From Runtime Verification to Evolvable Systems

Howard Barringer[1], Dov Gabbay[2], and David Rydeheard[1]

[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, where an evolvable component is an encapsulation of a supervisory component and its supervisee. Here, we extend our prior work on a revision-based logical modelling framework for such systems to incorporate programs within each component. We describe mechanisms for combining programs, possibly in different languages, from separate components and outline an operational semantics for programmed evolvable systems. We show how supervisory components extend run-time verifiers/monitors with capabilities for diagnosis and change. We illustrate the logical modelling using an example of an automated bank teller machine.

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. Many computational systems have this capability. 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.

Runtime verification techniques show considerable promise (and some return) for establishing the correctness of systems at runtime by monitoring system behaviour against a behavioural specification. This is particularly useful for systems that are too large for static verification techniques. Typically, in runtime monitoring and verification, when conformance fails, an error is reported and the system halted, possibly with some diagnostic data returned. This is fine for runtime verification applied during system simulation. However, for real-time on-line systems, fault diagnosis and system recovery is required, which in general will mean modification of the running system. When such additional capabilities are in place, the overall dynamically-monitored system becomes an *evolvable* system. The notion of evolvability which we explore here shares some features with Aspect-Oriented Programming [5].

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 evolvable (software and/or hardware) components. We model (and implement) evolvable components as a pairing of a supervisor and its supervisee component, where the supervisor dynamically monitors its supervisee as a runtime verifier, and possibly changes the supervisee so that its behaviour accords with that required by the supervisor. This approach is a generalisation of the software architecture principles that have been developed over a number of years, largely in the context of business process modelling [7]. Figure 1 depicts the pairing of a supervisor component with its supervisee as a new (evolvable) component. Figure 2 depicts a small hierarchical assembly of components. It shows both the *horizontal* composition of communicating components, namely EC1, C2 and EC3 yielding the component C, and the *vertical* composition of supervisors and their associated supervisees, namely E1 and C1 as EC1, E3 and C3 as EC3, and E and C as EC. Thus instead of one overall runtime verifier for a system, verification and evolution is localised to, and embedded within, components in a system hierarchy. This improves the manageability of runtime verification and system evolution

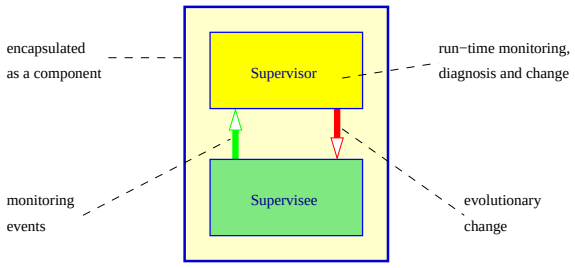

Fig. 1. An Evolvable Component Pairing

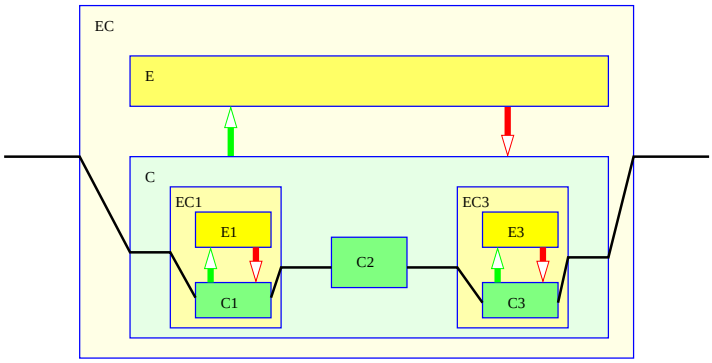

Fig. 2. An Example Hierarchical Assembly

for large systems and also enables us to use evolutionary behaviour as part of system design.

We provide a logical account of these evolvable systems in which the supervisor theory is described as a *meta-level* theory to the object-level supervisee theory. In other words, the supervisor theory has access to the logical structure of the theory of the supervisee, including its predicates, formulas, state, axioms, logical revision actions, and its subcomponent theories. This technically equips the supervisor with sufficient capability both to observe supervisee behaviour and to describe evolutionary object-level supervisee changes. Thus, not only can supervisor states record observations of its own state of computation, but they can also record observations about the object-level supervisee system. 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 supervisee system. This provides a logical account of how systems may evolve their structure during computation.

