Early Conflict Detection with Mined Models

Leonardo Mariani, Daniela Micucci and Fabrizio Pastore
University of Milano Bicocca
Milan, Italy
Email: {mariani,micucci,pastore}@disco.unimib.it

I. Conflict Detection

Developers are increasingly adopting Source Code Management (SCM) systems with extensive support to branching, parallel development, and merging, such as Git and Mercurial. For example, 62% of the Debian projects use modern SCM environments [1], and 40% of the medium and large enterprises surveyed in [2] use Git. The use of the branching logic provided by modern SCM gives flexibility to developers but also produces issues when multiple branches of development have to be merged together. For example, Brun et al. report that 24% of merge operations in open source projects generate textual conflicts, build problems, and test failures [3]; Microsoft developers report that most of the time dedicated to merge operations is spent on resolving conflicts and verifying correctness [4]. In a nutshell, *merging multiple branches is a painful, expensive, and error-prone process that requires specific techniques to be handled efficiently*.

Techniques for the early detection of conflicts and unexpected interactions among changes on multiple branches *can reduce the effort required to cope with software evolution and concurrent development by a significant factor*. So far, Brun et al. [3] and Guimaraes et al. [5] have investigated the idea of anticipating conflict detection by merging, locally to the developers' workspaces, the code in the working copy with the code extracted from other branches. The analysis executed on the developers' machines can detect textual conflicts, build problems, and test failures, before the merge actually takes place in the SCM system. These approaches, although useful to anticipate the discovery of some conflicts, suffer several practical limitations: the conflict identification mechanism is limited to textual conflicts and test case execution, which may miss several subtle faults hard to discover and fix [4]; the analysis is executed on the developer's machine while the verification of evolving software systems should take place on the SCM server, without bothering developers; and finally the behavior of software systems is not limited to functional behavior but also includes other dimensions, such as the temporal behavior, that may evolve across branches and that is ignored by these techniques.

The key idea introduced in this paper consists of *running a multi-branch server-side dynamic analysis at every commit operation*. The analysis will execute the test cases available in the SCM system to trace the behavior of the application and automatically *derive models* that capture how the program behaves according to multiple dimensions. For instance, the functional behavior of the program can be represented with method pre- and post-conditions, API usage protocols, and precedence rules among method invocations. The temporal behavior of a program can be represented with models that capture aspects, such as deadlines, periodicity, and constraints on the timing of the tasks. These models are used on the server side to run automated conflict detection.

Models are derived for every version in every branch, and automatically compared every time a change is introduced. *Comparing models allows identifying behavioral conflicts regardless the presence of textual conflicts, which do not need to be resolved to run the analysis*. We call this analysis *Behavioral Driven Continuous Integration* (BDCI).

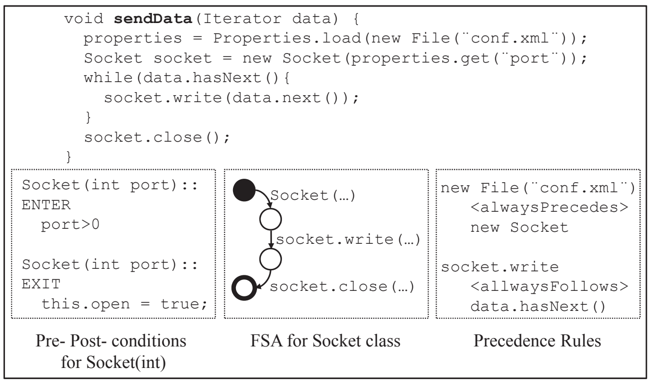
By raising the analysis to the behavioral level, BDCI can dramatically improve the rate of conflicts that are detected and resolved early in the process, as well as the rate of the bugs due to the concurrent modifications of the software that are revealed and fixed as soon as they are introduced. As a consequence, the cost and the effort required to complete merge operations will drastically decrease, and the capability to timely evolve software will significantly improve.

II. BDCI

The key idea of BDCI is to automatically derive models that represent the behavior of a program under evolution at multiple program locations, either recently changed or not, and raise the detection of conflicts from the source code level to the level of behavioral models. Although in the following we present an example referring to the analysis of the functional behavior of a program, the same kind of analysis can be instantiated to address other dimensions, such as resource consumption and timing of the operations.

Figure 1 shows some of the functional models that could be automatically mined with BDCI for a simple program. The method pre- and post-conditions capture information about the values that can be assigned to program variables. In the example, `port` must be positive when a socket is created and `open` must be true for the returned socket. Finite state automata (FSA) indicate how components interact. In the example, the automaton shows how the function `sendData` uses the socket library. Precedence rules indicate the execution order between operations. In the example, the creation of the conf.xml file must always precede the creation of the socket.

