Observable Behavior of Distributed Systems:
Component Reasoning for Concurrent Objects

Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, Olaf Owe

Precise Modeling and Analysis group (PMA),
University of Oslo

The 22nd Nordic Workshop on Programming Theory - NWPT ’10
10 November 2010, Turku, Finland
Motivation

- What kind of software systems do we focus on?
- Why it is challenging to reason about distributed systems?
- How do we solve these problems?

Compositional Verification System

Each component can be analyzed independently from its surrounding components.
Motivation

- What kind of software systems do we focus on?
- Why is it challenging to reason about distributed systems?
- How do we solve these problems?

Compositional Verification System

Each component can be analyzed independently from its surrounding components.
Motivation

- What kind of software systems do we focus on?
- Why it is challenging to reason about distributed systems?
- How do we solve these problems?

Compositional Verification System

Each component can be analyzed independently from its surrounding components.
Motivation

- What kind of software systems do we focus on?
- Why it is challenging to reason about distributed systems?
- How do we solve these problems?

Compositional Verification System

Each component can be analyzed independently from its surrounding components.
Main Purpose

- **Reasoning**
- Distributed, concurrent system
- **Main features**
  - encapsulation
  - asynchronous method calls
- **ABS is suitable language for this purpose**
Main Purpose

- Reasoning
- Distributed, concurrent system
- Main features
  - encapsulation
  - asynchronous method calls
- ABS is suitable language for this purpose
Outline

1. The ABS Language
2. Observable Behavior of Distributed Systems
3. Reasoning System for the ABS Classes
4. Object Composition for Concurrency and Object Generation
5. Summary
Abstract Behavioral Specification language (ABS)

- high-level imperative and OO language
- concurrent objects
- method call
- asynchronous
- no access to the internal state variables of other objects
- avoid intra-object interference (only one process is allowed)
- processor release point
  - conditional (await guard;)
  - unconditional (suspend;)
- abstract data types for data structures
- inspired by the Creol language but without inheritance
Abstract Behavioral Specification language (ABS)

- high-level imperative and OO language
- concurrent objects
  - method call
  - asynchronous
- no access to the internal state variables of other objects
- avoid intra-object interference (only one process is allowed)
- processor release point
  - conditional (await guard;)
  - unconditional (suspend;)
- abstract data types for data structures
- inspired by the Creol language but without inheritance
Abstract Behavioral Specification language (ABS)

- high-level imperative and OO language
- concurrent objects
- method call
- asynchronous
- no access to the internal state variables of other objects
- avoid intra-object interference (only one process is allowed)
- processor release point
  - conditional (await guard;)
  - unconditional (suspend;)
- abstract data types for data structures
- inspired by the Creol language but without inheritance
Abstract Behavioral Specification language (ABS)

- high-level imperative and OO language
- concurrent objects
- method call
- asynchronous
- no access to the internal state variables of other objects
- avoid intra-object interference (only one process is allowed)
- processor release point
  - conditional (await guard;)
  - unconditional (suspend;)
- abstract data types for data structures
- inspired by the Creol language but without inheritance
Abstract Behavioral Specification language (ABS)

- high-level imperative and OO language
- concurrent objects
- method call
- asynchronous
- no access to the internal state variables of other objects
- avoid intra-object interference (only one process is allowed)
  - processor release point
    - conditional (await guard;)
    - unconditional (suspend;)
- abstract data types for data structures
- inspired by the Creol language but without inheritance
Abstract Behavioral Specification language (ABS)

- high-level imperative and OO language
- concurrent objects
- method call
- asynchronous
- no access to the internal state variables of other objects
- avoid intra-object interference (only one process is allowed)
- processor release point
  - conditional (await guard;)
  - unconditional (suspend;)
- abstract data types for data structures
- inspired by the Creol language but without inheritance
Abstract Behavioral Specification language (ABS)

- high-level imperative and OO language
- concurrent objects
- method call
- asynchronous
- no access to the internal state variables of other objects
- avoid intra-object interference (only one process is allowed)
- processor release point
  - conditional (await guard;)
  - unconditional (suspend;)
- abstract data types for data structures
- inspired by the Creol language but without inheritance
Abstract Behavioral Specification language (ABS)

