Collaborative-Design Conflicts

Costs and Solutions

Jae young Bang, Kakao Corporation

Yuriy Brun, University of Massachusetts Amherst

Nenad Medvidović, University of Southern California

// Collaborative design exposes software architects to the risk of making conflicting modeling changes that either can't be merged or, when merged, violate consistency rules, nonfunctional requirements, or other system constraints. Proactive conflict detection can alleviate this risk. //

MODERN SOFTWARE SYSTEMS are often designed collaboratively by multiple software architects who make design decisions, document them in software models, and evolve the models as a team. This collaborative evolution is a complicated process, especially for large teams. To manage the high complexity, architects have adapted to using traditional copy-edit-merge-style version-control systems (VCSs) that enable parallel design in individual workspaces, synchronizing their work on demand.

However, architects might introduce two types of design conflicts—i.e., design changes that either

- conflict with each other and prevent merging the models or
- allow merging, but in a way that violates consistency rules, nonfunctional requirements, or system constraints.

Unfortunately, VCSs help discover only the first type, and only when architects attempt to synchronize their models. Discovery of the second type requires not only a merge but also that the architects elect to run the relevant analysis. When a conflict occurs, resolving it might require the architects to undo, redo, or even abandon their design and implementation work.

Proactive conflict detection (PCD) can alleviate the risk of design conflicts but requires tool support specific to collaborative design. Prior research on PCD at the code level[1] has shown that continuously making code-level analysis results available to developers[2] reduces conflict lifetime and improves developers' ability to make well-informed decisions. However, it is challenging to reuse existing PCD tools in collaborative design. They are not built to manage changes to graphical software models and are often limited to the specific development environments into which they are integrated. Moreover, many model analysis techniques are computationally intensive, and running them locally might disrupt the design activity. These challenges require a design-specific solution for collaborative conflict detection.

We present such a solution. Building on our prior research that presented the technical aspects and an extensive evaluation of PCD,[3] this article motivates the need for design conflict detection, describes its benefits to practitioners, and outlines the requirements for building detection tools. We

- define and classify design conflicts,
- identify the risks behind design conflicts and benefits to practitioners from PCD,

- discuss the features required for a collaborative-design environment aimed at PCD, and
- describe FLAME (Framework for Logging and Analyzing Modeling Events), our collaborative-design framework that interfaces with the architects' modeling and analysis tools to efficiently and continuously detect design conflicts.

Design Conflicts

To characterize design conflicts, we consider the following real-world case.[3] A team of architects was designing a large system. Although the team was distributed across three sites, a core group of senior architects physically colocated with the product manager for initial requirements analysis and architectural design. Once satisfied that the remaining design activities were appropriately divided, the core group members rejoined their original subteams. Each subteam proceeded to refine the design of its portion of the system, while, in parallel, development teams proceeded with implementation.

The architect teams captured the design using an in-house modeling tool. All design changes were saved into a shared VCS repository. The architects worked on design tasks alone or in small local groups. Design consistency was encouraged both locally, through daily status meetings and regular communication, and team-wide, through weekly videoconferences.

Despite the architects' best efforts, two types of issues arose regularly, requiring significant additional coordination among the architects and rework.

First, architects modified the design in mutually inconsistent ways. One 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. At the same time, 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 that it was unable to merge their changes.

Second, architects made local modifications that, when merged, violated nonfunctional properties. One example involved two teams trying to reduce message latency, via smart caching and pooling multiple payloads into a single message. Subsequent analysis showed that, together, these solutions sometimes increased latency and introduced unacceptably high memory consumption.

The previous two scenarios exemplify the two types of commonly occurring design conflicts: synchronization and high-order design conflicts. Synchronization conflicts (scenario 1) are mutually inconsistent design decisions that cannot be merged automatically by the VCS. High-order conflicts (scenario 2) are decisions that, once merged, violate one or more system requirements or constraints. Synchronization conflicts, also called context-free conflicts,[4] are analogous to textual[5] and direct[6] conflicts at the source code level. High-order conflicts, also called context-sensitive conflicts,[4] are analogous to higher-order[5] and indirect[6] conflicts at the code level.

PCD as a Solution

To ascertain the extent to which design conflicts are a real-world problem, we conducted interviews with 20 architects currently working at mature software companies.[7] Those architects reported that their companies might relocate architects to minimize the impact of geographic distribution. Colocation can simplify communication and integration, reducing conflicts and wasted effort.

