Design and Analysis of Executable Software Models: An Introduction and Overview

Reiner Hähnle
joint work with Antonio F. Montoya, Richard Bubel, Crystal C. Din and many others!

Technical University of Darmstadt
haehnle@cs.tu-darmstadt.de

14th International School on Formal Methods for the Design of Computer, Communication and Software Systems:
Executable Software Models
Bertinoro, 16th June 2014

http://www.envisage-project.eu
Part I

The ABS Modeling Language
What ABS Is All About

Consequences of design time decisions often realized only at runtime

- Modern SW development often model-/feature-driven
- Most modeling languages do not address behavior rigorously
- Mismatch among artefacts from analysis and coding phases
- “Built-in” disconnect between analysts and implementors
- Complicating factors: product variability, concurrency
Consequences of design time decisions often realized only at runtime

- Modern SW development often model-/feature-driven
- Most modeling languages do not address behavior rigorously
- Mismatch among artefacts from analysis and coding phases
- “Built-in” disconnect between analysts and implementors
- Complicating factors: product variability, concurrency
Main Design Goals of ABS

ABS is designed with analysis/code generation tools in mind

- Expressivity carefully traded off with analysability
  - enables **incremental/compositional** static and dynamic analyses

- State-of-art programming language concepts
  - ADTs + functions + objects
  - type-safety, data race-freeness by design
  - modules, components
  - pluggable type systems, annotations

- Layered concurrency model
  - Upper tier: asynchronous, no shared state, actor-based
  - Lower tier: synchronous, shared state, cooperative multitasking

- Modeling of variability/deployment with first-class language support
  - feature models, delta-oriented programming
  - deployment components

- Not only code analysis, but also **code generation/model mining**
ABS Design Principles

- Uniform, formal semantics
- Layered architecture: simplicity, separation of concerns
- Includes standard class-based, imperative and functional sublanguages
- Executability: simulation, rapid prototyping, visualization
- Abstraction: underspecification, non-determinism
- Realistic, yet language-independent concurrency model
- Module system (aka packages)
Layered ABS Language Design

- Feature Modeling
- Real-Time ABS Deployment Components
- Runtime Components

- Behavioral Interface Specs
- Local Contracts, Assertions
- Syntactic Modules
- Asynchronous Communication
- Concurrent Object Groups (COGs)
- Imperative Language
- Object Model
- Pure Functional Programs
- Algebraic (Parametric) Data Types

Full ABS
Core ABS
Built-In Data Types and Operators

Built-In Data Types

```haskell
data Bool = True | False;
data Unit = Unit;
data Int;  // 4, 2323, −23
data String; // "Hello World"
```
Built-In Data Types and Operators

Built-In Data Types

```plaintext
data Bool = True | False;
data Unit = Unit;
data Int; // 4, 2323, −23
data String; // "Hello World"
```

Built-In Operators (Java-like Syntax)

- All types:  `==`  `!=`
- `Bool`:  `~`  `&&`  `||`
- `Int`:  `+`  `-`  `*`  `/`  `%`  `<`  `>`  `<=`  `>=`
- `String`:  `+`
### User Defined Algebraic Data Types

**User-Defined Data Types**

```haskell
data Fruit = Apple | Banana | Cherry;
data Juice = Pure(Fruit) | Mixed(Juice, Juice);
type Saft = Juice; // type synonym
```
User Defined Algebraic Data Types

User-Defined Data Types

```haskell
data Fruit = Apple | Banana | Cherry;
data Juice = Pure(Fruit) | Mixed(Juice, Juice);
type Saft = Juice;  // type synonym
```

Parametric Data Types

```haskell
data List<T> = Nil | Cons(T, List<T>);
```
User Defined Algebraic Data Types

User-Defined Data Types

```haskell
data Fruit = Apple | Banana | Cherry;
data Juice = Pure(Fruit) | Mixed(Juice, Juice);
type Saft = Juice;  // type synonym
```

Parametric Data Types

```haskell
data List<T> = Nil | Cons(T, List<T>);
```

Selectors

```haskell
data Person = Person(String name, Int age, String address);
// if selector names present, they implicitly define selector functions
def String name(Person p) = ... ;
```
def Int length(IntList list) = // function names lower—case
  case list { // definition by case distinction and matching
    Nil => 0;
    Cons(n, ls) => 1 + length(ls); // data constructor pattern
    _ => 0; // underscore pattern (anonymous variable)
  };

def A head<A>(List<A> list) = // parametric function
  case list {
    Cons(x, xs) => x; // unbound variable used to extract value
  };
module ABS.StdLib;
export *
;

data Maybe\textlangle A \textrangle = Nothing | Just(A);
data Either\textlangle A, B \textrangle = Left(A) | Right(B);
data Pair\textlangle A, B \textrangle = Pair(A, B);
data List\textlangle T \textrangle = ...;
data Set\textlangle T \textrangle = ...;
data Map\textlangle K, V \textrangle = ...;

...

def Int size\textlangle A \textrangle(Set\textlangle A \textrangle xs) = ... 
def Set\textlangle A \textrangle union\textlangle A \textrangle(Set\textlangle A \textrangle set1, Set\textlangle A \textrangle set2) = ...
...

Object Model: Interfaces

Interfaces

- Provide types of objects (implementation abstraction)
- Multiple inheritance
- Subinterfaces

```
interface Baz { ... }
interface Bar extends Baz {
    // method signatures
    Unit m();
    Bool foo(Bool b);
}
```
Object Model: Classes

Classes

- Only for object construction
- No type
- No code inheritance (instead delta-oriented programming is used)

```java
// class declaration with parameters, implicitly defines constructor
class Foo(T x, U y) implements Bar, Baz {
    // field declarations
    Bool flag = False; // primitive types must be initialized
    U g; // object type field initialization optional
    {
        // optional class initialization block
        g = y;
    }
    Unit m() { } // method implementations
    Bool foo(Bool b) { return ~b; }
}
```
Active Classes

- Characterized by presence of `run()` method
- Objects from active classes start activity after initialization
- Passive classes react only to incoming calls

```java
Unit run() {
    // active behavior ...
}
```
Imperative Constructs

Sequential Control Flow

Loop \textbf{while} \ (x) \{ \ ... \ \}

Conditional \textbf{if} \ (x == y) \{ \ ... \ \} [\textbf{else} \{ \ ... \ \}]

Synchronous method call \texttt{x.m()}

Local State Update and Access (Assignment)

Object creation \texttt{new Car(Blue)};

Field read \texttt{x = [this.]f; (only on this object)}

Field assignment \texttt{[this.]f = 5; (only on this object)}

Blocks

- Sequence of variable declarations and statements
- Data type variables are initialized, reference types default to \texttt{null}
- Statements in block are \texttt{scope} for declared variables
Concurrency Model

Layered Concurrency Model

Upper tier: asynchronous, no shared state, actor-based
Lower tier: synchronous, shared state, cooperative multitasking

Concurrent Object Groups (COGs)

- Unit of distribution
- Own heap of objects
- Cooperative multitasking inside COGs
  - One processor, several tasks
  - Intra-group communication by synchronous/asynchronous method calls
  - Multiple tasks originating from asynchronous calls within COG
- Inter-group communication only via asynchronous method calls
Cooperative Multitasking inside COGs

**Multitasking**

- A COG can have multiple tasks
- Only one is active, all others are suspended
- Asynchronous calls create new tasks
- Synchronous calls block caller thread
  - Java-like syntax: `target.methodName(arg1, arg2, ...)`

**Scheduling**

- Cooperative by special scheduling statements
  - Explicit decision of modeller
  - No preemptive scheduling ⇒ no data races within COGs
- Non-deterministic otherwise
  - User-defined configuration of schedulers via annotations
ABS Concurrency Model

Method calls with shared heap access encapsulated in COGs

**COG**
- One activity at a time
- One lock
- Cooperative scheduling
- Callbacks (recursion) ok
- Shared access to data

```java
this:A
new B();
this:A b:B
```
Method calls with shared heap access encapsulated in COGs
ABS Concurrency Model

