A REVISION-BASED LOGICAL FRAMEWORK FOR EVOLVABLE SOFTWARE

Howard Barringer, David Rydeheard and Brian Warboys
School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL, UK
email: {howard.barringer,david.rydeheard,brian.warboys}@manchester.ac.uk

Dov Gabbay
Department of Computer Science
Kings College London
The Strand
London, WC2R 2LS, UK
email: dov.gabbay@kcl.ac.uk

ABSTRACT
We describe a natural revision-based logical modelling for evolvable component systems. In this paper, an evolvable component comprises two parts: a supervising process and its supervisee sub-component. The supervisor's role is to monitor and possibly evolve its supervisee, where evolutionary change may be determined purely internally from observations made by the supervisor of the supervisee's behaviour, or may be a response to external stimuli. We model these systems in a revision-based first-order logical framework in which the logic of the supervisor is a meta-logic to that of its supervisee. This enables evolutionary change of the supervisee to be induced by a supervisor's state revision at the meta-level. To describe the full hierarchical component-based structure requires considerable mathematical detail, so we introduce the basic ideas in a simple (single component) setting using the familiar Blocks World. We then indicate how this account extends to full structural hierarchies of evolvable components with supervisor processes at any level of the hierarchy. We also explain the difficulties that we encountered in devising a mathematical account of evolvable systems.

KEY WORDS
Evolvable Software Components, Run-time Monitoring, Logical Modelling, Revision Theory

1 Introduction

Computational systems can be viewed as evolvable at many levels of abstraction, from the rather low-level computational step evolution of a hardware system state, or of a program's computation state, to whole or partial system change or reconfiguration that may occur in the maintenance or installation of software or hardware updates. The execution of a program is computation state evolution. It is automatic and usually happens rather quickly, on a nanosecond timescale. On the other hand, system updates have largely been a non-automated process requiring explicit user action, although internet-based computing has changed this view, e.g. automated updates of virus detection software or security updates to operating systems. These updates are relatively infrequent, being on a timescale of weeks, months or years. Such timescales of change are studied as 'Software Evolution', see e.g. [15]. There are changes on timescales between these two extremes that can also be considered as evolution, or even adaptation. One example occurs in network routing where routing tables change dynamically as network nodes come and go, communication channels saturate, etc. In fact, there is an increasing focus on developing software systems that feature limited forms of autonomy, adaptation or evolution. This is quite natural given the ever-expanding application of computer-based processes for supporting human endeavour. The evolutionary nature of businesses, business processes and their computational modelling is a good example. It's fair to say, however, that mathematical theories of computation have largely ignored these more abstract levels of system evolution; attention has focussed instead on developing models of computation to support effective reasoning about fixed sequential, parallel and distributed software and hardware.

The issue for us is that as one changes the nature of systems so that adaptation and/or evolution become a dominant feature which is present at significantly higher levels of system organisation, how does one specify and reason formally about such systems? The introduction of evolutionary behaviour allows considerably more freedom in the way that systems may behave and it is not at all obvious that traditional methods of specification, and standard approaches to program logics and semantics, remain adequate. In the article [6], we presented a temporal logic-based approach in which we made a clear separation between *evolutionary* steps and *normal* computation steps. In essence we set up a two-level Kripke structure to model the view depicted in Figure 1. The higher-level Kripke struc-

