

A Logical Framework for Monitoring and Evolving Software Components

Howard Barringer¹, David Rydeheard¹, Dov Gabbay²

(1) School of Computer Science
University of Manchester

(2) King's College, London

June 2007

Overview

- Computational systems which exhibit evolutionary behaviour

Overview

- Computational systems which exhibit evolutionary behaviour
- Software architectures based on supervisory/monitoring processes

Overview

- Computational systems which exhibit evolutionary behaviour
- Software architectures based on supervisory/monitoring processes
- Hierarchical component-based systems - evolutionary behaviour at all levels

Overview

- Computational systems which exhibit evolutionary behaviour
- Software architectures based on supervisory/monitoring processes
- Hierarchical component-based systems - evolutionary behaviour at all levels
- Logical framework for modelling, reasoning, simulation and runtime monitoring, using **revision-based logics** and **meta-level descriptions**

Evolvable systems (I)

- We distinguish (1) **normal computational steps**, from (2) **evolutionary steps**

Evolvable systems (I)

- We distinguish (1) **normal computational steps**, from (2) **evolutionary steps**
- Evolutionary steps are usually 'less frequent' and more major changes: Changes of fixed internal structures, changes of programs or components, or major reconfigurations of systems

Evolvable systems (I)

- We distinguish (1) **normal computational steps**, from (2) **evolutionary steps**
- Evolutionary steps are usually 'less frequent' and more major changes: Changes of fixed internal structures, changes of programs or components, or major reconfigurations of systems
- **Controlled evolution**: components within a system control the evolution of other components – monitoring their behaviour and determining evolutionary change, either for internal or external reasons

Evolvable systems (II)

- Many systems are naturally structured as evolutionary:
Reactive planning systems, adaptive querying, responsive memory management, data-structure repair, business process modelling, adaptive hybrid systems, etc.

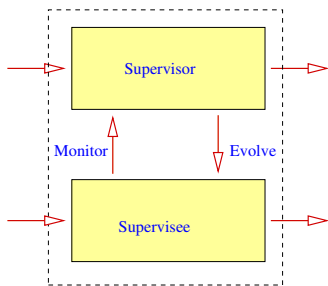
Evolvable systems (II)

- Many systems are naturally structured as evolutionary:
Reactive planning systems, adaptive querying, responsive memory management, data-structure repair, business process modelling, adaptive hybrid systems, etc.
- We introduce evolutionary behaviour as an integral part of the design and architecture of systems

Evolvable systems (II)

- Many systems are naturally structured as evolutionary:
Reactive planning systems, adaptive querying, responsive memory management, data-structure repair, business process modelling, adaptive hybrid systems, etc.
- We introduce evolutionary behaviour as an integral part of the design and architecture of systems
- Features in common with: Aspect-oriented programming, Autonomic programming, Runtime verification

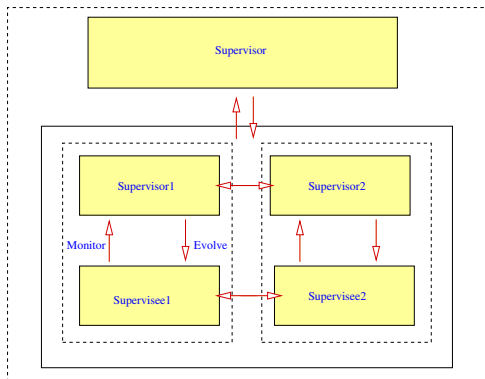
Supervisor-supervisee component modelling



Evolvable component:

The **supervisor** monitors the **supervisee** and may intervene to modify the supervisee's structure or behaviour.

Evolvable component hierarchies



Supervisory processes may occur at any level of a component hierarchy:

- **Vertical composition** via supervisor-supervisee pairing,
- **Horizontal composition** for communication between components via joint actions.

Logical modelling — Basic ideas ...

- We use a revision-based logic:
 - States of a component are sets of ground atomic formulae,
 - Computational steps are revision actions on these sets — adding and deleting formulae.
- A component is a logical theory - consisting of predicates, constraints (axioms) and revision actions.
- For a supervisor-supervisee pairing:
the supervisor theory is a meta-level theory for the object-level theory of the supervisee.

The supervisor theory is also a revision theory and access to the logical elements of the supervisee's theory.

State meta-view

Formally characterise the relationship between

meta-level state observations

and

object-level configuration

through the use of the *holds* predicate:

If $holds(\varphi, c_0)$ exists in meta-level state observation then φ must hold in the observation state of object-level configuration corresponding to c_0 .

Transition meta-view

Formally characterises the relationship between *PAIRS* of

meta level state observations

and

object level configurations

through the use of an *evolve* predicate.

