Towards a logical framework for tightly coupled monitoring and evolution of software components

Howard Barringer

in collaboration with Dov Gabbay, David Rydeheard

School of Computer Science
University of Manchester
email: howard.barringer@manchester.ac.uk

5th January 2007
Health Warning

Logicians at work!

What follows may seem rather
abstract, simplistic and too far from reality.

Please bear with us as we attempt to develop foundations!
Outline

Basic ideas

Introduction and Motivation
Logical modelling — basic ideas
Onwards to components

Where next

Programs . . .
Safe and controlled evolution
Motivation

- Evolvable systems
Motivation

- Evolvable systems
- Software architecture based on Evolver-Producer pairs
Motivation

- Evolvable systems
- Software architecture based on Evolver-Producer pairs
- Software component-based systems
Motivation

- Evolvable systems
- Software architecture based on Evolver-Producer pairs
- Software component-based systems
- Logical modelling and reasoning
Evolver-Producer Pairs

Warboys et al. introduced these as basic building blocks.

The evolver process may stop, decompose, modify, recompose and restart the producer process.

Hierarchical assemblies of such pairings — few constraints on connectivity.
Outline

Basic ideas

Introduction and Motivation
Logical modelling — basic ideas
Onwards to components

Where next
Programs . . .
Safe and controlled evolution
Imagine . . .

. . . a blocks world theory in a typed 1st order logic.

Use an observation predicate

\[
on : \text{Block} \times (\text{Block} \cup \text{Table})
\]

Specify an action, say,

\[
\text{Move} : \text{Block} \times (\text{Block} \cup \text{Table}) \times (\text{Block} \cup \text{Table})
\]

as a revision (e.g. as in STRIPS) on observation states (i.e. a set of ground atoms)

Use theory axioms/constraints to characterise size of tables/blocks, etc., as well as properties of observation predicates. Abstraction predicates can be introduced, e.g.

\[
\text{free} : \text{Table}
\]
Move action specification

Move($x : \text{Block}, y, z : (\text{Block} \cup \text{Table})$)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre</td>
<td>${ \text{on}(x, z), \text{free}(x), \text{free}(y), x \neq y }$</td>
</tr>
<tr>
<td>add</td>
<td>${ \text{on}(x, y) }$</td>
</tr>
<tr>
<td>del</td>
<td>${ \text{on}(x, z) }$</td>
</tr>
</tbody>
</table>
Now imagine ...

... another theory meta to the blocks world, also defined in a typed 1st order logic.

Use observation predicates

\[ \text{holds} : \text{FORMULA} \times \text{ConfigurationName} \]
\[ \text{current} : \text{ConfigurationName} \]
\[ \text{constraint} : \text{CONSTRAINTNAME} \]

Specify an action

\[ \text{Observe} : \text{FORMULAE} \times \text{ConfigurationName} \]

to revise the observation state by the addition of formulas given as argument.
Observing blocks world moves

Meta level view of blocks world at object level

Reflection of "abstracted configurations" at object level
to the "state descriptions" at meta level.

Blocks World ---- object level view
State Meta View

Formally characterises the relationship between

meta level state observations

and

object level configuration

through the use of the \textit{holds} predicate.

E.g. if \textit{holds}(\textit{on}(A, T), c_0) exists in meta-level state observation then \textit{on}(A, T) must exist in observation state of object-level configuration corresponding to \( c_0 \).
Evolving the blocks world

Meta level view of blocks world at object level

Reflection of "abstracted configurations" at object level

to the "state descriptions" at meta level.

Blocks World —— object level view
Transition Meta View

Formally characterises the relationship between \textit{PAIRS} of meta level state observations and object level configurations through the use of an \textit{evolve} predicate.
In the meta-level theory, specify an action:

\[ \text{\textit{Expand}}(m, n, c) \]

- **pre**: \{current(c), constraint(TableSize(T, m))\}
- **add**: \{current(s(c)), holds(free(T), s(c)), constraint(TableSize(T, n))\}
- **del**: \{current(c), constraint(TableSize(T, m))\}
Key points so far

- A configuration describes a logical execution state together with its associated theory instance.
Key points so far

- A configuration describes a logical execution state together with its associated theory instance
- Meta-level and object-level systems are specified in the same way
Key points so far

- A configuration describes a logical execution state together with its associated theory instance
- Meta-level and object-level systems are specified in the same way
- Meta-level and object-level systems are viewed as being executed in lock step synchrony
Key points so far