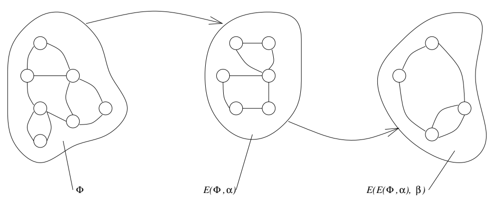

Figure 1. Evolving Worlds

ture's worlds are themselves Kripke structures, which, in turn, capture the "normal" computational behaviour. The higher-level accessibility relation, which in fact relates a state in one Kripke structure to a state in another, corresponds to the evolutionary changes that may occur. The paper argued the importance of abstraction in order to provide "constancy through change" for supporting global reasoning, and defined a two-level temporal logic equipped to reason locally over normal behaviour, and globally over evolutionary behaviour. The two-level temporal logic was based on ideas in the EAGLE temporal logic [5], designed as a highly expressive temporal logic for run-time monitoring purposes. The conceptual link between run-time monitoring and evolution features strongly in our current developments. A very natural architectural model is that the system consists of parts which are supervised at run-time by "higher-level" supervisory modules that monitor the behavior of their supervisees and may invoke evolutionary steps when particular conditions arise. These conditions may reflect an adaptive mode in which changes of environment determine changes in the system, or may be determined by internal imperatives in which the supervisor examines the behaviour of its supervisee and determines appropriate evolutionary actions when the behaviour becomes unacceptable. In fact, Warboys et al. [2, 17, 14, 13] in their modelling of business processes take this view to an extreme in that system architectures are constructed in such a way that every component/process consists of a pair of an 'evolver' (the supervisor) and a 'producer' (the supervisee) and, moreover, the evolvers can create their producers, thus effectively the evolvers determine the behaviour of the overall system. Our current work is heavily influenced by these ideas and also by developments in the EAGLE temporal logic and our previous work on the *executable* temporal logic METATEM [3]. Interestingly, it is not the executable view so much as the way METATEM had meta- and object-level embedded in one logic that is of relevance to the current work.

A satisfactory mathematical account of systems with evolutionary behaviour is not easy to produce — simple modifications of accounts of non-evolutionary systems do not appear to be adequate. Of course, such systems when viewed in their entirety, including the computational behaviour of the supervisory processes, can be treated as closed systems in the usual form and therefore open to standard treatments of semantics and logic. However, this account misses crucial common structure which is present in these systems and makes their analysis quite different from ordinary computational systems. The common structure resides in (1) the way that supervisory processes have the ability to examine the behaviour of the underlying system, (2) the way that evolutionary steps interrupt computation and modify systems and their associated behaviour. It is this common structure which is reflected both in their architecture and in their mathematical analysis. For (1) the logic of supervisory processes needs to have reference to the logic of their supervisees, in the most general form this means that these logics are *meta-logics*. For (2) the way that supervisors are linked to their supervisees and can influence their processing itself needs a logical description. This we provide in terms of relations called *meta-views* and through these relations evolutionary steps in the supervisor induce the required evolutions in their supervisees and thus provide a description of this process at a level of abstraction appropriate for the specification of, and reasoning about, such systems.

In this paper, we present a logical model of evolvable components where normal component behaviour is viewed as an object-level and the supervisor is described as a meta-level theory for the object-level. We represent a component's state as a set of positive atomic formulas, intuitively corresponding to observations being made of the component. We then follow approaches used in belief revision and treat component actions as revisions to the observation state. A supervisor is described as a separate meta-level system which monitors its supervisee by running in lock-step synchrony. This is a logical synchrony of revision steps and does not necessarily mean that implementations are synchronous. Of critical importance is the fact that the supervisor has the ability to change anything that exists in the description and behaviour of its supervisee. Our aim is to provide a tractable logical foundation for modelling evolvable systems that both improves our understanding of these systems and also supports effective reasoning about the behaviour of evolutionary systems.

In this overview paper, we introduce the elements of our analysis in a simple (one component) setting, using the well-known *Blocks World* as an example. We reflect on the various choices we faced in devising this formulation and then briefly outline how the techniques can be applied in the more complex setting of a hierarchical architecture of evolvable components, which is more extensively treated in [7].

2 Modelling Evolvable Systems

2.1 An Evolvable Software Systems Architecture

Figure 2 indicates an outermost modelling view of a component-based software system that we are constructing for supporting safe and controlled software evolution. As introduced above, an evolvable software component is structured as a special pairing of a supervisor and supervisee component. The supervisor dynamically manages its supervisee according to the principles and policy defined by its own program and its own upper supervisory management. The supervisor program, often referred to as the *evolver* has the power to stop the supervisee, modify it in any way that fits with its own policies, and restart it in a suitable state. Such evolutionary action may be determined necessary locally, or be determined by some higher-level supervisor. The highest level supervisor, referred to in the picture as *The Oracle*, will typically be a human supervisor. The supervisee component program, sometimes referred to as the *Producer* program, does the work. Any component may be built using sub-components, some of which may themselves be evolvable component pairings. A supervisor of an evolvable component pair has the capability to evolve any aspect of an evolvable component, including the supervisory part.

