team	2.50	0.92
	Teams with unresolved conflicts at session end	4/12	2/12
	High-order conflict lifetime (sec.)	256	150
	Conflicts resolved without synchronizing	28%	40%

Fig. 5: PCD's effect on high-order design conflicts.

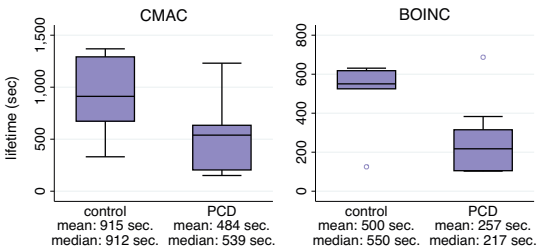


Fig. 6: Distribution of lifetimes of conflicts that lived longer than 100 seconds. Boxes show first, second (median), and third quartiles; whiskers show lowest and highest points within 1.5 times the interquartile range of lower and upper quartiles.

average, performed a higher number of modeling operations with a medium effect size (Welch test $p = 0.085$, Cohen's $\delta = 0.645$) and communicated more frequently with a medium effect size (Welch test $p = 0.066$, Cohen's $\delta = 0.697$) than the control teams. The number of synchronization activities, while higher for PCD teams, was not statistically significantly higher. The BOINC teams with PCD, on average, performed a higher number of modeling operations with a large effect size (Welch test $p = 0.031$, Cohen's $\delta = 0.800$) than the control teams. The amount of communication and the number of synchronization activities, while higher for PCD teams, were not statistically significantly higher. The increase in communication frequency is consistent with a prior study of source-code conflicts [49].

In the exit survey, the architects from both experiments remarked on PCD's effect on their productivity: *"[PCD] increased productivity as we were able to try more combinations [of modeling operations] in same amount of time."* and *"By comparison, [in the PCD mode] we have tried more combinations of the options (modeling operations) [...], and I think it means that our working efficiency has been improved."*

The increase in the number of performed operations can be explained by the increase in the participants' confidence in making new changes. Fear of conflicts could cause an engineer to avoid performing new operations [13]. Two participants remarked: *"Our confidence that the combined design would meet the requirements was much higher when using PCD."* and *"[PCD] helps me focus more on the design and modeling rather than worrying about the effect of a single change."* The participants identified fear as a deterrent to progress: *"Without PCD, deciding when to commit local changes was a challenge... None of us wanted to commit a change which would cause the merged model to violate requirements."*

RQ3: How did PCD affect resolving high-order design conflicts? Figure 5 summarizes PCD's effect on high-order conflicts. While the CMAC teams with PCD dealt with more conflicts on average, no conflict was left at the last commit and at the end of the session, whereas three control teams left such conflicts. The average lifetime of the conflicts was shorter for the PCD group than for the control group, although the difference was not statistically significant. The BOINC teams with PCD also had a shorter average conflict lifetime, although again not statistically significantly shorter. However, not all conflicts are created equal. Some high-order conflicts require more effort to resolve than others. Figure 6 summarizes the distributions of these *hard* conflicts whose lifetimes exceeded 100 seconds. (Because these distributions were not normal, we relied on nonparametric tests.) For both the CMAC and the BOINC experiments, the teams with PCD were able to resolve these hard conflicts faster, on average, than the control teams (for CMAC teams, 539 vs. 912 seconds, statistically significant difference and large effect size, Mann-Whitney-Wilcoxon test $p = 0.036$, common language effect size 0.836; for BOINC teams, 218 vs. 551 seconds, statistically significant difference with a medium effect size, Mann-Whitney-Wilcoxon test $p = 0.050$, common language effect size 0.794). These results are consistent with those reported for source-code conflicts [26], [49]. Architects observed: *"It was quicker and easier to detect conflicts and fix them immediately."* and *"[PCD was] making it easier to identify errors and fix them before further changes are made."*

