Random-Based Methods for Test-Case-Generation of C Programs

Phd Proposal

Motivation:

The programming language C is widely used for operating system level code which is both complex (due to the need for efficiency and the proximity to hardware) and safety critical. Building environments to verify C code has therefore attracted the interest of many research groups, leading to environments such as Frama-C or Visualstudio/VCC.

In this Phd, the goal is to explore static testing methods for a realistic subset of C, i.e. test-data are generated according certain criteria and techniques and run against „the real code“. A particular emphasis in this Phd is put on testing data-invariants of pointer structures, but also classical call-graph based test-case coverage criteria are considered, which were represented by an LTL like assertion language.

These assertions can be represented by graphs which can be checked against the set of „object graphs“ that can potentially evolve at runtime. Since such checks are expensive, therefore, this Phd will try to explore random-based techniques as developed on the Rukia-system of Johan Oudinet to alleviate the task: Instead of checking the entire graph for a particular assertion against the set of potential system states, the check will concentrate only a uniform random sample of paths in this graph. For them, path-conditions were generated representing a particular execution path on the concrete program, and concrete input can be constructed by applying constraint resolving techniques via constraint solvers such as Z3 or Alt-Ergo.

A substantial problem of this approach lies in the fact that random-based exploration requires a good approximation to the set of „feasible paths“, i.e. those paths for which the path-condition is actually satisfiable; approaches based on taking just the control-flow-graph of the program turned out to be ineffective because the likelihood to randomly choose an feasible path may be extremely low in practice. A major challenge is therefore to find a combination of:

- a „reasonably realistic“ memory model for C
- a combination of invariant-construction, abstract interpretation and code-slicing
- ... in order to get a good approximation of feasible paths before performing random-selection.

The practical part of this Phd consists in

- Integrating pre-existing AUGUSTE/RUKIA into Frama-C
- Developing an assertion language for data-structure consistency based on LTL and the pre-existing ADCL2 (Frama C Jessie Plugin)
- implementing new invariant-construction, abstract interpretation and other techniques (such as Dekkers Algorithm) in order to perform code-slicing before control-flow-graph construction.

- careful statistic exploration of larger program samples.

Example:

Example SortedList:

```
        variant struct obj =        Node {int content ; *obj  next}
                                    Empty{}
```

Explanation:

3.1 Temporal Properties

Temporal logics [7] are a powerful tool for expressing complex properties on the behaviour of systems. These modal logics provide, in addition to the traditional logic operators, temporal operators in such a way that it is possible to describe properties that hold along all (or some) executions of a system. Thus, for instance, temporal operators in LTL are $\bigcirc$ (*next*), $\Box$ (*always*), $\Diamond$ (*eventually*) and U (*until*). Formulae in LTL evaluate on execution traces, or sequence of states, starting at the first state, and recursively passing through the other states in a sequential manner. Informally, if f is an LTL formula, the meaning of the temporal operators is as follows. $\bigcirc f$: f must hold in the next state; $\Box f$: f must hold in every future state; $\Diamond f$: f must hold in some future state; and rUf: r must be true until, in some future state, f holds. Moreover, temporal logics can be used not only to specify properties over the system behaviour, but also over *the shape and content of dynamic data structure used during the system execution* as shown in [9].

When testing Java code, it is useful to consider both types of LTL properties:

1. Properties over dynamic data structures. In this case, LTL formulae are not evaluated on sequences of system states, as usual. On the contrary, they refer to some dynamic data structure (that can be naturally seen as a Kripke structure) and they are evaluated on it. Following this approach, we define the following properties over class SortedList given in Section 2.1:
    (a) $[l{:}header]\Box(next != null \rightarrow next{:}value{:}$compareTo$(value) \geq 0)$. This formula states that the list referenced by $l{:}header$ is always sorted upward.
    (b) $[l{:}header]\Box(next != null \rightarrow next{:}value{:}$compareTo$(value) \leq 0)$. This formula states that the list referenced by $l{:}header$ is always sorted downward.
    (c) $[l{:}header]\Box($valid$(next) \wedge \Diamond(next == null)$. This formula states that list $l{:}header$ is always a well formed linked list.

2. Properties over sequences of statements. In this case, LTL formulae refer to the sequences of states that constitute the executions of a system. This is the common use of logic LTL in model checkers like Spin. For instance, we can specify the following formulae over class SortedList:
    (a) $\Box l{:}nelements ==$ length(l). This formula states that field $nelements$ of list l and the length of l are equal in every state. Note that $length(l)$ is an external method that traverses the list and returns the length of the list starting at $l{:}header$. This formula is useful to detect memory leaks due to errors in the insert or remove methods provided by SortedList.
    (b) $\Box(\neg l{:}$isEmpty$() \rightarrow \Diamond l{:}$isEmpty$())$. This formula states that if the list has some element, then sometime in the future, the list will be empty, that is, the elements of the list will be removed.

It is worth noting that when using C objects in the LTL formulae, we are invoking a C runtime environment, and then giving the results to Spin. To do this, the value returned by the methods can not be a reference value, but it has to be a simple data that can be handled by Spin.

Collaborations:

- Benjamin Monate, CEA, [Frama-C]
- Frederic Voisin, ForTesSE [System-Integration, C Framework]
- Johan Oudinet, Alain Denise, Marie-Claude Gaudel [Rukia, Random-based testing]
- Groupe INRIA/LRI Demon

Bibliography:

1. Junit testing framework. http://www.junit.org.
2. P. de la Cámara, J. Castro, M. M. Gallardo, and P. Merino. Verification support for arinc-653-based avionics software. *Software Testing, Verification and Reliability*, page to be published, 2010.
3. E. A. Emerson. Automated temporal reasoning about reactive systems. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata, pages 41–101, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.*
4. *M. M. Gallardo, P. Merino, and D. Sanán. Model checking dynamic memory* allocation in operating systems. *Journal of Automated Reasoning*, 42(2):229–264, 2009.
5. Sascha Böhme, Michal Moskal, Wolfram Schulte, Burkhart Wolff: HOL-Boogie - An Interactive Prover-Backend for the Verifying C Compiler. J. Autom. Reasoning 44(1-2): 111-144 (2010)
6. G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.
7. D. Peled. Model checking and testing combined. In *ICALP*, pages 47–63, 2003
8. The Rukia Home Page: http://www.lri.fr/~oudinet/en/project.html
9. Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, et Sylvain Peyronnet. **Coverage-biased random exploration of large models**. Dans *ETAPS, Workshop on Model Based Testing, ENTCS*, pages 3-14, Mars 2008.
10. The Frama-C Home Page: http://frama-c.com/.
11. Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, Armand Puccetti: Experience report: OCaml for an industrial-strength static analysis framework. ICFP 2009: 281-286