Unifying Configuration Management with Merge Conflict Detection and Awareness Systems

H.-Christian Estler Martin Nordio Carlo A. Furia Bertrand Meyer
Chair of Software Engineering, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

***Abstract*—As software development becomes an increasingly collaborative effort, traditional development tools have to be extended to support seamless collaboration while minimizing the chances of conflicts. This paper describes CloudStudio, a collaboration framework that integrates a fine-grained software configuration management model and a real-time awareness system. CloudStudio's configuration management operates transparently by automatically sharing the changes of developers working on the same project; the real-time awareness system allows for dynamic views on the project selectively including or excluding other developers' changes. With this tight integration, conflicts are prevented in many cases, while leaving individual developers free to experiment without blocking others. The paper also describes a freely available prototype web-based implementation of CloudStudio and a case study that demonstrates the usability of the approach for collaborative software development.**

I. Introduction

Software development is overwhelmingly a group activity. Sure, innovative software products may still be conceived by—or even require—the stroke of genius of a solitary demiurge; but, if they eventually become widely used and successful, it is only through the contributions of multiple developers over several years.

With collaborative development becoming the norm, a number of standard processes and tools have emerged that support multiple developers working on the same codebase. Integrated Development Environments (IDEs) and Software Configuration Management systems (SCMs) have thus become the software developer's central tools. Traditional IDEs are essentially personal tools, where every member of a project works on a local copy of the software under development and periodically undergoes a process of synchronization with the other members using the functionalities offered by the SCM.

The development paradigm embodied by the combination of IDEs and SCMs has been incrementally refined and has successfully scaled up to very large development efforts. Its fundamental structure and mode of operation, however, have not changed significantly since their introduction, and its shortcomings are becoming more evident as the magnitude of software development efforts is ever increasing. Since each developer works off-line on a local copy of the codebase, *conflicts* between her changes and another developer's may emerge. Conflicts complicate and slow down collaborative development, because they require *resolution*: an often painful process of analysis and coordination to produce a unique consistent version that merges the conflicting views.

The software engineering research community is well aware of these shortcomings [14], [11], [18], [7] and, in response, has proposed a number of advanced techniques to support *conflict detection* as soon as possible, and to improve every developer's *awareness* of what his colleagues are doing to the codebase that may affect him. Conflict detection and awareness can both be instrumental in reducing the likelihood and severity of conflicts. The major limitation of the existing approaches (reviewed in Section II) is that they have been conceived and implemented in isolation: conflict detection systems remain centered around the notion of conflict and resolution without actively promoting conflict *avoidance*; awareness systems are typically oblivious of the abstractions used by the SCMs, and hence are of limited help to *resolve* conflicts when they cannot be avoided.

This paper describes CloudStudio, our proposal for a collaborative development framework where the software configuration management, conflict detection, and awareness systems are unitarily conceived and tightly integrated. CloudStudio's configuration management system continuously operates in the background to automatically share every developer's changes with everybody else. On top of this, the real-time awareness system lets each user decide to selectively display, notify, or hide the changes introduced by others. The whole development environment is aware of the current view, and can compile, execute, and debug the project accordingly. This tight integration of different features makes it possible to synergically avail their combined benefits: direct conflicts are *prevented* in most situations, but at the same time a developer's work need not block others. Real conflicts occur only when a developer deliberately decides to branch out a new version of the code independent of the others' work.

CloudStudio is also the name of a web-based IDE prototype that we have implemented to demonstrate the ideas of the CloudStudio framework. You can try it out online at cloudstudio.ethz.ch. Using this prototype, we have conducted a case study where three teams of two programmers worked on collaborative development tasks either with CloudStudio or with traditional a IDE and SCM (EiffelStudio and Subversion). Within the limits given by its scope, the case study substantiates our claims that the CloudStudio framework can facilitate collaborative development without interfering with the habitual practices of programmers.

The main contributions of the paper are as follows:

- A novel *software configuration management* model that

automatically maintains multiple synchronized versions of the codebase integrating the changes of different developers. This facilitates conflict *prevention* even in situations where multiple developers work closely on the same portion of code. The software configuration management model does not rely on the details of any specific SCM and can be applied to existing repositories.[1]
- A *real-time awareness system* that supports multiple views on the project, including or excluding other developers' changes, and unobtrusively making programmers aware of each other's work before conflicts occur.
- A *prototype implementation* of the CloudStudio collaborative development framework into a publicly available web-based IDE.
- A *case study* that gives preliminary evidence of the advantages brought by the integrated CloudStudio approach.

Outline. Section II presents related work about awareness systems and software configuration management; Section III gives an overview of how developers can collaborate using CloudStudio; Section IV describes CloudStudio's configuration management and awareness models; Section V discusses the main features of the prototype web-based IDE implementing the CloudStudio framework; Section VI presents a case study comparing CloudStudio against traditional SCM; and Section VII concludes.

II. Related Work

Several proposals for awareness and advanced conflict detection systems have been put forward in the last decade, sharing the same goals of facilitating collaborative development and improving over the standard practices based on traditional SCMs. This section discusses the main features of these systems, with focus on those that are directly relevant for the CloudStudio framework discussed in the rest of the paper; Table I gives a synoptic overview. The distinction between awareness and conflict detection is not sharp as most systems include some of both features; it is, however, useful to highlight the focus of each approach and to discuss how CloudStudio targets the tight integration of these two naturally related aspects. Since CloudStudio is currently available as a web-based IDE, we also discuss some outstanding examples of IDEs for collaborative work (Collabode [8] and Cloud9 [5])—even though the innovations of CloudStudio are in the underlying collaboration model rather than in its specific implementation in a web-based IDE.