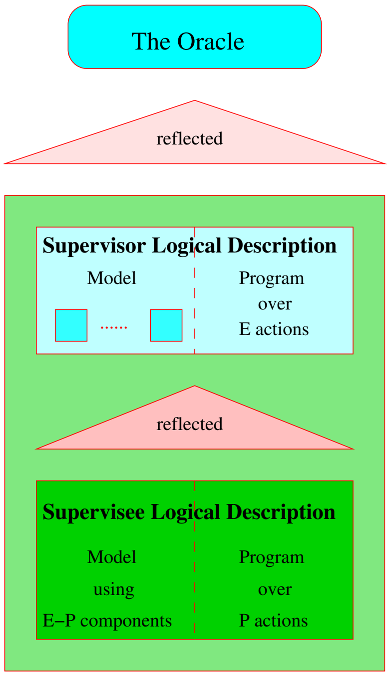

Figure 2. Evolvable Component System Overview

For logical modelling purposes we abstract completely from the implementation mechanisms used to effect evolutionary change by a supervisor program. We describe a component logically via a structural description (a component theory), together with a program over the actions of the component (and its sub-components). The abstract state of the component is given by a set of observations (represented by atomic formulas in the component's logical theory). So far this is fairly standard in formal specification methods for software. We differ, however, by describing actions via a revision process on the observation state. Such an approach is more usual in planning systems, belief revision systems, etc. For an evolvable component pairing, we further assume that the supervisor and supervisee programs are run synchronously with each normal action of the supervisee monitored, or observed, by its supervisor (and its own supervisor, etc.). A model of this as a synchronous parallel composition of the supervisor and supervisee fails to capture the special role and capabilities of the supervisor and a radically different account is required to capture the general mechanism of evolution. The supervisor's logic is a meta-logic describing the logic of the supervisee. This requires not only that the logical language used to describe the supervisor is a meta-language, meta to the language used for the supervisee, but that we define when an observation state of the supervisor is a meta-view of the supervisee, and when a supervisory action is meta to a supervisee's. This means that we are able to describe evolution as a revision action of the supervisor's state and that this action then induces the, often complex, evolutionary change in the supervisee.

2.2 An Evolvable Blocks World

Consider a *Blocks World* system in which there are a number of named blocks, a table, and a robot arm that can be used to move blocks around on the table. We use a many-sorted first order logic to describe particular Blocks World situations, or states, and actions that update/revise a given state. Figure 3 presents part of a schema for a logical theory for such a world.

$BlocksWorld$

TYPES

$$Blocks \stackrel{dfn}{=} \{A, B, C, D, E, F\}$$
$$Tables \stackrel{dfn}{=} \{T\}$$
$$Objects \stackrel{dfn}{=} Blocks \cup Tables$$

OBSERVATION PREDICATES

$on : Blocks \times Objects$

ABSTRACTION PREDICATES

$free : Objects$

$above : Blocks \times Objects$

CONSTRAINTS

$$BWC \stackrel{dfn}{=} \ldots$$
$$TableSize(T, 2) \stackrel{dfn}{=} \ldots$$
$$BlockSize(1) \stackrel{dfn}{=} \ldots$$

ACTIONS

$Move(x : Blocks, y, z : Objects)$	
pre	$\{on(x, z), free(x), free(y), x \neq y\}$
add	$\{on(x, y)\}$
del	$\{on(x, z)\}$

Figure 3. A Blocks World Theory

Such theory schemas not only define individuals, functions, predicates and axioms for describing and determining consistent states of the world, but also actions that can be used to revise states of the world. We divide the predicates according to whether they are used to represent direct observations, e.g. the truth of $on(A, T)$ means that the block named A is placed on the table named T, or whether they denote abstractions of the state, e.g. $free(T)$ is taken as an abstraction over the state and is used to denote that table T is free to accept another block placed upon it. The meaning of such abstraction predicates is given axiomatically via the named constraints. In this example, the parametrically named constraint $TableSize(T, 2)$ defines the capacity of the table T (as 2) and the formula $free(T)$ as below.