One architect stated, "Architects sometimes need to travel to be colocated when the complexity of the current task is very high." The architects especially stressed the need for frequent communication early on: "The first one-third of the time is put into the frequent meetings." It was repeatedly stressed that collaborative design often causes costly conflicts: "We often face inconsistencies between components developed by different engineers. … Half of the cases lead to full-scale reverting to earlier stages."

Furthermore, the architects saw communication and integration as activities that must be managed carefully: "One of the responsibilities of a senior architect is to facilitate communication between the junior architects to establish a common perspective with minimal necessity of integration points." This motivates the research on continuous PCD. The architects repeatedly expressed openness to using advanced technologies during collaborative design.

PCD can benefit collaborating architects. As the interviews suggest, the risk of design conflicts lies in the significant additional cost incurred by repeated design effort, increased communication, and even geographic relocation of architects. Fear of conflicts might also cause an architect to avoid making new model changes.[5] The adoption of PCD into the collaborative-design environment can alleviate these issues by continuously feeding conflict information to the architects so that they can minimize the risk of undiscovered conflicts and make well-informed design decisions.

Prior research corroborates this. Two controlled experiments involving 90 participants suggest that using PCD leads to greater architect efficiency, faster conflict resolution, and higher-quality designs.[3] In the post-design surveys, the participants expressed that they preferred using PCD (with a mean of 6.22 and standard deviation of 1.25 on a 7-point Likert scale).

However, implementing PCD in an existing collaborative-design environment is not without challenges. The existing PCD tools for collaborative implementation are designed primarily to handle source code and cannot be readily applied to software models. Tools that are designed to manage textual changes made to code are known to not work well with graphical software models.[8,9]

Moreover, unlike code-level conflict detection, continuous PCD at the design level might be prohibitively expensive: many model analyses (e.g., discrete-event simulation,[10] Markov-chain-based reliability analysis,[11] queueing-network-based performance analysis,[12] and symbolic model checking[13]) are highly computationally intensive. Continuously running these analyses might overwhelm the machine on which the analyses take place, slowing down conflict detection as well as other processes. Slow analyses further exacerbate the problem by inducing increasing numbers of pending analysis instances.

Adoption of PCD

A collaborative-design environment that aims to effectively provide PCD to software architects must support three overarching capabilities:

- It must continuously share model changes among the architects.
- It must continuously perform model merges in the background.
- It must continuously analyze the design models as they evolve.

An existing, traditional collaborative-design environment can be transformed into one that provides PCD by implementing these three capabilities. We elaborate on each of them next.

Continuous Sharing of New Model Changes

The key to PCD is to continuously speculate on and simulate architects' synchronization actions (commits and merges), perform design analyses in the background, and have the analysis results available to the architects preferably before the need for those analyses arises. To implement this as an automated process, newly made model changes must be transferred out of the workspace in which they are made (the architect's local copy of the model) and must be available for analyses before architects explicitly initiate the analyses. This reduces the gap present in traditional VCSs between the time when a new conflict is introduced and when it can be detected.

It is crucial to note that this feature differs from the shared workspace that group editors (e.g., Google Docs; https://docs.google.com) provide. When using a VCS, architects initially perform a checkout from a repository to create local copies of the model. Those are loosely synchronized individual workspaces in which architects perform their design activities in parallel. The architects later merge their changes back to the repository.

Unlike a VCS, a group editor immediately merges each new change as it is made. When a group editor is used, workspaces are fully synchronized; all copies of the model are updated together every time an architect makes a change. However, this shared workspace might discourage collaboration because frequent model changes prevent model analysis completion. In other words, it is hard to work when someone else is changing the model on you all the time.

A VCS with automatic sharing of changes allows architects to design in their individual workspaces and encourages parallel work. It enables PCD by continuously making new model changes available for analysis without forcing those changes on the architects' individual views.

Continuous Background Merging

Recall that VCSs discover synchronization conflicts only when architects explicitly attempt to merge their changes. In contrast, continuous background merging performs virtual merging of new model changes without involving the architects' views. The primary purpose of this capability is to proactively identify conflicts, and to do so outside architects' workspaces to prevent disruption of the architects' design activities.

