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

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

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

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

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

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

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

Parametric Data Types

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

User Defined Algebraic Data Types

User-Defined Data Types

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

Parametric Data Types

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

Selectors

data Person = Person(String name, Int age, String address);
// *if selector names present, they implicitly define selector functions*
def String name(Person p) = ... ;

Functions and Pattern Matching

```
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 contructor 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
  } ;
```

ABS Standard Library

ABS Standard Library

```
module ABS.StdLib;
export *;

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

...
def Int size<A>(Set<A> xs) = ...
def Set<A> union<A>(Set<A> set1, Set<A> 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)

```
// 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

Active Classes

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

```
Unit run() {
  // active behavior ...
}
```

Imperative Constructs

Sequential Control Flow

Loop `while (x) { ... }`

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

Synchronous method call `x.m()`

Local State Update and Access (Assignment)

Object creation `new Car(Blue);`

Field read `x = [this.]f;` (only on **this** object)

Field assignment `[this.]f = 5;` (only on **this** object)

Blocks

- Sequence of variable declarations and statements
- Data type variables are initialized, reference types default to `null`
- Statements in block are 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, ...)`

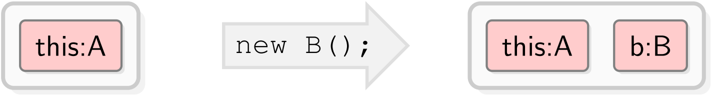
Scheduling

- Cooperative by special scheduling statements
  - Explicit decision of modeller
  - No preemptive scheduling $\Rightarrow$ no data races within COGs
- Non-deterministic otherwise
  - User-defined configuration of schedulers via annotations

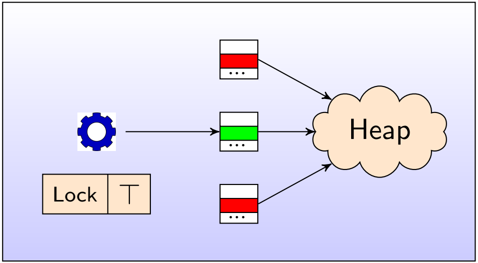
Method calls with shared heap access encapsulated in COGs

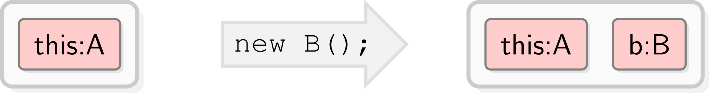
COG

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

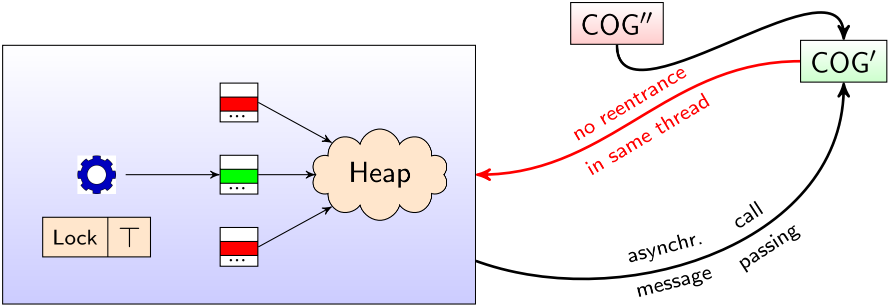


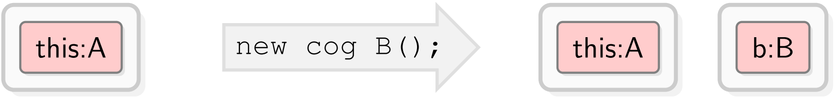
Method calls with shared heap access encapsulated in COGs





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





Asynchronous Method Calls

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);`

Asynchronous Method Calls

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

Asynchronous Method Calls

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)
  - `Fut<T> v = o!m(e);...; await v?; r = 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 comsumption)

Related to distributed systems and ABS models

- Bandwidth
- Executed instructions per component (COG)

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`

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;
      }
      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

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`
- 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

```
Account getAccount(Int aid) {
  Account result = null;   \
  Int n = length(as);       } 3
  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);          3
  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;
}
```