Distributed computation: async. calls/message passing/separate heap

```
this:A
new cog B();
this:A
b:B
```
# Asynchronous Method Calls

- **Syntax:** `target ! methodName(arg1, arg2, ...)`
- Creates new task in COG of target
- Caller continues execution and allocates a `future` to store the result
  - `Fut<T> v = o!m(e);`

## Conditional Scheduling (Waiting for the Future)

- `await g`, where `g` is a polling guard
  - Yields task execution until guard is true

## Reading Futures

- `f.get` - reads future `f` and blocks execution until result is available
- Deadlocks possible (use static analyzer for detection)
  - Programming idiom: `await f?` prevents blocking (safe access)

```java
Fut<T> v = o!m(e);...; await v?; r = v.get;
```
### Asynchronous Method Calls

**Syntax:**
```
target ! methodName(arg1, arg2, ...)  # Creates new task in COG of target
```

**Description:**
- Caller continues execution and allocates a `future` to store the result
  - `Fut<T> v = o!m(e);`

### Conditional Scheduling (Waiting for the Future)

- `await g`, where `g` is a polling `guard`
- Yields task execution until guard is true
Asynchronous Method Calls

Syntax:  
\[
\text{target} ! \text{methodName}(\text{arg1, arg2, ...})
\]

- Creates new task in COG of target
- Caller continues execution and allocates a future to store the result
  - \[
  \text{Fut}<\text{T}> \ v = \text{o!m(e)};
  \]

Conditional Scheduling (Waiting for the Future)

- \[
  \text{await g, where g is a polling guard}
  \]
- Yields task execution until guard is true

Reading Futures

- \[
  \text{f.get} - \text{reads future f and blocks execution until result is available}
  \]
- Deadlocks possible (use static analyzer for detection)
- Programming idiom: \[
  \text{await f? prevents blocking (safe access)}
  \]
  - \[
  \text{Fut}<\text{T}> \ v = \text{o!m(e)}; \ldots; \text{await v?}; \ r = \text{v.get};
  \]
Part II

Resource Analysis Of ABS Models
Static resource analysis attempts to infer upper bounds on the amount of resources that can be consumed by a system.

Some typical measured resources:
- Time (Computational complexity)
- Space (Memory consumption)

Related to distributed systems and ABS models:
- Bandwidth
- Executed instructions per component (COG)
An example that we want to analyze:

class DBImpl implements DB {
    List<Account> as = Nil;
    Account getAccount(Int aid) {
        Account result = null;
        Int n = length(as);
        Int cnt = 0;
        while (cnt < n) {
            Account a = nth(as, cnt);
            Fut<Int> idFut = a!getAid();
            Int id = idFut.get;
            if (aid == id) {
                result = a;
            }
            cnt = cnt+1;
        }
        return result;
    }
    ...
}
A ABS model fragment

An example that we want to analyze:

class DBImpl implements DB {
    List<Account> as = Nil;
    Account getAccount(Int aid) {
        Account result = null;
        Int n = length(as);
        Int cnt = 0;
        while (cnt < n) {
            Account a = nth(as,cnt);
            Fut<Int>idFut = a!getAid();
            Int id=idFut.get;
            if (aid == id) {
                result = a;
            }
            cnt = cnt+1;
        }
        return result;
    }
    ...
}

▶ A method getAccount that searches an Account aid
▶ It iterates over a field as
A ABS model fragment

An example that we want to analyze:

class DBImpl implements DB {
    List<Account> as = Nil;
    Account getAccount(Int aid) {
        Account result = null;
        Int n = length(as);
        Int cnt = 0;
        while (cnt < n) {
            Account a = nth(as,cnt);
            Fut<Int>idFut = a!getAid();
            Int id=idFut.get;
            if (aid == id) {
                result = a;
            } else {
                cnt = cnt+1;
            }
        }
        return result;
    }
    ...
}
A ABS model fragment

An example that we want to analyze:

```java
class DBImpl implements DB {
    List<Account> as = Nil;
    Account getAccount(Int aid) {
        Account result = null;
        Int n = length(as);
        Int cnt = 0;
        while (cnt < n) {
            Account a = nth(as, cnt);
            Fut<Int> idFut = a!getAid();
            Int id = idFut.get;
            if (aid == id) {
                result = a;
            }
            cnt = cnt + 1;
        }
        return result;
    }
    ...
}
```

- A method `getAccount` that searches an Account `aid`
- It iterates over a field `as`
- For each account `a` in `as`, call `getAid` and block execution until result is ready
- If `id` is the account we are looking for, store it in `result`
Basic approach (for Sequential Code)

1 Select a cost model
   - map each instruction to the amount of resources it consumes
   - dependant on the kind of resource to be measured

```java
Account getAccount(Int aid) {
    Account result = null;
    Int n = length(as);
    Int cnt = 0;
    while (cnt < n) {
        Account a = nth(as, cnt);  // 1 + nth(as, cnt)
        Fut<Int>idFut = a!getAid(); // 1 + getAid()
        Int id=idFut.get; // 1
        if (aid == id) {
            result = a; // 1
        }
        cnt = cnt+1; // 1
    }
    return result;
}
```
Basic approach (for Sequential Code)

2. Perform size analysis and generate abstract representation
   - Abstract data structures to an integer representation of their size
   - Abstract method calls to cost functions

Account getAccount(Int aid) {
    Account result = null;
    Int n = length(as);
    Int cnt = 0;
    while (cnt < n) {
        Account a = nth(as, cnt);
        Fut<Int> idFut = a!getAid();
        Int id = idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt + 1;
    }
    return result;
}
Basic approach (for Sequential Code)

Perform size analysis and generate abstract representation

- Abstract data structures to an integer representation of their size
- Abstract method calls to cost functions
- Generate a cost equation for each block of code

```java
Account getAccount(Int aid) {
    Account result = null;
    Int n = length(as);
    Int cnt = 0;
    while (cnt < n) {
        Account a = nth(as, cnt);
        Fut<Int>idFut = a!getAid();
        Int id=idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}
```

The list as is abstracted to its length
Basic approach (for Sequential Code)

2. Perform size analysis and generate abstract representation
   - Abstract data structures to an integer representation of their size
   - Abstract method calls to cost functions
   - Generate a cost equation for each block of code

```java
Account getAccount(Int aid) {
    Account result = null;
    Int n = length(as);
    Int cnt = 0;
    while (cnt < n) {
        Account a = nth(as, cnt);  // 1
        Fut<Int> idFut = a!getAid();  // 1 + getAid()
        Int id = idFut.get;  // 1
        if (aid == id) {  // 1
            result = a;  // 1
        }
        cnt = cnt + 1;  // 1
    }
    return result;
}
```

- The list `as` is abstracted to its length
Basic approach (for Sequential Code)

2. Perform size analysis and generate abstract representation
   • Abstract data structures to an integer representation of their size
   • Abstract method calls to cost functions
   • Generate a cost equation for each block of code

Account getAccount(Int aid) {
    Account result = null;
    Int n = length(as);
    Int cnt = 0;
    while (cnt < n) {
        Account a = nth(as, cnt);
        Fut<Int>idFut = a!getAid();
        Int id=idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}

The list as is abstracted to its length
Perform size analysis and generate abstract representation

- Abstract data structures to an integer representation of their size
- Abstract method calls to cost functions
- Generate a cost equation for each block of code

Account getAccount(Int aid) {
    Account result = null;
    Int n = length(as);
    Int cnt = 0;
    while (cnt < n) {
        Account a = nth(as,cnt); 1 + nth(as, cnt)
        Fut<Int>idFut = a!getAid(); 1 + getAid()
        Int id=idFut.get; 1
        if (aid == id) {
            result = a; 1
            getAccount(as, aid) = 3 + length(as) + while(0, n, aid, as) n = as
            while(cnt, n, aid, as) = 4 + nth(as, cnt) + getAid(a)+
            cnt = cnt+1; 1
            if(cnt, n, aid, id) + while(cnt+1, n, aid, as) cnt < n
        }
        while(cnt, n, aid, as) = 0
        cnt = cnt+1; 1
        if(cnt, n, aid, id) = 1
        id = aid
        if(cnt, n, aid, id) = 0
        id ≠ aid
    }
    return result;
}
Basic approach (for sequential models)