For BOINC, teams with PCD resolved conflicts without synchronizing their models more often than the control group, although the difference was not statistically significant. We attribute this phenomenon to the Head-and-Local merging strategy (recall Section II-B), which provides architects with awareness about conflicts involving their own uncommitted operations. This allows the architect to foresee conflicts in the working copy before committing and to resolve them (or abandon the conflicting operations) quickly. One architect observed: *"We can see whether an individual's atomic step can produce a high-order conflict and quickly rollback that operation."*

Finally, our exit survey asked the architects about their preferences regarding and against PCD, using the 7-point Likert scale (Figure 7). The participants overwhelmingly preferred using PCD (mean of 6.22; 7 indicates strong agreement), and

Question	mean	σ
I preferred the FLAME mode with PCD.	6.22	1.25
FLAME's PCD helped me deal with design conflicts.	6.15	1.03
It was difficult to understand the conflict detection information that FLAME provided.	2.80	1.60
Early detection of conflicts eased resolution.	6.07	0.88
The FLAME GUI was distracting.	2.48	1.44

Fig. 7: Five Likert-scale questions asked of the architects in an exit survey. The scale ranged from 1 (strongly disagree) to 7 (strongly agree). The standard deviation is denoted σ.

felt that it helped deal with conflicts (mean of 6.15) and made resolution easier (mean of 6.07). The participants were also favorable to FLAME's interface, but less strongly than to PCD.

Overall, teams with PCD designed higher-quality models, more efficiently, resolving more high-order conflicts faster.

B. Threats to Validity

Our study subjects were students. We mitigated this by relying on students in a professional graduate program and providing four weeks of training in the design environment and the subject systems. *The design tasks may not generalize.* We mitigated this by using popular, open-source projects with available design documents, and by creating realistic collaborative design scenarios based on those design documents. *We used resource consumption to approximate throughput and operations, communication, and synchronizations to approximate productivity.* These are reasonable proxies in the context of our studies. *We limited team sizes to two architects to help control task complexity.* This may not generalize to large-team design, but was necessary for the feasibility of the controlled experiment. *We did not vary the time FLAME analyses took to complete consistency checks*, while in a real-world setting, variation may influence the architects' reactions to PCD. *Our experiments used a single design environment and a small number of consistency analyses*, while a real-world setting may include many environments and analyses. We mitigated this in part by using three different consistency checking tools and by using a popular GME-based design environment. *The 30-minute design session length hides long-surviving conflicts.* However, this means that our measurements of the conflict lifetime are conservative underestimates, and the actual benefit of PCD may be greater than demonstrated.

IV. Performance Evaluation

This section describes the evaluation of FLAME's performance, including how much FLAME's conflict detection offloading reduces detection time, FLAME's scalability to large architect teams, and effects of FLAME's queue prioritization algorithm with respect to the theoretical worst-case bound.

A. Conflict Detection Offloading

FLAME's Detection Engines adapt to the variation in workload by offloading conflict detection to worker nodes (recall Section II-C). This reduces the delay that occurs when there are multiple model representations queued up to be analyzed.

To evaluate the effect of offloading on the delay, we conducted a study varying the number of worker nodes and measuring the *conflict detection time* — the time from the moment the operation is made until the analysis of the model representation including that operation completes — using the same set of model representation analyses. To properly measure the effect of offloading, we disabled the queue prioritization algorithm (recall Section II-D). Section IV-C evaluates that algorithm.

We used the collaborative architect behavior recorded during the BOINC experiment (recall Section III) to accurately represent the workload. FLAME's version control allowed us to record and replay the design activities, varying the size of the worker pool. Overall, the recorded behavior had 1,339 operations, and thus model representation analyses. We used the Google Compute Engine (GCE) [24] as the worker pool with nodes that had 2 virtual cores and 1.80 GB of memory.