In this paper, we outline how the logical modelling can be extended to components which contain programs of actions. All components may be active with programs running in concert, but we may also model passive service-provider systems in this approach. Components within one system may use different programming languages: this is common in practice, for example using a separate verification language, 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 vertical 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.

2 Upgrading ATMs

To illustrate our modelling approach to evolvable systems, we visit the world of banking and automated teller machines (ATMs). We focus, in particular, on how runtime monitoring programs for supervisors and basic supervisee component programs are semantically integrated. The Bank of New Island's old form of ATM, although comprising distinct hardware components, such as magnetic strip readers, note counters, keypads, displays, etc., had its local software built in an unstructured, monolithic fashion. Only limited security checks were programmed and certainly not easily changed (indeed the whole ATM network would need to be shutdown for at least a day to perform even minor upgrades). The design of the new system is such that each individual ATM will monitor, adapt and evolve its behaviour, in particular its security checking, to fit best with the bank's and their customers' desires and expectations. The individual software components used in the ATM will themselves also be evolvable and the network of ATMs will naturally support dynamic co-evolution.

The old banking system is modelled as a component assembly comprising a banking centre which holds the records and a number of automated teller machines (ATMs). Figure 3 shows an ATM linked to a central bank component; the ATM component has four communicating subcomponents. In [1], we outlined specifications for the overall structure of the banking system, its ATMs, together with simplified card-reader and note-dispenser components. Only basic actions were specified. In particular, we did not present a formalisation of the programs controlling the actions of the card reader and note dispenser. Here, we introduce programs over the specified actions of a component and incorporate these within the component. Given the differing roles of the supervisor and supervisee, different programming languages for these components may be appropriate. We first consider just the card-reader component and describe the control of its actions via a (very basic) guarded command style program. We then consider a supervisor component for the card reader, whose role is to monitor the patterns of acceptances and rejections of cards and, should the behaviour fall outside acceptable norms, modify, i.e. upgrade, the card reader's security level and associated card checking mechanisms. The temporal nature of the supervisory program can readily be captured via a combination of *declarative* temporal logic runtime monitors with *imperative* guarded command programs for the diagnosis and possible evolutionary change.

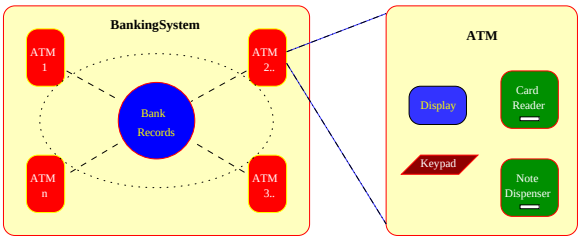

Fig. 3. The banking system component structure

The card-reader component for security level 0, $CardReader_0$, is simplified to holding the account number and PIN for the card currently in the card reader and any cards that have not been returned to the customer. The *cardIn* action, defined when no card is present, makes the account number and PIN of the card a state observation. The action $getUserPin(\_)$ is a shared action with the keyboard component (not specified here) and yields the user supplied PIN value. Validation of the current card is performed by the *checkPin* action; each call increments the number of attempts to verify the card's PIN and then, if the user supplied PIN is the same as the PIN of the current card, *cardAccepted* is added to the current state. The *cardOut* action simply removes the observation from the state. The *swallowCard* action removes the *currentCard* observation and adds the fact that the card is swallowed as well as its rejection.

The card reader's control program loops endlessly. It first reads the account number and PIN from the input card, gets a user supplied PIN and attempts to validate it. If validation succeeds in less than three attempts, the card is returned (we are not concerned with other account actions that may then have followed).

If validation does not succeed within three attempts, the card is swallowed and the reader becomes ready to accept another card.