The standard approach to software configuration management (implemented by tools such as CVS and Subversion) uses a client/server architecture with a central repository and local working copies with every developer. Synchronization between developers takes place indirectly through the central repository by explicit request of the clients: a client's *commit* operation propagates her local changes upward into the central repository; a client's *update* operation copies the central repository's content downward to her local copy. Whenever the local and central content diverge in irreconcilable ways, there is a conflict that must be addressed by *merging* the two different copies. While committing becomes a local operation with *distributed* SCMs (such as Git and Mercurial), where every client maintains a complete commit history of her changes, this does not makes conflicts less likely to occur, nor obviates the need for merging whenever a developer periodically *pushes* her local changes to the other team members' repositories.

This mode of operation can introduce two kinds of conflicts: *direct conflicts*—two users editing the same piece of code in different ways—and *indirect conflicts*—the edits of a user in a portion of the code break another portion of the code written by another user, such as when changing a method signature's requires the clients to change their invocations.

Awareness systems address the problem of conflicts by allowing each developer to see the changes introduced by others and what artifacts have been affected by the changes. If developer u can see that developer v has changed a line, u is aware that modifying the line may introduce a conflict, and that he should coordinate with v to avoid it. Significant examples of awareness systems are FastDash [1], Palantir [19], Syde [15], and CollabVS [13].

Awareness mostly targets direct conflicts, but only simple indirect conflicts such as changes to method signatures; this is the case of Palantir and CollabVS, whereas FastDash and Syde do not detect indirect conflicts at all. Since awareness systems are normally not integrated with the SCM, they fail to provide good support to resolve and merge conflicting versions when conflicts do occur. Exceptions are CollabVS and Syde, but their support for inspection of conflicting versions is comparable to textual diffs like those offered by traditional SCM.

Advanced conflict detection. Following the principle that problems are easier to fix the sooner they are discovered, *conflict detection* and *conflict prediction* systems monitor the activities of all members of a development team searching for causes of conflict and reporting them to the interested users as soon as possible. Tools such as Crystal [2] and WeCode [12] work by continuously (and transparently) trying to merge the local copies of the various users; whenever a speculative merge fails, it signals that a conflict may occur in the future. This mode of operation covers one of the main weakness of awareness systems: the detection of complex indirect conflicts, since the speculative merge may include compilation or even regression testing.

The evaluation of Crystal [2] shows that this information is quite useful to anticipate problems during development and to make merging less painful. On the other hand, since conflict detection systems are not fully integrated with the rest of the development environment, they do not offer flexible ways to inspect the work of others when conflicts are detected in order to facilitate *resolution*. A related issue is that conflict detection works only on stored or committed files, but user activity in-between such operations is invisible to others.

IDEs for collaboration. A diversity of tools are used

[1] In the current prototype, it is implemented on top of Git.

	direct conflict	indirect conflict	false positives	availability	Awareness				Editing		transfer changes	compile with view	Comment
					line-based	class-based	branch-based	customizable	shared editing	automatic merging			
FastDash [1]	no	no	–	real-time	yes	yes	yes	no	no	no	–	–	high-level information, not specifically for conflicts
Palantir [19]	detect	detect	?	real-time	no	yes	no	no	no	no	no	no	only basic indirect conflicts (not compilation)
Syde [15]	detect inspect	no	no	real-time	no	yes	no	no	no	no	no	no	cannot inspect other changes; diff-based inspection
CollabVS [13]	detect	detect inspect	yes	real-time	yes	yes	no	no	no	no	yes	no	only basic indirect conflicts (not compilation); diff-based inspection
Crystal [2]	detect	detect	no	commit	no	no	yes	no	no	no	no	no	cannot inspect other changes; conflicts detected after commit
WeCode [12]	detect	detect	no	saving	no	no	yes	no	no	no	no	no	cannot inspect other changes; conflicts detected after commit
Collabode [9]	no	no	–	no	no	no	no	no	yes	no	no	no	collaborative editing does not maintain separate versions for each user; no conflict detection
Cloud9 [5]	no	no	–	no	no	no	no	no	yes	no	no	no	collaborative editing does not maintain separate versions for each user; no conflict detection
CloudStudio	prevent detect inspect	detect inspect	no	real-time	yes	yes	yes	yes	no	yes	yes	yes	

TABLE I

MAIN FEATURES OF AWARENESS SYSTEMS AND CONFLICT DETECTION FRAMEWORKS.

For each system, we report: whether it supports detection of *direct conflicts* and of *indirect conflicts*; whether conflict reports may include *false positives*; whether conflicts are *available* in real-time or upon commit; the granularity of the *awareness* system (line, class, branch, and whether it is customizable); whether collaborative *editing* supports shared sessions a la Google Doc and automatic merging of versions; whether there are mechanisms to *transfer* the changes of one user to another; and to *compile* the version of the project under the current view; and the main *limitations* of the approach.

to simplify collaboration among distributed teams, including some commercial products such as IBM's Jazz [4] and Microsoft's Team Foundation [17]; a complete review is beyond the present paper's scope.

Recognizing the centrality of the IDE among development tools, tools such as CodeRun [6] have brought IDEs to the web. Using these tools requires no software installation or configuration but only a browser. Other features, however, are replicated as in traditional IDEs: every developer works on a different copy of the code, stored on a server, and SCM follows traditional practices. Thus, IDEs have followed the industry's trend to move to the web, but they do not address awareness or merge conflicts in any specific way.

In a few cases, for example Cloud9 [5] and Collabode [8], web-based IDEs support collaborative development through real-time code sharing: developers can simultaneously work on the same piece of code with the same view, as if they were editing a GoogleDoc shared document. Such an unrestricted form of collaboration is, however, useful only in certain circumstances where direct tight collaboration is required, such as in pair programming practices; during general development practices it is instead necessary to follow a more disciplined approach that integrates with standard SCMs.

III. A SESSION WITH CLOUDSTUDIO

This section gives an overview of the CloudStudio framework from the perspective of two users—Claudia and Stu—who are working on the same project using the CloudStudio web-based IDE. The use case scenario is shown in Figure 1, to which the following description repeatedly refers.