Solve the system of cost equations:

\[
\text{getAccount}(as, aid) = 3 + \text{length}(as) + \text{while}(0, n, aid, as) \quad n = as
\]
\[
\text{while}(cnt, n, aid, as) = 4 + \text{nth}(as, cnt) + \text{getAid}(a) +
\]
\[
\text{if}(cnt, n, aid, id) + \text{while}(cnt + 1, n, aid, as) \quad cn < n
\]
\[
\text{while}(cnt, n, aid, as) = 0 \quad cn \geq n
\]
\[
\text{if}(cnt, n, aid, id) = 1 \quad id = aid
\]
\[
\text{if}(cnt, n, aid, id) = 0 \quad id \neq aid
\]
Basic approach (for sequential models)

3. Solve the system of cost equations:

\[
\text{getAccount}(as, aid) = 3 + as + \text{while}(0, n, aid, as) \quad n = as
\]
\[
\text{while}(cnt, n, aid, as) = 4 + cnt + 1+
\]
\[
\text{if}(cnt, n, aid, id) + \text{while}(cnt + 1, n, aid, as) \quad cnt < n
\]
\[
\text{while}(cnt, n, aid, as) = 0 \quad cnt \geq n
\]
\[
\text{if}(cnt, n, aid, id) = 1 \quad id = aid
\]
\[
\text{if}(cnt, n, aid, id) = 0 \quad id \neq aid
\]

Assuming the following costs:

\[
\text{length}(as) = as
\]
\[
\text{nth}(as, cnt) = cnt
\]
\[
\text{getAid}(a) = 1
\]
Basic approach (for sequential models)

3 Solve the system of cost equations:

\[ \text{getAccount}(as, aid) = 3 + as + \text{while}(0, n, aid, as) \]
\[ n = as \]
\[ \text{while}(cnt, n, aid, as) = 4 + cnt + 1 + 1 + \text{while}(cnt + 1, n, aid, as) \]
\[ cnt < n \]
\[ \text{while}(cnt, n, aid, as) = 0 \]
\[ cnt \geq n \]
\[ \text{if}(cnt, n, aid, id) = 1 \]
\[ id = aid \]
\[ \text{if}(cnt, n, aid, id) = 0 \]
\[ id \neq aid \]

- An upper bound of \( \text{if}(cnt, n, aid, id) = 1 \)
Basic approach (for sequential models)

3. Solve the system of cost equations:

\[
\text{getAccount}(\text{as, aid}) = 3 + \text{as} + \text{while}(0, n, \text{aid}, \text{as}) \quad n = \text{as} \\
\text{while}(\text{cnt, n, aid, as}) = 4 + \text{cnt} + 1 + \\
1 + \text{while}(\text{cnt} + 1, n, \text{aid}, \text{as}) \quad \text{cnt} < n \\
\text{while}(\text{cnt, n, aid, as}) = 0 \quad \text{cnt} \geq n \\
\text{if}(\text{cnt, n, aid, id}) = 1 \quad \text{id} = \text{aid} \\
\text{if}(\text{cnt, n, aid, id}) = 0 \quad \text{id} \neq \text{aid}
\]

- An upper bound of \(\text{if}(\text{cnt, n, aid, id}) = 1\)
- The cost of any iteration of while \(6 + \text{cnt}\) is smaller than \(6 + n\)
Basic approach (for sequential models)

3 Solve the system of cost equations:

\[
\text{getAccount}(as, aid) = 3 + as + \text{while}(0, n, aid, as) \quad n = as
\]
\[
\text{while}(cnt, n, aid, as) = 4 + cnt + 1 + 1 + \text{while}(cnt + 1, n, aid, as) \quad cnt < n
\]
\[
\text{while}(cnt, n, aid, as) = 0 \quad cnt \geq n
\]
\[
\text{if}(cnt, n, aid, id) = 1 \quad id = aid
\]
\[
\text{if}(cnt, n, aid, id) = 0 \quad id \neq aid
\]

- An upper bound of if\((cnt, n, aid, id) = 1\)
- The cost of any iteration of while \(6 + cnt\) is smaller than \(6 + n\)
- while can iterate at most \(n\) times
Basic approach (for sequential models)

3. Solve the system of cost equations:

- \( \text{getAccount}(as, aid) = 3 + as + n^2 + 6n \quad n = as \)
- \( \text{while}(cnt, n, aid, as) = 4 + cnt + 1 + 1 + \text{while}(cnt + 1, n, aid, as) \quad cnt < n \)
- \( \text{while}(cnt, n, aid, as) = 0 \quad cnt \geq n \)
- \( \text{if}(cnt, n, aid, id) = 1 \quad id = aid \)
- \( \text{if}(cnt, n, aid, id) = 0 \quad id \neq aid \)

- An upper bound of \( \text{if}(cnt, n, aid, id) = 1 \)
- The cost of any iteration of while \( 6 + cnt \) is smaller than \( 6 + n \)
- \( \text{while} \) can iterate at most \( n \) times
- An upper bound of \( \text{while}(cnt, n, aid, as) = n^2 + 6n \)
Basic approach (for sequential models)

3 Solve the system of cost equations:

\[
\begin{align*}
\text{getAccount}(as, aid) &= 3 + as + n^2 + 6n \\
\text{while}(cnt, n, aid, as) &= 4 + cnt + 1 + \\
1 + \text{while}(cnt + 1, n, aid, as) &\quad \text{cnt} < n \\
\text{while}(cnt, n, aid, as) &= 0 \\
\text{if}(cnt, n, aid, id) &= 1 \\
\text{if}(cnt, n, aid, id) &= 0 \\
\end{align*}
\]

\[n = as\]

- An upper bound of \(\text{if}(cnt, n, aid, id) = 1\)
- The cost of any iteration of while \(6 + cnt\) is smaller than \(6 + n\)
- \textit{while} can iterate at most \(n\) times
- An upper bound of \(\text{while}(cnt, n, aid, as) = n^2 + 6n\)
- An upper bound of \(\text{getAccount}(as, aid) = as^2 + 7as + 3\)
Concurrency increases the degree of non-determinism

In ABS, fields can be modified by other methods during release points.

Consider the following modification of our example:

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as, cnt);
        Fut<Int>idFut = a!getAid();
        await idFut;
        Int id=idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}
```
Concurrency increases the degree of non-determinism
In ABS, fields can be modified by other methods during release points

Consider the following modification of our example:

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as, cnt);
        Fut<Int> idFut = a!getAid();
        await idFut;
        Int id = idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt + 1;
    }
    return result;
}
```

- Direct comparison with the length of `as`
Analyzing Concurrent Models

- Concurrency increases the degree of non-determinism
- In ABS, fields can be modified by other methods during release points

Consider the following modification of our example:

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as,cnt);
        Fut<Int>idFut = a!getAid();
        await idFut;
        Int id=idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}
```

- Direct comparison with the length of as
- Wait for idFut without blocking the object
Concurrency increases the degree of non-determinism
In ABS, fields can be modified by other methods during release points

Consider the following modification of our example:

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as,cnt);
        Fut<Int>idFut = a!getAid();
        await idFut;
        Int id=idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}
```

- Direct comparison with the length of as
- Wait for idFut without blocking the object

Problems
- as could be increased during await by another method
- Then the initial value of as is not a bound on the number of iterations
Use class invariants to capture the required information
Use class invariants to capture the required information

Class invariants

An ABS class invariant is a predicate over the fields of a class that holds at every release point

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as, cnt);
        Fut<Int>idFut = a!getAid();
        await idFut;
        Int id=idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}
```
Use class invariants to capture the required information

Class invariants
An ABS class invariant is a predicate over the fields of a class that holds at every release point

```
[as<=max(as)]
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as,cnt);
        Fut<Int>idFut = a!getAid();
        await idFut; [as<=max(as)]
        Int id=idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}
```

- Annotate the method with 
  
  \[ as<=\text{max}(as) \] at its release points

R. Hähnle (TUD)
Use class invariants to capture the required information

Class invariants

An ABS class invariant is a predicate over the fields of a class that holds at every release point

```java
[as<=max(as)]
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as,cnt);
        Fut<Int>idFut = a!getAid();
        await idFut;  [as<=max(as)]
        Int id=idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}
```

- Annotate the method with `[as<=max(as)]` at its release points
- Now it is guaranteed that the loop does not iterate more than `max(as)` times

Analysis of ESMs

R. Hähnle (TUD)
Use class invariants to capture the required information

Class invariants
An ABS class invariant is a predicate over the fields of a class that holds at every release point.

