Temporal Assertions with Parametrised Propositions

Volker Stolz `<vs@iist.unu.edu>`
United Nations University
Institute for Software Technology (UNU-IIST)

January 26, 2007

In this work, we present an extension to our previous approach to runtime verification of a single finite path against a formula in Next-free Linear-Time Logic (LTL) *with free variables and quantification.*

We introduce *parametrised propositions* that consist of a proposition name $(p, q, \ldots)$ with arity. The payload of such a proposition occurring on a trace contains values from some *object domain* according to its arity. In a formula, a proposition contains the appropriate amount of variables, e.g. $p(X, Y)$ or $q(Z)$.

Variables get instantiated if a proposition matches during evaluation of a trace. Multiple occurrences of the same variable are permitted and work similar to Prolog: if a variable is already bound when a proposition is evaluated, both the proposition occurring in the current state and the bound variables must match.

From our experience with J-LO, the Java Logical Observer [2, 1], we found it necessary to distinguish between read and write accesses to variables, based on a static analysis of the formula. Furthermore, evaluation of uninstantiated propositions had to be considered. As interpretation (through a human) of those formulae resulted difficult and error prone due to the binding semantics, in this article we introduce a special binary *binding operator* $\dot{\rightarrow}$ that simplifies our design in the following aspects:

- simpler binding semantics
- no static analysis necessary
- more general through quantification.

The left-hand side contains a single parameterised proposition, the right-hand side a temporal parametrised formula that may refer to the variables bound in the proposition, e.g.

$$\psi := p(X) \dot{\rightarrow} \varphi(X).$$

Negation is only permitted in propositional subformulae, We call the entire construct a *binding expression.*

Furthermore, we require that every variable occurring in a parametrised formula has previously been bound through the left-hand side of a binding operator. We can thus ensure by construction that evaluation will only encounter completely instantiated propositions, i.e. propositions, where a value for every variable is known. If the left-hand side does not match the current state during evaluation, the overall expression is evaluated to *false*.

Quantification plays a role when more than one matching proposition holds in the current state. Matching the proposition $p(X)$ against the state $\{p(1), p(3)\}$ yields two distinct bindings for variable X: $X/1$ and $X/3$. Quantifiers may only occur together with a parametrised proposition on the left-hand side of the binding operator. In a binding expression, all newly introduced variables through a proposition must also be quantified.

Additionally to the usual notion of LTL formulae augmented by quantified variables and bindings, we also permit *predicates* and *functions* over bound variables that can be used, for example, to compare values for inequality.

As an example, we consider the Lock-Order Reversal pattern [3], which captures a common error pattern where two processes repeatedly compete for two resources (locks), albeit in different order. This behaviour has the potential for a dead lock which can be detected by monitoring the order in which each process locks/unlocks the resources.

$$\begin{aligned}\Psi = \mathbf{G}\ &[\forall t_i \forall l_x : \mathtt{lock}(t_i, l_x) \mathbin{\dot{\rightarrow}} ([\neg\mathtt{unlock}(t_i, l_x)\ \mathbf{U}\ \exists l_{z'} : \mathtt{lock}(t_i, l_{z'}) \mathbin{\dot{\rightarrow}} l_{z'} \neq l_x] \\ &\quad \rightarrow \neg\mathtt{unlock}(t_i, l_x)\ \mathbf{U}\ \exists l_z : \mathtt{lock}(t_i, l_z) \mathbin{\dot{\rightarrow}} [l_z \neq l_x \\ &\qquad \wedge \forall l_y : \mathtt{lock}(t_i, l_y) \mathbin{\dot{\rightarrow}} (l_y \neq l_x \wedge G\ \neg(\exists t_j : \mathtt{lock}(t_j, l_y) \mathbin{\dot{\rightarrow}} [t_i \neq t_j \\ &\qquad\quad \wedge (\neg\mathtt{unlock}(t_j, l_y)\ \mathbf{U}\ \mathtt{lock}(t_j, l_x))]))])]\end{aligned}$$

`lock` and `unlock` are binary propositions, binding a thread- and a lock-id, $\neq$ is a predicate.

A declarative semantics is given by expanding quantified variables through values from the finite object domain and combining them through conjunction or disjunction according to the quantifier. Operationally, evaluation of such a Temporal Assertion proceeds by means of a variant of Alternating Finite Automata, augemented with a dictionary to maintain the current bindings for each subformula. For runtime verification, we give an algorithm based on sets in disjunctive normal form that traverses the automaton in a breadth-first fashion which requires processing each state in a path exactly once and in order. It is thus suitable for online checking where an error should be detected immediately.

References

[1] E. Bodden and V. Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In W. Löwe and M. Südholt, editors, *5th International Symposium on Software Composition (SC'06)*. To be published in Lecture Notes in Computer Science, Springer, 2006.

[2] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors, *Fifth Workshop on Runtime Verification (RV'05)*, volume 144 of *Electr. Notes in Theor. Comput. Sci.* Elsevier, 2005.

[3] V. Stolz and F. Huch. Runtime Verification of Concurrent Haskell Programms. In K. Havelund and G. Roşu, editors, *Proceedings of the Fourth Workshop on Runtime Verification (RV'04)*, volume 113 of *Electr. Notes in Theor. Comput. Sci.*, pages 201–216. Elsevier, 2005.