After logging in on cloudstudio.ethz.ch and selecting a project, Claudia (rightmost column in Figure 1) starts working on a class *PARAGRAPH*. CloudStudio displays the class current base version as plain text in Claudia's editor (C1).

Claudia is editing class *PARAGRAPH* concurrently with Stu, who is working at a different location. At any time, Claudia can show or hide Stu's changes to the code by toggling a button. When changes are shown, vertical bars of different colors mark each line of code according to its edit status: blue for lines changed or added by the current user; orange for lines changed or added by others; lines without a colored bar are unchanged by anyone. Claudia starts modifying class *PARAGRAPH* by adding a method *set_font_size* (C2 and C3), whose code is marked in blue in her editor.

In the meanwhile, Stu (leftmost column in Figure 1) starts editing the same *PARAGRAPH* class. Stu's setup is using another visualization option offered by CloudStudio: it displays only the *locations* of Claudia's current changes (marked with red arrow tips), but not the actual content of her changes. Stu notices that the "to do" comment line is marked (S1) and realizes that Claudia has modified that line. Stu switches view to see exactly Claudia's work (the implementation of *set_font_size*), which now appears marked in orange in his editor (S2). When Stu compiles the project, he can target the base version of the code (only unchanged lines), or include his or Claudia's changes, or both. This mechanisms make Claudia

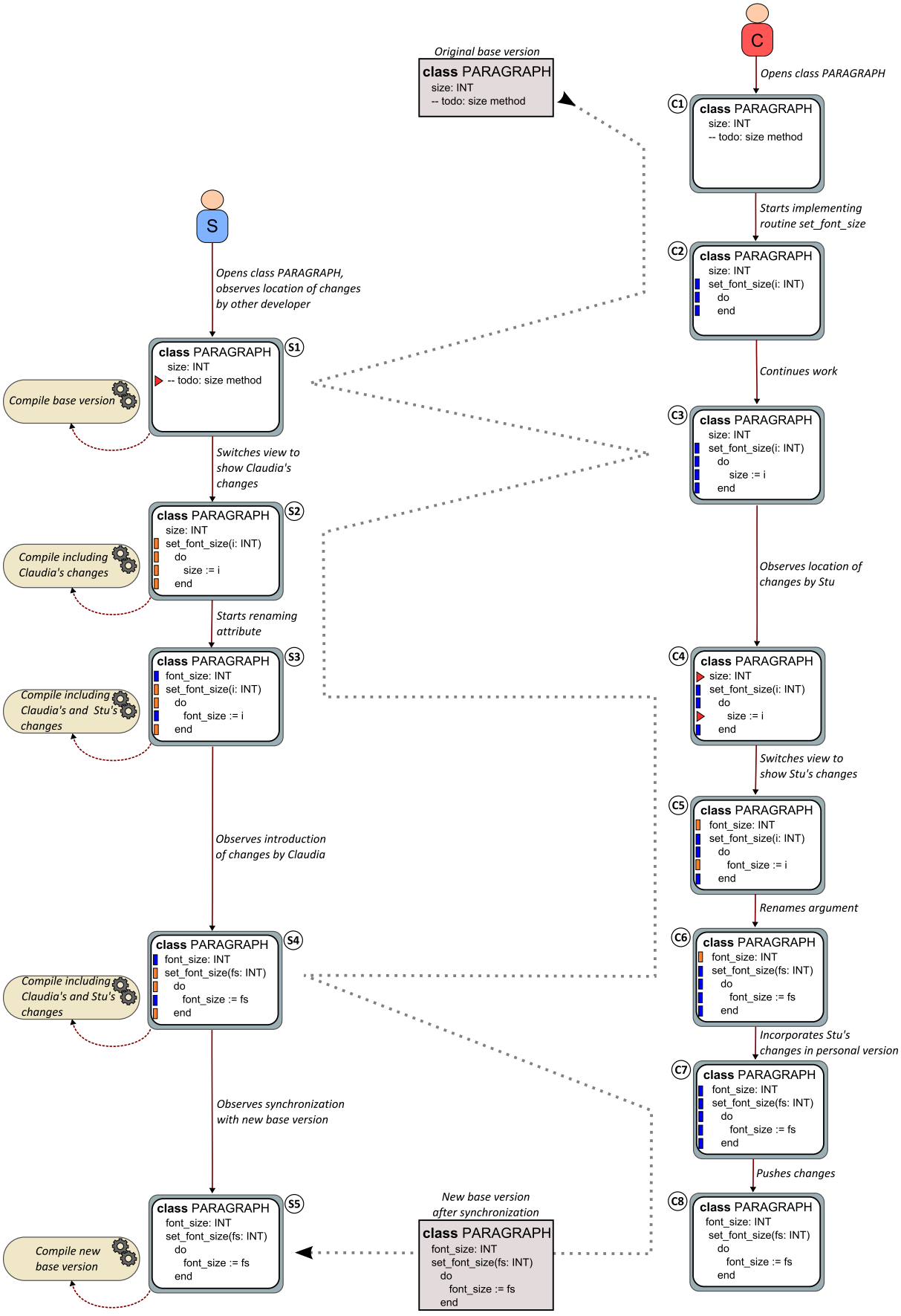

Fig. 1. A scenario demonstrating how the CloudStudio framework supports collaborative development.
Orange ▮ marks lines modified by others; blue ▮ lines modified by self; red arrow tips ▶ lines modified but not shown.

and Stu aware of each other's work; they do not have to block and immediately resolve conflicts, but they can continue working without stomping on each other's feet.

Fully aware of Claudia's concurrent editing, Stu does some light refactoring, consisting of renaming attribute *size* to *font_size* (S3). Claudia is aware of the change, because attribute *size*'s line becomes marked by a red arrow in her editor (C4). She decides to fully display Stu's changes (C5), so that she can check that Stu has diligently modified the body of *set_font_size* consistently with the refactoring.