$$
\begin{aligned}
&(\exists b_1, b_2 : Blocks \cdot on(b_1,T) \wedge on(b_2,T) \wedge (b_1 \neq b_2)) \\
&\qquad\qquad \Leftrightarrow \neg free(T) \quad \wedge \\
&\forall b_1, b_2, b_3 : Blocks \cdot on(b_1,T) \wedge on(b_2,T) \wedge on(b_3,T) \\
&\qquad\qquad \Rightarrow ((b_1 = b_2) \vee (b_2 = b_3) \vee (b_1 = b_3))
\end{aligned}
$$

The parametrically named constraint $BlockSize(1)$ defines $free$ for block arguments, in particular that only at most one block may be placed upon another, and BWC provides other axiomatic properties for the predicate on. The given Blocks World theory also presents an action schema for moving blocks from one position to another. These actions are defined as revisions on the states, revisions in a particularly simple form: just adding new formulas to the state, or deleting existing formulas. Revision by an action is only defined when the formulas appearing in the action's pre-condition set are deducible. The move action requires that the block being moved is free, i.e. has no block on top of it, and the place to which it is being moved is also free. Action revision is defined as follows.

Definition 2.1 *Let Δ be a set of ground atomic formulas built from observation predicates of theory W. Δ is consistent with respect to W iff it is not the case that $\not\models_W \Delta$.*

Definition 2.2 (Action Revision) *Given a typed first order theory W, let Δ denote a set of ground atomic observation formulas, i.e. a state description, and α a ground action of W. Let* pre-α, add-α *and* del-α *denote the pre-condition, addition and deletion sets for the action α. The revision of Δ by the ground action α, denoted by $\Delta * \alpha$, is defined when $\Delta \models_W \bigwedge$* pre-$\alpha$ *and yields a state description ($\Delta \cup$* add-$\alpha) \setminus$ del-α *that is consistent with respect to the theory W.*

Consider now a supervisory component for the Blocks World system. The supervisor is in a position to observe all aspects of the operation of the Blocks World system. In particular it may notice that the table T becomes full rather too quickly, or too often, impairing the overall performance. In such situations, it may be able to extend the existing table to increase its capacity, or even introduce another table. It may observe that a particular sequence of actions may be optimized by using two singly armed robots instead of just one, and so on. Of course, the supervisor is programmed to make such observations and take action upon them; there are no miracles here. We give a logical description of the supervisory system using similar schema to that used for the Blocks World. However, we need to use observation predicates to reflect what we want the supervisor to observe of the Blocks World. For example, there is a predicate $holds$ that has two arguments, a Blocks World formula and a name to reflect the current Blocks World observation state; we can then note in the supervisor state $holds(free(T), c)$ if the formula $free(T)$ is deducible from the Blocks World observation state that the supervisor names as c. There are also predicates to reflect other elements of the Blocks World theory, such as whether a named constraint is active, whether a particular action schema is defined, etc. In fact, we define the notion of configuration to capture both the theory and an observation description. A supervisor then is equipped with predicates over elements of the supervisee's configuration.

Definition 2.3 (Configuration) *In the context of a first order theory W, we define a state-action configuration as $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A} \rangle$ where:*

> *Δ is of type $State = 2^{Ob}$, where Ob denotes the set of ground atomic formula built using observation predicates;*
>
> *$\mathcal{C}$ is a finite consistent collection of parametrically named closed formulas of $\mathcal{L}$ — the W theory axioms;*
>
> *$\mathcal{A}$ is of type $ActionDefs = ActId \rightarrow (Vars \times 2^{Atom} \times 2^{Ob} \times 2^{Ob})$, and is the set of basic revision actions, each with a name in $ActID$, a list of parameters, and the sets of formulas which serve as pre-conditions, addition formulas and deletion formulas ($Atom$ is the set of atomic formulas of theory W).*

A state-action configuration $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A} \rangle$ is said to be consistent if and only if the state observation Δ is consistent with respect to the first order theory W.