```java
[as<=max(as)]
Account getAccount(Int aid) {
  Account result = null;
  Int cnt = 0;
  while (cnt < length(as)) {
    Account a = nth(as, cnt);
    Fut<Int>idFut = a!getAid();
    await idFut;
    Int id = idFut.get;
    if (aid == id) {
      result = a;
    }
    cnt = cnt + 1;
  }
  return result;
}
```

- Annotate the method with \([as\leq max(as)]\) at its release points.

But have to prove that the invariant holds!
Rely-Guarantee Reasoning (for Termination)

Adapt a rely-guarantee approach for proving termination as follows:

- Given a loop, assume its fields are not modified during release points
- Attempt to prove termination of the loop

Observation

If a loop terminates when its fields are not modified, it also terminates if the fields are modified a finite number of times

- Prove that the instructions that modify the fields involved in the termination proof (in between release points) are modified a finite number of times
- To do so, use same procedure in the loops that contain such instructions
- We fail if we find a circular dependency

A similar approach can be used to obtain upper bounds
Rely-Guarantee Reasoning: Example

To apply this procedure, we need a complete program—add a main method that:

```java
{
    Account a;
    DB db = new cog DBImpl();
    Int max = 10;
    Int i = 1;
    while(i <= max){
        a = new cog AccountImpl(i,0);
        Fut<Unit> aFut =
            db!insertAccount(a);
        await aFut?;
        i = i+1;
    }
    db!getAccount(3);
}
```
To apply this procedure, we need a complete program—
add a main method that:

- creates a database (DBImpl)

```java
{  
    Account a;
    DB db = new cog DBImpl();
    Int max = 10;
    Int i = 1;
    while (i <= max) {  
        a = new cog AccountImpl(i, 0);
        Fut<Unit> aFut =
            db!insertAccount(a);
        await aFut?;
        i = i + 1;
    }
    db!getAccount(3);
}
```
To apply this procedure, we need a complete program—
add a main method that:

- creates a database (`DBImpl`)
- adds 10 new accounts to the database (sequentially)

```java
{  
    Account a;
    DB db = new cog DBImpl();
    Int max = 10;
    Int i = 1;
    while (i <= max) {
        a = new cog AccountImpl(i, 0);
        Fut<Unit> aFut =
            db!insertAccount(a);
        await aFut?;
        i = i+1;
    }
    db!getAccount(3);
}
```
Rely-Guarantee Reasoning: Example

To apply this procedure, we need a complete program—add a main method that:

- creates a database (`DBImpl`)
- adds 10 new accounts to the database (sequentially)
- calls method `getAccount`

```java
{  
    Account a;
    DB db = new cog DBImpl();
    Int max = 10;
    Int i = 1;
    while (i <= max) {
        a = new cog AccountImpl(i, 0);
        Fut<Unit> aFut =
            db!insertAccount(a);
        await aFut?;
        i = i+1;
    }
    db!getAccount(3);
}
```
To apply this procedure, we need a complete program—add a main method that:

- creates a database (DBImpl)
- adds 10 new accounts to the database (sequentially)
- calls method `getAccount`

Assume `insertAccount` modifies the field as of the database as expected
Method applied to `getAccount`:

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as, cnt);
        Fut<Int> idFut = a!getAid();
        await idFut;
        Int id = idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt + 1;
    }
    return result;
}
```
Method applied to to `getAccount`:

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as, cnt);
        Fut<Int> idFut = a!getAid();
        await idFut;
        Int id = idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}
```

1. assume `as` is unmodified during `await`

2. obtain that `as` is an upper bound on the number of iterations

3. examine the program points where `as` can be modified

4. all instances of `insertAccount` must have finished when we execute `getAccount`

5. therefore, the upper bound is correct and we are done!
**Rely-Guarantee Reasoning: Example**

Method applied to to `getAccount`:

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as,cnt);
        Fut<Int>idFut = a!getAid();
        await idFut;
        Int id=idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt+1;
    }
    return result;
}
```

1. assume as is unmodified during await
2. obtain that as is an upper bound on the number of iterations
Method applied to to `getAccount`:

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as, cnt);
        Fut<Int> idFut = a!getAid();
        await idFut;
        Int id = idFut.get;
        if (aid == id) {
            result = a;
        }
        cnt = cnt + 1;
    }
    return result;
}
```

1. Assume `as` is unmodified during `await`.
2. Obtain that `as` is an upper bound on the number of iterations.
3. Examine the program points where `as` can be modified.
Method applied to `getAccount`:

```java
{  
  Account a;
  DB db = new cog DBImpl();
  Int max = 10;
  Int i = 1;
  while (i <= max) {  
    a = new cog AccountImpl(i, 0);
    Fut<Unit> aFut =
      db!insertAccount(a);
    await aFut?;
    i = i+1;
  }
  db!getAccount(3);
}
```

1. assume `as` is unmodified during `await`  
2. obtain that `as` is an upper bound on the number of iterations  
3. examine the program points where `as` can be modified  
4. all instances of `insertAccount` must have finished when we execute `getAccount`
Method applied to `getAccount`:

```java
Account getAccount(Int aid) {
    Account result = null;
    Int cnt = 0;
    while (cnt < length(as)) {
        Account a = nth(as, cnt);
        Fut<Int> idFut = a!getAid();
        await idFut;
        Int id = idFut.get;
        if (aid == id) {
            result = a;
            cnt = cnt + 1;
        }
    }
    return result;
}
```

1. *assume as is unmodified during await*
2. *obtain that as is an upper bound on the number of iterations*
3. *examine the program points where as can be modified*
4. *all instances of `insertAccount` must have finished when we execute `getAccount`*
5. *therefore, the upper bound is correct and we are done!*

```java
{ Account a;
  DB db = new cog DBImpl();
  Int max = 10;
  Int i = 1;
  while (i <= max) {
    a = new cog AccountImpl(i, 0);
    Fut<Unit> aFut =
      db!insertAccount(a);
    await aFut?;
    i = i + 1;
  }
  db!getAccount(3);
}
```
Consider the following modification:

```java
{ 
    Account a;
    DB db = new cog DBImpl();
    Int max = 10;
    Int i = 1;
    while(i <= max){
        a = new cog AccountImpl(i,0);
        Fut<Unit> aFut =
            db!insertAccount(a);
        await aFut?;
        i = i+1;
    }
    db!getAccount(3);
}
```
Consider the following modification:

```
{ 
  Account a;
  DB db = new cog DBImpl();
  Int max = 10;
  Int i = 1;
  while (i <= max) {
    a = new cog AccountImpl(i, 0);
    Fut<Unit> aFut =
      db!insertAccount(a);
      // without the await, some instances of insertAccount might execute in parallel with getAccount
    aFut?;
    i = i+1;
  }
  db!getAccount(3);
}
```
Consider the following modification:

```java
{  
    Account a;
    DB db = new cog DBImpl();
    Int max = 10;
    Int i = 1;
    while(i <= max){
        a = new cog AccountImpl(i,0);
        Fut<Unit> aFut =
            db!insertAccount(a);
        //await aFut%n;
        i = i+1;
    }
    db!getAccount(3);
}
```

4. without the await, some instances of `insertAccount` might execute in parallel with `getAccount`

5. we need to prove that the number of instances of `insertAccount` is finite
Consider the following modification:

```java
{
    Account a;
    DB db = new cog DBImpl();
    Int max = 10;
    Int i = 1;
    while (i <= max) {
        a = new cog AccountImpl(i, 0);
        Fut<Unit> aFut =
            db!insertAccount(a);
        await aFut?;
        i = i+1;
    }
    db!getAccount(3);
}
```

1. without the await, some instances of `insertAccount` might execute in parallel with `getAccount`
2. we need to prove that the number of instances of `insertAccount` is finite
3. apply rely-guarantee procedure to the while loop in the main block
Consider the following modification:

```java
{  
    Account a;
    DB db = new cog DBImpl();
    Int max = 10;
    Int i = 1;
    while(i <= max){
        a = new cog AccountImpl(i,0);
        Fut<Unit> aFut =
            db!insertAccount(a);
        //await aFut?;
        i = i+1;
    }
    db!getAccount(3);
}
```

4. without the await, some instances of `insertAccount` might execute in parallel with `getAccount`

5. we need to prove that the number of instances of `insertAccount` is finite

6. apply rely-guarantee procedure to the while loop in the main block

7. that loop terminates without any additional assumptions (it has at most 10 iterations)
Consider the following modification:

```java
{  
    Account a;
    DB db = new cog DBImpl();
    Int max = 10;
    Int i = 1;
    while(i <= max){
        a = new cog AccountImpl(i,0);
        Fut<Unit> aFut =
            db!insertAccount(a);
        //await aFut?;
        i = i+1;
    }
    db!getAccount(3);
}
```

4. without the await, some instances of `insertAccount` might execute in parallel with `getAccount`

5. we need to prove that the number of instances of `insertAccount` is finite

6. apply rely-guarantee procedure to the while loop in the main block

7. that loop terminates without any additional assumptions (it has at most 10 iterations)

8. we are done!
Part III

Deadlock Analysis of ABS Models
The ABS concurrency model excludes race conditions, but not deadlocks

**Definition (Deadlock)**

A **deadlock situation** is a state of a concurrent model in which one or more tasks are waiting for each others’ termination and none of them can make any progress.

In ABS, the main entities involved in deadlocks are:
- COGs, that represent the units of concurrency (processors)
- Method executions (a.k.a. tasks)
- Synchronization statements (`await g` and `get`)

R. Hähnle (TUD)
ABS Deadlock Example

```java
main
{
    a=new cog A();
    b=new cog B();
    c=new cog C();
    b!blk_c(c,a);
    a!blk_b(b);
}
```

Dependencies:
\{a=new cog A(); b=new cog B(); c=new cog C(); b!blk_c(c,a); a!blk_b(b);\}

Deadlock!

R. Hähnle (TUD)
ABS Deadlock Example

main
{
    a=new cog A();
    b=new cog B();
    c=new cog C();
    b!blk_c(c,a);
    a!blk_b(b);
}

B

A

C

Dependencies:
\{a=new cog A(); b=new cog B(); c=new cog C(); b!blk_c(c,a); a!blk_b(b);\}

Deadlock!

R. Hähnle (TUD) Analysis of ESMs 31 / 54
ABS Deadlock Example

```
main
{
    a=new cog A();
    b=new cog B();
    c=new cog C();
    b!blk_c(c,a);
    a!blk_b(b);
}

class B{
    blk_c(C c, A a){
        f=c!wait(a);
        f.get;
    }
    empt2(){}
}

class A{
    blk_b(B b){
        f=b!empt2();
        f.get;
    }
    empt(){}
}

class C{
    wait(A a){
        f=a!empt();
        await f?;
    }
}
```

Deadlock!

Dependencies:

\{ 
  a=new cog A();
  b=new cog B();
  c=new cog C();
  b!blk_c(c,a);
  a!blk_b(b);
\}
ABS Deadlock Example

main
{
    a=new cog A();
    b=new cog B();
    c=new cog C();
    b!blk_c(c,a);
    a!blk_b(b);
}

class B{
    blk_c(C c, A a){
        f=c!wait(a);
        f.get;
    }
    empt2(){}
}

class A{
    blk_b(B b){
        f=b!empt2();
        f.get;
    }
    empt(){}
}

class C{
    wait(A a){
        f=a!empt();
        await f?;
    }
}
ABS Deadlock Example

```
main
{
    a=new cog A();
    b=new cog B();
    c=new cog C();
    b!blk_c(c,a);
    a!blk_b(b);
}
```

```
class B{
    blk_c(C c, A a){
        f=c!wait(a);
        f.get;
    }
    empt2(){
    }
}
```

```
class A{
    blk_b(B b){
        f=b!empt2();
        f.get;
    }
    empt(){
    }
}
```

```
class C{
    wait(A a){
        f=a!empt();
        await f?;
    }
}
```

Dependencies:
{a=new cog A(); b=new cog B(); c=new cog C(); b!blk_c(c,a); a!blk_b(b);}

Deadlock!
ABS Deadlock Example

main
{
    a=new cog A();
    b=new cog B();
    c=new cog C();
    b!blk_c(c,a);
    a!blk_b(b);
}

class A{
    blk_b(B b){
        f=b!empt2();
        f.get;
    }
    empt(){}
}

class B{
    blk_c(C c,A a){
        f=c!wait(a);
        f.get;
    }
    empt2(){}
}

class C{
    wait(A a){
        f=a!empt();
        await f?;
    }
}

Dependencies:

1. $B \rightarrow C.wait$
ABS Deadlock Example

Dependencies:
1. $B \rightarrow C.wait$
2. $A \rightarrow Bempt2$
3. $Bempt2 \rightarrow B$
   (wait for $B$'s lock)

```java
class A{
    blk_b(B b){
        f=b!empt2();
        f.get;
    }
    empt(){}
}
class B{
    blk_c(C c,A a){
        f=c!wait(a);
        f.get;
    }
    empt2(){}
}
class C{
    wait(A a){
        f=a!empt();
        await f?;
    }
}
main
{
    a=new cog A();
    b=new cog B();
    c=new cog C();
    b!blk_c(c,a);
    a!blk_b(b);
}
```
ABS Deadlock Example

Dependencies:
1. $B \rightarrow C.wait$
2. $A \rightarrow B.empt2$
3. $B.empt2 \rightarrow B$ (wait for $B$'s lock)
ABS Deadlock Example

Dependencies:

1. $B \rightarrow C . wait$
2. $A \rightarrow B . empt2$
3. $B . empt2 \rightarrow B$
   (wait for $B$'s lock)
4. $C . wait \rightarrow A . empt$
5. $A . empt \rightarrow A$
   (wait for $A$'s lock)
ABS Deadlock Example

Dependencies:
1. \( B \rightarrow C.wait \)
2. \( A \rightarrow B.\text{empt}\_2 \)
3. \( B.\text{empt}\_2 \rightarrow B \) (wait for \( B \)'s lock)
4. \( C.wait \rightarrow A.\text{empt} \)
5. \( A.\text{empt} \rightarrow A \) (wait for \( A \)'s lock)

Deadlock!
Idea for Deadlock Analysis

Approximate statically “dependencies” that occur at runtime

Dependencies

Dependencies are created by tasks and COGs that wait for each other at synchronization points (await g or get)
Idea for Deadlock Analysis

Approximate statically “dependencies” that occur at runtime

Dependencies

Dependencies are created by tasks and COGs that wait for each other at synchronization points (await g or get)

Which tasks/COGs wait for which other tasks/COGs?
Idea for Deadlock Analysis

Approximate statically “dependencies” that occur at runtime

Dependencies

Dependencies are created by tasks and COGs that wait for each other at synchronization points (\texttt{await g} or \texttt{get})

Which tasks/COGs wait for which other tasks/COGs?

This is our plan:

- Obtain a \textit{finite} representation (approximation) of all possible objects, COGs and tasks that can be created
- Analyse the future variables dependencies at synchronization points
- Approximate this information through a points-to analysis
Abstract Dependencies

Definition (Kinds of dependencies in an abstract representation of ABS)

**COG-task** dependencies, when a task waits for the termination of another task but keeps the COG’s lock (using `get`)

**task-task** dependencies, when a task waits for the termination of another task with a non-blocking operation (`await g`) or with a blocking operation (using `get`)

**task-COG** dependencies between a task and the COG it belongs to
Abstract Dependencies

Definition (Kinds of dependencies in an abstract representation of ABS)

**COG-task** dependencies, when a task waits for the termination of another task but keeps the COG’s lock (using `get`).

**task-task** dependencies, when a task waits for the termination of another task with a non-blocking operation (`await g`) or with a blocking operation (using `get`).

**task-COG** dependencies between a task and the COG it belongs to.

Cycles within dependencies can indicate deadlock situations.
Building the Abstract Dependency Graph

Abstract COGs:
- main, A, B, and C

Synchronization instructions:
- f.get in B
- f.get in A
- await f? in C

main