At this point, Claudia and Stu continue with their concurrent editing without need for explicit synchronization (C6 and S4); this *prevents* conflicts during concurrent editing. What they see in their editors at any time is, however, only a real-time view constructed based on their visualization preference. Underlying the awareness system there is a full-fledged software configuration management system that maintains personal development branches for Stu and for Claudia. CloudStudio offers support to automatically synchronize and merge personal branches into the base version.

Claudia "approves" Stu's latest changes by storing them in her personal branch (C7) and by pushing the final set of changes to the master repository (C8). Since Stu has enabled automatic synchronization, his personal repository gets immediately synchronized with the latest version committed by Claudia through the master repository (S5).

IV. How CloudStudio Unifies Configuration Management, Conflict Detection, and Awareness

CloudStudio's software configuration management model combines flexibility and automation, and supports fine-grained conflict prevention and real-time awareness within an environment that facilitates collaborative development. This section describes the main technical details of its configuration management, conflict prevention, and awareness systems.

The CloudStudio framework builds on two fundamental abstractions: tasks and views. A *task* is a project branch organized around the activities of groups of developers. A *view* is the version of a project (or a subset thereof) obtained by combining the contributions of multiple developers.

A. Software Configuration Management Model

The software configuration management model maintains information about files and folders in combination with the task structure, which is used to consistently update the tasks and, by the awareness system, to automatically extract views on user demand. The rest of this subsection describes in some detail these abstractions and how they are implemented in CloudStudio; in order to do that, it first briefly revises standard notions used in SCMs (repositories, branches, and push/pull operations), on top of which CloudStudio's configuration management model is built.

1) Repositories: A *repository* is a collection of revisions (or snapshots) of a software project, consisting of files organized in folders. Repositories are organized in DAG structures: when it is created, a repository only contains an empty *root* branch. As new revisions are *committed*, the root grows linearly, until a new *branch* is created. Branches grow independently of the root until they possibly merge back into it. New branches can also spawn off other branches, thus creating subbranches. Figure 2 shows the structure of a repository where the root branch has revisions $r_{0.0}$ to $r_{0.6}$ and two branches originating in $r_{1.0}$ and $r_{2.0}$; the first branch has a subbranch at revision $r_{1.1.0}$ and the second branch merges back into the root at revision $r_{0.6}$.

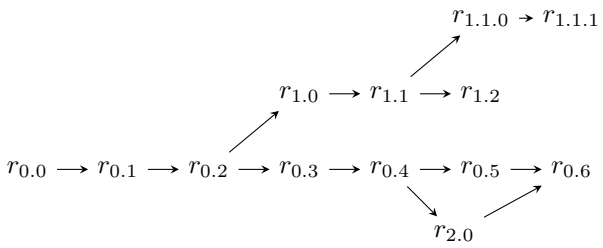

Fig. 2. A repository with branches and subbranches.

Synchronization between repositories occurs via push and pull operations, usually through a *master* repository. The *push* of a branch b from a repository R takes the content of b in R and merges it with the content of b in the master; if no conflicts occur, this corresponds to copying and appending b into the master. Conversely, the *pull* of a branch b from a repository R takes the content of b in the master and merges it with the content of b in R.

2) Tasks and Subtasks: Every CloudStudio project maintains a master repository plus personal repositories for each developer. Developers can work on the predefined *root task*, present in the master as well as in every repository and conventionally denoted T_0, or create new *tasks*. A task corresponds to a branch managed according to the synchronization policy of CloudStudio, which provides seamless and consistent synchronization among branches. Whenever a user u creates a new task T, both u's repository and the master spawn off a new branch for T; if, later, another user v joins task T, v's repository is updated with a branch for T as well. Tasks can also spawn *subtasks*, which are implemented as subbranches in the repositories. Figure 3 shows a CloudStudio project involving three developers—Claudia, Stu, and Ann—and three tasks—T_1, T_2, and T_3; T_1 involves Claudia and Stu, T_2 involves Stu and Ann, and T_3 involves Claudia and Ann; the root task T_0 is shared by everyone as usual.

3) Line Change Model: CloudStudio manages tasks according to a model of the *changes* introduced by the developers

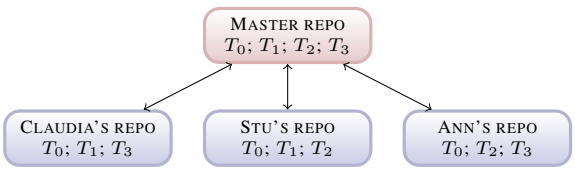

Fig. 3. A CloudStudio project involving developers working on four tasks.

working on each task. The change model is *fine-grained* as it logs changes to individual lines in every source file. For each line of every source file, CloudStudio stores a *line tree* of changes such as those in Figure 4. Line trees are ordered sequentially, according to how lines follow one another in the repository's files.

The root node of a tree initially stores the line in the latest version that all developers have pulled into their personal repositories; this is the base version initially shared by all developers on the project. Figure 4(a) shows the base version of a line with the assignment instruction *sum* := 3. The line is annotated with the tag "**M**: T_0", denoting that the line is also stored in the master repository **M** and belongs to the root task T_0.

When a developer modifies a line, the corresponding line tree gets extended with a new node that stores the changed line and who did the change. The root node, instead, still stores the version that all "other" users—that is, all users not mentioned in other nodes in the tree—have. If Stu initializes *sum* to *i* instead of 3, the line tree becomes as in Figure 4(b), where the new node is tagged "**Stu**: T_0" because it represents Stu's editing the root task.

