

From Runtime Verification to Evolvable Systems

Howard Barringer¹, Dov Gabbay², and David Rydeheard¹

¹ School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, UK.

{howard.barringer,david.rydeheard}@manchester.ac.uk

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

dov.gabbay@kcl.ac.uk

Abstract. We consider evolvable computational systems built as hierarchies of evolvable components. In a previous paper, we developed a revision-based logical modelling approach for such systems. This paper extends the logical framework to incorporate programs within each component, describing mechanisms for combining programs from separate components and an operational semantics for programmed evolvable systems. As a motivating example, we consider a simple model of a reactively planned remote roving vehicle.

1 Introduction

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 as evolvable systems. Examples include: supervisory control systems for, say, reactive planning, modelling evolving business processes, systems for adaptive querying, responsive memory management, dynamic network routing, autonomous software repair, data structure repair, and adaptive hybrid systems. Features of evolvable systems are also found in Aspect-Oriented Programming [8], whilst runtime verification [7] addresses issues arising in the execution of evolvable systems.

In [1, 3, 4], 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 [5]. An advantage of this approach to logic is its inherent persistence, which turns out to be an important feature for describing evolutionary behaviour.

We model evolvable components as a pairing of a supervisor and supervisee component where the supervisor monitors and possibly evolves its supervisee. In this logical account, the supervisor theory is described at a *meta-level* 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, 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 describe evolvable systems built from hierarchies of evolvable components.

In this paper, we outline how the logical modelling can be extended to components which contain programs of actions. The choice here of active componentry is not restrictive as it can also be used to model passive service-provider component models. Components within one system may use different programming languages: This is common in practice but seldom do such combinations come equipped with a logical account of the combined systems. We present a structural operational semantics 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.

2 Roving around the world of desserts

As an example of an evolvable system where part of the evolution is that of program change, we consider a simple example of reactive plan construction and execution. We present, in a revision-based logic, a simple model of the control of a semi-autonomous remote vehicle, e.g. a future Mars rover. Whilst supervisory control of planning-based systems is hardly new (see e.g. [6, 9]), this example neatly illustrates how the architecture of such systems and their programs are modelled in a logical framework that provides a foundation for static and dynamic reasoning.

Consider a situation as depicted in Figure 1, consisting of a rover vehicle at position s on a table, a destination d and a collection of obstacles - dessert dishes! The rover has a current plan, as a program, for reaching the destination. As it moves around, it views its environment and may find further obstacles, which mean that the current plan has to be revised. The replanning is a separate supervisory process which interacts with the rover but may be remote from it. An initial program for the rover may be

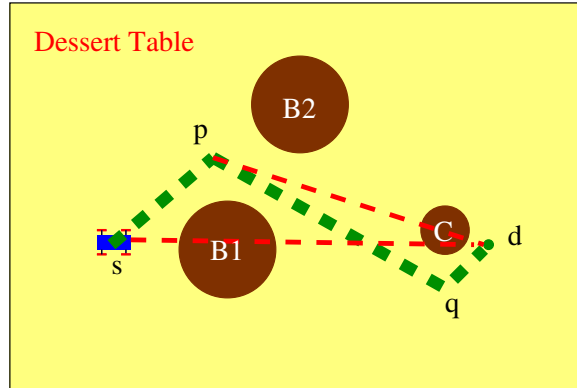


Fig. 1. Navigating the dessert dishes

for the rover may be

$$\text{view}; \text{setHeading}(d); \text{drive}.$$

When executed, this views dessert dishes $B1$ and $B2$, sets the heading for d and then when executing the drive action, aborts because of its knowledge of the obstructing dish $B1$. A replan program may then be

$$\text{setHeading}(p); \text{drive}; \text{view}; \text{setHeading}(d); \text{drive}.$$

The rover executes successfully the first *drive* action, but the second fails because of obstruction C recorded at the second view. A further replan program may then be

$$\text{setHeading}(q); \text{drive}; \text{view}; \text{setHeading}(d); \text{drive}.$$

This time, the rover is able to successfully complete its journey.

