A Logical Framework for Monitoring and Evolving Software Components

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

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

Abstract

We present a revision-based logical framework for modelling hierarchical assemblies of evolvable component systems. An evolvable component is a tight coupling of a pair of components, consisting of a supervisor and a supervisee, with the supervisor able to both monitor and evolve its supervisee. An evolvable component pair is itself a component so may have its own supervisor, or may be encapsulated as part of a larger component.

Components are modelled as logical theories containing actions which describe state revisions. Supervisor components are modelled as theories which are logically at a meta-level to their supervisee. Revision actions at the meta-level describe theory changes in the supervisee at the object-level. These correspond to various evolutionary changes in the component. We present this framework and show how it enables us to describe the architecture and logical structure of evolvable systems.

1. Introduction

We consider computational systems that exhibit evolutionary behaviour. In such systems, normal computational steps are distinguished from more radical change, which we call evolutionary. Evolutionary change may involve the revision of fixed structural elements, replacement of programs or components, or larger scale reconfigurations of systems. Evolutionary steps may be invoked through external stimuli or through the internal monitoring of a system's behaviour identifying the need for a change in structure.

Some systems are naturally structured as evolutionary. Consider, for example, supervisory control systems for, say, reactive planning [12], or systems for adaptive querying (evaluating queries over changing databases [9]), or responsive memory management (variable capacity memory allocation), or data structure repair [8], or business process modelling (businesses which adapt their processes according to internal and external imperatives [1]), or hybrid systems [14] which change their computational behaviour in response to environmental factors which they may themselves influence [14]. On longer timescales, computational systems may evolve either through user action or automated updates e.g. the automated revision of virus detection software or security upgrades to operating systems. Such timescales of change are studied as 'Software Evolution', see e.g. [15]. Some features of evolutionary systems, such as the monitoring of aspects across components, are also found in Aspect-oriented Programming [13] and in Monitor-oriented Programming [7]. Runtime verification [6] and runtime reflection address the monitoring of system behaviour against a specification.

In fact, there is an increasing presence of adaptive and evolutionary features in software, leading to systems with increasing degrees of autonomy. In this paper, we propose a framework which allows us to use evolutionary structuring as an integral part of the architecture of software systems, enabling evolution to be incorporated at various levels of a system design.

Evolutionary interventions have a common dynamic structure: An evolutionary step is invoked at an appropriate point in the normal processing, either when certain processes have terminated or immediately through an interruption of the computation. The system is then revised and normal processing resumed in a suitable state in the new computational system. This state is usually a modified form of the state before the evolution, with much of the state persisting through the evolutionary change. This mechanistic account of evolution, whilst useful for implementation purposes, is at a different level of abstraction from system design methodologies.

It is with a view to introducing evolution at a higher level of abstraction in system description that we present a logical account of evolutionary systems. In this account, software components are described as logical theories built

from predicates and axioms (or 'constraints'). The state of a component is a set of formulae of the theory. These 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 [11, 17, 19, 10]. This is one particular approach to describing systems with evolutionary behaviour. It is, however, an approach with built-in persistence — formulae change only when specified by a revision action, all other formulae remain unchanged.

What about evolutionary behaviour in this framework? We consider a process that monitors and may evolve a component. Such a process we call a 'supervisor' (other possible names are a 'monitor', a 'controller', a 'manager', or an 'evolver'), and sometimes we resort to the term 'supervisee' for the component being supervised. Supervisors are themselves components but stand in a special relationship to their supervisee component. In this logical account, this relationship is that of a meta-level theory for the object-level of the supervisee. That is, the theory of a supervisor has access to the logical structure of the theory of the supervisee, including its predicates, formulae, state, axioms, and its revision actions. This enables enough of the structure of the object-level to be available at the meta-level for the requisite evolutionary changes to be expressed. Meta-level states are able to record observations of the supervisor's state of computational but are also able to record observations of the object-level. Therefore, the meta-level and object-level states must be in accord and this we capture through the notion of state 'meta-views'. There has been considerable interest in meta-level computational systems and the associated notion of reflection, especially in AI (see, for example, [16], and also [2]). Revision actions at the meta-level transform the state of the supervisor and may induce an accompanying transformation of the object-level. These induced actions correspond to evolutionary steps at the object-level. In this way we give a logical account of evolutionary monitoring and intervention. Moreover, describing computational steps in terms of logical revisions allows us to build 'logical abstract machines' for the execution of these systems.

We develop a compositional framework for building systems from components. Components may be evolvable, i.e. consist of a supervisor and a supervisee as a closely coupled pair. In a hierarchical assembly of components, evolutionary behaviour can be incorporated at each level of the hierarchy by describing the requisite supervisory component. Indeed, what is considered to be evolutionary behaviour depends upon one's perspective: externally, an evolvable component is simply another component in the system. This approach gives considerable flexibility in the design of such systems, enabling us to distribute evolutionary behaviour amongst components at various levels in the hierarchy. We have explored this approach to system design with a range of examples.

In this paper, we will outline the logical framework and use a collection of simple examples, building buffering systems with variable capacities, to illustrate component-based software development in the presence of evolutionary behaviour. See [5] for a fuller account and more substantial examples of the design of evolutionary systems, and [4] for a summary of the basic logical structure of evolution that we here incorporate into component structures.