The arguments of *evolve* at the meta-level are the transformations required at the object-level. This is the transition meta-view relation.

Thus revisions at the meta-level *induce* changes at the object-level:
A logical account of evolutionary intervention.

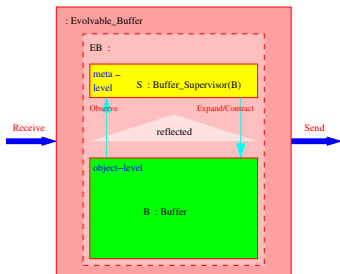
Logical modelling (continued)

Systems built hierarchically from components are modelled as a **tree of states and theories**.

- Evolutionary steps induce change in individual states or theories and/or in the theory tree itself – system reconfiguration.

A **configuration** consists of a tree-structured state, a tree of theory instances, and a collection of theory schema.

Example: An evolvable buffer



We model finite capacity FIFO buffers with `Send` and `Receive` actions.

The supervisor may alter the capacity of the buffer with `Expand` or `Contract` actions at the meta-level, as it monitors the buffer's usage.

A buffer theory

Buffer($N : \text{Int}$)

OBSERVATION PREDICATES

content : *Value-list*

ABSTRACTION PREDICATES

free

CONSTRAINTS

Uniqueness $\stackrel{\text{dfn}}{=} \forall l_1, l_2 : \text{Value-list}.$

$\text{content}(l_1) \wedge \text{content}(l_2) \Rightarrow l_1 = l_2$

Size($M : \text{Int}$ INITIALLY N) $\stackrel{\text{dfn}}{=} \text{INITIALLY } N$

$(\exists l : \text{Value-list} \cdot \text{content}(l) \wedge (|l| < M)) \Leftrightarrow \text{free} \wedge$

$\forall l : \text{Value-list} \cdot \text{content}(l) \Rightarrow (|l| \leq M)$

ACTIONS

<i>Send</i> (v)	
pre	$\{\text{content}(l :: v)\}$
add	$\{\text{content}(l)\}$
del	$\{\text{content}(l :: v)\}$

<i>Receive</i> (v)	
pre	$\{\text{free}, \text{content}(l)\}$
add	$\{\text{content}(v :: l)\}$
del	$\{\text{content}(l)\}$

A buffer supervisor

Buffer_Supervisor META TO *cid* : *Buffer*

TYPES

ConfigName

FUNCTIONS

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

OBSERVATION PREDICATES

$\textit{holds} : \text{FORMULA} \times \textit{ConfigName}$

$\textit{component} : \text{COMPONENTMAP}$

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

$\text{SCHEMADEFS} \times \textit{ConfigName}$

$\textit{current} : \textit{ConfigName}$

CONSTRAINTS

$\textit{Uniqueness} \stackrel{\textit{dfn}}{=}$

$\forall c_1, c_2 : \textit{ConfigName} \cdot \textit{current}(c_1) \wedge \textit{current}(c_2) \Rightarrow (c_1 = c_2)$

Continued...

ACTIONS

Observe($Q : \text{FORMULAE}$)

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

Expand

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

An evolvable buffer

Evolvable_Buffer

COMPONENTS

$EB : (E : Buffer_Supervisor \text{ META TO } B : Buffer(2))$

ACTIONS

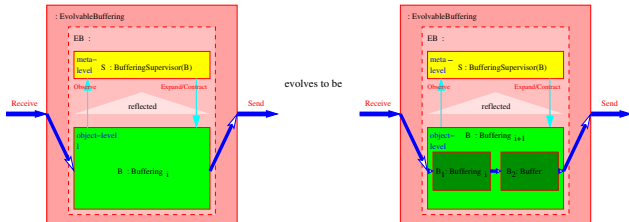
$Send(v) \stackrel{dfn}{=} EB.\langle E.Observe(Q), B.Send(v) \rangle$

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

$Internal \stackrel{dfn}{=} EB.\langle E.Expand, \rangle$

System reconfigurations as evolution (I)

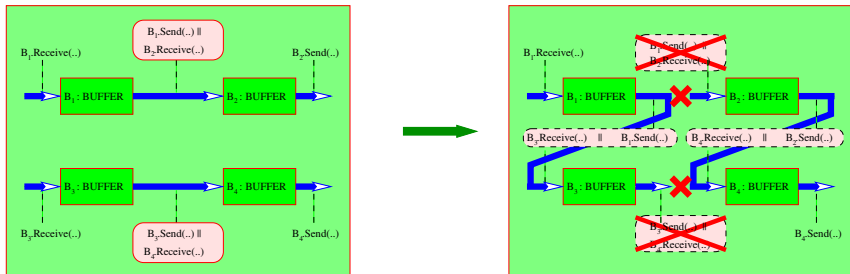
Example: Expanding buffers by concatenating with a fixed capacity buffer:



- The supervisor theory has a revision action of extending the buffer with another of fixed capacity,
- This action revises the collection of theory schema to add another, making the previous buffer schema a subcomponent and redefining the buffer actions on the new buffer,
- A renaming of components takes place so that the extended buffer has the component name of its previous incarnation.