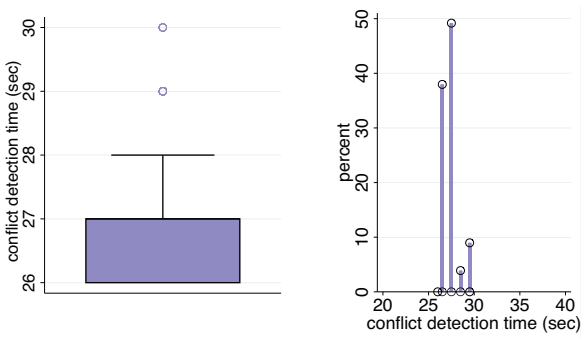


Fig. 8: Optimal conflict detection times for the 1,339 model representation analyses during the BOINC experiment from Section III. Mean time was 26.8s. Box shows first, second (median), and third quartiles; whiskers show lowest and highest points within 1.5 times the interquartile range of lower and upper quartiles. (Lower whisker overlaps with first quartile.)

We measured the optimal conflict detection time by manipulating the recorded design activity to ensure that all prior model representation analyses had finished processing before each new design operation. Figure 8 shows the distribution of optimal conflict detection times. The mean was 26.8s, median 27.0s, and maximum 30.0s, constituting a baseline for comparison.

Given that the longest analysis computation took 30s, we checked all 30s intervals of the recorded design activities and found that the architects never performed more than 9 operations, a rough upper limit on the number of workers needed to maximize parallelization. To be sure that extra workers would not help, we ran four experiments, giving FLAME access to 2, 4, 8, and 12 GCE worker nodes.

We sequentially replayed the recorded design activity from each of the 24 teams in the BOINC experiment on FLAME with access to each of the four worker pools. Figure 9 shows histograms of the detection times for different worker pool sizes. As expected, larger worker pools led to faster computations.

There are two measures one may wish to minimize: maximum detection time and mean detection time. The maximum detection time decreases from 133s for 2 workers, to 54s for 4 workers, and to 32s for 8 workers; there was no benefit for worker pools larger than 8. This suggests that reducing the maximum detection time to the optimal detection time requires using nearly as many workers as the maximum number of operations that take place within the time it takes to complete one analysis. However, reducing the mean detection time requires far fewer resources. The mean detection time decreases from 36.9s for 2 workers, to 27.5s for 4 workers; there was no benefit for worker pools larger than 4. Here, 4 workers were sufficient to reach nearly the optimal mean detection time.

This experiment demonstrates that FLAME's conflict detection offloading works as expected and that the mean detection time can be reduced to nearly the optimal with fairly few extra

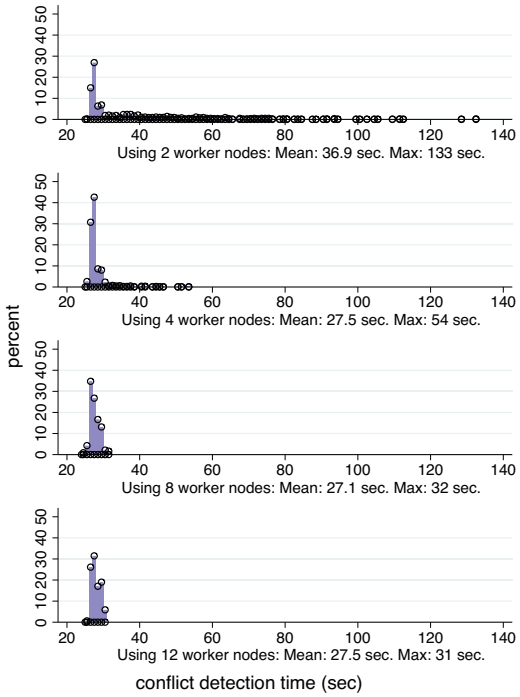


Fig. 9: Conflict detection times with varying worker pools.

