Testing Decision Procedures for Security-by-Contract: Extended Abstract [*]

Nataliia Bielova Fabio Massacci Ida Siahaan
Universitá di Trento, Italy
name.surname@disi.unitn.it

Abstract

The traditional realm of formal methods is off-line verification of formal properties of hardware and software. We report a different approach that uses formal methods (namely the integration of automata modulo theory with decision procedures) on-the-fly, at the time an application is downloaded on a mobile application such as PDA or a smart phone.

The idea behind security-by-contract is that a mobile applications comes equipped with a signed contract describing the security relevant behavior of the application and such contract should be matched against the mobile platform policy. Both specified as automata modulo theories and the operation is an on-the-fly emptiness test where edges are not only finite states of labels, but rather expressions which capture infinite transitions such as "connect only to urls starting with https://".

We will talk about prototype implementation, its integration with a state of the art decision solver (based on MathSAT and NuSMV) and the preliminary experiments for contract-policy matching.

1 Prototype Implementation

We implemented language inclusion as on-the-fly emptiness test a-la-SPIN with oracle calls to the decision procedures available in NuSMV [1]. Therefore, our design decision of automata modulo theories $\mathcal{AMT}$ makes reasoning about infinite transitions systems with finite states possible without symbolic manipulation procedures of zones and regions, or finite representation by equivalence classes whose memory intensive characteristic is not suitable for our application.

Our final objective is to do a run-time matching of the mobile's platform policy (called *policy*) against the midlet's security claims (called *contract*) expressed as $\mathcal{AMT}$. First, we implemented the contract-matching prototype in Java for Desktop version. Then, we ported the prototype into .NET on an HTC P3600 (3G PDA phone). We made experiments on both implementations to see the feasibility and to select the best design alternative.

Our algorithm checks whether or not the *contract* matches the *policy* using on-the-fly emptiness check. The on-the-fly procedure takes as input a contract automaton and a *complemented* policy automaton. The decision procedure part interacts with the SMT solver NuSMV[1] for satisfiability checks. The instance of the NuSMV class is created only once at the beginning of the On-the-Fly procedure; then we declare variables, add constraints and remove constraints from the library every time we call the solver. We used software Java SDK version 6 and Apache Ant[2] to compile the java sources and to run the tools automatically (see [2] for details).

2 Experiments on Desktop and on Device

To *decide the best configuration of integration with decision procedure*, we made different design decisions and run experiments on the alternatives. This analysis is important because of the resource constraints of mobile device; for achieving our goal even small changes in time makes sense.

[*]This work is partly supported by the project EU-FP6-IST-STREP-S3MS (www.s3ms.org). We would like to thank M. Roveri and A. Cimatti for the support in the usage of the NuSMV and MathSAT libraries and for hammering down a decision procedure for URLs. We also acknowledge Marco Dalla Torre for support in the integration of the tools.

[1]http://nusmv.fbk.eu/

[2]http://ant.apache.org/

Table 1: Running Problem Suit 10 Times

MUTEX_MC ONE_INSTANCE CACHING_SOLVER

Problem	Desktop				Mobile				Result
	ART (s)	CRT (s)	SV	TV	ART (s)	CRT (s)	SV	TV	
P1	2.4	2.4	2	6	4.3	4.3	2	6	Match
P2	2.4	4.8	2	6	4.1	8.4	2	6	Match
P3	2.4	7.2	3	11	3.9	12.3	3	11	Match
P4	2.4	9.6	2	6	4.0	16.3	2	6	Match
P5	4.7	14.3	3	11	4.1	20.4	3	11	Match
P6	2.9	2.9	4	4	3.8	3.8	3	6	Not Match
P7	2.8	5.7	5	7	3.8	7.6	2	4	Not Match
P8	2.9	8.6	5	7	3.8	11.4	3	6	Not Match
P100	9.3	9.3	102	307	11.3	11.3	102	307	Match

(a) Running Problem Suit

ART: Average Runtime for 10 runs SV: Number of Visited States
CRT: Cumulative Average Runtime TV: Number of Visited Transitions

(b) Abbreviations

We faced a number of design options: (1)One_vs_Many We could either create only one instance of solver, relying on the solver to assert and retract expressions on demand, or create a new instance of the solver every time we call the decision procedure. (2)MUTEX_SOLVER Method names are declared as mutex constants at the moment of declaring all variables on the solver, due to an edge in the automaton (correspond to a method) which is incompatible with another edge (correspond to a different method). MUTEX_MC allows the on-the-fly algorithm to check whether method names are the same. PRIORITY_MC Guards are evaluated using *priority or* and we can optimize the expressions sent to the decision procedure as minimized expression. CACHING_MC We saved time by caching the results of the matching. The solver itself has a caching mechanism that could be equally used (CACHING_SOLVER). However, One_vs_Many decision was not possible to be taken because of the garbage collection management both by the Java virtual machine and by the libraries of MathSAT/NuSMV (only one instance of solver exists at time).

We ran our experiments on a Desktop PC (Intel(R) Pentium(R) D CPU 3.40GHz, 3389.442MHz, 1.99GB of RAM, 2048 KB cache size) with operating system Linux version 2.6.20-16-generic, Kubuntu 7.04 (Feisty Fawn). We also ported the prototype to the mobile, namely HTC P3600 (3G PDA phone) with ROM 128MB, RAM 64MB, Samsung®SC32442A processor 400MHz and operating system Microsoft Windows Mobile®5.0. The experiments were made for different design decisions for different problem suits(see [3] for details). The problem suits covered different kind of categories such as network connectivity, use of costly functionalities, or private information management. Most problems have few states and transitions as a result the matching algorithm runtime is little (around 2.5s for running on Desktop and 4s for running on mobile). Even for pathological problem with 102 visited states, the runtime while running on a mobile platform is still acceptable (9.3s for Desktop, 11.3s on mobile).

We collected data on resources used, namely number of visited states, number of visited transitions, running time for each problem in each design alternative, and the number of solved problems against time. All methods seem to perform equally well and promising for the deployment on resource constrained in mobile device. Our current implementation uses PRIORITY_MC ONE_INSTANCE CACHING_MC configuration. PRIORITY_MC is preferred because of the nature of rules in policies which is *priority or*, also because MUTEX_SOLVER does not allow empty methods such as $\neg m_i \wedge \neg m_j$ which is possible in the matching algorithm. ONE_INSTANCE is chosen because of garbage collection problem. CACHING_MC is desired in order to save calls to solver for the already solved rules.

References

[1] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. of CAV'02*, 2002.

[2] N. Bielova, M. Dalla Torre, N. Dragoni, and I. Siahaan. Matching policies with security claims of mobile applications. In *Proc. of the 3rd Int. Conf. on Availability, Reliability and Security (ARES'08)*. IEEE Press, 2008.

[3] N. Bielova, F. Massacci, and I. Siahaan. Testing decision procedures for security-by-contract. In *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*, 2008. submitted.