Definition 2.4 *A pair of consistent state-action configurations*

$$\mathbf{C} = (\Delta, \mathcal{C}, \mathcal{A}) \text{ and}$$
$$\mathbf{C}' = (\Delta', \mathcal{C}, \mathcal{A})$$

are related by an action $\alpha \in GroundAction$ iff $\Delta' = (\Delta * \alpha)$. We write $\mathbf{C} \xrightarrow{\alpha} \mathbf{C}'$.

Figure 4 depicts the relationship between a sequence of actions, and associated configurations, of the Blocks World system and a trace of actions of a supervisor, described by the theory outlined in Figure 5. The supervisor has come equipped with three actions — a monitoring action $Observe$ and evolutionary actions $Expand$ and $Contract$. The monitoring action is pretty much self-explanatory and simply records particular Blocks World observations. $Expand$, on the other hand, has more subtlety. The action $Expand(2, 4, c)$ is defined if c is the supervisor's name for the current Blocks World configuration, and it has been recorded that $TableSize(T, 2)$ is an active named constraint of the Blocks World. The supervisor's action then records in its own observation state (i) that an evolution of the Blocks World configuration is required using the $evolve$ predicate and (ii) updates its own record of the table size constraint. The subtlety comes in how we now link the supervisor's trace with the supervisee's trace. First we define what it means for a supervisory state to be a reflection of its supervisee state.

Definition 2.5 (State meta-view) *Let W^M and W be first-order theories for the meta (evolver) and object-level systems respectively. We say that Δ^{M} (from a configuration of W^M) is a* state meta-view *of a configuration $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A} \rangle$ of theory W if*

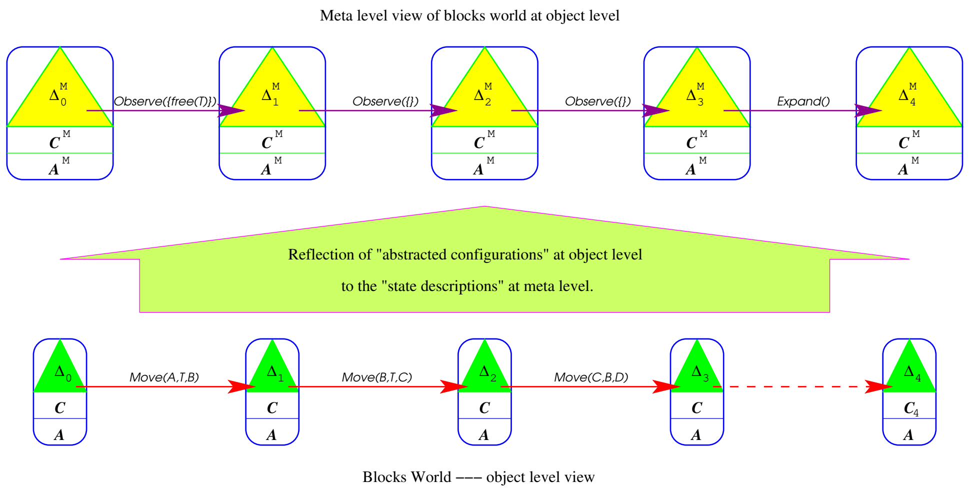

Figure 4. The correspondence between object-level and meta-level actions.

1. *for all object-level formulas* φ *and any* c*, if we have that* $\{current(c), holds(\varphi, c)\} \subseteq \Delta^{\mathrm{M}}$ *then we have* $\Delta \models_W \varphi$*;*

2. *for all constraint names* CN*, if we have that* $constraint(CN) \in \Delta^{\mathrm{M}}$ *then we have* $CN \in \mathcal{C}$*.*

Note that if we observe at the supervisory level, i.e. record in the supervisor's state, that some formula holds at the supervisee's level, then we are not insisting that it be an actual observation at the supervisee level, just that one can deduce it at that level.