A variety of strategies are possible to determine which changes to merge and when to merge them. For example, an architect might be interested to know whether his or her design conflicts with specific colleagues, whether merging all operations by all the architects in his or her group leads to a conflict, or whether merging formally committed versions will lead to a conflict. Different merging strategies aid varying kinds of collaborative awareness and are appropriate for possible variations in collaborative-design scenarios.

Continuous Model Analysis

Continuous analysis of the models produced by each virtual merge helps to detect high-order design conflicts. As we previously mentioned, continuously running model analyses locally might overwhelm the architect's machine, and thus not only disrupt the architect's work but also actually delay conflict detection. A collaborative-design environment that implements continuous model analysis must address this issue, potentially by offloading the burden to remote machines, parallelizing and distributing the computation, optimizing the order in which the analyses take place, etc.

FLAME

As a solution to the challenges inherent in collaborative design, we have designed and developed FLAME. FLAME implements the three capabilities described in the section "Adoption of PCD." It uses a novel event-based VCS and orchestrates architects' existing modeling and analysis tools to perform PCD.

Although the architecture of and ideas behind FLAME are independent of the employed architecture modeling and analysis tools, our implementation is built on top of the popular Generic Modeling Environment (GME; http://www.isis.vanderbilt.edu/Projects/gme) and uses the XTEAM (Extensible *T*ool-Chain for *E*valuation of *A*rchitectural *M*odels; https://softarch.usc.edu/~gedwards/xteam.html) architecture modeling and analysis framework. FLAME is open source, and our experimental data are publicly available at http://flamedesign.org.

FLAME differs from prior tools in its extensibility and operational granularity. FLAME is extensible by providing explicit 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 individual 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 each operation. Whereas 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. This real-time synchronization enables

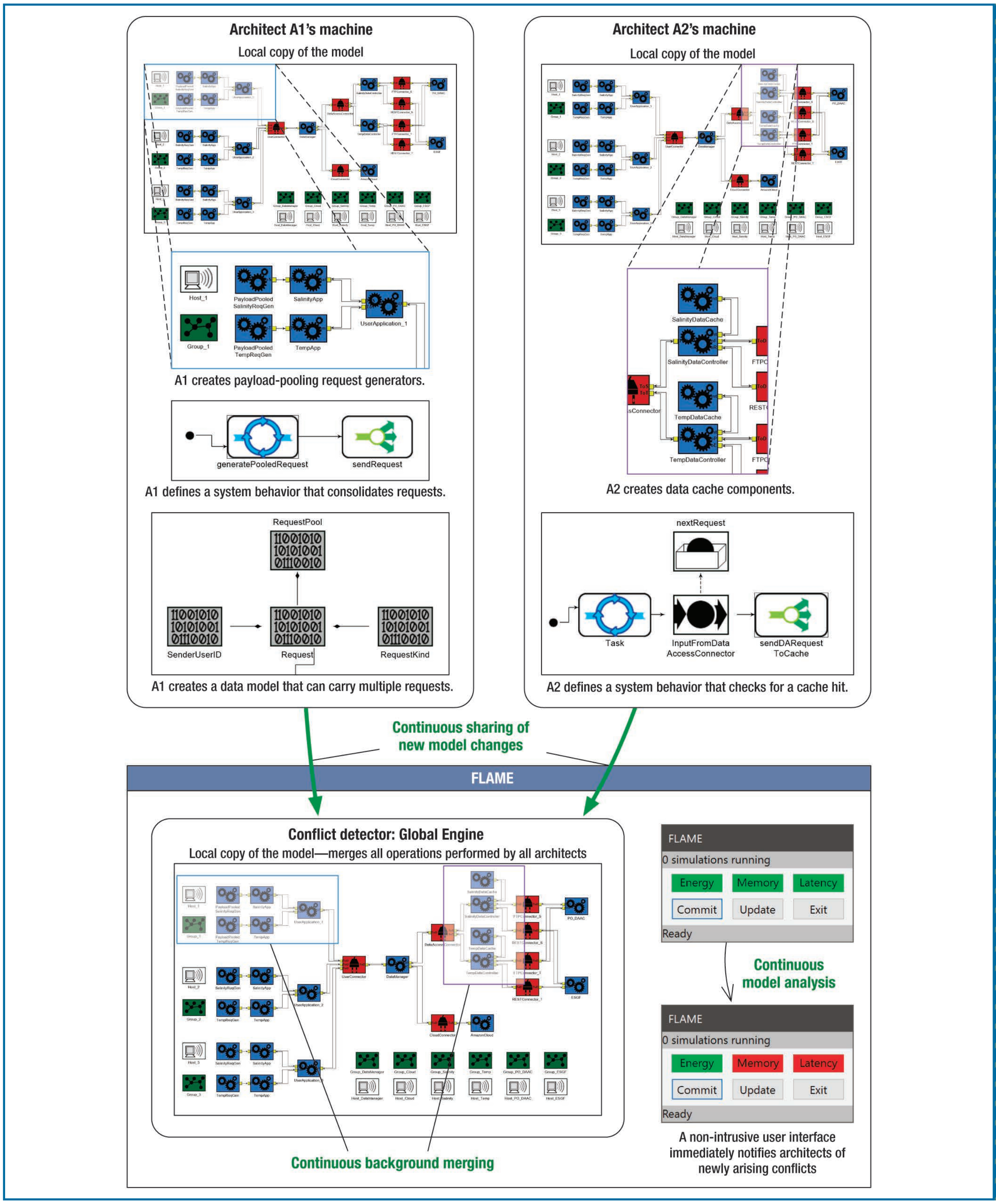
- continuous analysis execution even when an architect is the only one working and
- continuous proactive detection of synchronization and high-order conflicts.