$CardReader_0$

OBSERVATION PREDICATES

$currentCard : Account \times Pin$
$attempts : 0..3$
$cardAccepted, cardRejected$
$swallowedCard : Account \times Pin$

CONSTRAINTS

$$unique \stackrel{dfn}{=} \forall a_1, a_2 : Account, p_1, p_2 : Pin \cdot ((currentCard(a_1, p_1) \land currentCard(a_2, p_2)) \Rightarrow (a_1 = a_2 \land p_1 = p_2)) \land$$
$$\neg(cardAccepted \land cardRejected) \land$$
$$\forall a : Account, p_1, p_2 : Pin \cdot ((swallowedCard(a, p_1) \land swallowedCard(a, p_2)) \Rightarrow (p_1 = p_2))$$

ACTIONS

$cardIn(acc : Account, pin : Pin)$

pre	$\{\neg\exists a : Account, p : Pin \cdot currentCard(a, p)\}$
add	$\{currentCard(acc, pin), attempts(0)\}$
del	$\{cardAccepted, cardRejected, attempts(n) \mid n \in 1..3\}$

$cardOut()$

pre	$\{currentCard(acc, pin)\}$
add	$\{\}$
del	$\{currentCard(acc, pin)\}$

$getUserPin(userPin : Pin)$

pre	$\{\}$
add	$\{\}$
del	$\{\}$

$checkPin(userPin : Pin)$

pre	$\{attempts(n), n < 3, currentCard(acc, pin)\}$
add	$\{attempts(n+1)\} \cup \{cardAccepted \mid pin = userPin\}$
del	$\{attempts(n)\}$

$swallowCard()$

pre	$\{currentCard(acc, pin), \neg cardAccepted\}$
add	$\{swallowedCard(acc, pin), cardRejected\}$
del	$\{currentCard(acc, pin)\}$

PROGRAM

$$\begin{array}{l}
[\ cardIn(?acc, ?pin); \\
\quad [\neg(cardAccepted \lor cardRejected) \rightarrow \\
\qquad getUserPin(?userPin); \\
\qquad checkPin(userPin); \\
\qquad [\neg cardAccepted \land attempts(3) \rightarrow swallowCard() \\
\qquad \| cardAccepted \rightarrow cardOut() \\
\qquad] \\
\quad]^* \\
]^*
\end{array}$$

The new banking system is to be dynamically upgradable. The card reader is therefore reconstructed as an *evolvable* component by pairing it with a supervisory component and encapsulating the pair as a single component. A specification for the structure and actions of the supervisor component is given below. There are a number of different types of temporal criteria that may be dynamically monitored. For example, the system may monitor the ratio of rejected to accepted cards over a rolling 24-hour period, or on a daily basis, or over a fixed number of night-time hours, etc. The supervisor thus contains a predicate *criterion* pairing a criterion type and value – the latter may represent time-series data in order to compute rolling ratios, etc.

$CardReaderSupervisor$ META TO $cid : CardReader_{level}$

TYPES

$CriterionType \stackrel{dfn}{=} \{rejectsPerHour, usersPerHour, \ldots\}$

$CriterionValue \stackrel{dfn}{=} \ldots$

FUNCTIONS

$updateCriterion : CriterionType \times CriterionValue \times Int \times Time \rightarrow CriterionValue$

OBSERVATION PREDICATES

$clock : Time$

$criterion : CriterionType \times CriterionValue$

$securityUpgrade : Int \times$ stateTRANSFORMER $\times$ componentTRANSFORMER $\times$ schemaTRANSFORMER

$holds :$ FORMULA $\times ConfigName$

$current : ConfigName$

CONSTRAINTS ...

ACTIONS

$observeAccept(X : 2^{CriterionType})$

pre	$\{current(c), clock(t), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$
add	$\{holds(cid.cardAccepted, s(c)), current(s(c)),$ $\bigwedge_{ct \in X} criterion(ct, updateCriterion(ct, cv_{ct}, 1, t))\}$
del	$\{current(c), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$

$observeReject(X : 2^{CriterionType})$

pre	$\{current(c), clock(t), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$
add	$\{holds(cid.cardRejected, s(c)), current(s(c)),$ $\bigwedge_{ct \in X} criterion(ct, updateCriterion(ct, cv_{ct}, 0, t))\}$
del	$\{current(c), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$