```java
{  
a=new cog A();  
b=new cog B();  
c=new cog C();  
b!blk_c(c,a);  
a!blk_b(b);
}
```

class B{
    blk_c(C c, A a){
        f=c!wait(a);
        f.get;
    }
    empt2(){}
}

class C{
    wait(A a){
        f=a!empt();
        await f?;
    }
    }

class A{
    blk_b(B b){
        f=b!empt2();
        f.get;
    }
    empt(){}
}

R. Hähnle (TUD)
Building the Abstract Dependency Graph

Abstract COGs:
- `main`, `A`, `B`, and `C`

Synchronization instructions:
- `f.get` in `B`
- `f.get` in `A`
- `await f?` in `C`
Building the Abstract Dependency Graph

Abstract COGs:
- main, A, B, and C

Synchronization instructions:
- f.get in B
- f.get in A
- await f? in C
Building the Abstract Dependency Graph

Abstract COGs:
- main, A, B, and C

Synchronization instructions:
- f.get in B
- f.get in A
- await f? in C
Building the Abstract Dependency Graph

Abstract COGs: ▶ main, A, B, and C

Synchronization instructions:
▶ f.get in B
▶ f.get in A
▶ await f? in C
Building the Abstract Dependency Graph

Abstract COGs:
- main, A, B, and C

Synchronization instructions:
- f.get in B
- f.get in A
- await f? in C
Abstraction Is Too Coarse

- Dependency cycle remains in abstract graph
- But program is deadlock-free
- Not all dependencies can occur simultaneously

Add synchronization

```java
main
{
    a=new A();
b=new B();
c=new C();
f=b!blk_c(c,a);
await f?;
a!blk_b(b);
}
```

```java
class B{
    blk_c(C c, A a){
        f=c!wait(a);
        f.get;
    }
    empt2(){}
}
```

```java
class A{
    blk_b(B b){
        f=b!empt2();
        f.get;
    }
    empt(){}
}
```

```java
class C{
    wait(A a){
        f=a!empt();
        await f?;
    }
}
```
Problem Analysis

- Dependencies abstract away from all possible synchronizations.
- A cycle only represents a real deadlock risk when synchronizations may occur simultaneously.
- Annotate each dependency with the program point that causes it.
- A may-happen-in-parallel (MHP) analysis tells whether two program points can be executed in parallel.
Problem Analysis

- Dependencies abstract away from all possible synchronizations
- A cycle only represents a real deadlock risk when synchronizations may occur *simultaneously*
- Annotate each dependency with the program point that causes it
- A may-happen-in-parallel (MHP) analysis tells whether two program points can be executed in parallel

Definition (Feasible Cycle)

A cycle (in the abstract dependency graph) is **feasible** if all program points in the edge annotations can happen in parallel.
Applying MHP to Modified Example

```java
main
{
    a=new A();
    b=new B();
    c=new C();
    f=b!blk_c(c,a);
    await f?;
    a!blk_b(b);
}

class A{
    blk_b(B b){
        f=b!empt2();
        f.get;
    }
    empt(){}
}

class B{
    blk_c(C c, A a){
        f=c!wait(a);
        f.get;
    }
    empt2(){}
}

class C{
    wait(A a){
        f=a!empt();
        await f?;
    }
}

Result of MHP:
- await f? in main enforces completion of blk_c in B
- f.get in A \(\parallel\) f.get in B
```
Applying MHP to Annotated Dependency Graph

The cycle is unfeasible.

f.get in A \parallel f.get in B
Part IV

Deductive Verification Of ABS Models
Deductive verification of ABS

Functional verification of complex (first-order) properties

Uses ideas by
W. Ahrendt, M. Dylla, and C. Din et al.

Deductive Functional Verification

Based on a program logic for ABS that . . .

▶ Uses verification paradigm “logic rewriting as symbolic execution”: test generation, visualization, symbolic state debugging, . . .

▶ Is suitable for open-world verification hide internal structures of components from each other
Verification Workflow

```
<<interface>>
Account

Int deposit(Int x)
Int withdraw(Int x)
Bool transfer(...)
```

```
AccountImpl

Int aid;
Int balance;
...
```
Interface invariants express mostly restrictions on the control-flow
Class invariants relate the object state to the local system history
Verification Workflow

Interface Invariant

```
Account

Int deposit(Int x)
Int withdraw(Int x)
Bool transfer(...)
```

AccountImpl

```
Int aid;
Int balance;
...
```

Class Invariant

Proof-Obligation Generator

R. Hähnle (TUD)
Analysis of ESMs
Verification Workflow

Interface Invariant

```
interface Account

Int deposit(Int x)
Int withdraw(Int x)
Bool transfer(...)
```

AccountImpl

```
Int aid;
Int balance;
...
```

Class Invariant

Proof-Obligation Generator

ABS DL Formula

Verifier

X ✓
Dynamic Logic for ABS

Sorted first-order logic + ABS programs + modalities + updates

- $\langle p \rangle \phi$: Program $p$ terminates and in its final state $\phi$ holds.
- $[p] \phi$: If program $p$ terminates then in its final state $\phi$ holds.

We consider only partial correctness (box modality)
ABS-DL: Dynamic Logic

Dynamic Logic for ABS
Sorted first-order logic + ABS programs + modalities + updates

- $⟨p⟩φ$: Program $p$ terminates and in its final state $φ$ holds.
- $[p]φ$: If program $p$ terminates then in its final state $φ$ holds.

We consider only partial correctness (box modality)

Core Concepts

- Sequent calculus based on symbolic execution paradigm
- Assumption-commitment/rely-guarantee reasoning style
interface Account { ... }
class AccountImpl implements Account { ... }

data List = Cons(Int, List) | Nil
def List tail(List l) = case l {
  Cons(x, l) => l;
  Nil => Nil;
}

Sorts S
For each
  ▶ interface I,
  ▶ abstract datatype T (including built-in ADTs)
there are sorts I, T ∈ S
ABS-DL: Logic Setup

