From Runtime Verification to Evolvable Software

Howard Barringer, David Rydeheard *
EMAIL: {howard.barringer, david.rydeheard}@manchester.ac.uk

Dov Gabbay †
EMAIL: dov.gabbay@kcl.ac.uk

January 26, 2007

Summary

In [4] we developed a revision-based logical modelling approach for evolvable systems, built as hierarchical assemblies of components. A component may encapsulate *horizontal* compositions of interacting sub-components as well as specially paired *vertical* compositions of supervisor and supervisee subcomponents. Our work here extends this logical framework to incorporate programs within each component. We first consider a setting where the programs associated with both supervisor and supervisee components are written in the same guarded command style imperative language. However, as supervisor programs typically follow a monitor, diagnose and revise pattern, we then illustrate how temporal logic rule based supervisor programs, mixing declarative and imperative styles, can be semantically incorporated. Indeed, our modelling framework can fibre as many different programming languages as are necessary for the natural expression of the desired evolvable system behaviour. We use a model of a reactively planned remote roving vehicle as a motivating example.

Background and Motivation

We are interested in developing theories and tools to support the construction and running of safe, robust and controllable systems that have the capability to evolve or adapt their structure and behaviour dynamically according to both internal and external stimuli. We distinguish such evolutionary changes from the normal computational flow steps of a program; in particular, such changes may involve the revision of fixed structural elements, replacement of components and/or programs, or larger scale reconfigurations of systems. Evolution steps may be determined by internal monitoring of a system's behaviour identifying a need for change in structure or computation, or may be triggered by some external influence, e.g. a human user or some other computational agent. We refer to such systems in general terms as *evolvable* systems.

Many computational systems are naturally structured and modelled as evolvable systems. Examples include: business process modelling, which adapt their processes according to internal and external imperatives [1]; supervisory control systems for, say, reactive planning [9, 13]; systems for adaptive querying over changing databases [7]; autonomous software repair [12], data structure repair [6]; hybrid systems [11] that change their computational behavior in response to environmental factors that they may themselves influence. Features of evolvable systems, such as the monitoring of aspects across components, are also found in Aspect-oriented Programming [10] and Monitor-oriented Programming [5]. Clearly, the work in the field of runtime verification, addressing the monitoring of system behaviour against desired properties, or specification, is highly relevant to the design and structuring of such evolvable systems.

Our logical account of evolvable computational systems given in [4] aimed at a more refined understanding of these complex system behaviours. We introduced evolution at a level of abstraction that allows us to describe systems that are constructed as a hierarchical assembly of software and hardware components. Software (and hardware) components are modelled as logical theories built from predicates and axioms. The state of a component is a set of formulae of the theory; the formulae record observations that are valid at that stage of the computation. As components compute, their states change. For normal computational steps, these changes are described as revisions to the set of formulae, in a style familiar in revision-based logic [8]. This is just one particular approach to describing computational behaviour, however, it is an approach with built-in persistence — an important feature for describing evolutionary behaviour.

Components may also be constructed as a pairing of a supervisor and supervisee component in which the supervisor component embodies a process of monitoring and possibly evolving its supervisee

*School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK

†Department of Computer Science, Kings College London, The Strand, London, WC2R 2LS, UK

component. Although the supervisor is a component, it stands in a special relationship to its supervisee. In our logical account, this relationship is that of the supervisor theory being *meta* to the object-level supervisee theory. In other words, the supervisor theory has access to the (entirety of the) logical structure of the theory of the supervisee, thus including its predicates, formulae, state, axioms, revision actions, and its subcomponent theories. This equips the supervisor with sufficient capability to describe evolutionary object-level supervisee changes. Thus, not only can meta-level (supervisor) states record observations of its own state of computation, but they can also record observations about the object-level (supervisee) system. Revision actions at the meta-level update the state of the supervisor and, as a consequence of being meta to the supervisee, may also induce a transformation of the object-level, or supervisee, system. It is in this way that we capture evolutionary change. By introducing tree-structured logical descriptions and associated revision operations, we showed how the framework could be extended to evolvable systems built from hierarchies of evolvable components.

Programs for evolution

In this presentation, we outline how our modelling approach can be extended with the introduction of programs over the actions of the component theories. Without digressing into the debate on whether components should be viewed as active, or passive service providers, we enhance our component model and theory so that each component (and hence all of its subcomponents) comes equipped with its own "main" program, which is executed upon component instance creation. This choice of active componentry is not restrictive as it can easily be used to model passive service-provider component models. Furthermore, component instances of different component schema may use different programming languages. Of course, this is common in practice; for example, shell scripts supervising the execution of particular programs (in different languages), or temporal logic (or history/trace) based languages used for monitoring imperative C or Java programs. Seldom, however, do such combinations come equipped with a logical account of the combined systems.