We use an $evolve$ predicate to "induce" evolutionary action at the supervisee level. In order for this revision at the meta-level to change the state at the object-level, we define the linkage between the two levels through a *transition meta-view*:

Definition 2.6 (Transition Meta-View) *Given meta-level state descriptions,* Δ^{M} *and* $\Delta^{\mathrm{M}\prime}$ *of theory* W^M*, and object-level configurations,* $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A} \rangle$ *and* $\mathbf{C}' = \langle \Delta', \mathcal{C}', \mathcal{A}' \rangle$ *of theory* W*, such that* Δ^{M}*,* $\Delta^{\mathrm{M}\prime}$ *are state meta-views of* $\mathbf{C}$*,* $\mathbf{C}'$*, we say that the pair* $\langle \Delta^{\mathrm{M}}, \Delta^{\mathrm{M}\prime} \rangle$ *is a* transition meta-view *of* $\langle \mathbf{C}, \mathbf{C}' \rangle$ *if* $evolve(\delta^+_{\mathcal{D}}, \delta^-_{\mathcal{D}}, \delta^+_{\mathcal{C}}, \delta^-_{\mathcal{C}}, \delta^+_{\mathcal{A}}, \delta^-_{\mathcal{A}}, c), current(c) \in \Delta^{\mathrm{M}\prime}$ *and* $\Delta' = \Delta \cup \delta^+_{\mathcal{D}} \backslash \delta^-_{\mathcal{D}}$ *is theory* W' *consistent, where* W' *is the theory* W *with axiom set* $\mathcal{C}$ *updated to* $\mathcal{C}' = (\mathcal{C} \cup \delta^+_{\mathcal{C}} \backslash \delta^-_{\mathcal{C}})$*, then* $\mathbf{C}' = \langle \Delta', \mathcal{C}', \mathcal{A} \cup \delta^+_{\mathcal{A}} \backslash \delta^-_{\mathcal{A}} \rangle$*.*

Now for a supervisor's configuration trace to be a reflection of its supervisee configuration trace, we simply require that the state meta-view relation holds for all paired supervisor and supervisee states, and that the transition meta-view holds across appropriate matched pairs of supervisor and pairs of supervisee states.

When the Blocks World supervisor performs an $Expand$ action, there is no corresponding action in the Blocks World object-level schema. However, the Blocks World configuration will change according to the $evolve$ predicate's arguments, which, in this case, mean that there is no change in the state (i.e. expansion of the table does not alter the blocks on it! This is an example of the use of *persistence* in revision-based logic), the constraint on table capacity is replaced by another, and the set of actions remains unchanged. The conditions under which the $Expand$ and $Contract$ actions are invoked are determined by the supervisor's program and are written in terms of the observations that the supervisor makes of the object-level system. For example, we may wish $Expand$ to be invoked if the table is currently full i.e. $holds(\neg free(T), c)$ and $current(c)$ are in the supervisor's state. This concept of evolution as 'induced action' is one of the crucial aspects of this formulation of evolutionary behaviour. It abstracts from actual mechanisms of supervision and evolutionary interrupts to a level where we may describe the behaviour in a simple logical framework.

3 Review

A successful account of evolutionary behaviour must steer a course between two extremes. On the one hand, we may consider all computational steps of the system, normal steps and evolutionary steps, as equivalent and thus treat the system using standard techniques. However, as we have argued, this account omits crucial common structure present in evolvable systems, common structure which is exploited both in their architecture and in their mathematical analysis. On the other hand, since evolutionary steps may make considerable changes to the object-level system, indeed, in the extreme, may replace the system by an entirely different system, then this unlimited latitude in evolutionary change makes it impossible to give an account of evolution in such a way that we may reason about overall behaviour of evolutionary systems. What we have achieved in the account outlined above is a middle course where evolutionary steps

$BlocksWorldSupervisor$ META TO $BlocksWorld$

TYPES

$ConfigName$

FUNCTIONS

$s : ConfigName \to ConfigName$

OBSERVATION PREDICATES

$current : ConfigName$

$holds :$ FORMULAS $\times\ ConfigName$

$constraint :$ CONSTRNAME

$evolve :$ ATOMS $\times$ ATOMS $\times$ CONSTRNAMES $\times$ CONSTRNAMES $\times$ ACTIONNAMES $\times$ ACTIONNAMES $\times$ $ConfigName$

CONSTRAINTS

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

ACTIONS

$Observe(P :$ FORMULAS$, c : ConfigName)$

pre	$\{current(c)\}$
add	$\{holds(p, s(c)) \mid p \in P\} \cup \{current(s(c))\}$
del	$\{current(c)\}$