- high-level imperative and OO language
- concurrent objects
- method call
- asynchronous
- no access to the internal state variables of other objects
- avoid intra-object interference (only one process is allowed)
- processor release point
  - conditional (await guard;)
  - unconditional (suspend;)
- abstract data types for data structures
- inspired by the Creol language but without inheritance
Reader-Writer Example of ABS

class RWController() implements RW{
    DataSet readers; Obj writer; DB db; Int pr;
    {readers:=Empty;writer:=null;db:=new DataBase();pr:=0;}

    Void openR(){await writer=null;
                readers:=Cons(caller, readers);}

    Void openW(){await writer=null; writer:=caller;}
    Void closeR(){readers:=delete(caller,readers);}
    Void closeW(){await writer=caller; writer:=null;}

    Data read(Int key){Data result;
                        await isElement(caller,readers);pr:=pr+1;
                        await result:=db.read(key);pr:=pr-1; return result;}

    Void write(Int key, Data value){
        await caller=writer && readers=Empty && pr=0;
        db.write(key, value);
    }
}
Outline

1. The ABS Language
2. Observable Behavior of Distributed Systems
3. Reasoning System for the ABS Classes
4. Object Composition for Concurrency and Object Generation
5. Summary
A Method Call Cycle

\[ o \rightarrow o'.m(\bar{e}) \]
A Method Call Cycle

\[ o \rightarrow o'.m(\bar{e}) \]

\[ o \leftarrow o'.m(e) \]

\[ o \rightarrow o'.m(\bar{e}) \]
A Method Call Cycle

\[
o \rightarrow o'.m(\overline{e}) \quad o \leftarrow o'.m(v)
\]
A Method Call Cycle

\[ o \rightarrow o'.m(\overline{e}) \]

\[ o \leftarrow o'.m(v) \]

\[ o \rightarrow o'.m(\overline{e}) \]

\[ o \leftarrow o'.m(v) \]
The communication history (\( \mathcal{H} \)) of a (sub)system up to a given time is a finite sequence of events [Hoare’85, Dahl’87, Broy’01].

Example:

\[
\mathcal{H} := \mathcal{H} \vdash o \rightarrow o'.m(\overline{e})
\]
Communication History

The communication history ($\mathcal{H}$) of a (sub)system up to a given time is a finite sequence of events [Hoare’85, Dahl’87, Broy’01].

Example:

$\mathcal{H} := \mathcal{H} \vdash o \rightarrow o'.m(e)$
Communication History

The communication history ($\mathcal{H}$) of a (sub)system up to a given time is a finite sequence of events [Hoare’85, Dahl’87, Broy’01].

Example:

\[ \mathcal{H} := \mathcal{H} \vdash o \rightarrow o'.m(e) \]
Communication History

The communication history ($\mathcal{H}$) of a (sub)system up to a given time is a finite sequence of events [Hoare’85, Dahl’87, Broy’01].

Example:

$$\mathcal{H} := \mathcal{H} \vdash o \rightarrow o'.m(e)$$
Class Invariant

- contract between the different processes

- must hold
  - after initialization
  - after method termination
  - before suspension

- may assume
  - when method starts
  - after suspension
Verification Problem of $\text{openR()}$

- relates observable communication and internal state

$I(\text{readers}, \mathcal{H})$: $\text{Readers} (\mathcal{H}) = \text{readers}$

where $\text{Readers} (\mathcal{H})$
  - abstractly captures the registered readers
  - completed $\text{openR()}$ but not $\text{closeR()}$
Class Invariant

\[ I_C(\overline{w}, h_{\text{this}}) \triangleq I(\overline{w}, h_{\text{this}}) \land \text{wf}(h_{\text{this}}) \land h_{\text{this}} \text{ bw parent}(\text{this}) \rightarrow \text{this.new } C(cp). \]
Class Invariant

\[ I_C(\overline{w}, h_{this}) \triangleq I(\overline{w}, h_{this}) \land wf(h_{this}) \land h_{this} \text{ bw parent}(this) \rightarrow \text{this.new } C(cp). \]
Class Invariant

\[ I_C(w, h_{\text{this}}) \triangleq I(w, h_{\text{this}}) \land wf(h_{\text{this}}) \land h_{\text{this}} \sim \text{parent(this)} \Rightarrow \text{this.new } C(cp). \]
Class Invariant

\[ I_C(\overline{w}, h_{\text{this}}) \triangleq I(\overline{w}, h_{\text{this}}) \land wf(h_{\text{this}}) \land h_{\text{this}} \text{ bw } \text{parent}(\text{this}) \rightarrow \text{this.new } C(cp). \]
Outline

