Random-Based Methods for Runtime-Testing C Data-Structures

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 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 runtime-testing methods for a realistic subset of C. Runtime-testing means, that specifications for C code were represented by assertions which were compiled to optimized code that is injected into the running executions. A particular emphasis in this Phd is put on testing data-invariants, which were represented by an LTL like assertion language. These assertions can be represented by graphs which can be checked against the „real object graph" produced at runtime in the program. Such checks are ex–pensive, 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 the entire graph for a particular assertion, the check will concentrate only a uniform random sample of paths in this graph. Since programs tend to do little increments on data-structures, it can be assumed that randomized checks will have an accumulating effect on accurancy while not revealing a too large penalty wrt. runtime costs. However, random-sampling methods such as Rukia require „the entire graph" and provide a sampling for a given depth; however, this information is only known at runtime. Rukia provides methods to partially pre-compile automata-products, though, such that at least the part of the graph stemming from the specifi cation can be treated statically. It remains to explore how the costs for re-sampling can be deminished by knowledge available at runtime in order to make the overall method effective.

The practical part of this Phd consists in

- Integrating AUGUSTE/RUKIA into Frama-C
- Developing an assertion language for data-structure consistency based on LTL
- Converting LTL into Automatas for which RUKIA computes
  - a pre-conceived random selectio
  - or: a Markov Automata
  - or: a function that takes a current object-graph and compiles them into test samples turn this into code for run-time testing C data-invariants
- 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 O (*next*), □ (*always*), ◇ (*eventually*) and *U* (*until*). Formulae in LTL evaluate on execution traces, or sequence of states, starting at the fi rst 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. O*f*: *f* must hold in the next state; □*f*: *f* must hold in every future state; ◇*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 defi ne the following properties over class SortedList given in Section 2.1:

(a) [*l:header*]□(*next*! = *null* → *next:value:*compareTo(*value*) ¸ 0). This formula states that the list referenced by *l:header* is always sorted upward.

(b) [*l:header*]□(*next*! = *null* → *next:value:*compareTo(*value*) · 0). This formula states that the list referenced by *l:header* is always sorted downward.

(c) [*l:header*]□(valid(*next*)⋀◇(*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) □*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) □(¬*l:*isEmpty() → ◇*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]

Bibliography:

1. Junit testing framework. http://www.junit.org.
2. P. de la Cámara, J. Castro, M. M. Gallardo, and P. Merino. Verifi cation support for arinc-653-based avionics software. *Software Testing, Verifi cation 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. RUKIA
9. Frama-C