• A configuration describes a logical execution state together with its associated theory instance
• **Meta-level** and **object-level** systems are specified in the same way
• **Meta-level** and **object-level** systems are viewed as being executed in lock step synchrony
• The **meta-level system** has the capability to view anything of the **object-level configuration**
Key points so far

- A configuration describes a logical execution state together with its associated theory instance
- Meta-level and object-level systems are specified in the same way
- Meta-level and object-level systems are viewed as being executed in lock step synchrony
- The meta-level system has the capability to view anything of the object-level configuration
- Actions are specified as being logically atomic via a revision process.
Key points so far

- A configuration describes a logical execution state together with its associated theory instance
- **Meta-level** and **object-level** systems are specified in the same way
- **Meta-level** and **object-level** systems are viewed as being executed in lock step synchrony
- The **meta-level system** has the capability to view anything of the **object-level** configuration
- Actions are specified as being logically atomic via a revision process.
- At the **meta-level**, actions may observe, or induce change at the **object-level**.
Outline

Basic ideas
Introduction and Motivation
Logical modelling — basic ideas
Onwards to components

Where next
Programs . . .
Safe and controlled evolution
Component Theories

- Treat a component instance as an instance of a particular logical theory
Component Theories

- Treat a component instance as an instance of a particular logical theory
- Have extended the above approach to hierarchical assemblies of component instances
Component Theories

- Treat a component instance as an instance of a particular logical theory
- Have extended the above approach to hierarchical assemblies of component instances
- Use a simple unique naming mechanism to enable straightforward composition of logical theories
Component Theories

- Treat a component instance as an instance of a particular logical theory
- Have extended the above approach to hierarchical assemblies of component instances
- Use a simple unique naming mechanism to enable straightforward composition of logical theories
- Communication between component instances modelled by joint actions
A Buffer theory example

**BUFFER** \((N : \text{Int})\)

**Observation Predicates**

\(\text{content} : \text{Value-list} \)

**Abstraction Predicates**

\(\text{free} \)

**Constraints**

\(\text{Uniqueness} \equiv \forall l_1, l_2 : \text{Value-list} \cdot \text{content}(l_1) \land \text{content}(l_2) \Rightarrow l_1 = l_2 \)

\(\text{Size}(M : \text{Int} \text{ initially } N) \equiv (\exists l : \text{Value-list} \cdot \text{content}(l) \land (|l| < M)) \Leftrightarrow \text{free} \land \forall l : \text{Value-list} \cdot \text{content}(l) \Rightarrow (|l| \leq M) \)

**Actions**

\(\text{Send}(v)\)

<table>
<thead>
<tr>
<th>pre</th>
<th>{content(l::v)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>{content(l)}</td>
</tr>
<tr>
<td>del</td>
<td>{content(l::v)}</td>
</tr>
</tbody>
</table>

\(\text{Receive}(v)\)

<table>
<thead>
<tr>
<th>pre</th>
<th>{free, content(l)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>{content(v::l)}</td>
</tr>
<tr>
<td>del</td>
<td>{content(l)}</td>
</tr>
</tbody>
</table>
A coupled buffer

New logical theories can be constructed through combining instances of other theories.

**BUFFERING**

**COMPONENTS**

\[ B_1, B_2 : BUFFER(2) \]

**ACTIONS**

\[
\begin{align*}
    Send(v) & \overset{\text{dfn}}{=} B_2.Send(v) \\
    Receive(v) & \overset{\text{dfn}}{=} B_1.Receive(v) \\
    Internal & \overset{\text{dfn}}{=} B_1.Send(v) || B_2.Receive(v)
\end{align*}
\]

The specified actions are the only actions that an instance of a BUFFERING theory has.
Combining Meta-level and Object-level theories

- A meta-level theory is specified relative to an object-level theory
Combining Meta-level and Object-level theories

- A meta-level theory is specified relative to an object-level theory.
- The combination of a meta and associated object-level theory is special.
Combining Meta-level and Object-level theories

- A meta-level theory is specified relative to an object-level theory
- The combination of a meta and associated object level theory is special
- It requires identifying how actions can be coupled
Combining Meta-level and Object-level theories

- A meta-level theory is specified relative to an object-level theory.
- The combination of a meta and associated object level theory is special.
- It requires identifying how actions can be coupled.
- Meta-level actions that specify object level theory change occur in isolation.
Pictorially . . .
As a schema . . .