▶ 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);      } 3
  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;
}
```

► The list `as` is abstracted to its length

Basic approach (for Sequential Code)

❷ 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);      ⎬ 3
  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;
}
```

▶ The list `as` is abstracted to its length

$\mathrm{getAccount}(as, aid) = 3 + \mathrm{length}(as) + \mathrm{while}(0, n, aid, as) \qquad n = as$

Basic approach (for Sequential Code)

❷ 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);          } 3
  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;
}
```

▶ The list `as` is abstracted to its length

$$
\begin{aligned}
&\mathrm{getAccount}(as, aid) = 3 + \mathrm{length}(as) + \mathrm{while}(0, n, aid, as) && n = as \\
&\mathrm{while}(cnt, n, aid, as) = 4 + \mathrm{nth}(as, cnt) + \mathrm{getAid}(a) + \\
&\qquad \mathrm{if}(cnt, n, aid, id) + \mathrm{while}(cnt + 1, n, aid, as) && cnt < n \\
&\mathrm{while}(cnt, n, aid, as) = 0 && cnt \geq n
\end{aligned}
$$

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;      } 3
  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;
}
```

- The list `as` is abstracted to its length

$$
\begin{array}{ll}
getAccount(as, aid) = 3 + length(as) + while(0, n, aid, as) & n = as \\
while(cnt, n, aid, as) = 4 + nth(as, cnt) + getAid(a) + & \\
\quad if(cnt, n, aid, id) + while(cnt + 1, n, aid, as) & cnt < n \\
while(cnt, n, aid, as) = 0 & cnt \geq n \\
if(cnt, n, aid, id) = 1 & id = aid \\
if(cnt, n, aid, id) = 0 & id \neq aid
\end{array}
$$

Basic approach (for sequential models)

3. Solve the system of cost equations:

$$
\begin{array}{ll}
\mathrm{getAccount}(as, aid) = 3 + \mathrm{length}(as) + \mathrm{while}(0, n, aid, as) & n = as \\
\mathrm{while}(cnt, n, aid, as) = 4 + \mathrm{nth}(as, cnt) + \mathrm{getAid}(a) + & \\
\quad \mathrm{if}(cnt, n, aid, id) + \mathrm{while}(cnt + 1, n, aid, as) & cnt < n \\
\mathrm{while}(cnt, n, aid, as) = 0 & cnt \geq n \\
\mathrm{if}(cnt, n, aid, id) = 1 & id = aid \\
\mathrm{if}(cnt, n, aid, id) = 0 & id \neq aid
\end{array}
$$

Basic approach (for sequential models)

3. Solve the system of cost equations:

$$
\begin{array}{ll}
\text{getAccount}(as, aid) = 3 + as + \text{while}(0, n, aid, as) & n = as \\
\text{while}(cnt, n, aid, as) = 4 + cnt + 1 + & \\
\quad \text{if}(cnt, n, aid, id) + \text{while}(cnt + 1, n, aid, as) & cnt < n \\
\text{while}(cnt, n, aid, as) = 0 & cnt \geq n \\
\text{if}(cnt, n, aid, id) = 1 & id = aid \\
\text{if}(cnt, n, aid, id) = 0 & id \neq aid
\end{array}
$$

Assuming the following costs:

$$
\begin{array}{l}
\text{length}(as) = as \\
\text{nth}(as, cnt) = cnt \\
\text{getAid}(a) = 1
\end{array}
$$

Basic approach (for sequential models)

3. Solve the system of cost equations:

$$
\begin{array}{lr}
\text{getAccount}(as, aid) = 3 + as + \text{while}(0, n, aid, as) & n = as \\
\text{while}(cnt, n, aid, as) = 4 + cnt + 1 + & \\
\quad 1 + \text{while}(cnt + 1, n, aid, as) & cnt < n \\
\text{while}(cnt, n, aid, as) = 0 & cnt \geq n \\
\text{if}(cnt, n, aid, id) = 1 & id = aid \\
\text{if}(cnt, n, aid, id) = 0 & id \neq aid
\end{array}
$$

- An upper bound of $\text{if}(cnt, n, aid, id) = 1$

Basic approach (for sequential models)