We now show how such reactive planning systems can be naturally described using a logical modelling approach for evolvable systems. We model the rover and its observations of the dessert table as an object-level system and the reactive planner as a supervisor at a meta-level to the rover.

2.1 A two-level logical description

We describe a logical theory for the rover using predicates, constraints (axioms), revision actions, a program (for execution) and an initial state. Predicates are of two forms, observation predicates that can occur in states, and those that are abstractions of state properties. Constraints provide a link between these. See [1] for more details.

<i>Rover</i>	
OBSERVATION PREDICATES	
<i>at</i> : <i>Point</i>	
<i>next</i> : <i>Point</i>	
<i>blocked</i> : <i>Points</i>	
ABSTRACTION PREDICATES	
<i>obstruction</i> : <i>Point</i> × <i>Point</i>	
<i>between</i> : <i>Point</i> × <i>Point</i> × <i>Point</i>	
CONSTRAINTS	
<i>geometry</i> $\stackrel{dfn}{=}$	
$\forall p, r : Point \cdot obstruction(p, r) \Leftrightarrow$	
$\quad \exists Q : Points \cdot \exists q : Point \cdot q \in Q \wedge between(p, q, r)$	
$\forall p, q, r : Point \cdot between(p, q, r) \Leftrightarrow \dots$	
ACTIONS	
$\frac{view(h : Point, P : Points)}{pre \{at(h), blocked(Q)\}}$	$\frac{setHeading(t : Point)}{pre \{\}}$
$\frac{add \{blocked(P \cup Q)\}}{del \{blocked(Q)\}}$	$\frac{add \{next(t)\}}{del \{\}}$
$\frac{drive}{pre \{at(s), next(d), \neg obstruction(s, d)\}}$	
$\frac{add \{at(d)\}}{del \{at(s), next(d)\}}$	
PROGRAM	
NULL.	
INITIALLY	
\{\}	

The *Rover* system is specified with three observation predicates: *at*(*p*) is true if the rover is at location *p*; *next*(*p*) is true if the rover has been set to drive to point *p*; *blocked*(*Q*) is true if the rover has observed that the set of points given by *Q* are obstructed. Three actions are specified: *view*(*h*, *P*) is the action of determining the set of known (to the rover) blocked locations *P* from a given location *h*; *setHeading*(*d*) sets the rover up to drive to location *d*; *drive* is the action which moves the rover to its set heading. The *drive* action is defined when there is no obstruction between the rover's current location and its desired destination. Abstraction predicates *obstruction* and *between* are used to characterize this property; they determine whether an obstruction exists given the rover's knowledge of blocked points on the table. If there is no obstruction then the effect of *drive* is merely a revision in the rover's location. The *view*(*h*, *P*) action requires an interaction with the environment, which in this logical framework is expressed as joint action with the environment. The effect of the action is an update of the previous knowledge of blocked points. Such a definition is appropriate for a fixed environment. A dynamically changing environment would require a more sophisticated revision process. The rover's program of actions is initially defined as being the null program, and the initial state is empty.