The details of how a new node is added to a line tree depend on the awareness level of the developer introducing the change. Suppose developer u did the latest change to a line ℓ; CloudStudio stored u's change in a node N_u in ℓ's line tree. If another developer v modifies ℓ again, the new node N_v with v's changed line is added as a child of N_u *if and only if* v's awareness system is showing u's changes in real-time. In this case, there is no need to create a conflict because v is aware of u's work, and therefore we can expect that its own changes are consistent with and incremental over u's. Figure 4(c) shows an example of this where Ann modifies the assignment to *sum* again but she is aware of Stu's changes. In contrast, if v's awareness system is configured not to show u's changes in real-time, N_v is added as a sibling of N_u: v will be able to coordinate with u only later, but for the moment the two activities are separate. This is what happens in Figure 4(d), where we assume that, unlike Ann, Claudia is not displaying Stu's changes in real time.

If v is displaying u's changes in real-time but still prefers to branch off, it can either disable awareness of u, or switch to another task T' thus forcing the node N_v (with tag "v: T'") to become a sibling of N_u and to start a new series of independent changes. This is the case of Figure 4(e), where Ann spawns a new task T_1 to accommodate her change even if she is displaying Stu's changes.

4) Task Synchronization: CloudStudio uses the change model to organize the pushes of changes in tasks, in a way consistent with the information on awareness used in constructing the line trees to minimize conflicts.

To present the synchronization in some detail, we need the notion of *master ancestor*: the master ancestor of a node N for task T in a line tree is the ancestor node N_0 of N's such that N_0 has the line version stored in the master repository **M** for task T; in other words, the unique path from N_0 to N does not contain any node with tag "**M**: T" other than N_0. Suppose a node N_u stores a developer u's latest change ℓ_u to a line ℓ on task T, and let N_0 be the master ancestor of N_u for T in ℓ's line tree. Consider the subtree τ of ℓ's line tree rooted at N_0; and let τ_T be τ with all nodes not tagged with task T pruned. If τ_T is a linear sequence of nodes (every node has exactly one child), then N_u is *conflict free*. In this case, if developer u *pushes* its edits to ℓ, CloudStudio enforces a sequence of pushes, one for each node in τ_T in that order, and collapses the branch τ_T by replacing N_0 with the node $\langle \ell_u; \mathbf{M}, \mathbf{u}, \mathsf{U}\colon T\rangle$[2] while discarding the rest of τ_T until N_u included. All users other than u tagged in τ_T are aware of u's latest change, therefore CloudStudio notifies them and updates their personal repositories to coincide with the master on task T. The details of the notification are implementation dependent and customizable; for example, users may be asked to approve the push or may be lazily notified only when they want to edit line ℓ again.

Continuing the examples of Figure 4, consider again the line tree in Figure 4(c). The node with *sum* := *i* is conflict free; if Stu pushes his changes, the line tree becomes as in Figure 4(f), where the node with *sum* := *i*+*j* is also conflict free. If Ann pushes next, the tree becomes as in Figure 4(g); Stu is aware of Ann's changes, which have been automatically pulled into Stu's personal repository. The line tree ends up as in Figure 4(g) also if Ann pushes first (from the setup in Figure 4(c)). Using standard configuration management models, where there is no notion of who's aware of whom and whose changes can be merged without rising a conflict, a similar situation would force a conflict.

Branches in τ tagged with tasks other than T are joined with the collapsed τ_T so as to preserve the original information (this requires duplication of nodes in some cases, whose details are straightforward). If N_u is not conflict free, CloudStudio cannot push u's edits to ℓ automatically; in this case, a conflict is unavoidable, and the interested users are notified and required to resolve it before pushing is possible. They can do so by turning on awareness of each other's work, and then agreeing on a conflict free version of the line in question.

B. Real-time Awareness System

CloudStudio's awareness system displays the content of the project files based on the information in the software configuration management model (in particular, line trees) and displays it according to user preferences. A *view* is the version of the project determined by the current user preferences.

In the basic view, the editor shows the current user's edits, and, for each line not modified locally since the last pull, its base version as stored in the root node if its tree. On top of this, CloudStudio provides options to display the changes introduced by other developers in the current view. The current user can select any other developer v and choose to:

- display all changes introduced by v in real time;

[2] U denotes the (possibly empty) list of all other users tagged in τ_T.