System reconfigurations as evolution (II)

Example: Changing network connectivity:



Here a supervisor theory

- creates a new theory schema with the joint actions of the first system replaced by those of the second system,
- changes the schema associated (in the theory instance hierarchy) with the name of the first system to this new schema.

Proof theory

We have begun to investigate the proof theory of this framework, using ideas from process algebra.

For instance: for evolvable buffers in the first example above (i.e. which have an `Expand` action to change capacity), we can establish:

Theorem

*The `Evolvable_Buffer` component is **weakly bisimilar** to an unbounded buffer, where the internal action is the evolutionary action of `Expand` in `Evolvable_Buffer`.*

Conclusions

- Provided a logical framework for evolvable systems which allows us to model and to reason about the behaviour of such systems.
- We can add **programs** to components to sequence actions, with a transition-based operational semantics and a trace semantics.
- **A logical abstract machine**: Implement the revision steps (via automated reasoning tools) to prototype evolvable systems.
- Relationship with implemented evolvable systems (e.g. in Java) – static proofs and dynamic runtime verification.
- Website www.cs.manchester.ac.uk/evolve

Combining buffers - horizontal composition

Buffering

COMPONENTS

$B_1, B_2 : \text{Buffer}(2)$

ACTIONS

$\text{Send}(v) \stackrel{\text{dfn}}{=} B_2.\text{Send}(v)$

$\text{Receive}(v) \stackrel{\text{dfn}}{=} B_1.\text{Receive}(v)$

$\text{Internal}(v) \stackrel{\text{dfn}}{=} B_1.\text{Send}(v) \parallel B_2.\text{Receive}(v)$

Operational semantics - example rules

Evolve Action

$$\frac{\begin{array}{l} \uparrow_{\varepsilon} \Gamma[\Pi_E, \text{RUNNING}] \xrightarrow{\alpha_E} \uparrow_{\varepsilon} \Gamma'[\Pi'_E, \chi'_E] \\ \Pi'_C = \Pi(\uparrow_c \Gamma'), \text{ where } \text{tmcp}(\uparrow_{\varepsilon} \Gamma, \uparrow_{\varepsilon} \Gamma', \uparrow_c \Gamma, \uparrow_c \Gamma') \\ \chi' = \chi'_E \end{array}}{\Gamma[\Pi_E \text{ META TO } \Pi_C, \text{RUNNING}] \xrightarrow{\langle \alpha_E, \rangle} \Gamma'[\Pi'_E \text{ META TO } \Pi'_C, \chi']}$$

Operational semantics - example rules

Evolve Action

$$\frac{\begin{array}{l} \uparrow_{\varepsilon} \Gamma[\Pi_E, \text{RUNNING}] \xrightarrow{\alpha_E} \uparrow_{\varepsilon} \Gamma'[\Pi'_E, \chi'_E] \\ \Pi'_C = \Pi(\uparrow_c \Gamma'), \text{ where } \text{tmcp}(\uparrow_{\varepsilon} \Gamma, \uparrow_{\varepsilon} \Gamma', \uparrow_c \Gamma, \uparrow_c \Gamma') \\ \chi' = \chi'_E \end{array}}{\Gamma[\Pi_E \text{ META TO } \Pi_C, \text{RUNNING}] \xrightarrow{\langle \alpha_E, \rangle} \Gamma'[\Pi'_E \text{ META TO } \Pi'_C, \chi']}$$

Observation Action

$$\frac{\begin{array}{l} \uparrow_{\varepsilon} \Gamma[\Pi_E, \text{RUNNING}] \xrightarrow{\alpha_O} \uparrow_{\varepsilon} \Gamma'[\Pi'_E, \chi'_E] \\ \uparrow_c \Gamma[\Pi_C, \text{RUNNING}] \xrightarrow{\alpha} \uparrow_c \Gamma'[\Pi'_C, \chi'_C], \text{ where } \text{tmcp}(\uparrow_{\varepsilon} \Gamma, \uparrow_{\varepsilon} \Gamma', \uparrow_c \Gamma, \uparrow_c \Gamma') \\ \chi' = \chi'_E \end{array}}{\Gamma[\Pi_E \text{ META TO } \Pi_C, \text{RUNNING}] \xrightarrow{\langle \alpha_O, \alpha \rangle} \Gamma'[\Pi'_E \text{ META TO } \Pi'_C, \chi']}$$