3. Solve the system of cost equations:

$$
\begin{array}{ll}
\mathrm{getAccount}(as, aid) = 3 + as + \mathrm{while}(0, n, aid, as) & n = as \\
\mathrm{while}(cnt, n, aid, as) = 4 + cnt + 1 + & \\
\quad 1 + \mathrm{while}(cnt + 1, n, aid, as) & cnt < n \\
\mathrm{while}(cnt, n, aid, as) = 0 & cnt \geq n \\
\mathrm{if}(cnt, n, aid, id) = 1 & id = aid \\
\mathrm{if}(cnt, n, aid, id) = 0 & id \neq aid
\end{array}
$$

- An upper bound of $\mathrm{if}(cnt, n, aid, id) = 1$
- The cost of any iteration of while $6 + cnt$ is smaller than $6 + n$

Basic approach (for sequential models)

3. Solve the system of cost equations:

$$
\begin{array}{ll}
\text{getAccount}(as, aid) = 3 + as + \text{while}(0, n, aid, as) & n = as \\
\text{while}(cnt, n, aid, as) = 4 + cnt + 1 + & \\
\quad 1 + \text{while}(cnt + 1, n, aid, as) & cnt < n \\
\text{while}(cnt, n, aid, as) = 0 & cnt \geq n \\
\text{if}(cnt, n, aid, id) = 1 & id = aid \\
\text{if}(cnt, n, aid, id) = 0 & id \neq aid
\end{array}
$$

- An upper bound of $\text{if}(cnt, n, aid, id) = 1$
- The cost of any iteration of while $6 + cnt$ is smaller than $6 + n$
- *while* can iterate at most n times

Basic approach (for sequential models)

3. Solve the system of cost equations:

$$
\begin{array}{ll}
\mathrm{getAccount}(as, aid) = 3 + as + n^2 + 6n & n = as \\
\mathrm{while}(cnt, n, aid, as) = 4 + cnt + 1 + & \\
\quad 1 + \mathrm{while}(cnt + 1, n, aid, as) & cnt < n \\
\mathrm{while}(cnt, n, aid, as) = 0 & cnt \geq n \\
\mathrm{if}(cnt, n, aid, id) = 1 & id = aid \\
\mathrm{if}(cnt, n, aid, id) = 0 & id \neq aid
\end{array}
$$

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

Basic approach (for sequential models)

3. Solve the system of cost equations:

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

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

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:

```
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;
}
```

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:

```
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:

```
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

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:

```
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

Analyzing Concurrent Models: Class Invariants

Use class invariants to capture the required information

Analyzing Concurrent Models: Class Invariants

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

```
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;
}
```

Analyzing Concurrent Models: Class Invariants

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<=max(as)]` at its release points

Analyzing Concurrent Models: Class Invariants

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

Analyzing Concurrent Models: Class Invariants

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 resul
  Int cnt = 0;
  while (cnt <
    Account a =
    Fut<Int>id
    await  idF
    Int id=idF
    if (aid == id) {
      result = a;
    }
    cnt = cnt+1;
  }
  return result;
}
```

- Annotate the method with `[as<=max(as)]` at its release

guaranteed that the
s not iterate more than
) times

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:

```
{
  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`)

```
{
  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)

```
{
  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`

```
{
  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`

```
{
  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);
}
```

Assume `insertAccount` modifies the field `as` of the database as expected

Method applied to to `getAccount`:

```
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`:

```
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

Rely-Guarantee Reasoning: Example

Method applied to to `getAccount`:

```
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

Rely-Guarantee Reasoning: Example

Method applied to to `getAccount`:

```
{
  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

Method applied to to `getAccount`:

```
{
  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`

Rely-Guarantee Reasoning: Example

Method applied to to `getAccount`:

```
{
  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`
5. therefore, the upper bound is correct and we are done!

Rely-Guarantee Reasoning: Example(2)

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);
    await aFut?;
    i = i+1;
  }
  db!getAccount(3);
}
```

Rely-Guarantee Reasoning: Example(2)

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);
    await aFut?;
    i = i+1;
  }
  db!getAccount(3);
}
```

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

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);
    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

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);
    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

Rely-Guarantee Reasoning: Example(2)

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);
    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 asumptions (it has at most 10 iterations)

Rely-Guarantee Reasoning: Example(2)

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);
    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 asumptions (it has at most 10 iterations)
8. we are done!