2. Introducing components

We begin by describing components as theories in a revision-based logic. We then explain how, by combining theories, we can build systems hierarchically from components. This is preparatory to describing evolutionary behaviour in this framework. The presentation is example-driven throughout, using various buffering systems to illustrate how the logical framework allows us to describe evolutionary systems.

2.1. Buffers as components

We begin by describing a family of buffers, i.e. finite storage devices operating on a FIFO principle (First-in, First-out), with operations of 'Send', which removes an element (the first-in) from the buffer, and 'Receive', which adds at element to the buffer.

A component is defined by a theory consisting of a collection of predicates, named constraint formulae and actions. As an example consider the following theory for buffers:

$Buffer(N : Int)$

OBSERVATION PREDICATES

$content : Value\text{-}list$

ABSTRACTION PREDICATES

$free$

CONSTRAINTS

$Uniqueness \stackrel{dfn}{=} \forall l_1, l_2 : Value\text{-}list \cdot$
$content(l_1) \wedge content(l_2) \Rightarrow l_1 = l_2$

$Size(M : Int \text{ INITIALLY } N) \stackrel{dfn}{=}$
$(\exists l : Value\text{-}list \cdot content(l) \wedge (|l| < M)) \Leftrightarrow free \wedge$
$\forall l : Value\text{-}list \cdot content(l) \Rightarrow (|l| \leq M)$

ACTIONS

$Send(v)$	
pre	$\{content(l :: v)\}$
add	$\{content(l)\}$
del	$\{content(l :: v)\}$

$Receive(v)$	
pre	$\{free, content(l)\}$
add	$\{content(v :: l)\}$
del	$\{content(l)\}$

This $Buffer$ specification is a schema parameterized by an integer N which represents the initial capacity of the buffer. The schema presents two actions $Send(v)$ and $Receive(v)$, together with two predicates, $content$ and $free$. The formula $content(l)$ means that the content of the buffer is the list of values l, and $free$ means that the buffer can accept more input. The predicate $content$ is treated as an observation and thus can be present in an observation state of the buffer, whereas $free$ is used as an abstraction of a buffer state. Two constraints are specified: the first characterizing uniqueness of the buffer contents; the second characterizing the freeness and the capacity of the buffer. In this constraint, we set an initial value for the capacity so that each instance of the theory has an initial buffer size.

A state of the buffer is given as a set of ground (i.e. no free variables) atomic (i.e. built as a single predicate applied to arguments) formulae *in the observation predicates*. For this example, there is only one observation predicate, $content$. For buffers of integers, an example state may be:

$$\Delta = \{content([1, 2, 1, 3]\}.$$

The actions on the buffer, $Send$ and $Receive$, are defined as *revisions* of the state. For basic theories, revisions are just the addition and deletion of formulae from the state. The actions may be performed only if the preconditions are satisfied, that is, are valid in the theory specified by the constraints. We are now ready for the first definition:

Definition 2.1 (Action Revision) *Let W be a typed first-order theory, Δ a set of ground atomic formulae, and α a ground action of W. Let* pre-α, add-α *and* del-α *denote the precondition, 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-α and yields a state $(\Delta \cup$ add-$\alpha) \setminus$ del-α that is consistent with respect to the theory W.*

As an example, $Send(3)$ is defined on the state Δ above and results in the new state $\{[1, 2, 1]\}$. Whether the action $Receive(k)$ is defined on a state depends upon the capacity of the buffer, i.e. the value of the parameter N. For example, for theory $Buffer(4)$ and state Δ above, $Receive(k)$ is not defined as the precondition that $free$ holds fails in this case, but would hold for a buffer capacity larger than 4 and state Δ. This illustrates the role of abstraction predicates and constraints in ensuring correct behaviour relative to certain fixed aspects of the system, in this case the buffer capacity. Later, we shall show that this treatment of system specification allows us to modify such fixed aspects and hence to define evolving systems of variable capacity buffering.

Our aim is to use theory schema to create instances of components within other components. As an example, consider a system consisting of two buffers each of capacity 2 linked serially:

$Buffering$
COMPONENTS
$B_1, B_2 : Buffer(2)$
ACTIONS
$Send(v) \stackrel{dfn}{=} B_2.Send(v)$
$Receive(v) \stackrel{dfn}{=} B_1.Receive(v)$
$Internal(v) \stackrel{dfn}{=} B_1.Send(v) \| B_2.Receive(v)$

This theory has two named instances of the theory $Buffer(2)$. The predicates, constraints and actions of these two instances are available in the theory $Buffering$, but each prefixed with the instance name. The actions defined on the state of the $Buffering$ theory are those declared in the specification. Thus, there is a $Send$ action, defined as the $Send$ action of instance B_2 and similarly a $Receive$ action that is that of instance B_1. Finally, there is the internal communication of values between the two buffer instances. We specify this as a *joint* action, an action that will require $B_1.Send(v)$ to be undertaken synchronously with a $B_2.Receive(v)$ action. For simplicity, we will name the new joint action $Internal$. The joint combination of two revision actions is a revision action defined when the two are separately defined, and consists of the addition of the addition formulae of the two actions, and the deletion of the deletion formulae.

States of the $Buffering$ theory are trees, each node being labelled with a component path and having an associated set of formulae recording the observations for the component. In this example, there are no observation predicates at the root. An example of a state of the $Buffering$ theory is the tree with the empty set at the root, a subtree B_1 with set $\{content([1, 3]\}$ and a subtree B_2 with set $\{content([2, 4]\}$. Revision actions act on these trees. For example, $Send(4)$ is defined as the action $B_2.Send(4)$ and thus changes the set of observations at node B_2 to be $\{content([2]\}$. The internal communication transfers the last item of the contents of B_1 to become the first item of the contents of B_2 as expected.