We now specify a reactive planner as a supervisor for the rover. This is described as a theory which is logically at a meta-level to the above rover theory (see [1] for full details of meta-level descriptions for evolvable components). The planner has access to the predicates, formulae, constraints, actions and program of the rover theory. It also is able to record, using a *holds* predicate, any formulae which hold in the current rover state (which is tracked using a ‘configuration name’ to identify the current instance of the rover using a predicate *current*, and a successor function to indicate the name of the next computational instance of the rover). The *evolve* predicate describes evolutionary requirements on the rover’s configuration. Its arguments are: a tree-state transformer (describing the change of state at an evolution step), any new or replacement components, any new schema added, a replacement program, and the name of the new object-level configuration. The planner also records the current destination *goal*. We have specified three actions: an *observe* action used for monitoring purposes which records, using the *holds* predicate, which instances amongst a set of object-level formulae hold in the current state of the rover; an *initiate* action that evolves the rover to be at a specific location with an initial program (a simple plan to reach the given goal destination); and finally a *rePlan* action that will evolve the rover with an updated plan whenever necessary. The program of the planner first initiates the rover, then monitors the rover until it appears obstructed and unable to complete its given program (plan); for the obstructed situations, it re-routes the rover from its current location. We have expressed the supervisory program using a guarded command language, where the basic commands are the actions of the rover’s planner theory and the guards are (first order) formulae over the observation and abstraction predicates of that theory. The guards may be of a simple form or, in general, may need a theorem-prover to establish their validity to execute programs.

Planner META TO *R* : *Rover*

TYPES

ConfigName

CONSTANTS

$c_R : \text{ConfigName}$

FUNCTIONS

$s : \text{ConfigName} \rightarrow \text{ConfigName}$

$\text{replan} : \text{Point} \times \text{Point} \times \text{Points} \rightarrow \text{PROGRAM}$

$\text{replan}(s, d, Q) \stackrel{\text{dfn}}{=} \dots$

OBSERVATION PREDICATES

$\text{holds} : \text{FORMULAE} \times \text{ConfigName}$

$\text{evolve} : \text{STATETRANSFORMER} \times \text{COMPONENTMAP} \times \text{SCHEMADEFS} \times$

$\text{PROGRAM} \times \text{ConfigName}$

$\text{current} : \text{ConfigName}$

$\text{goal} : \text{Point}$

```

CONSTRAINTS
  PC  $\stackrel{dfn}{=}$ 
     $\forall c_1, c_2 : ConfigName \cdot current(c_1) \wedge current(c_2) \Rightarrow (c_1 = c_2) \wedge$ 
     $\vdots$ 
ACTIONS
   $\frac{observe(P : FORMULAE, c : ConfigName)}{pre \{current(c)\}}$ 
   $\frac{add \{holds(p[\bar{t}/\bar{x}], s(c)), current(s(c)) \mid p(\bar{x}) \in P, STATE \models p[\bar{t}/\bar{x}]\}}{del \{current(c)\}}$ 
   $\frac{initiate(p_s, p_d : Point)}{pre \{current(c)\}}$ 
   $\frac{add \{current(s(c)), goal(p_d), holds(at(p_s), s(c)), holds(blocked(\{\}), s(c)),$ 
     $evolve(update(R, \lambda S.S \cup \{at(p_s), blocked(\{\})\}), [], [],$ 
     $“\llbracket P \subseteq_2 Point \text{ true} \rightarrow view(p_s, P) \rrbracket; setHeading(p_d); drive.”, s(c)\}}{del \{current(c)\}}$ 
   $\frac{rePlan(p_s, p_d : Point)}{pre \{current(c), holds(blocked(Q), c), holds(next(t), c)\}}$ 
   $\frac{add \{current(s(c)), holds(at(p_s), s(c)),$ 
     $evolve(update(R, \lambda S.S \setminus \{next(t)\}), [], [], replan(p_s, p_d, Q), s(c)\}}{del \{current(c)\}}$ 
PROGRAM
  main(p_I, p_D : Point)  $\stackrel{dfn}{=}$ 
    initiate(p_I, p_D);
    [ holds(at(s), c)  $\wedge$  holds(next(t), c)  $\wedge$  holds(obstruction(s, t), c)  $\wedge$ 
      goal(d)  $\wedge$  current(c)  $\rightarrow$  rePlan(s, d)
       $\parallel$  (current(c)  $\wedge$   $\neg \exists d : Point \cdot holds(at(d), c) \wedge goal(d)$ 
         $\rightarrow$  [  $\neg \exists t : Point \cdot holds(next(t), c)$ 
           $\rightarrow$  [holds(at(s), c)  $\rightarrow$  observe({at(p), next(q), blocked(Q)}, c)]
           $\parallel$  holds(next(t), c)  $\wedge$  holds(at(s), c)
           $\rightarrow$  observe({at(p), next(q), blocked(Q), obstruction(s, t)}, c)
        ]
      ]
    ]*
INITIALLY
  {current(c_R)}