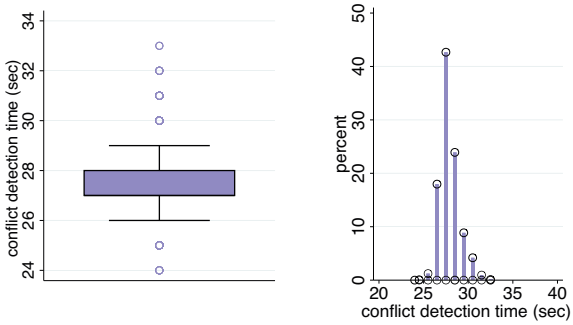


Fig. 10: Combined conflict detection times for the ten scenarios involving 24-architect teams, with FLAME using 48 GCE worker nodes. Mean time was 27.4s and maximum time was 33s. Box shows first, second (median), and third quartiles; whiskers show lowest and highest points within 1.5 times the interquartile range of lower and upper quartiles.

computational resources. Since architect activity may be hard to predict in some domains, and since cloud resources are elastic, FLAME's mechanisms for dynamically adding worker nodes at runtime are likely to be helpful in those domains.

B. Scalability

We used the recorded design activity of the 48 architects involved in the BOINC experiment to measure how FLAME's worker node management overhead scales to large collaborative teams. The rate of modeling operations grows with the size of the architect team. Since each operation generates a model representation for analysis, larger teams are likely to lead to more PCD computations. As Section IV-A showed, a larger node worker pool can help manage this computation and keep the conflict detection time low; however, managing the larger worker pool may increase FLAME's overhead, potentially introducing additional delay in conflict detection. Again, to properly measure the effect of scaling, we disabled the queue prioritization algorithm (recall Section II-D) for this evaluation. Section IV-C will evaluate that algorithm.

Using the recorded design activity, we randomly selected 12 of the 24 teams and merged their activity. We repeated this process ten times to generate ten logs, each representing a collaborative design scenario of a team of 24 architects. The logs included mutually incompatible modeling operations resulting in synchronization and high-order conflicts. Since our focus was on high-order conflicts, and since synchronization conflict detection is far less computationally costly, we replaced synchronization-conflict-causing operations with no-ops, which still created new model representations, preserving the total number of operations (and model representations) FLAME had to process. This approach provided a reasonable trade-off between the logistics of running and recording a 24-architect study and generating fully artificial activity logs. The resulting logs included real architect behavior and, if anything, overestimated how much activity may take place in a 24-architect team, as extra coordination in such a team is likely to reduce the activity, as compared to twelve 2-architect teams.

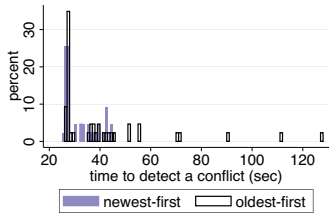
We replayed the resulting logs in FLAME with a 48-GCE-node (2 virtual cores and 1.80 GB of memory each) worker pool to measure the overhead FLAME's cloud management creates for a large team of architects using FLAME with many cloud nodes. To compare the conflict detection times to those for the 2-architect scenarios from Section IV-A, Figure 10 shows the distribution and histogram of the conflict detection times for the ten 24-architect scenarios. While the numbers of analyses and worker nodes were much higher, the mean and maximum conflict detection times were very similar to the optimal times for the 2-architect scenarios (recall Figure 8). This suggests that FLAME's additional cloud management overhead is negligible for teams of up to 24 architects, as long as sufficient cloud resources are available. The availability of commodity cloud infrastructures makes FLAME's use of third-party worker nodes practical and inexpensive.

C. Queue Prioritization

FLAME prioritizes the order in which it analyzes model representations to minimize conflict detection time (recall Section II-D). This section empirically analyzes the effect of prioritizing newer representations vs. analyzing the representations in their chronological order. Section II-D argued that the *newest-first* policy had a worst-case bound of $2t$ on the conflict detection time, where t is the time it takes to analyze a single representation. Meanwhile, the *oldest-first* policy (chronological order) has no theoretical bound.