$upgradeSecurityChecking()$

pre	$\{current(c), securityUpgrade(level, st, ct, cs),$ $component(thisComp \text{ as } [cid \mapsto \langle CardReader_{level}, \_, \_, \_\rangle])\}$
add	$\{current(s(c)), component(ct(thisComp))$ $evolve(st, ct(thisComp), cs(CardReader_{level}), s(c))\}$
del	$\{current(c), component(thisComp)\}$

Before presenting the supervisor's monitoring program, a few words of explanation on the above actions are necessary. The basic monitoring actions update the *criterion* predicate according for the associated criteria types. The evolutionary action *upgradeSecurityChecking*() specifies the potentially complex operation of updating the card reader's security checking procedures. To keep things simple, we suppose that the card reader supervisor has pre-programmed transformations that it can apply to the card reader. Recall that the card reader component performed very basic checking. A higher level vetting may include, for example, a check with the bank on the card's recent transaction history to determine whether its current use is out of the norm, and then, if so, to proceed through further security checks, e.g. via questions agreed previously with the customer. It may also be possible to invoke other forms of unique customer identification, e.g. finger prints, iris prints, etc., depending upon hardware capability and information stored on chip. The *upgradeSecurityChecking*() action schema abstracts the update via three transformations that are stored in the supervisor's state. The predicate $securityUpgrade(level, st, ct, cs)$ records the fact that *st*, *ct* and *cs* are, respectively, state, component instance and component schema transformers which yield a card reader at security level *level*. These transformers are applied in the appropriate way to the observation state, component instance map and schema map of the object-level configuration for the card reader component by the addition of a suitably instantiated *evolve* predicate in the supervisor's observation state.

From past analyses of card use, the bank finds that it is acceptable for (i) the hourly average of retries on PINs not to exceed one during daytime unless there's been very high usage over the past 24 hour period, and (ii) the hourly average of retries to be no more than 2 during the wee night hours, again unless the usage has been exceptionally low over the past 24 hours. The Bank of New Island governors believe that patterns of behaviour falling outside these norms warrant a higher level of security checking. We can capture this monitoring via a supervisor program in which temporal formulas, for example EAGLE formulas [2], are used to define the acceptable norms. The program construct

MONITOR USING ⟨Actions⟩ WHERE ⟨Bindings⟩
BEHAVIOUR ⟨Formula⟩
[ON SUCCESS ⟨Program⟩]
[ON FAILURE ⟨Program⟩]

describes a runtime monitor that checks conformance of the supervisor's state against the given formula whenever any of the specified actions are executed. As soon as the observed temporal behaviour matches the specified logical formula, the (optional) success continuation program is executed. On the other hand, as soon as the run-time behaviour can be determined not to match the specified behaviour, the (optional) failure continuation program is executed. As an example, we give a simple card reader monitoring program:

PROGRAM
  [MONITOR USING $observeAccept, observeReject$ WHERE
      $a \stackrel{dfn}{=} \iota x$ ST $criterion(rejectsPerHour, x)$
      $u \stackrel{dfn}{=} \iota y$ ST $criterion(usersPerHour, y)$
      $t \stackrel{dfn}{=} \iota z$ ST $clock(z)$
      $daytime(t) \stackrel{dfn}{=} 3 \leq hour(t) \wedge hour(t) < 22$
    BEHAVIOUR
      ALWAYS($(daytime(t) \Rightarrow a \leq 1 \vee$ OVERPASTDAY$(t, u > 20)) \wedge$
              $(\neg daytime(t) \Rightarrow a \leq 2 \vee$ OVERPASTDAY$(t, u < 2)))$
    ON SUCCESS [STATUS(STOP) $\rightarrow$ STOP
               $[\!]$ STATUS(ABORT) $\rightarrow$ $resetCardReader()$]
    ON FAILURE $upgradeSecurityChecking()$
  ]*

The card reader supervisor observes the accepts and rejects of the card reader via *observeAccept*() and *observeReject*() actions. The bindings define the average number of rejects per hour and of users per hour given by the *criterion* predicate, and also the time given by the *clock* predicate. The temporal formula characterises the desired behaviour. Because it is an ALWAYS formula, it can evaluate to true only when the program of the card reader terminates. For normal termination with status STOP, the monitoring program also stops. For abnormal termination with status ABORT, the supervisor resets the card reader (we do not define this here). On the other hand, should the sequence of observations lead to criterion values that do not satisfy the temporal formula, then the failure continuation program, the *upgradeSecurityChecking*() action of the card reader supervisor, is executed to upgrade the card reader to a higher security level. As the monitor construct is embedded within a loop, once the upgrade is complete, monitoring will be resumed.

