

ESAT: A Tool for Animating Logic-Based Specifications of Evolvable Component Systems

Djihed Affi, David E. Rydeheard, and Howard Barringer

The University of Manchester, School of Computer Science,
Kilburn Building, Oxford Road, Manchester, M13 9PL, United Kingdom
{djihed,david,howard}@cs.man.ac.uk

1 Introduction

An increasingly important area of runtime monitoring is the incorporation of techniques for diagnosis and repair, for example, in autonomic control applications [9], in robotics, and in e-business process change [12]. In particular, a runtime monitor becomes a ‘supervisor’ - a process which not only monitors but may evolve the running system dynamically. In [4], a framework for the logical modelling of hierarchically structured supervised component systems was set out. The modelling captures the following key behavioural concepts: at runtime, a supervisory component can (i) monitor its supervisee to ensure conformance against desired behaviour, (ii) analyse reasons for non-conformance, should that arise, (iii) evolve its supervisee in a pre-programmed way following diagnosis, or via external stimulus received from higher-level supervisory components. Structurally, components may contain sub-components, actions over the state of the component, and programs over the actions. In this logical framework, components are specified by first-order logic theories. Actions are either basic revisions to the state of the component or combinations of actions. Crucially, a supervisory component is treated as a logical theory meta to its supervisee, thus providing access to all facets of the supervisee’s structure. A supervisory component program is executed meta to its supervisee’s program. Synchronisation between the two may occur through a variety of schemes, from lock-step synchronisation to asynchronous execution with defined synchronisation points. A supervisory program action may evolve its supervisee by making changes to its state, to its actions, to its sub-components, or to its program. This occurs in the logical framework via a theory change induced from the meta-level.

The logical framework introduces a new design methodology whereby evolutionary concerns are built into system designs at various levels. The hierarchical aspect of this framework allows for localised monitoring and evolution, improving the manageability of evolution in large systems. As the logic for specifying supervised component systems is revision-based, programs over the actions can be directly executed. This execution is performed using ESAT, the Evolvable Systems Animator Tool. ESAT is written in Java and makes use of automatic theorem provers to simulate systems. ESAT animates abstract logical specifications of evolvable systems. The tool was developed in order to support case

studies which explored and tested this particular design methodology and to prototype systems. Furthermore, the animation of logical models enables the reasoning and verification of various properties of models of evolvable systems. ESAT differs from other formal specification tools such as Perfect Developer [6] and Maude [11] in its support for meta-level descriptions and runtime evolutionary change.

2 ESAT: Evolvable Systems Animator Tool

2.1 Input Component Specifications

An overview of the tool is given in Fig. 1. The input to the tool is a textual representation of component schema definitions. Figure 2 outlines an example specification of an autonomic rover system. It consists of three specifications of theories: *Rover* is an abstract description of an autonomous rover system, *Planner* is a specification of a supervisor as a planning agent, and *Supervised_Rover* combines instances of the previous specifications. The logic of the specification is a many-sorted (typed) first-order logic with enumerated types, sub-typing, product types and lists. A schema may declare types, functions, predicates, sub-components, constraints, an initial state, actions and a program. In addition to the system-wide types such as *Int* and *String*, a schema may introduce its own types as well as functions. A schema may introduce predicates as either: (i) *observation* predicates: the state of a component is defined as a subset of the positive ground atoms of these predicates (ii) *abstraction* predicates: other predicates that may appear in the schema's constraints or actions. For a supervisor component, special predicates are used for its meta relation with its supervisee. As an example, the supervisor formula "*holds(ϕ, c)*" is used to denote that an object formula ϕ holds at the supervisee level at an object configuration named by the supervisor as c . A schema may define sub-components that will be instantiated from other schemas. Components can be either standalone or supervisor-supervisee pairings. The constraints of a schema are a set of parametrised schematic first-order logic formulae. Actions can be one of four kinds (i) basic actions specified in a STRIPS-style via pre-conditions, and additions and deletions of state atomic formulae, i.e. via state revisions. Pre-conditions may contain free variables that are bound at execution time, (ii) paired actions: the pairing of a supervisor action with a supervisee action, (iii) joint actions: the lock step parallel execution of several actions from several components, (vi) choice actions: the non-deterministic choice between several actions. ESAT incorporates a guarded choice language (with non-deterministic choice) with iteration for specifying a program for each component.