This is the essence of how we orient logical descriptions to systems built from components, using the notion of joint action to allow communication between components. It is a fairly standard treatment in a revision-based logic. We now turn to evolutionary behaviour and meta-level descriptions.

3. Evolvable components

We now consider the structure of evolvable components. We begin with an example, that of an evolvable buffer, one that adjusts its capacity according to usage. We deal with a simple case in which the buffer size may be doubled (other size changes, for example contraction of the buffer, are handled in the same way). To do this a supervisory component

needs to access the size constraint at the object-level and to modify it. Indeed, the supervisor, in general, needs access to all the logical structure of the object-level component — that is it is a meta-level theory for the object-level theory of the component.

One of the key aspects of this logical account is that we do not code evolutionary steps at the object-level (for example, the revision of the size of a buffer) but induce it by a suitable revision action of the supervisor at the meta-level. Both the component and the supervisor are defined as revision theories i.e. collections of predicates, constraints and actions defined as revisions. As actions are undertaken by the component, the supervisor tracks these actions with actions of its own. The computational sequences of the component and of its supervisor must be in accord. One aspect of this is that observations recorded by the supervisor of the component's state must in fact be valid in the component's state. Other formulae present in the supervisor's state may refer to other elements of the structure at the object-level, and may refer to revision changes at the object-level. We capture the relationship between object-level and meta-level computational sequences through 'meta-view' relations, which we define later.

In order for a supervisor to track the activity of its supervisee and record facts about its state as computation proceeds, the supervisor requires a naming scheme for the computational stages of the supervisee. Each supervisor has, in its state, a record of its observations of the object-level system, each observation being tagged with such a name. We adopt here a simple naming scheme in which each supervisor theory has a type of (configuration) names $ConfigName$ whose elements $c_0, c_1, \ldots$ are linearly ordered with an associated successor operation.

The specification $Buffer\_Evolver$ below defines a supervisor theory for the $Buffer$ component described above. An instance of this supervisor which is meta to an instance of a $Buffer(2)$ component is defined in the specification $Evolvable\_Buffer$ (below) to describe an evolvable buffer.

Let us examine these specifications in detail, beginning with the supervisor theory, $Buffer\_Evolver$. This is at a meta-level to the an instance of the buffer theory named cid. The observation predicates are used as follows: The predicate $holds$ enables us to record, in meta-level states, facts valid in object-level states (where FORMULA is the type of object-level formulae). For example, for configuration name c, we may have at the meta-level $holds(content([1,2]), c)$. Notice that the first argument is a *formula*, i.e. is a syntactic item in the object-level, as are other items in the meta-level description. The predicate $component$ enables us to record the component structure of the object-level. The predicate $evolve$ is the key to defining induced evolutionary steps. Its arguments (which we define formally later) are a transformer of the object-level state, the component structure of the object-level, schema giving descriptions of the object-level components, and, finally, a configuration name. The current configuration name is recorded by the predicate $current$. The constraints formally express the fact that the current configuration name is unique, and also that at most one $evolve$-formulae may be present for the current configuration in the meta-level state.

The actions of $Buffer\_Evolver$ are revisions of the meta-level state. There are two: The $Observe$ action allows us to record, at the meta-level, valid facts about the object-level state. It does so using the $holds$ predicate, and updates the current configuration name to its successor.

$Buffer\_Evolver$ META TO $cid : Buffer$

TYPES

$ConfigName$

FUNCTIONS

$s : ConfigName \rightarrow ConfigName$

OBSERVATION PREDICATES

$holds$: FORMULA $\times$ $ConfigName$
$component$: COMPONENTMAP
$evolve$: STATETRANSFORMER $\times$ COMPONENTMAP $\times$ SCHEMADEFS $\times$ $ConfigName$
$current : ConfigName$

CONSTRAINTS

$Uniqueness \stackrel{dfn}{=}$
$\forall c_1, c_2 : ConfigName \cdot$
$current(c_1) \wedge current(c_2) \Rightarrow (c_1 = c_2)$
$\forall \delta, \delta'$: STATETRANSFORMER,
$\delta_M, \delta_M{}'$: COMPONENTMAP,
$\delta_S, \delta_S{}'$: SCHEMADEFS,
$c : ConfigName \cdot$
$(evolve(\delta, \delta_M, \delta_S, c) \wedge$
$evolve(\delta', \delta_M{}', \delta_S{}', c)) \Rightarrow$
$((\delta = \delta') \wedge (\delta_M = \delta_M{}') \wedge (\delta_S = \delta_S{}'))$

ACTIONS

$Observe(Q$: FORMULAE$)$

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

$Expand$

pre	$\{current(c),$ $component([cid \mapsto bc]),$ $bc = (sid, ts, cs, cm),$ $Size(m) \in cs\}$
add	$\{component([cid \mapsto$ $bc[(cs \cup \{Size(2 * m)\} \setminus \{Size(m)\})/cs]]),$ $evolve(\lambda\Delta \cdot \Delta,$ $[cid \mapsto bc[(cs \cup$ $\{Size(2 * m)\} \setminus \{Size(m)\})/cs]], [], s(c)),$ $current(s(c))\}$
del	$\{component([cid \mapsto bc]), current(c)\}$

The $Expand$ action is the increase in size of the buffer. Its preconditions allow access to the object-level compo-

nent structure and ensure that there is a constraint named $Size(m)$ for some m in the supervisee component. We then add to the meta-level state an updated component observation where the size constraint has been doubled, an updated current state, and finally, an $evolve$-formula which says that the state of the buffer (its contents) is unchanged through an $Expand$ action, that the constraint is modified and there are no other changes to the object-level.