To measure the effect of prioritization on the conflict detection time, we replayed the workloads from the BOINC experiment (recall Section III) on two FLAME configurations, one with newest-first prioritization and the other with oldest-

(a) 1 worker node
newest-first: mean: 38.7s median: 39s max: 53s
oldest-first: mean: 92.5s median: 77s max: 267s



(b) 2 worker nodes
newest-first: mean: 31.5s median: 27s max: 45s
oldest-first: mean: 40.9s median: 35s max: 128s

Fig. 11: Histograms of the conflict detection times using (a) 1 and (b) 2 worker nodes. Theoretical worst-case bound on the conflict detection time using the newest-first prioritization is $2t = 60$s.

first prioritization (i.e., no prioritization). The configurations were otherwise identical. We ran these experiments with a pool of 1 worker node, and a pool of 2 worker nodes. Since Section IV-A showed that using only 2 worker nodes causes significant delays in conflict detection time (and this delay would be even greater for a single worker node), these configurations are sufficient to reveal the effects of prioritization.

Figure 11 shows that the newest-first prioritization substantially decreases the conflict detection times with a medium or large effect size, compared to the chronological, oldest-first prioritization scheme (Mann-Whitney-Wilcoxon tests, $p < 0.001$ and common language effect size 0.812 for 1-worker scenario, $p = 0.023$ and common language effect size 0.640 for 2-worker scenario). For the 1-worker scenario, mean time decreases from 92.5s to 38.7s (58% improvement), and the maximum time decreases from 267s to 53s (80% improvement). For the 2-worker scenario, mean time decreases from 40.9s to 31.5s (23% improvement), and maximum time decreases from 128s to 45s (65% improvement). The positive benefits of prioritization increase in more resource-constrained scenarios. In all cases, the maximum conflict detection times with newest-first prioritization — 53s and 45s — were well below the $2t = 60$s theoretical worst-case bound from Section II-D.

Conflict detection with newest-first prioritization may identify conflicts in model representations subsequent to the first representation that reveals the conflict (recall Section II-D). In the 1-worker scenario, this happened for 60.5% of the conflicts, and in the 2-worker scenario, for 32.6% of the conflicts. This is more likely to occur in resource-constrained scenarios because more analyses are delayed in those scenarios.

Overall, the newest-first queue prioritization scheme yielded significant benefit in resource-constrained environments, and incurred virtually no cost even for unconstrained environments.

V. Related Work

Source-code conflicts are common in collaborative development [12], [13], [26], [50]. FLAME is the first to tackle design-level conflicts. CollabVS [18], Palantír [51], Safe-commit [60], Crystal [11], [12], [13], WeCode [26], and Syde [27], [28], [33] have shown the positive impact of proactive, continuous code-level conflict detection. FLAME uses speculative analysis [10], [12] to predict architects' actions and to inform the architects of consequences of those actions early.

General-purpose VCSs work poorly for software design models [31], [36], [44]. They are inflexible, inextensible, and limited to specific design environments [1]. Operation-based, real-time group editors [7], [25] provide a shared workspace, but this discourages collaboration: operations that prevent model analysis completion prevent all collaborators from analyzing the model. By contrast, FLAME provides an individual workspace to each architect. Operation-based conflict detection techniques monitor the sequence of performed modeling operations to detect design conflicts [7], [8], [31]. However, these techniques target cheap-to-compute conflicts, while FLAME's focus is on computationally expensive design conflict detection. These techniques are complementary and can be integrated into FLAME. Finally, AMOR [1], [9] is a VCS that aids collaborative design but does not proactively detect conflicts.

Automatically merging design models may reduce synchronization conflicts [43], [55], but not necessarily high-order conflicts. This paper focused on using model analyses from prior work [4], [15], [22], [37], [48], [52], [59]. FLAME natively extends to all analyses implemented for XTEAM [21], and its ideas apply to all architectural analyses.

Our work has been inspired by techniques that continuously perform software analyses in the background. Argo/UML's [46] critics provide continuous design review and improvement suggestions as architects make modeling changes. Argo/UML throttles CPU resources used for model analysis to avoid draining the local computation resources, which may delay computing the analysis results and is not ideal for long-running analyses. Codebase Replication [38], [39] is a framework for turning offline analyses into continuous ones in Eclipse by maintaining an up-to-date replica of the source code on which to perform the analyses. By contrast, FLAME targets often computationally expensive design-level conflicts caused by modeling changes made by multiple architects in parallel, as opposed to local inconsistencies.