2.2 Animation

For animating a specification, ESAT provides both a command line interface (CLI) and a graphical user interface (GUI). The CLI simulates an input specification from start to finish (or to a pre-determined action count execution limit) without user intervention and is suitable for running large simulations. The GUI

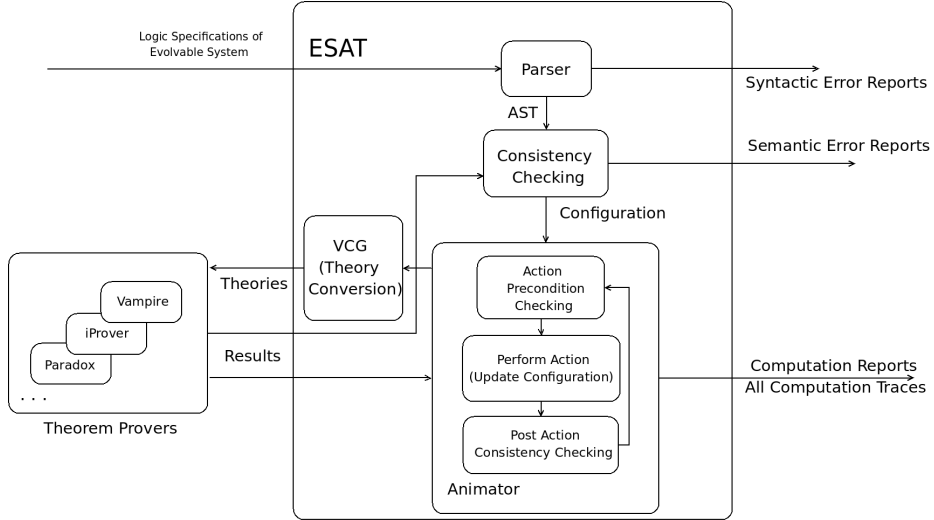


Fig. 1. ESAT Overview

<pre> Rover OBSERVATION PREDICATES at : Point next : Point blocked : Points ABSTRACTION PREDICATES obstruction : Point x Point between : Point x Point x Point CONSTRAINTS geometry $\stackrel{dfn}{=} \dots$ ACTIONS view(h : Point, P : Points) pre {at(h), blocked(Q)} add {blocked(P \cup Q)} del {blocked(Q)} setHeading(t : Point) ... drive ... PROGRAM NULL. INITIALLY {} Supervised_Rover COMPONENTS ER : (P : Planner() META TO R : Rover()) PROGRAM ... </pre>	<pre> META Planner TYPES ConfigName CONSTANTS cR : ConfigName FUNCTIONS s : ConfigName \rightarrow ConfigName replan : Point x Point x Points \rightarrow PROGRAM OBSERVATION PREDICATES holds : FORMULAE x ConfigName evolve : ... current : ConfigName goal : Point CONSTRAINTS EWC $\stackrel{dfn}{=} \dots$ ACTIONS query(P : FORMULAE) ... observe(P : FORMULAE, c : ConfigName) ... rePlan(p_s, p_d : Point) ... PROGRAM [goal(p) \wedge current(c) \wedge holds(at(s), c) \wedge start \neq p \rightarrow rePlan(s, d) goal(p) \wedge current(c) \wedge holds(at(p), c) \rightarrow query(at(p))]* INITIALLY {current(cR)} </pre>
--	--

Fig. 2. Rover Example Specification

allows for running, pausing and stepping through a simulation, and provides detailed feedback about the actions being executed and the state of the system at any point during animation.

The input specification is animated by executing its programs and producing a tree of all traces. This animation requires the use of automated theorem provers (see [1]) in order to: (i) Verify the consistency of the theory and of the state of each component at system start-up, after executing actions and after performing evolutions. This checking may throw logical error reports when the theory of a component does not have a model i.e. it is unsatisfiable, (ii) Establish the validity of each action's pre-conditions from the component's theory and state, (iii) Check that the meta-level relations hold, i.e. that the supervisee's reflection in the supervisor in each supervisor-supervisee pair is correct.

ESAT can use any automated theorem prover for first-order logic which supports the TPTP [16] format and which has the capability to determine the satisfiability of a set of formulae and the deducibility of a formula from a formula set.

Although the component specifications use typed formulae, these are encoded as untyped first-order formulae by ESAT as TPTP input to the theorem provers. The translation from typed to untyped logic adds axioms and predicates to encode typing information (see [8]). When a decision is required from a theorem prover, multiple provers may be fired in parallel: theorem provers differ in their proof capability and speed of decision making. The tool supports using different theorem provers for satisfiability and deducibility. We found that using multiple theorem provers increases the overall simulation speed by an average of 30% in our case studies as it is often the case that one theorem prover will be particularly fast at the given problem. We have experimented with 14 theorem provers and model finders, with the main emphasis on Paradox [5], iProver [10], Vampire [14] and E [15].

The animation of a system potentially generates a large number of proof obligations. As the overhead of discharging proof obligations can be as much as 90% of the running time, the simulation of a simple system may spend a substantial amount of time communicating theories and results with theorem provers. A simple caching mechanism is used by ESAT to eliminate, in the examples we have run, over 60% of these obligations. For each component, the caching mechanism associates a mini-cache that stores the list of previously proven formulae as well as the set of previously unproven sets of formulae. In the case of a cache hit, the lookup is much faster than firing external theorem provers. The caching also improves the performance of theorem provers by supplying the previously proven formulae as axioms. On the Rover system, the caching mechanism reduces the simulation time from 6 minutes to 90 seconds.