$Evolvable\_Buffer$

COMPONENTS

$EB : (E : Buffer\_Evolver$
META TO $B : Buffer(2))$

ACTIONS

$Send(v) \stackrel{dfn}{=} EB.\langle E.Observe(Q), B.Send(v)\rangle$
$Receive(v) \stackrel{dfn}{=} EB.\langle E.Observe(Q), B.Receive(v)\rangle$
$Internal \stackrel{dfn}{=} EB.\langle E.Expand, \rangle$

The new component $Evolvable\_Buffer$ is formed by the infix operator META TO which creates a dependent pair of an object-level system and a system at a meta-level to it. In this example, we create a component with instance name EB from a supervisor instance named E of schema type $Buffer\_Evolver$ and a supervisee component instance named B of schema type $Buffer(2)$. The new component schema $Evolvable\_Buffer$ has actions $Send$, $Receive$ and $Internal$. The $Send$ and $Receive$ actions are the paired actions of an $Observe$ action of the E instance of the supervisor $Buffer\_Evolver$ together with a $Send$ (or $Receive$) action of the B instance of $Buffer(2)$. The set of formulae Q present in the $Observe$ actions may be appropriately chosen. The $Internal$ action is an evolutionary $Expand$ action of the supervisor inducing change in the $Buffer$ instance B.

4. A configuration structure for components

We now turn to the mathematical structures underlying the specifications above and define the logical concepts that express evolutionary behaviour in terms of relations between logical systems. In doing so, we ensure that the specifications above behave as required, in particular that $Evolvable\_Buffer$ really does behave as an evolvable buffer. The structures are fairly complicated involving recursively defined tree hierarchies of component theories, theory schema (similarly recursively defined), revision actions on these tree structures and relations between meta-level and object-level structures.

We begin with the definition of a 'configuration' which contains all the structure of the component specifications and their schema as well as the current states of the components in the system.

Definition 4.1 *A* configuration *for a component system is a triple containing the observation state of the component, the component instance and the component schema definitions.*

$$\begin{aligned} Configuration = \\ & ObservationState \times \\ & ComponentInstance \times \\ & SchemaDefs \end{aligned}$$

Here, $ObservationState$ is a tree-structured collection of states of the components, $ComponentInstance$ defines the component instance hierarchy, and $SchemaDefs$ holds the definitions of component schema. We now give formal definitions of these structures, beginning with $SchemaDefs$. (Here $A \rightharpoonup B$ is the set of partial functions from A to B.)

Definition 4.2

$$SchemaDefs = SchemaID \rightharpoonup ComponentSchema$$

$$\begin{aligned} ComponentSchema = Vars \times \\ & Predicates \times \\ & (ComponentID \rightharpoonup \\ & \quad (SchemaID \times Terms)) \times \\ & ConstraintSchemaDefs \times \\ & ActionSchemaDefs \end{aligned}$$

$$ConstraintSchemaDefs = ConstraintSchema^*$$

$$\begin{aligned} ConstraintSchema = \\ & ConstraintID \times Vars \times Terms \times Formula \end{aligned}$$

$$ActionSchemaDefs = ActionSchema^*$$

$$ActionSchema = ActionID \times Vars \times ActionBody$$

$$\begin{aligned} ActionBody = BasicAction \mid PairedActions \mid \\ & JointActions \end{aligned}$$

$$BasicAction = Pre\text{-}set \times Add\text{-}set \times Del\text{-}set$$

$$Pre\text{-}set = 2^{Formula}$$
$$Add\text{-}set = 2^{AtomicFormula}$$
$$Del\text{-}set = 2^{AtomicFormula}$$

$$ActionName = ComponentIDs \times ActionID$$

$$ComponentIDs = ComponentID^*$$

$$Action = ActionName \times Terms$$

$$PairedActions = E\_Action \mid EC\_Action$$

$$E\_Action = Action$$

$$EC\_Action = Action \times Action$$

$$JointActions = Action^n, n \geq 2$$

A $ComponentSchema$ describes the specifications we have presented so far: It contains the set of formal argument names, observation predicates, the subcomponents (which consist of a mapping from component identifiers to their associated schema identifiers and actual arguments), a list of constraint schema definitions and a list of action definitions. Actions may be defined as atomic actions (in terms of addition and deletion of formulae), as definitions in terms of

action names and as paired actions (see below). Other actions, such 'choice actions' for non-determinism, may be included here. The structure takes into account supervisor/supervisee pairings for evolvable components, such as that of the evolvable buffer. These we explain in more detail below.

The second element of a configuration holds the current component instance hierarchy, together with the constraints associated with each component. For a basic component, i.e. one given without an associated supervisor, the structure records the component identifier, the schema identifier and the actual arguments applied to create the instance, together with the component constraints (derived from the relevant schema definition) and any subcomponent instances. For an instance of an evolvable component, the structure records the component identifier and the component instance for the supervisor and for the supervisee.