1 The ABS Language
2 Observable Behavior of Distributed Systems
3 Reasoning System for the ABS Classes
4 Object Composition for Concurrency and Object Generation
5 Summary
Semantic Definition of ABS by a Syntactic Encoding

In [Apt’81,Apt’84], Apt shows that the usual proof system is sound and relative complete in the following sequential language with the syntax:

\[
\text{skip} \mid \text{abort} \mid \overline{\nu} := \overline{e} \mid s_1; s_2 \mid \text{if } b \text{ then } s_1 \text{ [else } s_2]\? \text{ fi.}
\]

Additionally, we add extra statements (S) and the corresponding weakest liberal preconditions (WLP):

<table>
<thead>
<tr>
<th>statement S</th>
<th>wlp(S,Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y := \text{some } x . \ P(x) )</td>
<td>( \forall x . (P(x) \Rightarrow Q_{x}^{y}) )</td>
</tr>
<tr>
<td>assert ( b )</td>
<td>( b \land Q )</td>
</tr>
<tr>
<td>assume ( b )</td>
<td>( b \Rightarrow Q )</td>
</tr>
</tbody>
</table>
Semantic Definition of ABS by a Syntactic Encoding

In [Apt’81,Apt’84], Apt shows that the usual proof system is sound and relative complete in the following sequential language with the syntax:

\[
\text{skip} \mid \text{abort} \mid \overline{v} := \overline{e} \mid s_1; s_2 \mid \text{if } b \text{ then } s_1 \text{ [else } s_2]\text{? } \text{fi.}
\]

Additionally, we add extra statements (S) and the corresponding weakest liberal preconditions (WLP):

<table>
<thead>
<tr>
<th>statement S</th>
<th>wlp(S,Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\overline{y} := \text{some } \overline{x} \cdot P(\overline{x}))</td>
<td>(\forall \overline{x} \cdot (P(\overline{x}) \Rightarrow Q_{\overline{y}}))</td>
</tr>
<tr>
<td>\text{assert } b</td>
<td>(b \land Q)</td>
</tr>
<tr>
<td>\text{assume } b</td>
<td>(b \Rightarrow Q)</td>
</tr>
</tbody>
</table>
Semantic Definition of ABS by a Syntactic Encoding

In [Apt’81,Apt’84], Apt shows that the usual proof system is sound and relative complete in the following sequential language with the syntax:

\[
\text{skip} | \text{abort} | \overline{v} := \overline{e} | s_1; s_2 | \text{if } b \text{ then } s_1 [\text{else } s_2]? \text{ fi.}
\]

Additionally, we add extra statements \((S)\) and the corresponding weakest liberal preconditions \((WLP)\):

<table>
<thead>
<tr>
<th>statement (S)</th>
<th>(wlp(S,Q))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\overline{y} := \text{some } \overline{x}. P(\overline{x}))</td>
<td>(\forall \overline{x}. (P(\overline{x}) \Rightarrow Q_{\overline{y}}))</td>
</tr>
<tr>
<td>\text{assert } b</td>
<td>(b \land Q)</td>
</tr>
<tr>
<td>\text{assume } b</td>
<td>(b \Rightarrow Q)</td>
</tr>
</tbody>
</table>
Semantic Definition of ABS by a Syntactic Encoding

In [Apt’81,Apt’84], Apt shows that the usual proof system is sound and relative complete in the following sequential language with the syntax:

\[
\text{skip} \mid \text{abort} \mid \nu := \bar{e} \mid s_1; s_2 \mid \text{if } b \text{ then } s_1 \text{ [else } s_2] ? \text{ fi}.
\]

Additionally, we add extra statements (S) and the corresponding weakest liberal preconditions (WLP):