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 ATM 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 formulas which are ground, i.e. no free variables, and atomic, i.e. consisting only of applications of predicates to terms. Such formulas are 'observations' of a system's computational state. For example, the set $\{currentCard(5435123456789012, 1234), attempts(3)\}$ is a possible state of the card reader described above.

Computations are expressed in terms of actions which 'revise' states. For states which are sets of formulas, these revisions take on a particularly simple

form, namely the addition of new formulas, possibly with the deletion of some existing formulas. For example, the *swallowCard* action of the card reader revises the above state to become the state:

$$\{swallowedCard(5435123456789012, 1234), cardRejected, attempts(3)\}.$$

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 formulas 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 description of an evolvable card reader consisting of an object-level component, $CardReader_0$, and a meta-level component, $CardReaderSupervisor$, 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 $\Delta^{\mathtt{M}}$ (from a configuration $\Gamma^{\mathtt{M}}$ 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 $\Delta^{\mathtt{M}}$

- for all object-level formulas φ and any configuration name c, if $p.\{current(c), holds(\varphi, c)\} \subseteq \downarrow \Delta^{\mathtt{M}}$, then $\downarrow \Delta \models_W \varphi$;
- for all component instance maps θ, if $p.component(\theta) \in \downarrow \Delta^{\mathtt{M}}$, then $\theta \subseteq \Theta$;
- for all schema definition maps σ, if $p.schema(\sigma) \in \downarrow \Delta^{\mathtt{M}}$, then $\sigma \subseteq \Sigma$;
- for all program structures π, if $p.program(\pi) \in \downarrow \Delta^{\mathtt{M}}$, 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, $\Gamma^{\mathtt{M}} = \langle \Delta^{\mathtt{M}}, \Theta^{\mathtt{M}}, \Sigma^{\mathtt{M}}, \Pi^{\mathtt{M}}, \chi^{\mathtt{M}} \rangle$ and $\Gamma^{\mathtt{M}\prime} = \langle \Delta^{\mathtt{M}\prime}, \Theta^{\mathtt{M}\prime}, \Sigma^{\mathtt{M}\prime}, \Pi^{\mathtt{M}\prime}, \chi^{\mathtt{M}\prime} \rangle$ 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 $\Delta^{\mathtt{M}}$, $\Delta^{\mathtt{M}\prime}$ are, respectively, state meta-views of Γ, Γ', we say that the pair $\langle \Delta^{\mathtt{M}}, \Delta^{\mathtt{M}\prime} \rangle$ 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 $\Delta^{\mathtt{M}}$,

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

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 $\Gamma' = \langle \Delta', \Theta', \Sigma', \Pi', \text{RUN} \rangle$.

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

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?
- How are the monitoring, diagnostic and evolutionary aspects of a supervisor expressed in terms of a program?

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, and (4) introduce a specific monitoring language for supervisors.

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

$$\Pi \text{ WITH } \Pi_1, \Pi_2$$

denoting the main program Π of a component instance C with sub-component programs Π_1 and Π_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

$$\Pi_M \text{ META TO } \Pi_O$$

for the combination of a supervisor's program Π_M (at a meta-level) with that of the program Π_O of its supervisee (at an object-level).