Definition 4.3

$$ComponentMap = ComponentID \rightharpoonup ComponentInstance$$

$$ComponentInstance = BasicComponent \mid EvolvableComponent$$

$$BasicComponent = SchemaID \times Terms \times Constraints \times ComponentMap$$

$$EvolvableComponent = ComponentID \times ComponentInstance \times ComponentID \times ComponentInstance$$

$$Constraint = ConstraintID \times Terms$$
$$Constraints = Constraint^*$$

The observation state of a component is a tree. A node of the tree is the state of either a basic component or an evolvable component: for the former, the state is a set of ground atomic formulae (the local state) together with states of each of the subcomponents; for the latter, the state is a pair, the first element being the state of the supervisor, and the second being the state of the supervisee:

Definition 4.4

$$LocalState = 2^{GroundAtom}$$

$$ComponentState = C\_State \mid EC\_State$$
$$C\_State = LocalState \times ObservationState$$
$$EC\_State = ComponentID \times ComponentState \times ComponentID \times ComponentState$$

$$ObservationState = ComponentID \rightharpoonup ComponentState$$

A well-formedness condition on configurations is required:

Definition 4.5 (Well-formedness of configurations) *Let $\Gamma = \langle \Delta, \Pi, \Sigma \rangle$ be a configuration. Γ is a well-formed configuration instance if and only if*

- *Δ and Π have the same component structure, which accords with the schema definitions Σ;*
- *predicate names appearing in local state descriptions within Δ, for any (sub)component of Π are present in the associated schema definitions.*

4.1. Revision actions and meta-views

We are now in a position to define (1) the revisions associated with action definitions, and (2) the formal relationship that is required to hold between the object-level and the meta-level, which we call 'meta-views' and which 'reflect' object-level structure at the meta-level. The structures involved are fairly complex and consequently the definitions given below are quite involved, but then so are the specifications that we build, consisting of communicating components which are structured hierarchically with the possibility of supervisory components at any level of the hierarchy, with evolutionary steps that may reconfigure the associated tree structures.

We begin with a 'flattening' operation on tree-structured states, yielding a set of path-named ground atomic formulae:

$$\downarrow : ObservationState \rightarrow LocalState$$
$$\downarrow\Delta \stackrel{dfn}{=} \bigcup\{cid.(\downarrow\Delta(cid)) \mid cid \in \mathbf{dom}\Delta\}$$
$$\downarrow : ComponentState \rightarrow LocalState$$

For C_States this is defined by

$$\downarrow(\sigma, \mu) \stackrel{dfn}{=} \sigma \cup \downarrow\mu$$

and for EC_States this is defined by

$$\downarrow(cid_1, \sigma_1, cid_2, \sigma_2) \stackrel{dfn}{=} cid_1.(\downarrow\sigma_1) \cup cid_2.(\downarrow\sigma_2)$$

We define the revision operation associated with a basic action, i.e. one defined in term of the addition and deletion of formulae, extending Definition 2.1 to trees.

Definition 4.6 (Local Basic Action Revision) *Let*

$\Gamma = \langle \Delta, \Pi, \Sigma \rangle$ be a well-formed configuration,

$\delta = [cid \mapsto (\sigma, \mu)]$ be an observation state within Δ for a component instance named by the path $p.cid$ of component schema sid,

$\alpha = \langle aid, \bar{t} \rangle$ be an applied basic action of component instance located at $p.cid$ of schema sid, and β be the action body associated with α in the configuration.

The revision of δ by α is defined when there exists a substitution $[\bar{y} \mapsto \bar{u}]$ such that $\downarrow\delta \models_{\Gamma_{p.cid}} \bigwedge$ pre-$\beta[\bar{t}/\bar{x}][\bar{u}/\bar{y}]$ and yields an observation state $[cid \mapsto (\sigma', \mu)]$ where $\sigma' = (\sigma \cup$ add-$\beta[\bar{t}/\bar{x}][\bar{u}/\bar{y}]) \setminus$ del-$\beta[\bar{t}/\bar{x}][\bar{u}/\bar{y}]$.

Definition 4.7 (Basic Action State Revision) *Let $\Gamma = \langle \Delta, \Pi, \Sigma \rangle$ be a well-formed configuration and $\alpha = ((p.cid, aid), \bar{t})$ a well-formed action name denoting a basic action of the component located in Π by $p.cid$. The revision $\Delta * \alpha$ is given by $\Delta' = update(p.cid, \Delta, \delta')$ (i.e. the replacement of the state at $p.cid$ in Δ by the state δ') where δ' is the update of the state in Δ at $p.cid$ by $\langle aid, \bar{t} \rangle$ when Δ' is consistent for Γ. We extend basic action revision to configurations as follows: $\Gamma * \alpha = \langle \Delta * \alpha, \Pi, \Sigma \rangle$.*

We now define two relations, *State meta-view* and *Transition meta-view*, relating meta-level and object-level systems. We begin with the state meta-view. The essence of the following definition is that if φ is asserted to hold at the meta-level in the current configuration, then indeed φ holds in the current object-level state. Moreover, if a constraint is asserted to hold at the meta-level, it really is present in the object-level configuration, likewise for schemas that are asserted at the meta-level.

Definition 4.8 (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 $\Gamma = \langle \Delta, \Pi, \Sigma \rangle$ of theory W if, for any valid non-empty path of component identifiers p in Δ^{M}*

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

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

Components may be hierarchically structured, with supervisor/supervisee pairs themselves as first-class components. We extend the above definition to such situations. We define the notion of *state meta-consistency* for components.