A problem with running large system simulations is the complexity of the execution trace and the difficulty of inspecting a tree of traces in a linear fashion. The tool facilitates system modelling and execution by providing a GUI with the following views:

- A text editor with syntax highlighting for creating system specification files. Import statements can be used to include schema definitions from multiple files.
- A trace viewer that graphically displays a system’s execution trace. The user can step through the execution of a specification’s programs. The proof obligations that were generated for each action are displayed for inspection. Statistics about the number of proof obligations, the provers that successfully returned a result and the proof obligations that were eliminated using the caching mechanism are summarised in this view.
- A configuration viewer that graphically displays the system’s component tree and the properties of each component such as its state and program. This view is helpful for examining evolutions that change the hierarchical structure of a system.
- A theory editor for testing logical theories. In this view, the user can directly write theories in the input format of the theorem provers and test satisfiability or deducibility. Proof obligations that were generated during the simulation of a specification can be verified here to examine the model generated in satisfiability mode or the proof generated in deducibility mode. This is useful for refining the theories of the component schema specifications.

3 Concluding Remarks

ESAT has been used on a variety of case studies which include (i) an evolvable version of the traditional ‘blocks world’ in which a supervisor monitors a table and blocks being moved around the table, and can invoke changes to the system e.g. changing the table size or number of tables, (ii) a simple model of a banking system comprising a network of ATMs in which not only are there standard local evolutions such as stocking notes and upgrading card readers, but also diagnostic system-wide recognition of potential fraud and evolution of security mechanisms, and (iii) an abstraction of a hierarchic reactively planned autonomous rover where a putative rover’s exploration plan can be updated by supervisors as more information about the environment and the rover’s internal state is received and analysed. The development of the tool and animation of these case studies enabled us to thoroughly test both the mathematical setting and the syntactic descriptions of this logical framework for evolvable systems. In the future, we see ESAT as both a tool for prototyping evolvable systems and also as part of a runtime monitoring system of implemented (e.g. in Java) evolvable systems.

ESAT is still under development. Currently, only TPTP theorem provers that accept classical first-order logic are used to determine the computation steps. System specifications that use arithmetic or other theories such as lists or arrays need to axiomatise the theories as first-order logic formulae suitable for these provers, e.g. Presburger Arithmetic. In the future, TPTP theorem provers that support arithmetic such as SPASS+T [13] and MetiTarski [2] will be explored. Also, a new verification condition generator needs to be implemented to use SMT solvers that support the INTS, REALS and ARRAYS theories, such as Z3 [7] and CVC [3]. Furthermore, ESAT can be extended to enable the runtime monitoring and evolution of components written as real Java programs.

References

1. Affi, D., Rydeheard, D., Barringer, H.: Automated reasoning in the simulation of evolvable systems. In: Workshop on Practical Aspects of Automated Reasoning (PAAR 2010), Edinburgh, UK (2010)
2. Akbarpour, B., Paulson, L.C.: Towards automatic proofs of inequalities involving elementary functions. PDPAR 2006: Pragmatical Aspects of Decision Procedures in Automated Reasoning, p. 27 (2006)
3. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
4. Barringer, H., Gabbay, D., Rydeheard, D.: Modelling evolvable component systems: Part I: A logical framework. *Logic Jnl IGPL* 17(6), 631–696 (2009)
5. Claessen, K., Sörensson, N.: New techniques that improve MACE-style model finding. In: Proc. of Workshop on Model Computation, MODEL (2003)
6. Crocker, D.: Perfect developer: A tool for object-oriented formal specification and refinement. tools exhibition notes at formal methods europe. In: Tools Exhibition Notes at Formal Methods Europe, p. 2003 (2003)
7. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Enderton, H.B., NetLibrary, I.: A mathematical introduction to logic. Academic Press, New York (1972)
9. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* 36(1), 41–50 (2003)
10. Korovin, K.: iProver - an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
11. Lincoln, P., Clavel, M., Eker, S., Meseguer, J.: Principles of maude. In: Meseguer, J. (ed.) *Electronic Notes in Theoretical Computer Science*, vol. 4, Elsevier Science Publishers, Amsterdam (2000)
12. Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H., Occhipinti, C.: ArchWare: Architecting Evolvable Software. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 257–271. Springer, Heidelberg (2004)
13. Prevosto, V., Waldmann, U.: Spass+ t. ESCoR: Empirically Successful Computerized Reasoning 192, 88 (2006)
14. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. *AI Communications* 15(2-3), 91–110 (2002)
15. Schulz, S.: E-a brainiac theorem prover. *AI Communications* 15(2), 111–126 (2002)
16. Sutcliffe, G., Suttner, C.B.: The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning* 21(2), 177–203 (1998)