Part III

Deadlock Analyis 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`)

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);
}
```

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

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

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

A

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

C

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

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

A

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

C

```
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);
}
```

B

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

A

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

C

```
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);
}
```

B

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

A

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

C

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

Dependencies:

1. $B \rightarrow C.wait$

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

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

A

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

C

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

Dependencies:

1. $B \rightarrow C.wait$
2. $A \rightarrow B.empt2$
3. $B.empt2 \rightarrow B$ (wait for B's lock)

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);
}
```

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

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

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

Dependencies:

1. $B \rightarrow C.wait$
2. $A \rightarrow B.empt2$
3. $B.empt2 \rightarrow B$
   (wait for B's lock)

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

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

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

Dependencies:

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

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

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

A

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

C

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

Dependencies:

1. $B \to C.wait$
2. $A \to B.empt2$
3. $B.empt2 \to B$ (wait for B's lock)
4. $C.wait \to A.empt$
5. $A.empt \to A$ (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 (`await g` or `get`)

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

This is our plan:

- Obtain a 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

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

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

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

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

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

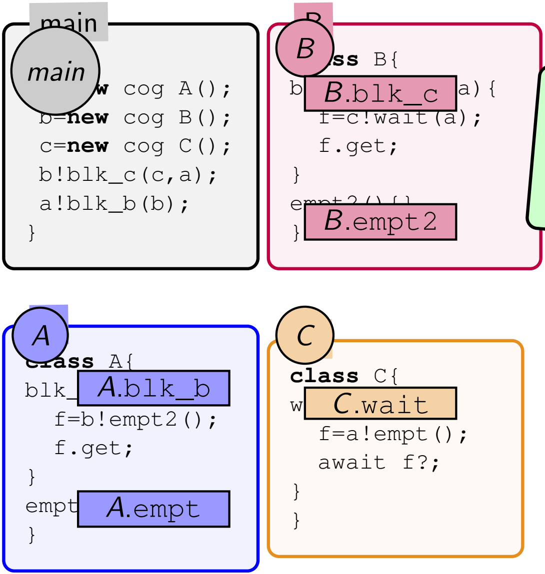
Abstract COGs:

- *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 & tasks

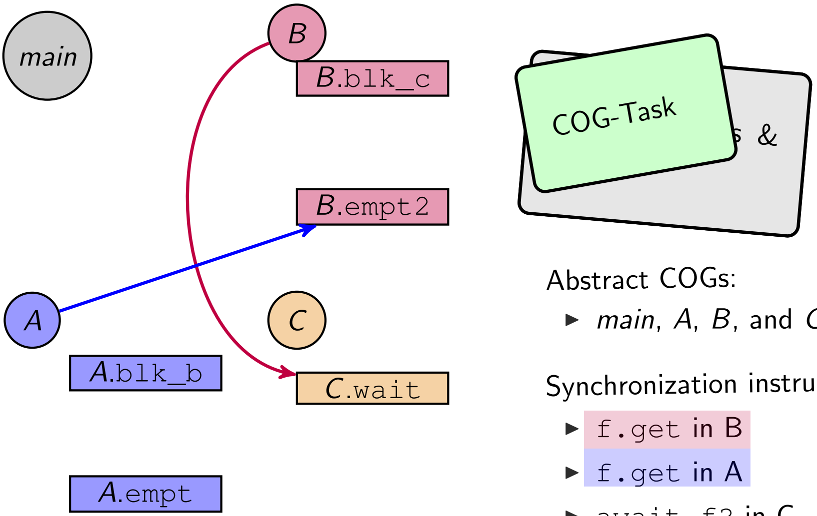
Abstract COGs:

- *main*, *A*, *B*, and *C*

Synchronization instructions:

- `f.get` in B
- `f.get` in A
- `await f?` in C

Building the Abstract Dependency Graph

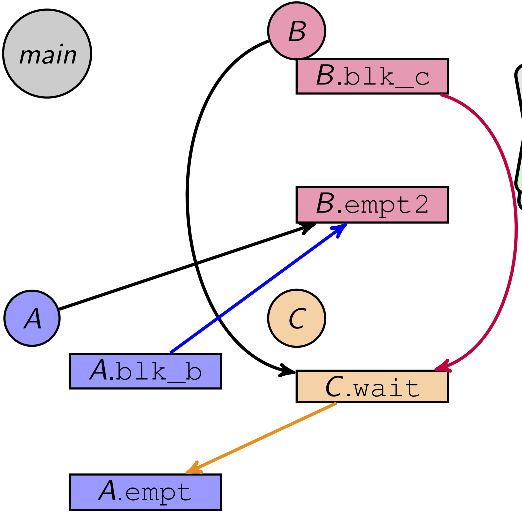


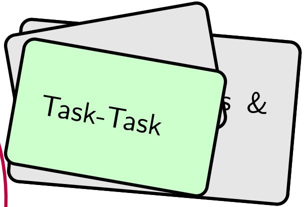
Abstract COGs:

- *main*, *A*, *B*, and *C*

Synchronization instructions:

- `f.get` in B
- `f.get` in A
- `await f?` in C





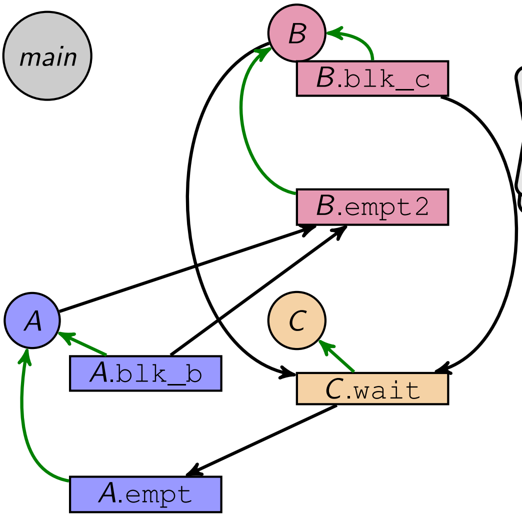
Abstract COGs:

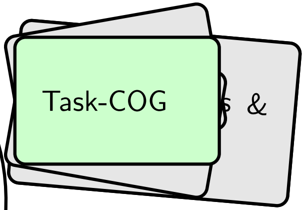
- *main*, *A*, *B*, and *C*

Synchronization instructions:

- `f.get` in B
- `f.get` in A
- `await f?` in C

Building the Abstract Dependency Graph





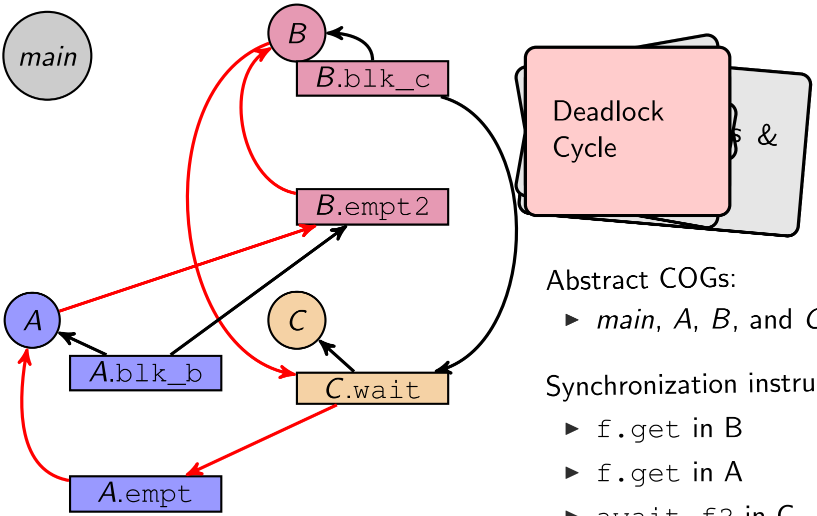
Abstract COGs:

- *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

main

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

B

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

A

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

C

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

Add synchronization

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

May-Happen-in-Parallel Information

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

May-Happen-in-Parallel Information

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

main

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

B

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

A

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

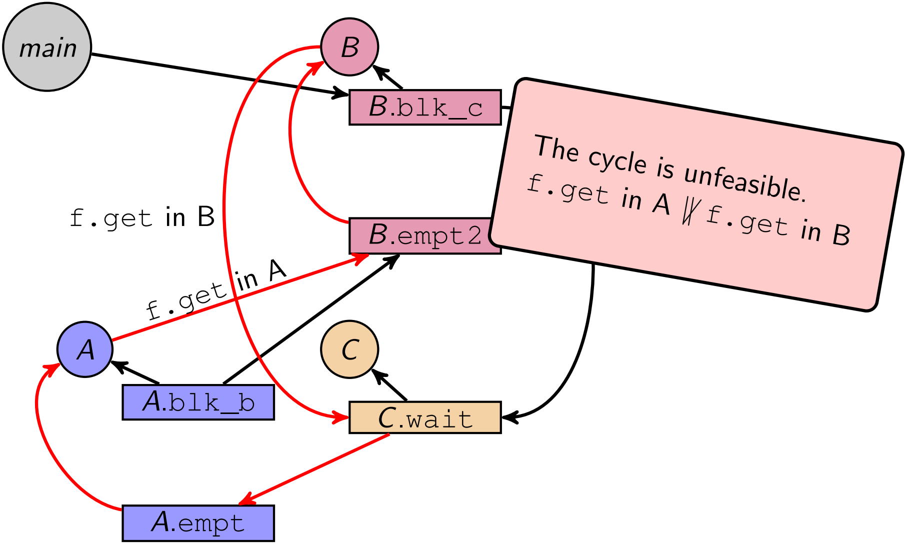
C

```
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` $\nparallel$ `f.get` in `B`

Applying MHP to Annotated Dependency Graph



Part IV

Deductive Verification Of ABS Models

Deductive verification of ABS

Functional verification of complex (first-order) properties

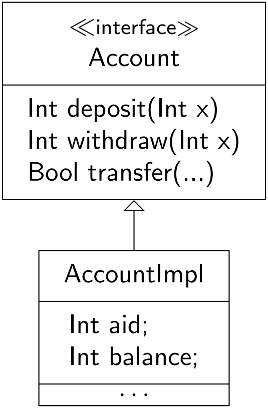
Uses ideas by
W. Ahrendt, M. Dylla, and C. Din et al.

Deductive Functional Verification

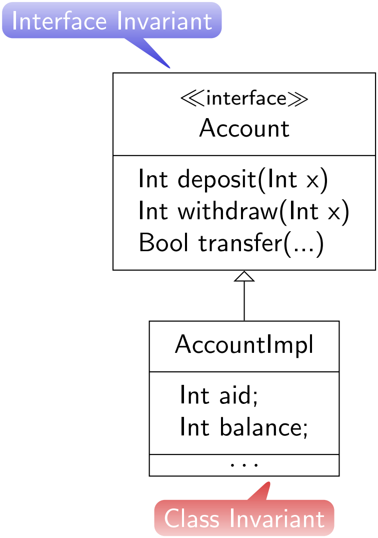
Based on a program logic for ABS that ...

- Uses verification paradigma "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

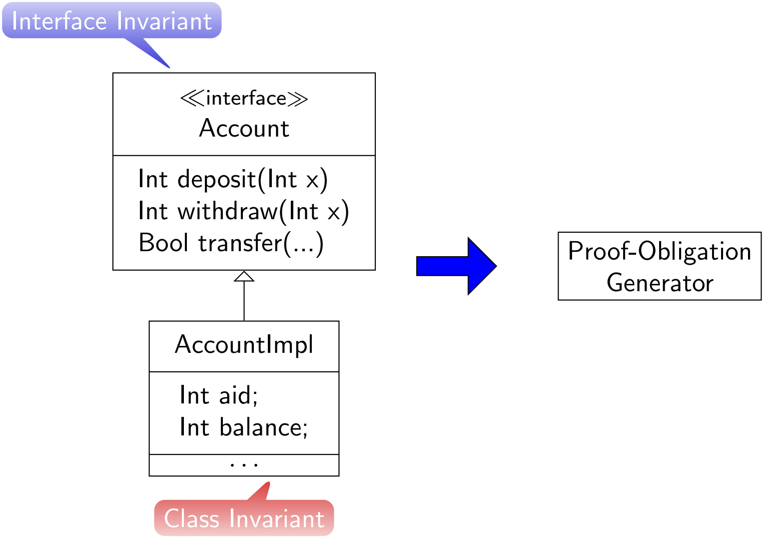
Verification Workflow



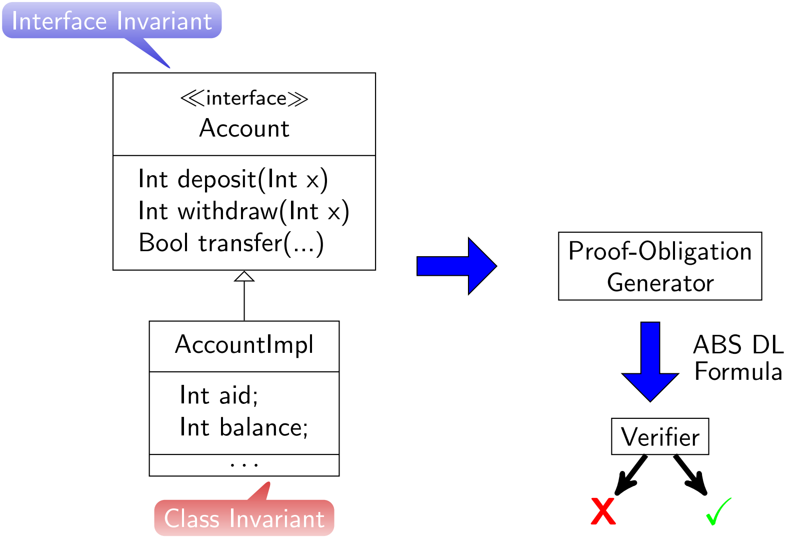
Verification Workflow



Interface invariants	express mostly restrictions on the control-flow
Class invariants	relate the object state to the local system history


Interface Invariant
≪interface≫
Account
Int deposit(Int x)
Int withdraw(Int x)
Bool transfer(...)
AccountImpl
Int aid;
Int balance;
. . .
Class Invariant
Proof-Obligation Generator

Verification Workflow



ABS-DL: Dynamic Logic

Dynamic Logic for ABS

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

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

We consider only partial correctness (box modality)

ABS-DL: Dynamic Logic

Dynamic Logic for ABS

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

- $\langle \mathrm{p} \rangle \phi$: Program p terminates and in its final state ϕ holds.
- $[\mathrm{p}]\phi$: 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

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;
}
```