A structural operational semantics, as well as a trace-based denotational semantics, has been provided for the various ways that component programs may be combined, including, in particular, the supervisor-supervisee combination of evolvable components. This provides not only a foundation for static proof analysis of an evolvable component hierarchy but also a natural setting for dynamic, reasoned and programmed, control of a system's evolution as a generalization of standard runtime verification techniques. In addition, we illustrate a temporal rule-based language, blending concepts from EAGLE [3] and METATEM [2], for supervisory programming.

A roving example

We will motivate our approach to the inclusion of programs in component models using an abstraction of a reactive planning-based remote roving vehicle. Let us here give just an informal idea. At the base level, we model a rover as a simple linear-plan execution engine. The rover's plans, i.e. programs, are sequences of actions such as taking a picture, setting a destination heading, and driving towards the destination. A reactive planner is then modelled as a supervisor for this base engine. The supervisor sets an initial plan, then monitors the execution engine's behaviour. If a planned action fails, e.g. a drive action fails because of some unexpected obstruction, the supervisor must diagnose the problem, replan and reinstall a more appropriate plan for the rover. In this basic application, the supervisory control is modelled using a straightforward guarded command programming language; monitoring is reduced to looking for action failures. Obviously, more complex monitoring, both temporal and spatial, would be required for a more sophisticated, potentially predictive, supervisor. Furthermore, a hierarchy of supervisors may also be necessary. For example, suppose the replan action of a first-level supervisor fails because there is no unobstructed route to the given destination. A higher-level supervisor (re-planner) may well revise the goal to drive to another location or may be able to employ some other technology to remove part of the obstruction.

Whilst supervisory control of planning-based systems is hardly new, this example neatly illustrates how the architecture of such systems and their programs are modelled in a logical framework that provides foundation for static and dynamic reasoning.

References

[1] D. Balasubramaniam, R. Morrison, G.N.C. Kirby, K. Mickan, B.C. Warboys, I. Robertson, B. Snowdon, R.M. Greenwood and W. Seet. A software architecture approach for structuring autonomic systems. In *ICSE 2005 Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005)*, St Louis, MO, USA. ACM Digital Library. 2005.

[2] H. Barringer, M. Fisher, D. Gabbay, G. Gough and R. Owens. METATEM: An introduction. *Formal Aspects of Computing*, 7(5): 533–549, 1995.

[3] H. Barringer, A. Goldberg, K. Havelund and K. Sen. Rule-Based Runtime Verification. Proceedings of the *VMCAI'04, 5th International Conference on Verification, Model Checking and Abstract interpretation*, Venice. Volume 2937, Lecture Notes in Computer Science, Springer-Verlag. 2004.

[4] H. Barringer, D. Gabbay and D. Rydeheard. Logical Modelling of Evolvable Systems. Submitted for publication, 2006. See also http://www.cs.manchester.ac.uk/evolve

[5] F. Chen and G. Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003. http://www.elsevier.nl/locate/entcs/volume89.html

[6] B. Demsky and M. Rinard. Data Structure Repair Using Goal-Directed Reasoning. *Proc. 2005 International Conference on Software Engineering.* St. Louis, Missouri. 2005.

[7] K. Eurviriyanukul, A.A.A. Fernandes and N.W. Paton. A Foundation for the Replacement of Pipelined Physical Join Operators in Adaptive Query Processing. *Current Trends in Database Technology (EDBT Workshops)*, Springer, 589-600. 2006.

[8] R.E. Fikes and N.J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3–4):189–208. 1971.

[9] M.P. Georgeoff and A.L. Lansky. Reactive Reasoning and Planning. *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, WA. 677–682, July 1987.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proc. of the European Conference on Object-Oriented Programming*, 1241, pp 220–242. 1997.

[11] X.D. Koutsoukos, P.J. Antsaklis, M.D. Lemmon and J.A. Stiver. Supervisory Control of Hybrid Systems. *Proc. of the IEEE, Special Issue on Hybrid Systems*, 88(7),1026-1049. 2000.

[12] R. Levinson. Unified Planning and Execution for Autonomous Software Repair. *Workshop of Plan Execution: A Reality Check, ICAPS'05*, 2005.

[13] N. Muscettola and G. Dorais and C. Fry and R. Levinson and C. Plaunt. Idea: Planning at the core of autonomous reactive agents. Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, October 2002.