Performing continuous model analysis on a single machine might delay conflict detection and exacerbate the problem a tool is trying to solve. FLAME distributes the work to multiple conflict-detection engines; each maintains an internal copy of the model, follows its own merging strategy, and generates model representations for deploying analysis. The engine executes the analysis either locally or remotely using worker nodes (e.g., cloud instances that provide computational resources for model analysis). FLAME offloads multiple analysis executions to be performed in parallel, one for each version of a model. A detection engine selects a model representation, deploys its analysis onto a worker node, and relays the result back to the architects.

FLAME can employ a variety of merging strategies to provide varying levels of conflict awareness. We developed two merging strategies: Global Engine and Head-and-Local Engine. Global Engine merges all architects' operations. The result is the most current design that gives PCD the most predictive power, albeit risking detecting false-positive conflicts that do not materialize because the architects might revert uncommitted operations. Figure 1 depicts an example collaborative-design scenario using FLAME with this strategy.

Head-and-Local Engine merges an architect's latest operations with all the other architects' committed operations, reducing false positives but delaying the detection of some conflicts. This merging strategy is implemented by many existing code-level PCD tools.

FLAME implements a prioritization algorithm that selects which representations to analyze first in order to quickly provide information on newly arising conflicts to architects. 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. This algorithm bounds the time required to detect the high-order design conflicts to twice the running time of the employed analysis that finds the conflict.[3]



Architect A1's machine
Local copy of the model
Host_1
PayloadPooled SalinityReqGen
SalinityApp
UserApplication_1
Group_1
PayloadPooled TempReqGen
TempApp
A1 creates payload-pooling request generators.
generatePooledRequest
sendRequest
A1 defines a system behavior that consolidates requests.
RequestPool
SenderUserID
Request
RequestKind
A1 creates a data model that can carry multiple requests.
Architect A2's machine
Local copy of the model
SalinityDataCache
SalinityDataController
TempDataCache
TempDataController
A2 creates data cache components.
nextRequest
Task
InputFromData AccessConnector
sendDARequest ToCache
A2 defines a system behavior that checks for a cache hit.
Continuous sharing of new model changes
FLAME
Conflict detector: Global Engine
Local copy of the model—merges all operations performed by all architects
FLAME
0 simulations running
Energy
Memory
Latency
Commit
Update
Exit
Ready
Continuous model analysis
FLAME
0 simulations running
Energy
Memory
Latency
Commit
Update
Exit
Ready
A non-intrusive user interface immediately notifies architects of newly arising conflicts
Continuous background merging

FIGURE 1. Design model changes made concurrently by multiple architects can be merged together but result in a high-order conflict. Here, merging A1's change to pool payloads and A2's change to add smart caching results in increased latency and high memory consumption (recall scenario 2 in the section "Design Conflicts"). FLAME (Framework for Logging and Analyzing Modeling Events) detects this high-order conflict quickly after it is introduced.

ABOUT THE AUTHORS

JAE YOUNG BANG is a software engineer and researcher at Kakao. His research interests include reliability and security in distributed software systems and collaborative software development. Bang received a PhD in computer science from the University of Southern California. Contact him at jae.bang@kakaocorp.com; http://ronia.net.