To make the semantics specific and to correspond to the example above of automated bank teller machines, we introduce two simple programming languages. The first is a language of guarded commands, built from basic actions α, and standard constructs:

$$\Pi ::= \alpha \mid \text{STOP} \mid \Pi_1; \Pi_2 \mid [\,[\!]_i g_i \rightarrow \Pi_i] \mid \Pi^*$$

The second language is that of supervisory control for meta-level components. We reuse the language of guarded commands, extending it with a monitoring construct:

$$monitor(A, \varphi, \Pi_1, \Pi_2)$$

This is abstract syntax for the monitoring programs that we introduced in the banking example above. The set A is that of supervisor actions at which the monitoring events take place, φ is the monitoring formula (in the above example we use a temporal logic to express monitoring formulas, but other logics may be used instead), Π_1 is the program that runs in the case when the monitoring succeeds i.e. the formula becomes satisfied, and Π_2 is the program that runs when the monitoring fails i.e the formula becomes falsified.

4.2 An operational semantics

We provide an SOS-style [6] transition semantics. The semantics of a program structure Π 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 $\Gamma = \langle \Delta, \Theta, \Sigma, \Pi, \chi \rangle$, we write $\Gamma[\Pi', \chi']$ for the configuration $\langle \Delta, \Theta, \Sigma, \Pi', \chi' \rangle$. Much of the semantics follows standard guarded-command language semantics [6]. We concentrate here on monitoring programs and the combinators corresponding to component assembly.

The first rule states that the semantic relation $\xrightarrow{\alpha}$ is indeed an extension of the revision relation and we introduce the RUN program status. Thus, for a program which consists of a single action α with precondition pre-α[1]:

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

[1] For the case of an action for which the precondition is *not* satisfied, the resulting program status is not RUN but is ABORT, with suitable rules for the ABORT status.

The semantics of monitoring programs of the form $monitor(A, \varphi, \Pi_1, \Pi_2)$ require us to 'unfold' the monitoring formula φ as the computation proceeds. The exact form of this depends upon the logic used to express monitoring formulas, in particular, temporal operators unfold as future obligations become satisfied. Techniques for this are well-known (see e.g [2]). We thus assume a relation of the form $\Gamma, \varphi \xrightarrow{\alpha} \Gamma', \varphi'$ where φ' is the unfolding of φ after the action α in the context of the two configurations Γ and Γ'. The rules for monitoring are:

$$\frac{\alpha \in A, \qquad \Gamma, \varphi \xrightarrow{\alpha} \Gamma', \varphi', \quad \varphi \notin \{\top, \bot\}}{\Gamma[monitor(A, \varphi, \Pi_1, \Pi_2), \text{RUN}] \xrightarrow{\alpha} \Gamma'[monitor(A, \varphi', \Pi_1, \Pi_2), \text{RUN}]}$$

$$\frac{\alpha \in A, \qquad \Gamma, \varphi \xrightarrow{\alpha} \Gamma', \top}{\Gamma[monitor(A, \varphi, \Pi_1, \Pi_2), \text{RUN}] \xrightarrow{\alpha} \Gamma'[\Pi_1, \text{RUN}]}$$

$$\frac{\alpha \in A, \qquad \Gamma, \varphi \xrightarrow{\alpha} \Gamma', \bot}{\Gamma[monitor(A, \varphi, \Pi_1, \Pi_2), \text{RUN}] \xrightarrow{\alpha} \Gamma'[\Pi_2, \text{RUN}]}$$

The first rule is the case when monitoring continues with a revised formula, the second and third rules are the cases when monitoring succeeds and the formula is satisfied, and the case when monitoring fails and the formula is falsified. There are also rules for the case where the object-level system terminates, either normally or with failure.

We now turn to evolvable components, i.e. the supervisor/supervisee pairing of a a meta-level to an object-level system. The actions of such a combination are of three forms:

> $\langle \alpha_{observe}, \alpha \rangle$, a *paired action* consisting of a meta-level observation action $\alpha_{observe}$ executed in synchrony with an object-level component action α;
> $\langle \alpha_{query}, \rangle$, a meta-level *query* action[1] α_{query} executed in isolation of the object-level component, but leaving the object-level system unchanged;
> $\langle \alpha_{evolve}, \rangle$, a meta-level *evolution* action α_{evolve} with no explicit object-level action, but inducing an object-level system change.