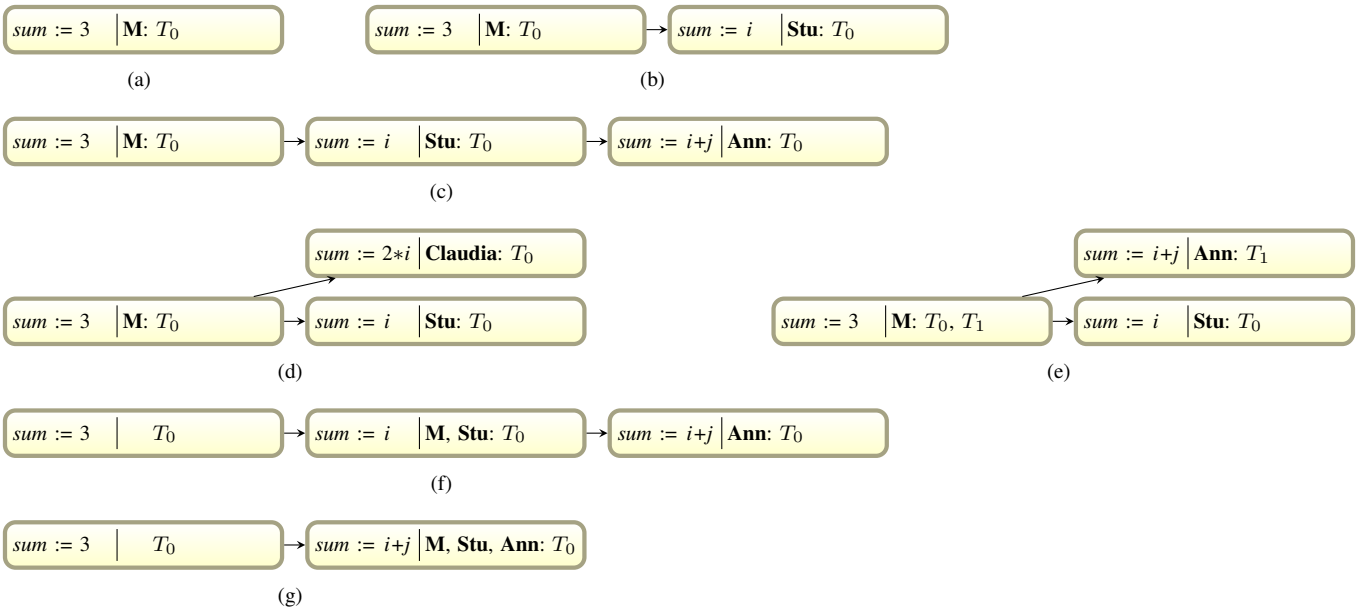

Fig. 4. The CloudStudio change model of a line with and assignment.

- display all changes introduced by v up to the latest successful compilation;[3]
- display where v introduced changes but do not show them;
- do not display changes by v at all;
- display all changes introduced in the current task by any other user.

Unlike most related work—where only committed changes are available to the awareness system [2], [8]—CloudStudio updates the information about changes in real-time and makes it available to all users.

CloudStudio's real-time awareness system integrates with the rest of the development environment through views. The compiler and every other tool working on the project files (such as debuggers or testing environments) have access to the project in the current view, even if the underlying changes have not been pushed to the personal or master repositories yet. This makes for a seamless integration of the awareness system and software configuration management within the developers' overall activity.

C. Collaborative Editing

On top of the mechanisms supporting tasks, fine-grained versioning, and views, the CloudStudio framework includes functionality to automatically control the synchronization of tasks between developers and facilitate collaborative editing.

Using the awareness system, it is possible to import code from one user's personal repository to another's and to start collaborating. CloudStudio supports the *importing* function that clones a portion of code (such as a method or a whole class) from a user u's personal repository to another user v's and turns on u and v's awareness systems so that they can work together on the code by relying on the conflict prevention mechanisms described above.

The CloudStudio development environment includes a set of options to trigger synchronization between developers automatically following a compilation event. When enabled, automatic synchronization triggers the following actions whenever a user u saves and successfully compiles the project under its current view:

1) it pushes u's current task to the master repository;
2) it pulls these changes to the personal repositories of all other users collaborating with u (and who have enabled automatic synchronization).

D. Conflict Detection and Prevention

In this section so far, the presentation has focused on *direct* conflicts, also known as *syntactic* conflicts, which occur when two developers modify the same line. CloudStudio's tight integration of awareness system and configuration management *prevents* direct conflicts by letting developers see each other's work in collaborative editing and by branching out when they deliberately decide to work in parallel ignoring each other.

In contrast, *indirect* conflicts, also known as *semantic* conflicts, occur when a developer's change to a portion of the code breaks the dependency with the work of another developer in another portion of the code. For example, changing a method's signature by adding a new argument introduces an indirect conflict in all clients that invoke the method anywhere in the code, which have to change their calls to conform to the new signature. CloudStudio supports indirect conflict *detection* through views. Users can compile the project under the current

[3]To extract the changes up to the latest successful compilation, all nodes in the line tree that denote the current version are tagged after every successful compilation. This is a straightforward extension of the line tree model discussed previously.

view, thus getting an error if any of the other users' changes included in the view has introduced an indirect conflict. The option to include in the view only the changes that can be successfully compiled adds another degree of flexibility, where developers collaborate on "stable" versions but are still free to experiment changes on their own that may break compilation. In all such cases, CloudStudio's real-time awareness features can be useful to let two developers collaborate with the goal of removing an indirect conflict that involve their work.

E. Limitations of CloudStudio's SCM

CloudStudio's change model works at the level of individual lines of code. This achieves language-independence but also does not take the structure and semantics of the code into account [15]; sound refactorings, in particular, may still be considered conflicting. While CloudStudio mainly aims at preventing conflicts and supports compilation using other developers' changes, a mechanism for conflict detection based on speculative compilation [3] could still provide an additional dimension of automation. We will consider these extensions in future work.

V. CloudStudio's Prototype Implementation

The CloudStudio web-based IDE is a prototype implementing the CloudStudio framework described in Section IV. The prototype is freely available at cloudstudio.ethz.ch; since it is web-based, using it does not require downloading any software. The implementation combines an editor written in Java using the Google Web Toolkit v. 2.3, and leverages a MySQL database and the Git SCM as back-ends.

CloudStudio currently supports development of projects in Java, JavaScript and Eiffel, but its architecture is extensible to other programming languages. Besides the innovative integration of the configuration management and real-time awareness system described in Section IV, CloudStudio offers some of the basic functionalities of traditional IDEs such as VisualStudio or Eclipse: an editor with syntax highlighting, a class browser to navigate the project, integration with the compiler, minimal support for execution, and a debugging environment (currently available only for JavaScript projects).

The awareness system uses the color code described in Section III to highlight lines changed in real-time by other users that have been selected by the current user. If the awareness system is configured to display change locations without showing the actual changes, red arrows in the margin mark the location of changes. Hovering over a colored bar or an arrow shows the user who has introduced the change, as well as different versions of the line as edited by other users active on the same task.

In continuity with our related work on formal verification [20], the CloudStudio IDE also integrates verification tools to help developers improve software quality. It currently supports testing with the AutoTest framework [16], [22], and formal correctness proofs with AutoProof [21]. AutoTest performs random testing of object-oriented programs with contracts, and it has proved extremely effective in detecting hundreds of errors in production software; AutoProof provides a static verification environment for Eiffel. Both tools are fully automatic and integrated with CloudStudio's SCM system: testing and proving sessions work on the current view selected by CloudStudio users, which flexibly may or may not include concurrent edits by other developers (as described in Section IV).