**Evolvable Buffer**

**Components**

\[ EB : (E : BUFFER_EVOLVER \text{ meta to } B : BUFFER(2)) \]

**Actions**

\[
\begin{align*}
\text{Send}(v) & \overset{\text{dfn}}{=} EB.\langle E.\text{Observe}(Q), B.\text{Send}(v) \rangle \\
\text{Receive}(v) & \overset{\text{dfn}}{=} EB.\langle E.\text{Observe}(Q), B.\text{Receive}(v) \rangle \\
\text{Internal} & \overset{\text{dfn}}{=} EB.\langle E.\text{Expand}, \rangle
\end{align*}
\]

\(EB\) names an instance of the special combination of meta and object-level component theories. The actions of the **Evolvable Buffer** component are specified as combinations of the meta and object component actions.
A more structural object-level change
For components, so far ...

- We have a logical framework for supporting the specification of hierarchical assemblies of supervised components.
For components, so far ...

- We have a logical framework for supporting the specification of hierarchical assemblies of supervised components
- Configuration structures for such assemblies are fairly complex
For components, so far ...

- We have a logical framework for supporting the specification of hierarchical assemblies of supervised components.
- Configuration structures for such assemblies are fairly complex.
- Revision is no longer a simple process of augmenting and/or restricting sets of formulae — complex tree manipulations may be required.
- Components communicate vertically through evolutionary processes and horizontally through joint actions defined as disjoint revisions on the subcomponent states.
- The interaction of component hierarchies with evolutionary behaviour appear to yield a powerful methodology for the development of adaptive and evolvable software systems.
For components, so far ...

- We have a logical framework for supporting the specification of hierarchical assemblies of supervised components.
- Configuration structures for such assemblies are fairly complex.
- Revision is no longer a simple process of augmenting and/or restricting sets of formulae — complex tree manipulations may be required.
- Standard revision logics consider only simple revisions — considerable work to be done on proof theory.
For components, so far ...

- We have a logical framework for supporting the specification of hierarchical assemblies of supervised components.
- Configuration structures for such assemblies are fairly complex.
- Revision is no longer a simple process of augmenting and/or restricting sets of formulae — complex tree manipulations may be required.
- Standard revision logics consider only simple revisions — considerable work to be done on proof theory.
- Components communicate vertically through evolutionary processes.
For components, so far...

- We have a logical framework for supporting the specification of hierarchical assemblies of supervised components.
- Configuration structures for such assemblies are fairly complex.
- Revision is no longer a simple process of augmenting and/or restricting sets of formulae — complex tree manipulations may be required.
- Standard revision logics consider only simple revisions — considerable work to be done on proof theory.
- Components communicate *vertically* through evolutionary processes.
- and *horizontally* through joint actions defined as disjoint revisions on the subcomponent states.
For components, so far ...

- We have a logical framework for supporting the specification of hierarchical assemblies of supervised components.
- Configuration structures for such assemblies are fairly complex.
- Revision is no longer a simple process of augmenting and/or restricting sets of formulae — complex tree manipulations may be required.
- Standard revision logics consider only simple revisions — considerable work to be done on proof theory.
- Components communicate vertically through evolutionary processes.
- and horizontally through joint actions defined as disjoint revisions on the subcomponent states.
- The interaction of component hierarchies with evolutionary behaviour appear to yield a powerful methodology for the development of adaptive and evolvable software systems.
Outline

Basic ideas
- Introduction and Motivation
- Logical modelling — basic ideas
- Onwards to components

Where next
- Programs ...
- Safe and controlled evolution
Encapsulating programs

- Where and how do programs over the actions of a component fit in?
Encapsulating programs

- Where and how do programs over the actions of a component fit in?
- Have been experimenting with the view that components are active and come equipped with a single main program?
Encapsulating programs

- Where and how do programs over the actions of a component fit in?
- Have been experimenting with the view that components are active and come equipped with a single main program?
- Components may have different associated languages!
Encapsulating programs

- Where and how do programs over the actions of a component fit in?
- Have been experimenting with the view that components are active and come equipped with a single main program?
- Components may have different associated languages!
- Meta-level components may thus not only change programs, but also programming language of associated object-level systems.
Outline

Basic ideas
- Introduction and Motivation
- Logical modelling — basic ideas
- Onwards to components

Where next
- Programs . . .
- Safe and controlled evolution
An implementation aim!!!!

Whilst we only show a single supervisor that has been implemented in Java, the supervisory tower may be as high, but no higher than the tower of logically modelled supervisors for a component.

The lhs of the picture provides the Evolve Java virtual machine context for safe and controlled execution of evolvable Java programs.
There is an evolving page
http://www.cs.manchester.ac.uk/evolve

Thank you for your patience