$Expand(m : Int, n : Int, c : ConfigName)$

pre	$\{current(c),$ $constraint(TableSize(T, m))\}$
add	$\{current(s(c)), holds(free(T), s(c)),$ $evolve(\{\}, \{\}, \{TableSize(T, n)\}, \{TableSize(T, m)\}, \{\}, \{\}, s(c)),$ $constraint(TableSize(T, n))\}$
del	$\{current(c),$ $constraint(TableSize(T, m))\}$

$Contract(m : Int, c : ConfigName)$

pre	$\{current(c),$ $constraint(TableSize(T, m))\}$
add	$\{current(s(c)),$ $evolve(\{\}, \{\}, \{TableSize(T, m-1)\}, \{TableSize(T, m)\}, \{\}, \{\}, s(c)),$ $constraint(TableSize(T, m-1))\}$
del	$\{current(c),$ $constraint(TableSize(T, m))\}$

Figure 5. A Blocks World Supervisor Theory

are treated as different from those of the object-level system and are indeed powerful enough to invoke a complete replacement of the object-level system, and yet the effect of evolutionary change on the overall behaviour of the system is determined by the relationship between the meta-level and object-level systems. Moreover, in this account supervisors are revision processes in exactly the same form as their supervisees and so we may form hierarchies of supervised evolvable systems, although it should be noted that meta-level descriptions of meta-level systems can be quite unwieldy.

The observation that meta-logics provide an account of the way supervisory processes interact with the logic of the supervisees is far-reaching. The meta-logics of the supervisory processes need to record all the structure of these processes as well as that *of the logic* of the supervisee — its predicates, formulas, consequence relation etc, and moreover this logic is changing as the system evolves! Of course, this structural feature is generic both in recording the structure of the object-level logic, and also in the genericity of the supervisory actions of observation and evolution.

The revision-based approach actually provides an abstract machine, running revision actions on sets of formulas — a machine that may be used to prototype evolvable systems. Notice that the sets of formulas that occur as states of the system are used in two ways — through membership, testing whether a formula occurs in the set, and through deductive consequence, testing whether a formula is deducible in a theory from those in the set. The role of these two is carefully balanced. Different choices lead to somewhat different accounts of the logical framework.

4 Issues in Evolvable Component Modelling

So far, the Blocks World system we have described is monolithic — it consists of a single component. More generally, we wish to assemble evolvable systems from evolvable components in a hierarchical fashion allowing supervisors at each level of the hierarchy able to monitor and evolve not just the components below but also the behaviour of their supervisors.

We have extended the above account to introduce component-based system assembly for evolvable systems (see [7]). That this is possible depends, in part, on the fact that we have described both supervisor and supervisee systems in exactly the same form — as revision processes on sets of formulas.

We outline here changes needed to the above account in order to incorporate component-based system assembly.

The first change is simply naming: state and configurations associated with named components are themselves named by the component-name. The hierarchy of components is then captured as a tree structure and a system is described as a tree of its component states and configurations. All this is standard for component-based systems.

However, several new features are required to describe evolvable component-based systems:

- Revision is no longer the simple process of augmenting and/or restricting sets of formulas. Consider, for example, an evolutionary step which consists of the replacement of a component (which may have subcomponents). In this case, not only will states and configurations in the tree change, but the structure of the tree itself will change, and for evolutions involving more general reconfigurations of a system, these tree manipulations can themselves be complex. Standard revision logics consider only simple revisions of the form we have considered above, but it is possible to extend the notion of revision to these hierarchical systems.

- Component-based systems are built from components which are either unsupervised or consist of a pair of a supervisor and a supervisee. In the latter case, what is any higher supervisor observing? The answer is the synchronous pairs of actions of the supervisor and supervisee. These *paired actions* consist either of (1) a normal computational step of the supervisee together with a monitoring step, or (2) an evolutionary step invoked by the supervisor and the corresponding induced evolutionary action in the supervisee.