A. Limitations of Current Implementation

The current CloudStudio prototype implementation has a number of limitations, which makes it immature compared to most commercial IDEs. In particular, it lacks advanced features to compare and merge different versions; and its performance scales poorly with the number of users concurrently accessing the server. Improving all these points belongs to future work.

VI. Case Study

In order to have a preliminary assessment of the CloudStudio framework and its advantages for collaborative development over traditional IDEs and SCM techniques, this section presents a case study of two-programmer teams working on collaborative development tasks. While small in extension, the case study provides preliminary evidence that CloudStudio can improve the performance of programmers working collaboratively.

A. Development Tasks

The case study included three program development tasks, two focused on refactoring and one on testing; all applications were written in Eiffel.

- R1: Task R1 targets an application implementing a card game (the card deck and the game logic); the complete application includes 210 lines of code over 4 classes. Task R1 requires refactoring of three classes, and development of new functionalities by extending the refactored classes; the task is collaborative because the new functionalities must work with the classes after refactoring. Refactoring included: method and field renaming; enforcement of Eiffel coding standards (e.g., capitalization, comments); rearrangement of methods in groups (marked by the **feature** Eiffel keyword) according to their functionalities; code extraction into a new class.
- R2: Task R2 targets an application modeling a coffee vending machine; users of the application have basic options to select coffee and can pay and receive change. The application includes 230 lines of code over 3 classes. Task R2 is similar to R1 except that it targets the coffee machine application: R2 requires refactoring and development of new functionalities by extending the refactored classes.
- T1: Task T1 targets the same coffee machine application as task R2. It requires development of new functionalities (namely, the option to add milk to the coffee,

and the dispatching of different cup sizes) and writing of test cases that achieve 100% code coverage on the new code. Task T1 is also inherently collaborative as the development of new functionalities and of test cases occur concurrently, according to the concept of *test-driven pair programming* [10].

B. Subjects and Experimental Setup

The subjects used in the study were six PhD students from our research group. All of them are experienced Eiffel programmers who frequently develop with EiffelStudio and Subversion (SVN) as part of their PhD research; none of them had used CloudStudio before the study, had taken part in its development, or has much experience with collaborative development.

We randomly arranged the six subjects in three pairs: Team1, Team2, Team3. Team1 first performed task R1 with CloudStudio and then task T1 with EiffelStudio and SVN. Team2 first performed task R1 with EiffelStudio and SVN and then task T1 with CloudStudio. Team3 first performed task R1 with CloudStudio and then task R2 with EiffelStudio and SVN.

Each team performed its sessions according to the following protocol. The two team members sat at the opposite corners of a large table with their laptops connected to the network. Before beginning, the first author (henceforth "the experimenter") gave a brief (5 min.) introduction to the CloudStudio SCM and web-based IDE to both programmers at the same time, where he showed them how to log-in and the basics of the SCM system without any reference to the development tasks. Then, he gave them a sheet of paper with a description of the task they had to perform (the second task was introduced only after completion of the first). The two programmers received identical instruction sheets and had to coordinate in order to split the work between them.

During the study nobody other than the experimenter and the two programmers was in the room. The programmers were only allowed to use instant messaging to communicate; their position in the room and the experimenter ensured that no other communication channel was available. The experimenter did not interfere with the programmers other than to clarify possible unclear points in the task description (but this was never necessary).

There was no time limit to complete the tasks: each session continued until the current task was completed (the experimenter checked completeness *a posteriori* by manual inspection of the codebase). After each session, the experimenter recorded the total number of words exchanged via instant messaging and the overall time spent to complete the task. An *a posteriori* analysis of the communication logs, discussed in Section VI-D, supports the hypothesis that these two measures (words and time) are reasonable proxies for the actual amount of communication between the two programmers that took place during the experiments.

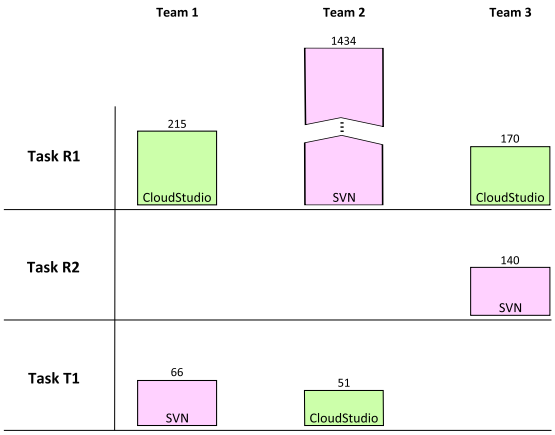

Fig. 5. Results of the case study (the scale is not uniform).

C. Results

Figure 5 reports the amount of communication between programmers while performing the various tasks. While all participants are competent programmers, their speed and development style vary significantly; as a result, the random assignment formed heterogeneous groups which may not be directly comparable. The results in Figure 5, however, show a consistent advantage for teams using CloudStudio over teams using SVN: the difference is sometimes small (as for task T1), sometimes conspicuous (as for task R1 between Team2 and Team1); in all cases, CloudStudio required less communication for the same task than SVN, even if the study's programmers used it for the first time. Let us now describe the performance of the various teams in more detail.

Team1 delivered the best overall performance and was fluent both with SVN and with CloudStudio; the two programmers worked well together and required a limited amount of communication to synchronize properly. The comparison with Team2 on the same tasks suggests that using CloudStudio is beneficial: Team1 outperformed Team2 almost by an order of magnitude when using CloudStudio on task R1, whereas their performance became similar on task T1 where Team1 used SVN. It was clear that Team1 was overall faster than Team2, but the peculiarities of task R1 magnified the difference in favor of who could rely on better collaboration tools.