Definition 4.9 (State meta-consistency) *A well-formed component configuration $\Gamma = \langle \Delta, \Pi, \Sigma \rangle$ is said to be* state meta-consistent *if the component map Π refers*

- *to a basic component containing no subcomponents;*
- *to a basic component and the extracted configurations for each subcomponent are state meta-consistent;*
- *to an evolvable component (supervisor/supervisee pair) such that the supervisor configuration is a meta-configuration of the underlying component configuration, and both the supervisor and supervisee configurations are each state meta-consistent.*

The transition meta-view, which we now define, captures the notion of actions induced at the object-level by formulae at the meta-level. The essence of this are the $evolve$-formulae, which describe how an object-level configuration is to be transformed:

Definition 4.10 (Transition meta-view) *Given meta-level configurations, $\Gamma^{\mathsf{M}} = \langle \Delta^{\mathsf{M}}, \Pi^{\mathsf{M}}, \Sigma^{\mathsf{M}} \rangle$ and $\Gamma^{\mathsf{M}\prime} = \langle \Delta^{\mathsf{M}\prime}, \Pi^{\mathsf{M}\prime}, \Sigma^{\mathsf{M}\prime} \rangle$ in component theory W^M, and object-level configurations, $\Gamma = \langle \Delta, \Pi, \Sigma \rangle$ and $\Gamma' = \langle \Delta', \Pi', \Sigma' \rangle$ of component theory W, such that Δ^{M}, $\Delta^{\mathsf{M}\prime}$ are, respectively, state meta-views of Γ, Γ', we say that the pair $\langle \Delta^{\mathsf{M}}, \Delta^{\mathsf{M}\prime} \rangle$ is a* transition meta-view *of $\langle \Gamma, \Gamma' \rangle$*

> *if whenever for any valid non-empty path p in Δ^{M}, $p.\{evolve(\delta, \pi, \sigma, c), current(c)\} \subseteq \downarrow\Delta^{\mathsf{M}\prime}$ and $\Delta' = \delta(\Delta)$ is theory W' consistent, where W' is the component theory W with component instance map Π updated to $\Pi' = \Pi \dagger \pi$ and component schema definitions Σ updated to $\Sigma' = \Sigma \dagger \sigma$,*
>
> *then $\Gamma' = \langle \Delta', \Pi', \Sigma' \rangle$.*

(Here, $\dagger$ is map update, i.e. the overwriting of a value-result pair with another.) We also say that the configuration pair $\langle \Gamma^{\mathsf{M}}, \Gamma^{\mathsf{M}\prime} \rangle$ is a transition meta-configuration pair *for $\langle \Gamma, \Gamma' \rangle$.*

Finally, we define the actions of components which consist of a supervisor/supervisee pairing. The actions are of two forms, either a normal computational action of the supervisee and an associated action of the supervisor, or an evolutionary action of the supervisor, with no corresponding action of the supervisee defined, but nevertheless an induced change to the supervisee configuration:

Definition 4.11 (Local revision - paired actions) *Given a well-formed configuration $\Gamma = \langle \Delta, \Pi, \Sigma \rangle$ for an evolvable component instance ec and an action $\alpha = ((\langle ec \rangle, aid), \bar{t})$ referring to the paired action body $\beta = (\alpha_E, \alpha_C)$, let $\langle \Delta_E, \Pi_E, \Sigma_E \rangle = \uparrow_{\mathcal{E}}(\Gamma)$ and $\langle \Delta_C, \Pi_C, \Sigma_C \rangle = \uparrow_{\mathcal{C}}(\Gamma)$. Let $\Gamma'_E = \Gamma_E * \alpha_E[\bar{t}/\bar{x}]$ and $\Gamma'_C = \Gamma_C * \alpha_C[\bar{t}/\bar{x}]$. Under the assumption that both Γ'_E and Γ'_C are defined and that Γ'_E is a state meta-configuration for Γ'_C, the revision of Γ by α is given by $\mathcal{EC}(\Gamma'_E, \Gamma'_C)$.*

Here $\uparrow_{\mathcal{E}}$ extracts the supervisor configuration from the component configuration, $\uparrow_{\mathcal{C}}$ the supervisee configuration, and $\mathcal{EC}$ combines a supervisor and supervisee configuration into a component configuration. Thus, for a paired action comprising a supervisor action with an action of the supervisee,

a revision of a configuration by the action can be determined by simply deconstructing the configuration, revising by the separate supervisor and supervisee actions and then reconstructing the configuration. More interesting are the evolutionary actions:

Definition 4.12 (Local revision - evolutionary actions) *Given a well-formed configuration* $\Gamma = \langle \Delta, \Pi, \Sigma \rangle$ *for an evolvable component instance* ec *and an action* $\alpha = ((\langle ec \rangle, aid), \bar{t})$ *referring to the evolution action body* $\beta = \alpha_E$*, let* $\langle \Delta_E, \Pi_E, \Sigma_E \rangle = \uparrow_{\mathcal{E}}(\Gamma)$ *and* $\langle \Delta_C, \Pi_C, \Sigma_C \rangle = \uparrow_{\mathcal{C}}(\Gamma)$*. Assuming that* $\Gamma'_E = \uparrow_{\mathcal{E}}(\Gamma) * \alpha_E[\bar{t}/\bar{x}]$ *is defined, the revision of* Γ *by* α *is given by*

$$\mathcal{EC}(\Gamma'_E, \Gamma'_C)$$

for some Γ'_C *such that the pair of supervisor configurations* $\langle \uparrow_{\mathcal{E}}(\Gamma), \Gamma'_E \rangle$ *is a transition meta-configuration pair for the object-level configurations* $\langle \uparrow_{\mathcal{C}}(\Gamma), \Gamma'_C \rangle$.