<table>
<thead>
<tr>
<th>statement S</th>
<th>wlp(S,Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{y} := \text{some } \bar{x} . P(\bar{x})$</td>
<td>$\forall \bar{x} . (P(\bar{x}) \Rightarrow Q_{\bar{y}})$</td>
</tr>
<tr>
<td>assert $b$</td>
<td>$b \land Q$</td>
</tr>
<tr>
<td>assume $b$</td>
<td>$b \Rightarrow Q$</td>
</tr>
</tbody>
</table>
Semantic Definition of ABS by a Syntactic Encoding

In [Apt’81,Apt’84], Apt shows that the usual proof system is sound and relative complete in the following sequential language with the syntax:

\[
\text{skip} \mid \text{abort} \mid \overline{v} := \overline{e} \mid s_1 ; s_2 \mid \text{if } b \text{ then } s_1 [\text{else } s_2]? \text{ fi.}
\]

Additionally, we add extra statements (S) and the corresponding weakest liberal preconditions (WLP):

<table>
<thead>
<tr>
<th>statement S</th>
<th>wlp(S,Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y := \text{some } x . \ P(x))</td>
<td>(\forall x . (P(x) \Rightarrow Q_{\overline{y}}))</td>
</tr>
<tr>
<td>assert b</td>
<td>(b \land Q)</td>
</tr>
<tr>
<td>assume b</td>
<td>(b \Rightarrow Q)</td>
</tr>
</tbody>
</table>
Semantic Definition of ABS by a Syntactic Encoding

In [Apt’81,Apt’84], Apt shows that the usual proof system is sound and relative complete in the following sequential language with the syntax:

\[
\text{skip} | \text{abort} | \overline{v} := \overline{e} | s_1; s_2 | \text{if } b \text{ then } s_1 \text{ [else } s_2]\text{? fi.}
\]

Additionally, we add extra statements (S) and the corresponding weakest liberal preconditions (WLP):

<table>
<thead>
<tr>
<th>statement S</th>
<th>wlp(S,Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\overline{y} := \text{some} \overline{x} \cdot P(\overline{x}))</td>
<td>(\forall \overline{x} \cdot (P(\overline{x}) \Rightarrow Q_{\overline{x}}))</td>
</tr>
<tr>
<td>assert b</td>
<td>b \land Q</td>
</tr>
<tr>
<td>assume b</td>
<td>b \Rightarrow Q</td>
</tr>
</tbody>
</table>
Semantic Definition of ABS by a Syntactic Encoding

In [Apt’81,Apt’84], Apt shows that the usual proof system is sound and relative complete in the following sequential language with the syntax:

\[
\text{skip} \mid \text{abort} \mid \overline{v} := \overline{e} \mid s_1; s_2 \mid \text{if } b \text{ then } s_1 \text{ [else } s_2] ? \text{ fi.}
\]

Additionally, we add extra statements (S) and the corresponding weakest liberal preconditions (WLP):

<table>
<thead>
<tr>
<th>statement S</th>
<th>wlp(S,Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \overline{y} := \text{some } \overline{x} \cdot P(\overline{x}) )</td>
<td>( \forall \overline{x} . (P(\overline{x}) \Rightarrow Q_{\overline{x}}^{\overline{y}}) )</td>
</tr>
<tr>
<td>assert ( b )</td>
<td>( b \land Q )</td>
</tr>
<tr>
<td>assume ( b )</td>
<td>( b \Rightarrow Q )</td>
</tr>
</tbody>
</table>
Semantic Definition of ABS by a Syntactic Encoding

In [Apt’81,Apt’84], Apt shows that the usual proof system is sound and relative complete in the following sequential language with the syntax:

\[
\text{skip} | \text{abort} | \overline{v} := \overline{e} | s_1; s_2 | \text{if} \ b \ \text{then} \ s_1 \ [\text{else} \ s_2] \ ? \ \text{fi}.
\]

Additionally, we add extra statements (S) and the corresponding weakest liberal preconditions (WLP):