```

The planner acts as a supervisor to the rover, that is, these two components are combined (as a ‘vertical’ combination) to form a new component – the evolvable rover. Valid forms of action of this evolvable rover are determined by the relevant combination of the planner and rover programs which we describe in Section 4.

The *view* action of the rover requires that the evolvable rover system interacts with its environment – the table and its obstacles. In this logical framework, the environment is also modelled as a revision theory, the combination (a ‘horizontal’ combination) of the rover and the environment forms a new component, one of whose actions is the joint action of a *view* from the rover and a corre-

sponding *view* from the environment which supplies the obstructed points from a particular viewpoint.

In the scenario described in Figure 1, the rover eventually succeeds in reaching its goal. However, this may not always happen, for example the goal may be unreachable, or the replanning may enter a cycle. In such cases, a hierarchy of supervisors may be built, each one supervising the behaviour of its supervisee, which is itself an evolvable component. Because of the uniform structure of supervisors and supervisees, both being specified as revision theories, such hierarchies can be expressed naturally in this logical framework.

3 A logical framework

We now give an overview of a revision-based logical framework which provides an interpretation for descriptions of evolvable component systems, such as that of the rover above. A full description of this framework may be found in [1].

3.1 States, configurations and revision actions

States of systems are expressed in terms of sets of formulae which are ground, i.e. no free variables, and atomic, i.e. consisting only of applications of predicates to terms. Such formulae are ‘observations’ of a system’s computational state. For example, the set $\{at(0, 0), next(1, 1), blocked(\{(1, 0)\})\}$ is a possible state of the rover.

Computations are described in terms of actions which ‘revise’ states. For states which are sets of formulae, these revisions take on a particularly simple form, namely the addition of new formulae, possibly with the deletion of some existing formulae. For example, the *drive* action of the rover revises the above state to become the state: $\{at(1, 1), blocked(\{(1, 0)\})\}$. When a state Δ is updated by an action α to become state Δ' , we write $\Delta \xrightarrow{\alpha} \Delta'$.

A *configuration* corresponds to the full logical structure of a component hierarchy. A configuration $\Gamma = \langle \Delta, \Theta, \Sigma, \Pi, \chi \rangle$ consists of a tree-structured state Δ , i.e. a set of ground atomic formulae allocated to each node of the hierarchy, a component instance hierarchy Θ and a schema hierarchy Σ . Access to elements of these hierarchies are provided by well-formed *paths*. Full details of this structure are found in [1]. New to this account are the remaining elements of the configuration, consisting of a program structure Π and a program status χ . The form of these is described in the next section.

The definition of revision by actions may be extended to tree-structured states, using paths to identify the location of a revision.

3.2 Meta-view relations

In the above description of a rover as an object-level component and a planner as its supervisor at a meta-level, the states of the two components must be in accord, in that what is asserted to hold at the meta-level of the object-level system,

must indeed hold. Moreover, the supervisor state may assert the existence of constraints, actions and programs at the object-level, which therefore must exist. Further, when an *evolve* predicate is present in the meta-level state, the required change of object-level structure must occur. These requirements are expressed as ‘meta-view’ relations.