We extend these definitions from local actions to actions on full configurations in the same way as we did above for basic action revisions.

We now indicate how this framework allows us not only to define evolutionary systems, but also to reason about them. Consider the buffer example. For the evolvable buffer, whilst at any instance it is of bounded capacity, we ought to be able to establish that it behaves (in a sense to be given) as an unbounded buffer. The specification of an unbounded buffer $Unbounded\_Buffer$ is that of $Buffer$, above, modified by removing the parameterisation and the constraint $Size$, and removing the abstraction predicate $free$, including from the precondition of $Receive$. This then means that a buffer with contents is always capable of performing a $Receive$ action, and the size of the contents is always finite but unlimited.

As an example of reasoning about behaviour, with the above definitions we are able to establish the following result: Using the evolutionary action of $Expand$ in $Evolvable\_Buffer$ as an internal action, the $Evolvable\_Buffer$ component is *weakly bisimilar* [18] to the unbounded buffer component $Unbounded\_Buffer$.

For further details of reasoning about evolutionary systems in this logical framework, see [5].

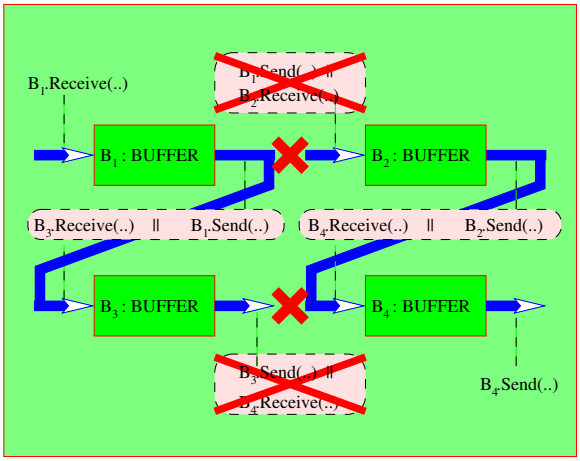
5. System reconfigurations as evolution

As we have indicated, there are many different ways in which a system may evolve. In the world of component hierarchies, we are able to consider evolutions of a system which not only modify existing components, as in the case of buffer expansion in a previous section, but also add new components into the system. More generally, an evolution may be an entire reconfiguration of a system of components. We discuss here how various methods of reconfiguring systems can be incorporated into the logical framework for evolution. In this summary paper, we cannot present the full details of these evolutionary systems (see [5] for a more detailed treatment) but we indicate how various component reconfigurations are treated.

As an example, an alternative way to expand a buffer, and perhaps a more natural one in the component world, is to create a new buffer component and couple it to the existing buffer in such a way that the composite appears like a buffer. In pictorial terms, we want a supervisor to achieve the structural change depicted in Figure 1. Here, a family of buffers $Buffering_i$ is created by a supervisor, which serially adds a buffer of fixed capacity to an already created member of the family to extend its capacity.

The essence of expressing this in the revision-logic framework is the definition of the theory schema for the family of buffers. This definition makes the current buffer a subcomponent of the new buffer schema, and redefines the actions of the buffer ($Send$ and $Receive$) for this new expanded buffer. An $Expand$ action is then defined in the supervisory theory, in much the same way as in the evolutionary example of Section 3, by adding an $evolve$-formula in the state at the meta-level to induce this schema change at the object-level. Notice a subtlety of evolution through reconfiguration: It appears from the depiction that the supervisor theory schema remains unchanged through the evolution. However, the supervisor is now at a meta-level to a different component with a different schema and thus its references to structure at the object-level are now to this updated theory.

More radical reconfigurations are expressible in this framework. For example, consider the reconfiguration in the diagram below.



The starting point is a network of connected buffers, in which the buffer B_1 has its $Send$ action joint with B_2's $Receive$ action, i.e. establishing a connection from

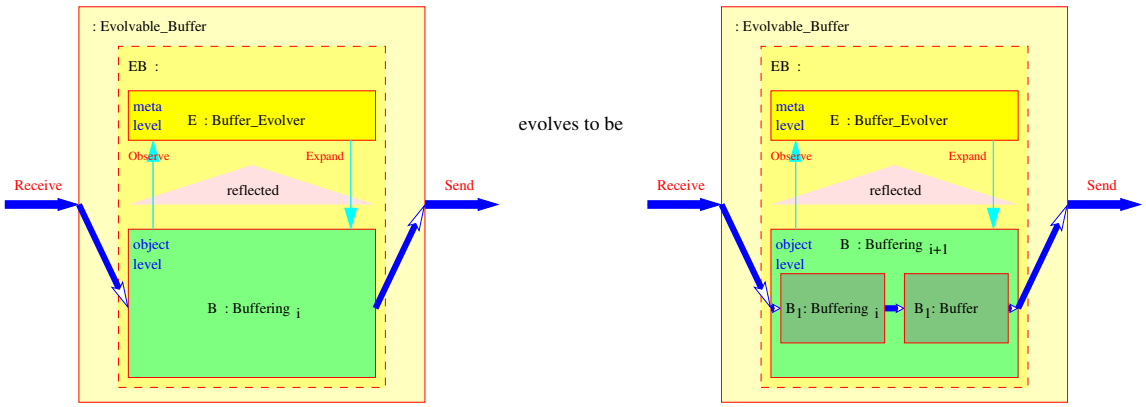


Figure 1. Expanding buffers by adding components.