```
interface Account { ... }
class AccountImpl implements Account { ... }

data List = Cons(Int, List) | Nil
def List tail(List l) = case l {
    Cons(x, l) => l;
    Nil => Nil;
}
```

**Functions**

For each constructor, function of an ADT there is a rigid function symbol of same name and arity in ABS-DL

Calculus contains (automatically generated) rules for:
- different constructors denote different entities
- rewrite rules for each function definition
Gentzen-Style Sequent Calculus: $\Gamma \Rightarrow \Delta$

$$\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigvee_{\delta \in \Delta} \delta$$
Gentzen-Style Sequent Calculus: $\Gamma \Rightarrow \Delta$

$$\bigwedge_{\gamma \in \Gamma} \gamma \to \bigvee_{\delta \in \Delta} \delta$$

Propositional Rules (classical logic)

- **andLeft**
  \[
  \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \land B \Rightarrow \Delta}
  \]

- **andRight**
  \[
  \frac{\Gamma \Rightarrow A, \Delta \quad \Gamma \Rightarrow B, \Delta}{\Gamma \Rightarrow A \land B, \Delta}
  \]

- **impRight**
  \[
  \frac{\Gamma, A \Rightarrow B, \Delta}{\Gamma \Rightarrow A \to B, \Delta}
  \]
Symbolic Representation of State Transitions
Generalization of explicit substitutions

**Elementary update** (val has no side effects)

\[ \text{loc} := \text{val} \]

Same meaning as an assignment
Symbolic Representation of State Transitions
Generalization of explicit substitutions

**Elementary update** (val has no side effects)

\[
\text{loc} := \text{val}
\]

Same meaning as an assignment

**Parallel update**

\[
\text{loc}_1 := \text{val}_1 \parallel \text{loc}_2 := \text{val}_2
\]
Symbolic Representation of State Transitions
Generalization of explicit substitutions

**Elementary update** (val has no side effects)

\[ \text{loc} := \text{val} \]

Same meaning as an assignment

**Parallel update**

\[ \text{loc}_1 := \text{val}_1 \parallel \text{loc}_2 := \text{val}_2 \]

**Update application**

- on term: \( \{U\} t \)
- on formula: \( \{U\} \phi \)
Symbolic Representation of State Transitions
Generalization of explicit substitutions

Elementary update (val has no side effects)
\[ \text{loc} := \text{val} \]

Same meaning as an assignment

Parallel update
\[ \text{loc}_1 := \text{val}_1 \parallel \text{loc}_2 := \text{val}_2 \]

Update application

- on term: \( \{U\} t \)
- on formula: \( \{U\} \phi \)

Example (Update simplification)
\[ \{i := j \parallel j := i\}(i = i_{old}) \sim \]

R. Hähnle (TUD)
Symbolic Representation of State Transitions
Generalization of explicit substitutions

Elementary update (\(\text{val}\) has no side effects)

\[
\text{loc} := \text{val}
\]

Same meaning as an assignment

Parallel update

\[
\text{loc}_1 := \text{val}_1 \parallel \text{loc}_2 := \text{val}_2
\]

Update application

on term: \(\{U\} t\)  on formula: \(\{U\} \phi\)

Example (Update simplification)

\[
\{i := j\parallel j := i\}(i = i_{old}) \leadsto \\
(\{i := j\parallel j := i\}i) = (\{i := j\parallel j := i\}i_{old}) \leadsto
\]
Symbolic Representation of State Transitions
Generalization of explicit substitutions

Elementary update (val has no side effects)

\[ \text{loc} := \text{val} \]

Same meaning as an assignment

Parallel update

\[ \text{loc}_1 := \text{val}_1 \parallel \text{loc}_2 := \text{val}_2 \]

Update application

on term: \( \{U\} t \) on formula: \( \{U\} \phi \)

Example (Update simplification)

\[ \{i := j \parallel j := i\}(i = i_{old}) \rightsquigarrow \]
\[ (\{i := j \parallel j := i\}i) = (\{i := j \parallel j := i\}i_{old}) \rightsquigarrow j = i_{old} \]
ABS-DL: Support for Imperative ABS

Assignment Rule

\[
\Gamma \Rightarrow \{U\}\{x := e\} [\omega] \phi, \Delta \\
\Gamma \Rightarrow \{U\}[x=e; \omega] \phi, \Delta
\]

e side effect free (pure) expression
**ABS-DL: Support for Imperative ABS**

### Assignment Rule

\[
\frac{\Gamma \Rightarrow \{U\}\{x := e\} [\omega]\phi, \Delta}{\Gamma \Rightarrow \{U\}[x=e; \omega]\phi, \Delta}
\]

- \(e\) side effect free (pure) expression

### Conditional Rule

\[
\frac{\Gamma, \{U\} e \Rightarrow \{U\}[p; \omega]\phi, \Delta \quad \Gamma, \{U\} \neg e \Rightarrow \{U\}[q; \omega]\phi, \Delta}{\Gamma \Rightarrow \{U\}[\text{if } (e) \{p\} \text{ else } \{q\} \omega]\phi, \Delta}
\]

- \(e\) pure expression
- Rule splits proof in a **then** and **else** branch
Explicit Heap Model

- Global program variable: *heap*
- Axiomatized using theory of arrays:

\[
\text{select(store}(heap, this, field, 5), this, field)\]

\[
\text{this.field = 5;}
\]

Advantage

Easy to assign all fields an unknown value
- anonymization of field values at control release
Verification of distributed systems achieved by **sequential** means
Verification of distributed systems achieved by \textit{sequential} means

\[ \rightarrow \text{Event Histories: Sequences of Messages [Din et al.]} \]

\begin{itemize}
  \item bank:Bank
  \item acc:AccountImpl
\end{itemize}
Verification of distributed systems achieved by **sequential** means

⇒ Event ** Histories: Sequences of Messages** [Din et al.]

![Diagram showing invocation and completion events](image)

**History Event:** \( \text{inEv} \text{ (bank, acc, fut, withdraw, (amount))} \)

- fut: newly created future for asynchronous message
- (⋯): list of arguments
Verification of distributed systems achieved by **sequential** means

⇒ Event **Histories**: Sequences of Messages [Din et al.]

```
Invocation Event (inEv)
```

```
bank:Bank
```

```
 Invocation Reaction Event (inREv)
 withdraw(amount)
```

```
acc:AccountImpl
```

```
Invocation Reaction Event (inREv)
```

**History Event**: **inREv** (bank, acc, fut, withdraw, *(amount)*)

- event created upon execution of method
- fut: future to be resolved upon completion
Verification of distributed systems achieved by **sequential** means

⇒ Event **Histories**: Sequences of Messages [Din et al.]

---

Invocation Event (inEv)

\[ \text{bank:Bank} \]

Invocation Reaction Event (inREv)

\[ \text{withdraw(amount)} \]

Completion Event (compEv)

\[ \text{return} \]

Completion Reaction Event (compREv)

History Event: \textbf{compEv}(\text{acc, fut, withdraw, r})

- event created upon completion of method
- \( r \): return value of method for future fut

---

R. Hähnle (TUD)
Analysis of ESMs
48 / 54
Verification of distributed systems achieved by \textit{sequential} means

⇒ Event \textbf{Histories:} Sequences of Messages [Din et al.]

\begin{itemize}
  \item \textbf{Invocation Event (inEv)}
  \item \textbf{Invocation Reaction (inREv)}
  \item \textbf{Completion Event (compEv)}
  \item \textbf{Completion Reaction Event (compREv)}
\end{itemize}

\begin{center}
\begin{tikzpicture}
  \node (bank) at (0,0) {bank:Bank};
  \node (acc) at (4,0) {acc:AccountImpl};
  \draw[->] (bank) -- node[above] {$\text{withdraw(amount)}$} (acc);

  \node (inEv) at (-2,-2) {Invocation Event (inEv)};
  \node (inREv) at (2,-2) {Invocation Reaction Event (inREv)};
  \node (compEv) at (-2,2) {Completion Event (compEv)};
  \node (compREv) at (2,2) {Completion Reaction Event (compREv)};

  \draw[->] (bank) -- (inEv);
  \draw[->] (acc) -- (inREv);
  \draw[->] (compEv) -- (compREv);
\end{tikzpicture}
\end{center}

\textbf{History Event:} \texttt{compREv}(bank, fut, r)

\begin{itemize}
  \item event created upon resolving fut
\end{itemize}
ABS Dynamic Logic (DL): History Events

<table>
<thead>
<tr>
<th>bank:Bank</th>
<th>Invocation Event (inEv)</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc:AccountImpl</td>
<td>Invocation Reaction Event (inREv)</td>
</tr>
</tbody>
</table>

Invocation Event: `withdraw(amount)`

Completion Reaction Event (compREv): `return`

History Events: Observations
- Time delay between events and reaction events
- Implicit restrictions on event order
- Need to specify properties of event histories (complex quantifiers)

Many subgoals during verification are concerned with the event order.
Messages: Structured Labels/Events

\[ \text{inEv}(\text{caller}, \text{callee}, \text{future}, \text{method}, \overline{\text{args}}) \]
\[ \text{inREv}(\text{caller}, \text{callee}, \text{future}, \text{method}, \overline{\text{args}}) \]
\[ \text{compEv}(\text{callee}, \text{future}, \text{method}, \text{return value}) \]
\[ \text{compREv}(\text{receiver}, \text{future}, \text{return value}) \]
ABS-DL: Concurrent Constructs

Messages: Structured Labels/Events

\[
\begin{align*}
\text{inEv}(\text{caller}, \text{callee}, \text{future}, \text{method}, \text{args}) \\
\text{inREv}(\text{caller}, \text{callee}, \text{future}, \text{method}, \text{args}) \\
\text{compEv}(\text{callee}, \text{future}, \text{method}, \text{return value}) \\
\text{compREv}(\text{receiver}, \text{future}, \text{return value})
\end{align*}
\]

History

Represented as program variable of type History

- Standard sequent axiomatisation
- Specification predicates and functions:
  - \( \text{wfHist}, \text{isCompletionEv}, \text{isInvocationEv} \)
  - \( \text{getResultFrom(label)}, \text{etc.} \)
ABS-DL: Concurrent Constructs

Messages: Structured Labels/Events

\[ \text{inEv}(\text{caller}, \text{callee}, \text{future}, \text{method}, \overline{\text{args}}) \]
\[ \text{inREv}(\text{caller}, \text{callee}, \text{future}, \text{method}, \overline{\text{args}}) \]
\[ \text{compEv}(\text{callee}, \text{future}, \text{method}, \text{return value}) \]
\[ \text{compREv}(\text{receiver}, \text{future}, \text{return value}) \]

Wellformedness predicate \( wfHist \)

- invocation event < invocation reaction event <
  < completion event < completion reaction event
- For each reaction event there must be a precursor event
- Only one of each such events for a specific invocation
- Futures are unique and not "reused"
Example: Invariants of class Account

We can now express:

**Balance of class Account always non-negative**