The semantics of paired actions is:

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

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 may make a $\langle \alpha_M, \alpha_O \rangle$ transition, provided that

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

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 holds. The program status of the final system is that of the *supervisor* after its action, giving the supervisor overall control of the computation. The rule for query actions is similar, except that there is no α_O action and therefore the configuration of the supervisee remains unchanged.

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

$$\frac{\uparrow_{\mathcal{M}} \Gamma[\Pi_M, \text{RUN}] \xrightarrow{\alpha_M} \uparrow_{\mathcal{M}} \Gamma'[\Pi'_M, \chi'_M] \quad \Pi'_O = \Pi(\uparrow_{\mathcal{O}} \Gamma'), \text{ where } tmcp(\uparrow_{\mathcal{M}} \Gamma, \uparrow_{\mathcal{M}} \Gamma', \uparrow_{\mathcal{O}} \Gamma, \uparrow_{\mathcal{O}} \Gamma')}{\Gamma[\Pi_M \text{ META TO } \Pi_O, \text{RUN}] \xrightarrow{\langle \alpha_M, \rangle} \Gamma'[\Pi'_M \text{ META TO } \Pi'_O, \chi'_M]}$$

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. For a configuration Γ consisting of a component with configuration $\uparrow_0 \Gamma$ which has two immediate subcomponents with configurations $\uparrow_1 \Gamma$ and $\uparrow_2 \Gamma$ several actions are possible. We consider here only one case, the action of a component which consists of a 'communication' between its two 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:

$$\frac{\uparrow_0 \Gamma[\Pi, \text{RUN}] \xrightarrow{\alpha} \uparrow_0 \Gamma'[\Pi', \chi'_0] \qquad \uparrow_1 \Gamma[\Pi_1, \text{RUN}] \xrightarrow{\alpha_1} \uparrow_1 \Gamma'[\Pi'_1, \chi'_1] \quad \uparrow_2 \Gamma[\Pi_2, \text{RUN}] \xrightarrow{\alpha_2} \uparrow_2 \Gamma'[\Pi'_2, \chi'_2] \qquad \chi' = (\chi'_1 = \text{ABORT}? \chi'_1 : \chi'_2)}{\Gamma[\Pi \text{ WITH } \Pi_1, \Pi_2, \text{ RUN}] \xrightarrow{\alpha} \Gamma'[\Pi' \text{ WITH } \Pi'_1, \Pi'_2, \chi']}$$

As an example of the semantics, consider the specification of an evolvable bank card reader in Section 2. The *CardReaderSupervisor* has a monitoring program which, when the monitoring fails because the pattern of activities falls outside its requirements, invokes the *upgradeSecurityChecking* action. To interpret this, the *evolution action* rule applies. This says that, if the supervisor's status is RUN, then the result of the *upgradeSecurityChecking* action is a supervisor/supervisee configuration Γ' whose object-level configuration $\uparrow_{\mathcal{O}} \Gamma'$ is related to the meta-level as a transition meta-configuration pair (Definition 2). This relation says that the object-level card reader is that provided by the *evolve*-formula added to the state by the *upgradeSecurityChecking* action, i.e. a new card reader with an upgraded security vetting system. The rule says that the program for the supervisor is the continuation after the evolution step and the program of the card reader is that supplied with the new card reader.

We have thus demonstrated how programmed monitoring and evolutionary change may be described in terms of a 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 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*. 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 evolvable systems, or, when run alongside an actual evolvable system, 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 system evolution as a generalization of standard runtime verification.

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, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with LTL in Eagle. In *Proceedings of PADTAD'04, Parallel and Distributed Systems: Testing and Debugging*, Santa Fe, New Mexico, USA, 2004.
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. 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.
6. G. D. Plotkin. A structural approach to operational semantics. Technical Report, DAIMI FN-19, University of Aarhus, 1981.
7. B. C. Warboys, R. A. Snowdon, R. M. Greenwood, W. Seet, I. Robertson, R. Morrison, D. Balasubramaniam, G. Kirby, and K. Mickan. An active architecture approach to cots integration. *IEEE Software - Special Issue on Incorporating COTS into the Development Process*, 22(4):20–27, 2005.