VI. Contributions

FLAME, a proactive, continuous design conflict detection framework, informs software architects of newly arising design conflicts that would otherwise remain hidden. This enables well-informed decisions and early conflict resolution while relevant details are fresh in the architects' minds, and prevents undoing and redoing work. FLAME uses cloud resources and

detection prioritization to guarantee a bound on the conflict detection time, without disturbing the ongoing collaborative design activities. Two controlled experiments demonstrate that architects who use FLAME produce higher-quality designs, repair conflicts faster, and enjoy using FLAME. Empirical evidence supports claims that FLAME efficiently uses cloud resources, scales to large architect teams, and complies with the theoretical worst-case bound on conflict detection time. Our results suggest that practitioners may benefit from FLAME.

REFERENCES

[1] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR — Towards adaptable model versioning. In *Workshop on Model Co-Evolution and Consistency Management*, volume 8, pages 4–50, 2008.

[2] K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *IJWIS*, 5(3), 2009.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *CACM*, 45(11), 2002.

[4] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *TSE*, 30(5), 2004.

[5] J. Bang, I. Krka, N. Medvidovic, N. Kulkarni, and S. Padmanabhuni. How software architects collaborate: Insights from collaborative software design in practice. In *CHASE*, 2013.

[6] J. Bang and N. Medvidovic. Proactive detection of higher-order software design conflicts. In *WICSA*, 2015.

[7] J. Bang, D. Popescu, G. Edwards, N. Medvidovic, N. Kulkarni, G. M. Rama, and S. Padmanabhuni. CoDesign: A highly extensible collaborative software modeling framework. In *ICSE demo*, 2010.

[8] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *ICSE*, 2008.

[9] P. Brosch, M. Seidl, K. Wieland, M. Wimmer, and P. Langer. We can work it out: Collaborative conflict resolution in model versioning. In *ECSCW*, 2009.

[10] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: Exploring future states of software. In *FoSER*, 2010.

[11] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Crystal: Precise and unobtrusive conflict warnings. In *ESEC/FSE demo*, pages 444–447, 2011.

[12] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *ESEC/FSE*, 2011.

[13] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Early detection of collaboration conflicts and risks. *TSE*, 39(10), 2013.

[14] A. G. Cass and C. S. T. Fernandes. Modeling dependencies for cascading selective undo. In *CHI*, 2005.

[15] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. Multi-valued symbolic model-checking. *TOSEM*, 12(4), 2003.

[16] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 2nd edition, 1988.

[17] D. Damian, R. Helms, I. Kwan, S. Marczak, and B. Koelewijn. The role of domain knowledge and cross-functional communication in socio-technical coordination. In *ICSE*, 2013.

[18] P. Dewan and R. Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *ECSCW*, 2007.

[19] G. Edwards. The eXtensible tool-chain for evaluation of architectural models. http://softarch.usc.edu/~gedwards/xteam.html, 2014.

[20] G. Edwards and N. Medvidovic. A methodology and framework for creating domain-specific development infrastructures. In *ASE*, 2008.

[21] G. Edwards, C. Seo, and N. Medvidovic. Model interpreter frameworks: A foundation for the analysis of domain-specific software architectures. *JUCS*, 2008.

[22] A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *TSE*, 37(2), 2011.

[23] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD*, 1989.

[24] Google Compute Engine. https://cloud.google.com/compute/, 2015.

[25] Google Docs. https://docs.google.com, 2014.

[26] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *ICSE*, 2012.

[27] L. Hattori and M. Lanza. Syde: A tool for collaborative software development. In *ICSE demo*, 2010.

[28] L. Hattori, M. Lanza, and M. D'Ambros. A qualitative user study on preemptive conflict detection. In *ICGSE*, 2012.