```
\textbf{invariants}(\text{Seq theHistory, Heap theHeap, ABSAnyInterface self}) \{ 
  \text{nonNegativeBalance : AccountImpl} \{ 
    \text{int::select(theHeap, self, balance) }\geq 0 
  \}; } 
```

For the above invariant to be preserved, we need also:

```
\textbf{invariants}(\text{Seq theHistory, Heap theHeap, ABSAnyInterface self}) \{ 
  \text{amountOfDepositNonNegative : AccountImpl} \{ 
    \forall \text{Event ev}; 
    \forall \text{int i; ( i }\geq 0 \& i < \text{seqLen(theHistory)} - 1 \& ( \text{ev = Event::seqGet(theHistory, i) \& (isInvocationEv(ev) \lor isInvocationREv(ev)) \& (getMethod(ev) = Account::deposit \& int::seqGet(getArguments(ev), 0) }\geq 0 ) )} 
  \}; } 
```
Example: Invariants of class Account

We can now express:

Balance of class Account always non-negative

\[
\textbf{invariants}(\text{Seq theHistory, Heap theHeap, ABSAnyInterface self}) \{ \\
\quad \text{nonNegativeBalance : AccountImpl} \{ \\
\quad\quad \text{int::select(theHeap, self, balance) } \geq 0 \\
\quad\}; \\
\}
\]

For the above invariant to be preserved, we need also:

Method deposit(Int) is always invoked with non-negative argument

\[
\textbf{invariants}(\text{Seq theHistory, Heap theHeap, ABSAnyInterface self}) \{ \\
\quad \text{amountOfDepositNonNegative : AccountImpl} \{ \\
\quad\quad \textbf{forall} \text{ Event ev; } \textbf{forall} \text{ int i; } ( i \geq 0 \& i < \text{seqLen}(\text{theHistory}) \rightarrow \\
\quad\quad\quad ( \text{ ev = Event::seqGet(\text{theHistory, i}) \& } \\
\quad\quad\quad\text{ isInvocEv(ev) } \mid \text{isInvocREv(ev)) } \& \\
\quad\quad\quad \text{getMethod(ev) = Account::deposit } \rightarrow \\
\quad\quad\quad\quad \text{int::seqGet(getArguments(ev), 0) } \geq 0 \\
\quad\}; \\
\}
\]
Asynchronous Method Invocation

Asynchronous Method Call

\[ \Gamma \Rightarrow \{ \mathcal{U} \} (o \neq \text{null} \land \text{wfHist}(\mathcal{H})), \Delta \]
\[ \Gamma \Rightarrow \{ \mathcal{U} \} (\text{futureUnused}(\text{frc}, \mathcal{H}) \rightarrow \\
\{ \text{fr} := \text{frc} \mid \mathcal{H} := \mathcal{H} \circ \text{inEv}(\text{this}, o, \text{frc}, m, \overline{\text{args}}) \}(R_I(\mathcal{H}, o) \land [\omega] \phi)), \Delta \]
\[ \Gamma \Rightarrow \{ \mathcal{U} \}[r = o!m(\overline{\text{args}}); \omega] \phi, \Delta \]

Rule has two premisses . . .

- First premiss: callee is not null and history is wellformed
- Second premiss (continuation of symbolic execution):
  - new future created as part of invocation event
  - invocation event appended to history
  - upon asynchronous call all interface invariants (and preconditions of method) are established
  - postcondition \( \phi \) holds after execution of remaining program
Asynchronous Method Invocation

Awaiting Completion

\[
\begin{align*}
\Gamma & \Rightarrow CInv(C)(heap, \mathcal{H}, this), \Delta \\
\Gamma & \Rightarrow \{ \text{heap := newHeap } \mid \mathcal{H} := \mathcal{H} \circ A_{\mathcal{H}} \circ \text{compREv}(\ldots) \}
(CInv(C)(heap, \mathcal{H}, this) \land E_I(\mathcal{H}, this) \land \text{wfHist}(\mathcal{H}) \rightarrow [\omega]\phi), \Delta \\
\Gamma & \Rightarrow [\text{await } r?; \omega]\phi, \Delta
\end{align*}
\]
Asynchronous Method Invocation

Awaiting Completion

\[ \Gamma \Rightarrow \text{ClInv}(C)(\text{heap}, \mathcal{H}, \text{this}), \Delta \]
\[ \Gamma \Rightarrow \{ \text{heap} := \text{newHeap} \parallel \mathcal{H} := \mathcal{H} \circ \mathcal{A}_\mathcal{H} \circ \text{compREv}(\ldots) \} \]
\[ (\text{ClInv}(C)(\text{heap}, \mathcal{H}, \text{this}) \land E_I(\mathcal{H}, \text{this}) \land \text{wfHist}(\mathcal{H}) \rightarrow \lbrack \omega \rbrack \phi), \Delta \]
\[ \Gamma \Rightarrow [\text{await } r?; \omega] \phi, \Delta \]

Rule has two premisses . . .

- First premiss: class invariant is established (await releases control)
- Second premiss (continuation of symbolic execution):
  - heap is anonymised
  - history is extended by an unspecified sequence of events (\( \mathcal{A}_\mathcal{H} \)) followed by the completion reaction event
  - \( E_I(\ldots) \): assume class invariant & interface invariants (+ postcondition of method)
Tool support: Current Status

- Verification tool based on KeY
- Maturity: Late alpha stage
### Summary

#### The ABS Language
- Executable modeling language with formal semantics
- Suitable for modeling distributed and concurrent systems
- Cooperative multitasking + asynchronous calls with futures

#### Resource Analysis
- Abstract representation of programs: cost equations
- Rely-guarantee reasoning to prove termination of concurrent code

#### Deadlock Analysis
- Abstract dependency graph captures potential deadlocks
- MHP-analysis improves precision

#### Deductive Verification
- Histories to achieve sequential style (compositional) reasoning
- No explicit modeling of process queues or scheduling needed