<table>
<thead>
<tr>
<th>statement S</th>
<th>\text{wlp}(S,Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \overline{y} := \text{some} \ \overline{x} . P(\overline{x}) )</td>
<td>( \forall \ \overline{x} . (P(\overline{x}) \Rightarrow Q_{\overline{y}}) )</td>
</tr>
<tr>
<td>\text{assert} b</td>
<td>b \land Q</td>
</tr>
<tr>
<td>\text{assume} b</td>
<td>b \Rightarrow Q</td>
</tr>
</tbody>
</table>
Syntactic Encoding and WLP of \textbf{Methods}

\[
\langle\langle m(x) \ B \rangle\rangle \triangleq \\
\mathcal{H} := \mathcal{H} \vdash \text{caller } \rightarrow \ \text{this.m}(\text{return}); \langle\langle B \rangle\rangle \\
\mathcal{H} := \mathcal{H} \vdash \text{caller } \leftarrow \ \text{this.m}(\text{return}); \ \text{assume} \ \text{wf}(\mathcal{H})
\]

\[
wlp(m(x) \ B, Q) \triangleq \\
wlp(\mathcal{H} := \mathcal{H} \vdash \text{caller } \rightarrow \ \text{this.m}(\text{return}); B; \\
\mathcal{H} := \mathcal{H} \vdash \text{caller } \leftarrow \ \text{this.return}, \ \text{wf}(\mathcal{H}) \ \Rightarrow \ Q)
\]
Syntactic Encoding and WLP of `await b`

\[
\langle\text{await} \ b\rangle \triangleq \\
\text{if } b \text{ then skip else assert } I_C(\overline{w}, \mathcal{H}); \\
\overline{w}, \mathcal{H} := \text{some } \overline{w}, \mathcal{H}' \cdot \mathcal{H} \leq \mathcal{H}' \land I_C(\overline{w}, \mathcal{H}') \land b \text{ fi}
\]

\[
wlp(\text{await} \ b, Q) \triangleq \\
\text{if } b \text{ then } Q \text{ else } I_C(\overline{w}, \mathcal{H}) \land \\
\forall \overline{w}, \mathcal{H}' . (\mathcal{H} \leq \mathcal{H}' \land I_C(\overline{w}, \mathcal{H}') \land b) \implies Q^\mathcal{H}_{\mathcal{H}'}
\]
Hoare Reasoning

\{P\} \ S \ \{Q\} \ \text{is the same as} \ P \ \Rightarrow \ wlp(S, Q)

Given that \ wlp \ \text{is Sound and Complete}

\{P\} \ S \ \{Q\} \ \text{is sound if} \ P \ \Rightarrow \ wlp(S, Q).
\{P\} \ S \ \{Q\} \ \text{is complete if} \ P \ \Leftrightarrow \ wlp(S, Q).
Hoare Reasoning

\{P\} S \{Q\} is the same as \( P \Rightarrow \text{wlp}(S, Q) \)

Given that \( \text{wlp} \) is Sound and Complete

\{P\} S \{Q\} is sound if \( P \Rightarrow \text{wlp}(S, Q) \).
\{P\} S \{Q\} is complete if \( P \Leftrightarrow \text{wlp}(S, Q) \).
Derived Hoare Rule for Methods

- Derived Hoare Rule for \( m(\overline{x}) \) B:

\[
\{ I_{C_{\text{pop}(\mathcal{H})}}^{\mathcal{H}} \wedge \mathcal{H} \text{ew caller } \rightarrow \text{this.m(}\overline{x}\text{)} \} \ B \ \{ I_{C_{\mathcal{H}_{-}\text{caller}\leftarrow\text{this.m(return)}}}^{\mathcal{H}} \}
\]

- Follows by WLP for \( m(\overline{x}) \) B
Derived Hoare Rule for **Methods**

- Derived Hoare Rule for $m(x)$ B:

  $$\{ I_{C_{\text{pop}(\mathcal{H})}} \land \mathcal{H} \text{ ew caller } \rightarrow this.m(x) \} \ B \ \{ I_{C_{\mathcal{H} \leftarrow \text{caller} \leftarrow this.m(return)}} \}$$