- Components may communicate *vertically* through supervisory processes, or *horizontally* with each other (including the possibility of supervisors communicating with each other) through *joint actions* $\alpha||\beta$ which are defined as revisions on the two component states when α and β are revisions on the individual states.

Introducing components does not alter the basic logical structure of evolution but the interaction of component hierarchies with evolutionary behaviour yields a powerful methodology for the development of adaptive and evolutionary systems.

5 Conclusions

The approach of this paper, based on revision logic and a logical description of the process of monitoring and evolution, appears to provide an adequate formulation of a fairly general notion of evolvable system. Other approaches suggest themselves: we have already begun to explore the role of temporal logics [6]; we have also considered process-theoretic ideas (such as bisimulation) which arise when we view revision systems as transition systems induced by the revision actions; also of relevance may be continuation-based denotational semantics to express the different forms of computational step present in evolvable systems.

Much remains to be done to expand these ideas into a full account, including (1) an implemented form of this evolution ('Evolvable Java'); (2) establishing a formal relationship between this logical account and the mechanisms of monitoring and evolution present in an implementation; and (3) exploiting this logical account to establish properties of evolvable systems, both properties that are preserved under evolution and those which undergo controlled evolutionary change.

References

[1] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M.R. Lowry, C.S. Pasareanu, G. Rosu, K. Sen, W. Visser and R. Washington. Combining tset case generation and runtime verification. *Theor. Comput. Sci.* 336(2–3): 209–234. 2005.

[2] 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.

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

[4] H. Barringer, M. Fisher, D. Gabbay, R. Owens and M. Reynolds. *The Imperative Future: Principles of Executable Temporal Logic*. Research Studies Press. 1996.

[5] 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.

[6] H. Barringer and D. Rydeheard. Modelling Evolvable Systems: A Temporal Logic View. *We Will Show Them! Essays in honour of Dov Gabbay on his 60th Birthday*, Volume 1, Artemov, Barringer, d'Avila Garcez, Lamb, Woods (ed.), 195–228, College Publications, 2005.

[7] 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

[8] C. Caleiro, A. Sernadas and C. Sernadas. Fibring Logics: Past, Present and Future. *We Will Show Them! Essays in honour of Dov Gabbay on his 60th Birthday*, Volume 1, Artemov, Barringer, d'Avila Garcez, Lamb, Woods (ed.), 363–388, College Publications, 2005.

[9] P.Y. Cunin, R.M. Greenwood, L. Francou, I. Robertson and B.C. Warboys. The PIE Methodology - Concept and Application. *EWSPT 2001 Proceedings 8th European Software Process Technology Workshop*, LNCS 2077, 3–26. Springer Verlag. 2001.

[10] 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

[11] D.M. Gabbay. *Fibring Logics*. Oxford University Press. 1999.

[12] D.M. Gabbay, O. Rodrigues, and J. Woods. Belief Contraction, Anti-formulae and Resource Overdraft: Part I Deletion in Resource Bounded Logics. *Logic Journal of the IGPL*, 10(6): 601–652. Oxford Univ. Press. 2002

[13] R.M. Greenwood, I. Robertson, B.C. Warboys, and B.S. Yeomans. An Evolutionary Approach to Process System Development. Proceedings of the *International Process Technology Workshop*, Villard de Lans (Grenoble). 1999.

[14] R.M. Greenwood, B.C. Warboys, R. Harrison and P. Henderson. An Empirical Study of the Evolution of a Software System. Proceedings of the *13th IEEE Conference on Automated Software Engineering*, Honolulu. IEEE Computer Society Press. 1998.

[15] M.M. Lehman and Juan F. Ramil. Software Evolution: Background, Theory, Practice. *Information Processing Letters* 88 (1-2) 33–44. 2003.

[16] A. Sernadas, C. Sernadas and J.F. Costa. Object Specification Logic. *Journal of Logic and Computation*, 5(5): 603–630. 1995.

[17] B. Warboys, B. Snowdon, R.M. Greenwood, W. Seet, I. Robertson, R. Morrison, D. Balasubramaniam, G. Kirby, and K. Mickan. An Active Architecture Approach to COTS Integration. In *IEEE Software - Special issue on Incorporating COTS into the Development Process*, 22(4):20-27, 2005.