Definition 1 (State meta-view). *Let W^M and W be the typed first-order theories for meta-level and object-level systems respectively. We say that Δ^M (from a configuration Γ^M of W^M) is a state meta-view of a configuration $\Gamma = \langle \Delta, \Theta, \Sigma, \Pi, \chi \rangle$ of theory W if, for any valid non-empty path of basic (i.e. non-evolvable) component identifiers p in Δ^M*

- for all object-level formulae φ and any configuration name c , if $p.\{current(c), holds(\varphi, c)\} \subseteq \downarrow \Delta^M$, then $\downarrow \Delta \models_W \varphi$;
- for all component instance maps θ , if $p.component(\theta) \in \downarrow \Delta^M$, then $\theta \subseteq \Theta$;
- for all schema definition maps σ , if $p.schema(\sigma) \in \downarrow \Delta^M$, then $\sigma \subseteq \Sigma$;
- for all program structures π , if $p.program(\pi) \in \downarrow \Delta^M$, then $\pi = \Pi$.

We also say that Γ^M is a meta-configuration for Γ .

Here, $\downarrow \Delta$ is the flattened form of the tree-structured state Δ . When this relationship is extended to all levels of a component hierarchy in a configuration, we say that the configuration is *state meta-consistent*.

Definition 2 (Transition meta-view). *Given meta-level configurations, $\Gamma^M = \langle \Delta^M, \Theta^M, \Sigma^M, \Pi^M, \chi^M \rangle$ and $\Gamma^{M'} = \langle \Delta^{M'}, \Theta^{M'}, \Sigma^{M'}, \Pi^{M'}, \chi^{M'} \rangle$ of component theory W^M , and, at object-level, $\Gamma = \langle \Delta, \Theta, \Sigma, \Pi, \chi \rangle$ and $\Gamma' = \langle \Delta', \Theta', \Sigma', \Pi', \chi' \rangle$ of component theory W , such that $\Delta^M, \Delta^{M'}$ are, respectively, state meta-views of Γ, Γ' , we say that the pair $\langle \Delta^M, \Delta^{M'} \rangle$ is a transition meta-view of $\langle \Gamma, \Gamma' \rangle$, if whenever for any valid non-empty path of basic (i.e. non-evolvable) component identifiers p in Δ^M ,*

$$p.\{evolve(\delta, \theta, \sigma, \pi, c), current(c)\} \subseteq \downarrow \Delta^{M'}$$

and $\Delta' = \delta(\Delta)$ is consistent in theory W' , where W' is the component theory W with component instance map Θ updated to $\Theta' = \Theta \dagger \theta$, component schema definitions Σ updated to $\Sigma' = \Sigma \dagger \sigma$, and program structure updated Π updated to $\Pi' = \pi(\Pi)$, then $\Gamma' = \langle \Delta', \Theta', \Sigma', \Pi', \text{RUNNING} \rangle$.

Furthermore, we say that the configuration pair $\langle \Gamma^M, \Gamma^{M'} \rangle$ is a transition meta-configuration pair for $\langle \Gamma, \Gamma' \rangle$ and write $tmcp(\Gamma^M, \Gamma^{M'}, \Gamma, \Gamma')$.

4 Including programs in component theories

4.1 Evolvable component structures

We now consider how to incorporate programs into a hierarchy of evolvable components.

There are several issues which need to be addressed when each individual component has a program associated with it:

- In an assembly of components, how do we determine the overall computation from that of the individual programs?
- In cases where programs may terminate normally or abort their computation abnormally, how does this behaviour in a component affect the overall computational behaviour of the system?

To formalise answers to these, we (1) introduce combinators for programs corresponding to way we assemble components, (2) present an operational semantics of these combinators, (3) include explicitly the notion of the ‘status’ of a program in the semantics, so that the effect of the status of individual programs on the overall computation can be expressed.

For evolvable systems, there are two ways that components may be combined. The ‘horizontal’ combination of components allows components to communicate via synchronised joint actions. The corresponding combination of programs is

$$II \text{ WITH } II_1, II_2$$

denoting the main program II of a component instance C with sub-component programs II_1 and II_2 of sub-component instances C_1 and C_2 of C .