- Follows by WLP for $m(x)$ B
Derived Hoare Rules for \textbf{await} \ b

\[
\{\mathcal{H}_0 = \mathcal{H}\} \quad \text{await} \ b \quad \{\mathcal{H}_0 \leq \mathcal{H} \land b\}
\]

\[
\{l_c\} \quad \text{await} \ b \quad \{l_c \land b\}
\]

\[
\{Q \land b\} \quad \text{await} \ b \quad \{Q \land b\}
\]

- Follows by WLP for \textbf{await} \ b
- The await rules can be shown complete (assuming standard adaptation and disjunction rules).
Derived Hoare Rules for `await b`

- \[
\{H_0 = H\} \ await \ b \ \{H_0 \leq H \land b\}
\]

- \[
\{l_C\} \ await \ b \ \{l_C \land b\}
\]

- \[
\{Q \land b\} \ await \ b \ \{Q \land b\}
\]

- Follows by WLP for `await b`

- The await rules can be shown complete (assuming standard adaptation and disjunction rules).
Derived Hoare Rules for `await b`

- \(\{\mathcal{H}_0 = \mathcal{H}\} \quad \text{await} \quad b \quad \{\mathcal{H}_0 \leq \mathcal{H} \land b\}\)

- \(\{l_c\} \quad \text{await} \quad b \quad \{l_c \land b\}\)

- \(\{Q \land b\} \quad \text{await} \quad b \quad \{Q \land b\}\)

- Follows by WLP for `await b`

- The await rules can be shown complete (assuming standard adaptation and disjunction rules).
Reasoning About openR in RW Example

\{ I : \text{Readers}(\mathcal{H}) = \text{readers} \}\n
openR()
\{
\{ \mathcal{H}^{\text{pop}(\mathcal{H})} \land \mathcal{H} \text{ew} \text{ caller} \rightarrow \text{this.openR} \}
\{ \text{Readers}(\mathcal{H}) = \text{readers} \}\n
\textbf{await writer} = \text{NULL};
\{ \text{Readers}(\mathcal{H}) = \text{readers} \land \text{writer} = \text{NULL} \}
\{ \text{Readers}(\mathcal{H}) \cup \{ \text{caller} \} = \text{Cons}(\text{caller}, \text{readers}) \}\n
\text{readers} := \text{Cons}(\text{caller}, \text{readers})
\{ \mathcal{H}^{\text{H}_{\text{H} \rightarrow \text{caller} \rightarrow \text{this.openR}}} \}
\}\n
\{ I : \text{Readers}(\mathcal{H}) = \text{readers} \}
Reasoning About openR in RW Example

\{ l : Readers(\mathcal{H}) = readers \}

\texttt{openR()}
\begin{align*}
& \begin{cases}
{l^\mathcal{H}_{\text{pop}}(\mathcal{H}) \land \mathcal{H} \text{ew} \text{ caller } \rightarrow this.openR} \\
{Readers(\mathcal{H}) = readers}
\end{cases} \\
& \texttt{await writer = NULL;} \\
& \begin{cases}
{Readers(\mathcal{H}) = readers \land writer = NULL} \\
{Readers(\mathcal{H}) \cup \{\text{caller}\} = \text{Cons}(\text{caller}, readers)}
\end{cases} \\
& \texttt{readers := Cons(caller, readers)} \\
& \begin{cases}
{l^\mathcal{H}_{\text{caller} \leftarrow this.openR}}
\end{cases}
\end{align*}

\{ l : Readers(\mathcal{H}) = readers \}
Reasoning About openR in RW Example

\{ I : \text{Readers}(H) = \text{readers} \}

\text{openR()}
\{
\{ \exists H_{\text{pop}}(H) \land H \ \text{ew} \ \text{caller} \rightarrow \text{this.openR} \}
\{ \text{Readers}(H) = \text{readers} \}
\text{await writer} = \text{NULL};
\{ \text{Readers}(H) = \text{readers} \land \text{writer} = \text{NULL} \}
\{ \text{Readers}(H) \cup \{ \text{caller} \} = \text{Cons}(\text{caller}, \text{readers}) \}
\text{readers} := \text{Cons}(\text{caller}, \text{readers})
\{ I^H \}
\{ I^H_{\text{caller} \rightarrow \text{this.openR}} \}
\}

