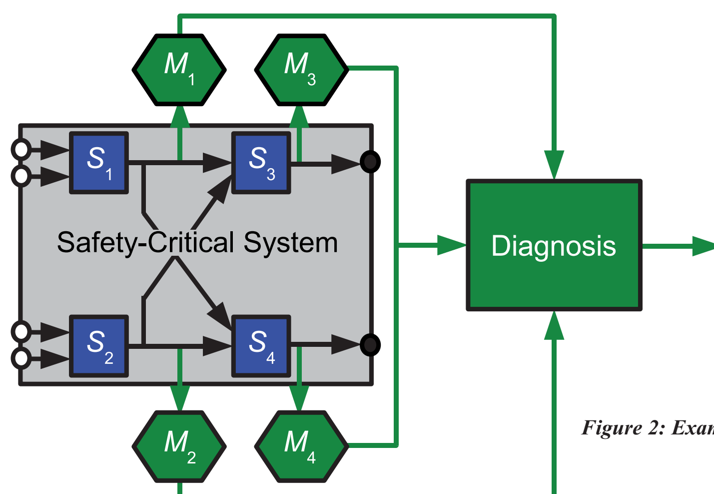

Figure 2: Example of the logical architecture of a reflective system.

system, the generated monitors will respond with an alarm signal for subsequent diagnosis.

Following FDIR, we separate the detection of faults from the identification of failures. The diagnosis layer collects the verdicts of the distributed monitors and deduces an explanation for the current system state. For this purpose, the diagnosis layer may infer a (minimal) set of system components, which must be assumed to be faulty in order to explain the currently observed system state. The procedure is based solely upon the results of the monitors and general information on the system. Thus, the diagnostic layer is not directly communicating with the application. It can easily be implemented in a generic manner based on SAT solving techniques. An example of the logical architecture of a reflective system in shown in Figure 2.

The results of the system's diagnosis are then used to recover the system and if possible mitigate the failure. However, depending on the diagnosis and the particular failure, it may not always be possible to re-establish a particular system behaviour. In some situations, such as the occurrence of fatal errors, a recovery system may only be able to store detailed diagnosis information for offline treatment.

In cooperation with NASA JPL, runtime verification and runtime reflection techniques are currently being enhanced and tailored for application in the embedded systems world, as found for example in the automotive sector.

Links:
http://www.runtime-verification.org
http://runtime.in.tum.de

Please contact:
Martin Leucker
Technical University Munich, Germany
Tel: +49 89 289 17376
E-mail: leucker@informatik.tu-muenchen.de

LaQuSo: Using Formal Methods for Analysis, Verification and Improvement of Safety-Critical Software

by Sjaak Smetsers and Marko van Eekelen

Due to its great complexity, it is inevitable that modern software will suffer from the presence of numerous errors. These software bugs are frequent sources of security vulnerabilities, and in safety-critical systems, they are not simply expensive annoyances: they can endanger lives. The growing demand for high availability and reliability of computer systems has led to a formal verification of such systems.

There are two principal approaches to formal verification: model checking and theorem proving.

Goal

The goal of this project is to develop a methodology for improving the quality of software by combining model checking with theorem proving such that the advantages of both methods are fully exploited. Additionally, we enhance existing or develop new techniques to support the various translations of conversions that are needed in this process.

The proposed methodology resembles what in software engineering is called code refactoring. However, the latter replaces existing code with the same functionality. It does not fix bugs; rather it improves the understandability of the code by changing its internal structure.

Approach

The proposed methodology consists of the following steps:

- the software system is analysed to identify and localize 'suspicious' parts
- a formal model of these parts is created
- the model is verified by the model checker and, if necessary, improved until it is approved by the checker
- for full reliability the model is converted into a PVS specification. PVS (Prototype Verification System) is a proof tool providing a theorem specification language and support for developing correctness proofs. This tool is used to construct a full formal proof of the system
- from this PVS specification new code is derived that replaces the original (possibly erroneous) code. In this way one obtains highly reliable code, since it corresponds directly to a fully proven formal model.

These steps are illustrated in Figure 1. We will briefly discuss each of them.

- *Identification.* Most of the time, the software system will be too large to be handled completely. Instead one must first identify the safety-critical or erroneous parts. The process of determining the parts of a program relevant to the property to be checked is often called slicing.

- *Abstraction.* There exist several techniques by which formal models can be obtained from source code. Due to the state explosion problem, a direct instruction-wise translation of source code is unsuitable. The methods for alleviating this problem can be divided into two categories: abstraction and symbolic evaluation. With abstraction, the state space is reduced by employing specific knowledge about the system in order to model only the relevant features. In the symbolic evaluation approach, the abstraction occurs at the level of the model itself. Using succinct data structures and symbolic algorithms, state information can be compressed, thus helping to keep state explosion under control.

- *Conversion.* At present, no tool exists to convert a model into a PVS specification. This is future work.

- *Code generation.* From the PVS specification, code will be generated that can replace the original (possibly faulty) code. Most theorem provers (Isabelle, Coq, PVS) are already equipped with code generators, which have in common that they produce functional code. We are currently developing a generator capable of generating code for imperative languages like Java. An interesting aspect is the extension of Java code with proof information leading to so-called proof-carrying code.

The figure has two feedback steps:

- *Counterexample.* If the attempt to check the abstract model fails, there is a counterexample which can be used directly to adjust the source code, provided that the counterexample was genuine. If the latter is not the case, our abstract model is probably too coarse and must be refined further. A technique is developed which iteratively uses spurious counter¬examples to create an abstract model with the right level of abstraction.

- *Validation.* The theorem-proving process is very sensitive to the way in which properties are formulated. In the case of state transition systems, this boils down to the right formulation of invariants. The problem with these invariants is that they are usually very subtle and therefore very difficult to specify. One may easily overlook a detail, thus making the invariant invalid. Instead of immediately checking invariants in the theorem prover itself, we propose using the model checker to quickly trace and repair flaws before we start proving the invariants. This step requires a translation of PVS properties to equivalent statements in the language of the model checker. As far as we know, this has not been done before.

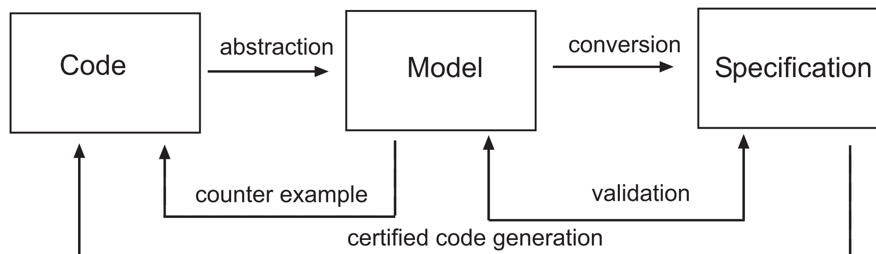

Figure 1: The LaQuSo FM based approach.

Applications

We apply our methodology in the context of the Laboratory for Quality Software (LaQuSo), which is a joint activity between the Technical University Eindhoven and the Radboud University Nijmegen. Within LaQuSo, successful case studies have been performed for safety-critical software components. In the near future we expect to apply our approach to cases such as control software for the Dutch 'Maeslantkering' water barrier and to programmable hardware to replace traditional components in nuclear plants

Links:
LaQuSo:
http://www.laquso.nl

Related publications:
http://www.cs.ru.nl/~marko/research/pubs

Please contact:
Marko van Eekelen
Radboud University Nijmegen, The Netherlands
Tel: +31 24 365 3410
E-mail: M.vanEekelen@cs.ru.nl