[29] J. D. Herbsleb. Global software engineering: The future of socio-technical coordination. In *FOSE*, 2007.

[30] Institute for Software Integrated Systems, Vanderbilt University. Generic modeling environment. http://www.isis.vanderbilt.edu/projects/gme, 2014.

[31] M. Koegel, J. Helming, and S. Seyboth. Operation-based conflict detection and resolution. In *Workshop on Comparison and Versioning of Software Models*, 2009.

[32] P. Kruchten. The software architect. In *Software Architecture*. 1999.

[33] M. Lanza, L. Hattori, and A. Guzzi. Supporting collaboration awareness with real-time visualization of development activity. In *CSMR*, 2010.

[34] B. Magnusson, U. Asklund, and S. Minör. Fine-grained revision control for collaborative software development. In *FSE*, 1993.

[35] C. A. Mattmann, C. S. Lynnes, L. Cinquini, P. M. Ramirez, A. F. Hart, D. Williams, D. Waliser, and P. Rinsland. Next generation cyberinfrastructure to support comparison of satellite observations with climate models. In *BiDS*, 2014.

[36] A. Mehra, J. Grundy, and J. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE*, 2005.

[37] T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *MoDELS*, 2006.

[38] K. Muşlu, Y. Brun, M. D. Ernst, and D. Notkin. Making offline analyses continuous. In *ESEC/FSE*, 2013.

[39] K. Muşlu, Y. Brun, M. D. Ernst, and D. Notkin. Reducing feedback delay of software development tools via continuous analyses. *TSE*, 41(8), 2015.

[40] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. In *OOPSLA*, 2012.

[41] K. Muşlu, L. Swart, Y. Brun, and M. D. Ernst. Simplifying development history information retrieval via multi-grained views. In *ASE*, 2015.

[42] NASA. Computational Modeling Algorithms and Cyberinfrastructure (CMAC). https://www.earthsystemcog.org/projects/cmac/, 2013.

[43] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *ICSE*, 2007.

[44] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE*, 2005.

[45] D. Ohst and U. Kelter. A fine-grained version and configuration model in analysis and design. In *ICSM*, 2002.

[46] J. E. Robbins and D. F. Redmiles. Cognitive support, UML adherence, and XMI interchange in Argo/UML. *IST*, 42(2), 2000.

[47] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, 2004.

[48] R. Salay and M. Chechik. A generalized formal framework for partial modeling. In *FASE*, 2015.

[49] A. Sarma, D. Redmiles, and A. van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *FSE*, 2008.

[50] A. Sarma, D. Redmiles, and A. van der Hoek. Categorizing the spectrum of coordination technology. *Computer*, 2010.

[51] A. Sarma, D. F. Redmiles, and A. van der Hoek. Palantír: Early detection of development conflicts arising from parallel code changes. *TSE*, 2012.

[52] T. Schriber and D. Brunner. Inside discrete-event simulation software: How it works and why it matters. In *Winter Simulation Conference*, 2005.

[53] C. Sun and D. Chen. A multi-version approach to conflict resolution in distributed groupware systems. In *ICDCS*, 2000.

[54] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[55] S. Uchitel and M. Chechik. Merging partial behavioural models. In *FSE*, 2004.

[56] University of California. BOINC. http://boinc.berkeley.edu.

[57] M. Vakilian, A. Phaosawasdi, M. Ernst, and R. Johnson. Cascade: A universal programmer-assisted type qualifier inference tool. In *ICSE*, 2015.

[58] B. Westfechtel. Merging of EMF models. *SoSyM*, 13(2), 2014.

[59] J. A. Whittaker and M. Thomason. A markov chain model for statistical software testing. *TSE*, 20(10), 1994.

[60] J. Wloka, B. Ryder, F. Tip, and X. Ren. Safe-commit analysis to facilitate team software development. In *ICSE*, 2009.

[61] Y. Yoon and B. A. Myers. Supporting selective undo in a code editor. In *ICSE*, 2015.