The ‘vertical’ combination of components is that of the supervisor/supervisee pairing used to model evolvable components. We write

$$II_M \text{ META TO } II_O$$

for the combination of a supervisor’s program II_M (at a meta-level) with that of the program II_O of its supervisee (at an object-level).

We now introduce a simple programming language based on guarded commands as an example, and consider an operational semantics for its programs and the component combinators above. The language of guarded commands, built from basic actions α , is standard:

$$II ::= \alpha \mid \text{STOP} \mid II_1; II_2 \mid [\llbracket_i g_i \rightarrow II_i \rrbracket \mid II^*.$$

4.2 An operational semantics

We provide an SOS-style [10] transition semantics. The semantics of a program structure II is a labelled relation between configurations which we write as

$$\Gamma \xrightarrow{\alpha} \Gamma',$$

where α is the current action undertaken to transform configuration Γ to Γ' . For a component configuration $\Gamma = \langle \Delta, \Theta, \Sigma, II, \chi \rangle$, we write $\Gamma[II', \chi']$ for the configuration $\langle \Delta, \Theta, \Sigma, II', \chi' \rangle$.

Much of the semantics follows standard guarded-command language semantics [10]. We concentrate here on the combinators corresponding to component assembly. A full description of the semantics is available in [2].

The first rules state that the semantic relation $\xrightarrow{\alpha}$ is indeed an extension of the revision relation, and we introduce both the RUNNING program status, and

the ABORTED program status, the latter expressing the attempt to run actions whose preconditions are not satisfied. Thus, for a program which consists of a single action α with precondition $\text{pre-}\alpha$:

$$\frac{\downarrow \Delta \models \text{pre-}\alpha \quad \Delta \xrightarrow{\alpha} \Delta'}{\langle \Delta, \Theta, \Sigma, \alpha, \text{RUNNING} \rangle \xrightarrow{\alpha} \langle \Delta', \Theta, \Sigma, \text{NULL}, \text{RUNNING} \rangle}$$

$$\frac{\downarrow \Delta \not\models \text{pre-}\alpha}{\langle \Delta, \Theta, \Sigma, \alpha, \text{RUNNING} \rangle \xrightarrow{\alpha} \langle \Delta, \Theta, \Sigma, \alpha, \text{ABORTED} \rangle}$$

We now turn to evolvable components, i.e. the supervisor/supervisee pairing of a supervisor at a meta-level to an object-level supervisee. In this account, the transitions of such a combination are of three forms:

- $\langle \alpha_O, \alpha \rangle$, a meta-level *observation* action α_O executed in synchrony with an object-level component action α ;
- $\langle \alpha_Q, \rangle$, a meta-level *query* action¹ α_E executed in isolation of the object-level component, but leaving the object-level system unchanged;
- $\langle \alpha_E, \rangle$, a meta-level *evolution* action α_Q with no explicit object-level action, but inducing an object-level system change.

The semantics of observation actions is:

$$\frac{\begin{array}{l} \uparrow_{\mathcal{M}} \Gamma[\Pi_M, \text{RUNNING}] \xrightarrow{\alpha_M} \uparrow_{\mathcal{M}} \Gamma'[\Pi'_M, \chi'_M] \\ \uparrow_{\mathcal{O}} \Gamma[\Pi_O, \text{RUNNING}] \xrightarrow{\alpha_O} \uparrow_{\mathcal{O}} \Gamma'[\Pi'_O, \chi'_O], \\ \text{where } \text{tmcp}(\uparrow_{\mathcal{M}} \Gamma, \uparrow_{\mathcal{M}} \Gamma', \uparrow_{\mathcal{O}} \Gamma, \uparrow_{\mathcal{O}} \Gamma') \end{array}}{\Gamma[\Pi_M \text{ META TO } \Pi_O, \text{RUNNING}] \xrightarrow{\langle \alpha_M, \alpha_O \rangle} \Gamma'[\Pi'_M \text{ META TO } \Pi'_O, \chi'_M]}$$