YURIY BRUN is an associate professor in the College of Information and Computer Sciences at the University of Massachusetts Amherst. His research interests include software engineering, software fairness and bias, and self-adaptive systems. Brun received a PhD in computer science from the University of Southern California. Brun has received the US National Science Foundation CAREER Award and the IEEE Technical Committee on Scalable Computing Young Achiever in Scalable Computing Award. He's a senior member of ACM and IEEE. Contact him at brun@cs.umass.edu.

NENAD MEDVIDOVIĆ is a professor in the University of Southern California's Computer Science Department and Informatics Program. His research interests include architecture-based software development. Medvidović received a PhD from the Department of Information and Computer Science at the University of California, Irvine. He has been the chair of the ACM Special Interest Group on Software Engineering and is editor in chief of *IEEE Transactions on Software Engineering*. He's an ACM Distinguished Scientist and IEEE Fellow. Contact him at neno@usc.edu.

The development cost of integrating FLAME into an existing collaborative software design environment is relatively minor. One needs to develop an adaptor for the modeling tool to

- capture modeling operations an architect performs and
- apply operations from other architects back to the model.

FLAME's own implementation serves as a proof of this concept, requiring 6,500 lines of Java and C++ code on top of GME and XTEAM.[3]

Conflicts in collaborative software design are frequent, and the risks they pose are costly. Research has suggested how software architects might benefit from PCD to reduce these conflicts and their risks.[3] Although PCD does not directly affect parts of the development lifecycle other than design, it can be integrated into any development process if the process allows design collaboration. Incorporating the results of this research in industrial practice is likely to reduce the costs of collaborative design.

Further research on improving collaborative PCD includes improving the conflict notification interfaces to provide better awareness without distraction, and automatically developing conflict resolutions and effectively recommending them to the architects.

Acknowledgments

The US National Science Foundation partially supported this work under grants CCF-1453474, CNS-1513055, CCF-1564162, CCF-1618231, CCF-1629977, and CCF-1717963.

References

1. Y. Brun et al., "Proactive Detection of Collaboration Conflicts," *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. Foundations of Software Eng.* (ESEC/FSE 11), 2011, pp. 168–178.
2. K. Muşlu et al., "Reducing Feedback Delay of Software Development Tools via Continuous Analyses," *IEEE Trans. Software Eng.*, vol. 41, no. 8, 2015, pp. 745–763.
3. J. Bang, Y. Brun, and N. Medvidović, "Continuous Analysis of Collaborative Design," *Proc. 2017 IEEE Int'l Conf. Software Architecture* (ICSA 17), 2017, pp. 97–106.
4. B. Westfechtel, "Merging of EMF Models," *Int'l J. Software and Systems Modeling*, vol. 13, no. 2, 2014, pp. 757–788.
5. Y. Brun et al., "Early Detection of Collaboration Conflicts and Risks," *IEEE Trans. Software Eng.*, vol. 39, no. 10, 2013, pp. 1358–1375.
6. A. Sarma, D.F. Redmiles, and A. van der Hoek, "Palantír: Early Detection

of Development Conflicts Arising from Parallel Code Changes," *IEEE Trans. Software Eng.*, vol. 38, no. 4, 2012, pp. 889–908.

7. J. Bang et al., "How Software Architects Collaborate: Insights from Collaborative Software Design in Practice," *Proc. 6th Int'l Workshop Cooperative and Human Aspects of Software Eng.* (CHASE 13), 2013, pp. 41–48.
8. T.N. Nguyen et al., "An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), 2005, pp. 215–224.
9. A. Mehra, J. Grundy, and J. Hosking, "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design," *Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 05), 2005, pp. 204–213.
10. T. Schriber and D. Brunner, "Inside Discrete-Event Simulation Software: How It Works and Why It Matters," *Proc. Winter Simulation Conf. 2014*, 2014, pp. 132–146.
11. J.A. Whittaker and M. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Trans. Software Eng.*, vol. 20, no. 10, 1994, pp. 812–824.
12. Balsamo et al., "Model-Based Performance Prediction in Software Development: A Survey," *IEEE Trans. Software Eng.*, vol. 30, no. 5, 2004, pp. 295–310.
13. M. Chechik et al., "Multi-valued Symbolic Model-Checking," *ACM Trans. Software Eng. and Methodology*, vol. 12, no. 4, 2003, pp. 371–408.