The programmers in **Team2** had the greatest communication problems in the study, as shown by their performance in task R1. The log of their message exchanges shows that they had to debate several points of disagreement about how to perform the refactorings, and that not being able to see in real-time what the other was doing (as it happened when working with SVN) exacerbated their disagreement and frustration.

Unlike the members of the other teams, the two programmers in **Team3** worked with wildly different speed, to the point that in both tasks R1 and R2 a programmer completed his part of the task when the other was still exploring the system and understanding the instructions. The overall performance of Team3 required little communication in all cases, but this is

mostly a result of the fact that the different programmer speed forced a serialization between the two programmers; hence, synchronization was not a big issue because the development was not really collaborative and interactive.

We do not discuss in detail the *time* taken by programmers because the assignments emphasized correctness of the solution and did not pressure the teams for time. Anyway, and perhaps unsurprisingly, the overall time turned out to be correlated with the amount of communication, and hence all the experimental data point to the same qualitative conclusions.

D. Discussion

A *post mortem* analysis of the instant messaging logs shows recurring patterns of communications between programmers. The initial part of every session starts with a discussion of the task, after which the two programmers negotiate a division of the labor and agree on some synchronization mechanism. During development with SVN, messages such as "Did you update your project?" and "I'm done with implementing X and have committed" are frequent. With CloudStudio, the same messages occurs much more sparingly, and some of the remaining instances can probably be attributed to the programmers' limited familiarity with CloudStudio and how it works (in fact, in some cases of redundant notification messages using CloudStudio, the recipient replied with sentences such as "Just go ahead, I can see your changes live").

After the case study, we asked the participants to complete a simple questionnaire about their experience and with requests for feedback. The participants unanimously appreciated CloudStudio's mechanisms for the real-time awareness of other people's changes, and for the prevention and easy resolution of conflicts. Disagreement existed on how severe a problem merge conflicts are in everyday's software development: four programmers consider it a serious hassle and appreciate better mechanisms to prevent or manage conflicts; the other two maintained that merge conflicts can be reduced to a minimum with a little coordination.

In all, the participants to the study tend to agree with our conclusions that CloudStudio offers valuable features for collaborative development and a more flexible paradigm of SCM. The generalizability of our results is necessarily limited by the case study's scope and size, as well as by its reliance on specific development tasks that emphasize real-time collaboration but may affect only a limited part of large software projects. In this sense, the reaction of one of the programmers in our study to task R1 is instructive: he was initially skeptical and remarked that he "would never do refactoring while another programmer is implementing new functionalities"; after using CloudStudio, however, he acknowledged that, with the right tools, such tasks can indeed be performed in parallel.

VII. Conclusions

We described the CloudStudio framework that integrates software configuration management with real-time awareness to help detect and prevent merge conflicts in collaborative software development. We implemented the CloudStudio framework in a web-based IDE, and conducted a small case study to have a preliminary assessment of its usefulness compared to traditional IDEs and SCMs.

Acknowledgments

We thank Le Minh Duc, Alexandru Dima, Alejandro Garcia, and Sandra Weber for contributing to the prototype implementation of CloudStudio. Julian Tschannen and Yi (Jason) Wei contributed to the integration of the verification tools, and participated to the case study with Yu (Max) Pei, Marco Piccioni, Marco Trudel, Scott West. CloudStudio's startup funding through the Gebert-Ruf Stiftung is gratefully acknowledged.

References

[1] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. FASTDash: A visual dashboard for fostering awareness in software teams. In *CHI*, pages 1313–1322. ACM, 2007.

[2] Y. Brun, R. Holmes, M. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *ESEC/FSE*, pages 168–178. ACM, 2011.

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

[4] L.-T. Cheng, C. R. de Souza, S. Hupfer, J. Patterson, and S. Ross. Building collaboration into ides. *Queue*, 1(9):40–50, December 2003.

[5] Cloud9 IDE. http://www.cloud9ide.com.

[6] CodeRun Studio. http://www.coderun.com.

[7] J. Estublier and S. Garcia. Process model and awareness in SCM. In *SCM*, pages 59–74, 2005.

[8] M. Goldman, G. Little, and R. C. Miller. Collabode: Collaborative coding in the browser. In *CHASE*, pages 65–68. ACM, 2011.

[9] M. Goldman, G. Little, and R. C. Miller. Real-time collaborative coding in a web IDE. In *UIST*, pages 155–164. ACM, 2011.

[10] M. Goldman and R. C. Miller. Test-driven roles for pair programming. In *CHASE*, pages 13–20. ACM, 2010.

[11] R. E. Grinter. Using a configuration management tool to coordinate software development. In *COOCS*, pages 168–177, 1995.

[12] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *ICSE*, pages 342–352. IEEE Press, 2012.

[13] R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. In *ASE*, pages 178–187, 2008.

[14] S. Horwitz, J. Prins, and T. W. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[15] M. Lanza, L. Hattori, and A. Guzzi. Supporting collaboration awareness with real-time visualization of development activity. In *CSMR*, pages 202 –211, 2010.

[16] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Software*, pages 22–24, 2009.

[17] Microsoft Team Foundation. http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server/overview, 2012.

[18] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, 2001.

[19] A. Sarma, G. Bortis, and A. van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *ASE*, pages 94–103, New York, NY, USA, 2007.

[20] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *SEFM*, pages 382–398. Springer, 2011.

[21] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Automatic verification of advanced object-oriented features: The AutoProof approach. In *LASER Tools for Practical Software Verification*, volume 7682 of *LNCS*, pages 134–156. Springer, 2012.

[22] Y. Wei, H. Roth, C. A. Furia, Y. Pei, A. Horton, M. Steindorfer, M. Nordio, and B. Meyer. Stateful testing: Finding more errors in code and contracts. In *ASE*, pages 440–443. IEEE, 2011.