Here, for a configuration Γ of a supervisor/supervisee pairing, $\uparrow_{\mathcal{M}} \Gamma$ is the configuration of the supervisor (at the meta-level) and $\uparrow_{\mathcal{O}} \Gamma$ is the configuration of the supervisee (at the object-level). This rule says: if the supervisor program makes an α_M transition, and the supervisee program makes an α_O transition, then the combination program makes a $\langle \alpha_M, \alpha_O \rangle$ transition, provided that an important condition applies, namely the configurations of the supervisor before and after the transition, and those of the supervisee, are related as a *transition meta-configuration pair* (see Definition 2), i.e. the action of the supervisor tracks that of the supervisee so that the required relationship on the states holds. Notice that the program status of the final system is that of the *supervisor* after its action. This condition corresponds to the supervisor observing the supervisee and having overall control of the computation.

The rule for query actions is similar, except that there is no α_O action. Instead we insist that the configuration of the supervisee remains unchanged with the transition meta-view relation still holding.

The evolution action is a key to the whole account. Here an action is undertaken by the supervisor which *induces a change* in the supervisee, without

¹ The query action is typically used when the supervisee program has terminated and the supervisor needs to query the reason for termination.

an explicit supervisee action. The semantics of this is expressed in the following rule.

$$\frac{\begin{array}{c} \uparrow_{\mathcal{M}} \Gamma[\Pi_M, \text{RUNNING}] \xrightarrow{\alpha_M} \uparrow_{\mathcal{M}} \Gamma'[\Pi'_M, \chi'_M] \\ \Pi'_O = \Pi(\uparrow_{\mathcal{O}} \Gamma'), \text{ where } \text{tmcp}(\uparrow_{\mathcal{M}} \Gamma, \uparrow_{\mathcal{M}} \Gamma', \uparrow_{\mathcal{O}} \Gamma, \uparrow_{\mathcal{O}} \Gamma') \end{array}}{\Gamma[\Pi_M \text{ META TO } \Pi_O, \text{RUNNING}] \xrightarrow{(\alpha_M, \cdot)} \Gamma'[\Pi'_M \text{ META TO } \Pi'_O, \chi'_M]}$$

Again, the crucial condition linking the configuration of the supervisee before and after the supervisor's evolution action is the transition meta-view relation.

We now look briefly at the semantics of the horizontal composition of components, in particular, joint actions which allow communication between components.

For a configuration Γ consisting of a component with configuration $\uparrow_0 \Gamma$ which has two immediate subcomponents with configurations $\uparrow_1 \Gamma$ and $\uparrow_2 \Gamma$ then several actions are possible. The component itself may have actions defined which are independent of those of the subcomponents, with the following semantics.

$$\frac{\begin{array}{c} \uparrow_0 \Gamma[\Pi, \text{RUNNING}] \xrightarrow{\alpha} \uparrow_0 \Gamma'[\Pi', \chi'] \\ \uparrow_1 \Gamma = \uparrow_1 \Gamma' \quad \uparrow_2 \Gamma = \uparrow_2 \Gamma' \end{array}}{\Gamma[\Pi \text{ WITH } \Pi_1, \Pi_2, \text{RUNNING}] \xrightarrow{\alpha} \Gamma'[\Pi' \text{ WITH } \Pi_1, \Pi_2, \chi']}$$

On the other hand, an action of the component may be defined in terms of actions of subcomponents. There are several cases of this. For example, the action of a component may consist of a 'communication' between the subcomponents. In this case, the action α of the component is defined to be the joint action $\alpha_1 || \alpha_2$ of the two subcomponents, with semantics:

$$\frac{\begin{array}{c} \uparrow_0 \Gamma[\Pi, \text{RUNNING}] \xrightarrow{\alpha} \uparrow_0 \Gamma'[\Pi', \chi_0] \\ \uparrow_1 \Gamma[\Pi_1, \text{RUNNING}] \xrightarrow{\alpha_1} \uparrow_1 \Gamma'[\Pi'_1, \chi'_1] \\ \uparrow_2 \Gamma[\Pi_2, \text{RUNNING}] \xrightarrow{\alpha_2} \uparrow_2 \Gamma'[\Pi'_2, \chi'_2] \\ \chi' = (\chi'_1 = \text{ABORTED} ? \chi'_1 : \chi'_2) \end{array}}{\Gamma[\Pi \text{ WITH } \Pi_1, \Pi_2, \text{RUNNING}] \xrightarrow{\alpha} \Gamma'[\Pi' \text{ WITH } \Pi'_1, \Pi'_2, \chi']}$$

As an example of the use of the semantics, consider the rover planner and its *rePlan* action. This is treated as an evolution action by the supervisor/supervisee pairing of the planner and the rover. Thus the evolution action rule above applies. This says that, if the planner's status is *RUNNING*, then the result of the *rePlan* action is a system with program structure $\Pi'_M \text{ META TO } \Pi'_O$, where Π'_M is the new planner program (that remaining after execution of the *rePlan* action) and Π'_O , the new rover program, is that in the resultant object-level configuration which satisfies the transition meta-view relation. This means that it is the program provided by the *evolve*-formula added to the state by the *rePlan* action, i.e. the new plan for the rover.

We have thus demonstrated how programmed evolutionary systems may be described in terms of revision-based logic and a transition-based operational semantics.

5 Conclusions

One starting point for this work lies in the relationship between supervisory control systems and runtime monitoring and verification. To explore this link, we have shown how programs may be incorporated into a logical account of evolvable component systems, using a (rather elegant) transition-based operational semantics to capture the interaction of programs amongst components, in particular for components which have supervisory monitoring and control. We are currently developing a corresponding trace-based denotational semantics.

As a revision-based logic, the framework may be implemented to provide a logical abstract machine whose actions are the revision actions of the logic. The implementation requires automated reasoning tools to establish the validity of action application and of meta-view relations. Such a machine can be used to prototype evolutionary systems, or, when run alongside an evolutionary system implemented in, say, Java, it can provide a mechanism for runtime verification. This work thus provides not only a foundation for static proof analysis but also a natural setting for dynamic, reasoned and programmed, control of a system's evolution as a generalization of standard runtime verification techniques.

References

1. H. Barringer, D. Gabbay, and D. Rydeheard. Logical modelling of evolvable component systems: Part (i) a logical framework. Submitted for publication, See <http://www.cs.manchester.ac.uk/evolve>, 2007.
2. H. Barringer, D. Gabbay, and D. Rydeheard. Logical modelling of evolvable component systems: Part (ii) including programs in components. See <http://www.cs.manchester.ac.uk/evolve>, 2007.
3. H. Barringer, D. Rydeheard, and D. Gabbay. A logical framework for monitoring and evolving software components. In *Proceeding of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Computer Science (TASE 2007)*, Shanghai, China, June 2007. IEEE Computer Society Press.
4. H. Barringer, D. Rydeheard, B. Warboys, and D. Gabbay. A revision-based logical framework for evolvable software. In *Proceeding of IASTED Multi-Conference: Software Engineering (SE07)*, pages 78–83, Innsbruck, Austria, 2007.
5. 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.
6. M.P. Georgeoff and A.L. Lansky. Reactive reasoning and planning. In *Proceedings of the 6th National Conference on AI*, pages 677–682, Seattle, WA., July 1987.
7. K. Havelund, G. Holzmann, I. Lee, and G. Rosu. Runtime verification website: <http://www.runtime-verification.org/>.
8. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, 1997.
9. N. Muscettola, G. Dorais, C. Fry, R. Levinson, and C. Plaunt. Idea: Planning at the core of autonomous reactive agents. In *Proc. 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.
10. G. D. Plotkin. A structural approach to operational semantics. Technical Report, DAIMI FN-19, University of Aarhus, 1981.