\{ I : \text{Readers}(H) = \text{readers} \}
Reasoning About openR in RW Example

\{ I : \text{Readers}(\mathcal{H}) = \text{readers} \}

\text{openR()}
\{ \}
\{ I^\mathcal{H}_{\text{pop}} \land \mathcal{H} \text{ ew} \text{ caller} \rightarrow \text{this.openR} \}
\{ \text{Readers}(\mathcal{H}) = \text{readers} \}
\text{await writer = NULL;}
\{ \text{Readers}(\mathcal{H}) = \text{readers} \land \text{writer} = \text{NULL} \}
\{ \text{Readers}(\mathcal{H}) \cup \{ \text{caller} \} = \text{Cons} (\text{caller}, \text{readers}) \}
\text{readers} := \text{Cons} (\text{caller}, \text{readers})
\{ I^\mathcal{H}_{\text{caller} \leftarrow \text{this.openR}} \}
\}

\{ I : \text{Readers}(\mathcal{H}) = \text{readers} \}
Reasoning About openR in RW Example

\{ I : \textit{Readers}(\mathcal{H}) = \textit{readers} \} \\

\texttt{openR()}

\{ \} \\
\ \{ \textit{I}_{\text{pop}}(\mathcal{H}) \land \mathcal{H} \texttt{ew} \textit{caller} \rightarrow \textit{this.openR} \} \\
\{ \textit{Readers}(\mathcal{H}) = \textit{readers} \} \\
\texttt{await writer = NULL;}

\{ \textit{Readers}(\mathcal{H}) = \textit{readers} \land writer = \texttt{NULL} \} \\
\{ \textit{Readers}(\mathcal{H}) \cup \{ \textit{caller} \} = \text{Cons}(\textit{caller}, \textit{readers}) \} \\
\textit{readers} := \text{Cons}(\textit{caller}, \textit{readers}) \\
\{ \textit{I}_{\mathcal{H}}^{\text{caller} \leftarrow \textit{this.openR}} \} \\
\} \\

\{ I : \textit{Readers}(\mathcal{H}) = \textit{readers} \}
Outline

1. The ABS Language
2. Observable Behavior of Distributed Systems
3. Reasoning System for the ABS Classes
4. Object Composition for Concurrency and Object Generation
5. Summary
History Invariant

History invariant is a predicate over the communication history expressing safety properties [Dahl’87, Dahl’92].
Composition Rule

- **Object Invariant**

\[ l_{o:C(E)}(h_o) \triangleq \exists \overline{w} . (l_{C(\overline{w}, h_{\text{this}})})^{\text{this, cp}}_{o,E} \]

abstracting away the internal state

- **Global Invariant**

\[ I(H) \triangleq ( \bigwedge_{(o:C(\overline{e})) \in \text{ob}(H)} l_{o:C(\overline{e})}(H/o)) \land \text{wf}(H) \]

reflecting concurrent composition of objects and dynamic object creation where \( H/o = h_o \)
Composition Rule

- **Object Invariant**

\[
I_{o:C(E)}(h_o) \triangleq \exists w . (I_{C(w, h_{this})})^{tthis,cp}_{o,E}
\]

abstracting away the internal state

- **Global Invariant**

\[
I(H) \triangleq (\bigwedge_{(o:C(e)) \in \text{ob}(H)} I_{o:C(e)}(H/o)) \land \text{wf}(H)
\]

reflecting concurrent composition of objects and dynamic object creation where \(H/o = h_o\)
Outline

1. The ABS Language
2. Observable Behavior of Distributed Systems
3. Reasoning System for the ABS Classes
4. Object Composition for Concurrency and Object Generation
5. Summary
Conclusion

- A sound and complete reasoning system for the ABS language
- Classes can be specified independently from the surroundings
- Modularity is achieved
- Global specification is realized from composing local specifications
Comparison to Related Work

- The work is based on [Dovland’08, Ahrent’10] but simpler
- No concept of input insensitivity and need not prove it
- Different notion of locality and events
- No nondeterministic extension of the history with environment activity
- Unrestricted use of assumptions on the environment
- Valuable when reasoning about objects in an open environment
Future Work

- Extend the system for ABS future variables
- Implementation in KeY
- Semi-automatic verification
- Large case studies
References


References


Thank You