While in a traditional SCM only the test cases and the project source files are stored for each change, in a BDCI environment each program version and its behavioral models



```
void sendData(Iterator data) {
  properties = Properties.load(new File(¨conf.xml¨));
  Socket socket = new Socket(properties.get(¨port¨));
  while(data.hasNext(){
    socket.write(data.next());
  }
  socket.close();
}
```

```
Socket(int port)::
ENTER
  port>0

Socket(int port)::
EXIT
  this.open = true;
```

```
new File(¨conf.xml¨)
   <alwaysPrecedes>
   new Socket

socket.write
   <allwaysFollows>
   data.hasNext()
```

Fig. 1. Sample functional behavioral models

are also stored. The availability of the models provides unique opportunities for the early detection of conflicts and side effects produced by changes that would not be detectable otherwise. In particular, models can be compared to identify:

- *behavioral changes*: BDCI compares two versions on a same branch to discover the behaviors that have been affected by the change; this information is useful to assess the impact of changes;
- *behavioral conflicts*: BDCI compares the behavioral changes of two versions belonging to different branches to detect concurrent and incompatible changes in a same behavioral model and report the conflict.

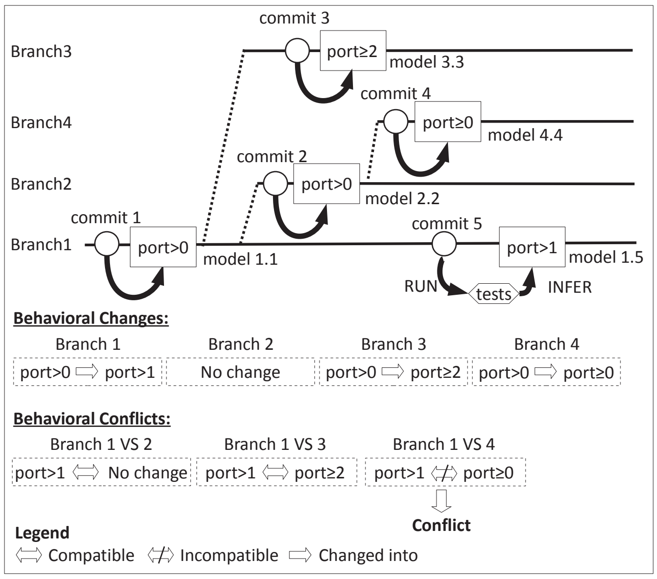


Fig. 2. The Commit Process

Figure 2 shows how BDCI can detect conflicts on the functional behavior in a simplified scenario where four branches of development are active. For simplicity we consider the case of a single functional model derived for a single program location: the functional model captures the values assigned to the integer variable `port`. When the commit 5 operation is performed, the SCM server augmented with BDCI automatically checks-out the project, executes the tests, collects the traces, generates the models (model 1.5), and detects behavioral conflicts with the other branches.

When comparing two branches for conflict detection (e.g., branch 1 and branch 2), BDCI derives behavioral changes between the most recent models available on the two branches (models 1.5 and 2.2) and the models available in the most recent version common to the two branches (models 1.1). The (concurrent) behavioral changes in the example for the variable `port` that are automatically detected are listed in Figure 2: branch 1 changes the model into $port > 1$; branch 2 includes no behavioral changes; branch 3 changes the model into $port \geq 2$; and branch 4 changes the model into $port \geq 0$. Two incompatible behavioral changes on two different branches generate a behavioral conflict. In the example, branches 1 and 2 show a different behavior for `port`, but the change occurs in one branch only and thus BDCI does not report any conflict. Branches 1 and 3 include two concurrent modifications to the same behavior of the program but do not produce a conflict because the changes are compatible ($port \geq 2$ and $port > 1$ are equivalent expressions). Finally, branches 1 and 4 represent the case of a conflict because there are concurrent and non-equivalent changes on the two branches: one change allows more port numbers to be used by the system, while the other change restricts the number of ports used by the system.

The capabilities offered by BDCI does not come for free. The cost of the BDCI analysis is higher than the cost of classic conflict detection algorithms, which do not require running tests and checking models. When these operations are particularly expensive, we envision scenarios where BDCI, instead of immediately checking commit operations, is executed overnight to check all the changes of the day.

III. Conclusion

The BDCI analysis introduced in this paper can promisingly produce three key benefits: (1) the capability to timely discover the conflicts and side-effects that are introduced during concurrent changes, with little manual effort; (2) the drastic reduction of the cost of merge operations thanks to the early resolution of conflicts; and (3) the significant reduction of the subtle problems that might escape merging operations, because working at the level of the behavioral models allows detecting problems that cannot be detected with textual comparison, build process, and test case execution.

References

[1] Z. Zacchiroli, “Vcs usage for debian source packages,” http://upsilon.cc/ zack/stuff/vcs-usage/.
[2] K. Noyes, “Git turns 8, sees wide adoption in the enterprise,” http://www.linux.com/news/enterprise/systems-management/715287-git-turns-8-enterprise-wide-adoption/.
[3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in *Proceedings of the ACM Symposium and the European Conference on Foundations of Software Engineering*, 2011.
[4] C. Bird and T. Zimmermann, “Assessing the value of branches with what-if analysis,” in *Proceedings of the ACM International Symposium on the Foundations of Software Engineering*, 2012.
[5] M. L. Guimaraes and A. R. Silva, “Improving early detection of software merge conflicts,” in *Proceedings of the International Conference on Software Engineering*, 2012.