Challenges and Demands on Automated Software Revision (extended abstract)

Borzoo Bonakdarpour and Sandeep S. Kulkarni

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
Email: {borzoo,sandeep}@cse.msu.edu

1 Motivation

In the past three decades, automated program verification has undoubtedly been one of the most successful contributions of formal methods to software development. However, when verification of a program against a logical specification discovers bugs in the program, manual manipulation of the program is needed in order to repair it. Thus, in the face of existence of numerous unverified and uncertified legacy software in virtually any organization, tools that enable engineers to automatically verify and subsequently *fix* existing programs are highly desirable. In addition, since requirements of software systems often evolve during the software life cycle, the issue of *incomplete specification* has become a customary fact in many design and development teams. Thus, automated techniques that *revise* existing programs according to new specifications are of great assistance to designers, developers, and maintenance engineers. As a result, incorporating *program synthesis* techniques where an algorithm generates a program, that is *correct-by-construction*, seems to be a necessity.

The notion of manual program repair described above turns out to be even more complex when programs are integrated with large collections of sensors and actuators in hostile physical environments in the so-called *cyber-physical systems*. When such systems are safety/mission-critical (e.g., in avionics systems), it is essential that the system reacts to physical events such as faults, delays, signals, attacks, etc, so that the system specification is not violated. In fact, since it is impossible to anticipate all possible such physical events at design time, it is highly desirable to have automated techniques that revise programs with respect to newly identified physical events according to the system specification. Thus, one can observe that while formal software verification plays an important role in ensuring the correctness of systems, it is equally important to address the following fundamental question:

In the face of constant evolution of existing computing systems and their physical environment, how should we revise them according to their specification and how should we cure their vulnerabilities (e.g., failures, time unpredictability, insecurity, etc) in an incremental and automated fashion?

2 Current Results

The notion of program revision (repair) was independently introduced by Bonakdarpour, Ebnenasir, and Kulkarni [Fmics'06, Opodis'05] and Jobstmann, Griesmayer, and Bloem [Cav'05]. In our work, we have focused on developing a theory of automated program revision from different perspectives such as time-predictability, fault-tolerance, and distribution. The main focus of this theory is to identify instances where sound and complete automated revision of programs can be achieved in polynomial-time, and, where it is hard in some class of complexity. Complexity analysis identifies cases where program revision is (1) likely to be successful via developing efficient algorithms and heuristics, or (2) unlikely to have an impact. Completeness of a revision algorithm is important in the sense that if the algorithm fails to revise a program with respect to a property, it implies that the program in its current form is not *fixable* and, hence, a more comprehensive approach (e.g., synthesis from specification) must be applied. Thus far, the theory has been established in the following contexts:

1. We concentrated on automatic addition of untimed (respectively, real-time) Unity properties to programs in the form of a finite state automata (respectively, timed automata) such that revised programs continue to satisfy universally quantified properties of the original program [Fmics'06, Opodis'05].
2. We have extended the basic theory by considering systems where programs are subject to a set of uncontrollable *faults* [Sss'06]. We considered synthesizing three levels of fault-tolerance, namely *failsafe*, *nonmasking*, and *masking*, based on satisfaction of safety and liveness properties in the presence of faults. For failsafe and masking fault-tolerance, we considered two additional levels, namely *soft* and *hard*, based on satisfaction of timing constraints in the presence of faults. In our case studies, besides the factual benefits of automated addition of fault-tolerance, we observed that our synthesis methods can be potentially used to determine incompleteness of specification as well. We also introduced the notion of *bounded-time phased recovery* [Fm'08] where simple recovery to the program's normal behavior is necessary, but not sufficient. For such programs, it is necessary to accomplish recovery in a sequence of phases, each ensuring certain constraints.
3. In order to make synthesis algorithms efficient so that they can be used in tools in practice, we have developed a set of symbolic heuristics for automatic synthesis of fault-tolerant distributed untimed programs [Icdcs'07]. Our experimental results on synthesis of classic fault-tolerant distributed problems showed that synthesis for these problems is feasible for state space of size 10^{30} and beyond. The tool Sycraft (*SYmboliC synthesizeR and Adder of Fault-Tolerance*) implements the aforementioned heuristics.

The correctness of a selection of our synthesis algorithms is verified by the theorem prover PVS [Afm'06, Lopstr'04]. This verification essentially shows that any program synthesized by our algorithms is indeed correct-by-construction.

3 Related Work

Our formulation of the revision problem is in spirit close to *controller synthesis*, where program and fault transitions may be modeled as controllable and uncontrollable actions, and *game theory*, where program and fault transitions may be modeled in terms of two players. In controller synthesis (respectively, game theory) the objective is to restrict a *plant* (respectively, an *adversary*) at each state through synthesizing a controller (respectively, a wining strategy) such that the behavior of the entire system always meets some safety and/or reachability conditions. Note, however, that there are several distinctions. First, in addition to safety and reachability constraints, our notion of fault-tolerance is also concerned with adding new *recovery* behaviors to the given program as well, which is normally not a concern in controller synthesis and game theory. Secondly, we model distributed systems by imposing read-write restrictions over variables of each process in a shared-memory model. Finally, rather than addressing any arbitrary specification, we concentrate on properties typically used in specifying systems.

4 Future Research Directions

A grand challenge in dealing with formal analysis of cyber-physical systems is to develop abstractions, models of computation, formal frameworks, and efficient automated techniques to specify and reason about such systems.
Formal specification of cyber-physical systems. This direction includes (1) structural specification, which models how components work and how they are interconnected, and (2) behavioral specification, which models how each component responds to an internal or external event.
Bridging the gap between specification and implementation. Another direction is to explore mechanisms for ensuring that implementation of cyber-physical systems refines their specification. To this end, one may generalize our existing synthesis/revision algorithms and tools to bridge the gap between formal specification and implementation of *multi-tolerant hybrid* cyber-physical systems.
Establishing interfaces between components operating in different contexts. As recognized by the research community, cyber-physical systems must be reliable, secure, safe, efficient, distributed, and operate in real-time. We plan to study how to express and reason about multiple (and often conflicting) concerns by considering the state of knowledge of agents in a distributed system using *epistemic logic*.
Making the developed methods scalable. The main challenge in developing verification and synthesis algorithms is scalability. Thus, we plan to accommodate *model checking techniques* in the context of program synthesis so that synthesis tools can be exploited by engineers and designers in practice.