B_1 to B_2, and similarly for buffers B_3 and B_4. Suppose a supervisor responsible for this network needs to change the network configuration to be that depicted. Thus the connection from B_1 to B_2, i.e. the joint action $B1.Send(v)||B2.Receive(v)$, is to be deleted and replaced with a new connection from B_1 to B_3, i.e. a new joint action $B1.Send(v)||B3.Receive(v)$. Similarly for the other connection. In order to evolve the network as desired, a new component schema is required. Assume that the component schema associated with the first network of buffers is $Network1$. The supervisor's action therefore creates a new schema, $Network2$, built from $Network1$ but with the appropriate change in joint actions, and then ensures the instance name of the first network becomes of schema type $Network2$. The component instance tree and state will remain the same. We have already discussed all of the evolutionary changes needed to specify this supervisor within the revision-logic framework — revision of actions, revision of schema, and renaming of components.

We have indicated how we can model a range of evolutionary actions in this framework, from simple changes of constraints, changes of actions that systems may execute, through to changes of logical systems and complex changes of network connectivity and the reconfiguration of systems with new and existing components.

5.1. Hierarchies of evolvability

We treat a supervisor/supervisee pair as a first-class component. This means that the framework supports hierarchies of supervisor/supervisee pairs. Consider a simple example depicted in Figure 2. It consists of a component built from two evolvable buffers, for example those specified in Section 3, coupled to an encoder. The buffers are used to, say, smooth out variable data rates of connected devices.

The buffers, being evolvable, are able to expand (and, possibly, contract) according to demand. The way the

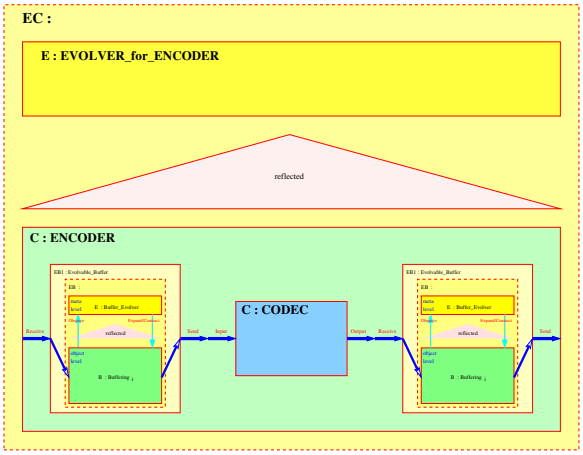


Figure 2. An example of a hierarchy.

buffers expand or contract, however, is fixed and controlled by the evolvable buffer's own internal supervisor. An observer of these evolvable buffers may detect an undesirably high frequency of expansion and contraction and consequently 'tune' the evolvable buffers. In this situation, a supervisor *evolves* a supervisor subcomponent. Such tiered hierarchies of evolvable components with supervisors at various levels are readily definable in this framework, allowing us to model complex hierarchies of evolvability.

6. Conclusions

We have outlined a logical framework for describing evolutionary component-based systems and illustrated the approach with examples based on evolvable buffers. We have developed more substantial examples elsewhere, including an evolving blocks world, and simple models of a bank teller machine system with variable security levels,

and an adaptive router for 'rover' vehicles (see [5]).

Notice the level of abstraction of the system descriptions. These descriptions are not pitched at the level of the overall behaviour of an evolvable system, which, for example, may be formulated in a temporal logic. The descriptions are not at the level of implementations either. In particular, the mechanism by which supervisors monitor and evolve systems is not part of the description. One of the more difficult aspects of developing a successful formulation of evolution has been to get the level of abstraction correct — so that the structuring of a system around evolvable components is present but not at an implementation level.

Clearly there is more to do to develop this approach. Major topics under investigation are (1) linking the logical descriptions of systems to overall system behaviour, (2) introducing programs into this framework so that we can determine sequencing of computational and evolutionary steps [5], (3) implementing the logical abstract machine and integrating with automated reasoning tools, and (4) establishing the relationship between logical descriptions of systems and actual implemented evolvable systems (e.g. in Java).

References

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

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

[3] H. Barringer 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.

[4] H. Barringer, D. Rydeheard, B. Warboys, and D. Gabbay. A Revision-based Logical Framework for Evolvable Software. To appear: *Proc. IASTED International Conference on Software Engineering (SE07)*, Innsbruck. 2007.

[5] H. Barringer, D. Gabbay and D. Rydeheard. Logical Modelling of Evolvable Systems. See http://www.cs.manchester.ac.uk/evolve.

[6] Runtime Verification website. http://www.runtime-verification.org/

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

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

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

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

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

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

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

[16] P. Maes and D. Nardi (Eds) *Meta-Level Architectures and Reflection* North-Holland. 1988.

[17] J. McCarthy and P.J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence 4*, Ed. B. Meltzer and D. Michie, 463–502, Edinburgh University Press. 1969.

[18] R. Milner. *Communication and Concurrency*. Prentice-Hall. 1989.

[19] T. Winograd. *Understanding Natural Language*. Academic Press, New York. 1972.