Sorts S

For each

- interface I,
- abstract datatype T (including built-in ADTs)

there are sorts $I, T \in S$

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

ABS-DL: Sequent Calculus

Gentzen-Style Sequent Calculus: $\Gamma \Rightarrow \Delta$

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

ABS-DL: Sequent Calculus

Gentzen-Style Sequent Calculus: $\Gamma \Rightarrow \Delta$

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

Propositional Rules (classical logic)

$$\text{andLeft} \; \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \wedge B \Rightarrow \Delta} \qquad \text{andRight} \; \frac{\Gamma \Rightarrow A, \Delta \qquad \Gamma \Rightarrow B, \Delta}{\Gamma \Rightarrow A \wedge B, \Delta}$$

$$\text{impRight} \; \frac{\Gamma, A \Rightarrow B, \Delta}{\Gamma \Rightarrow A \to B, \Delta}$$

ABS-DL: Dynamic Logic and Updates

Symbolic Representation of State Transitions

Generalization of explicit substitutions

Elementary update (val has no side effects)

$$\mathsf{loc} := \mathsf{val}$$

Same meaning as an assignment

ABS-DL: Dynamic Logic and Updates

Symbolic Representation of State Transitions

Generalization of explicit substitutions

Elementary update (val has no side effects)

$$\mathsf{loc} := \mathsf{val}$$

Same meaning as an assignment

Parallel update

$$\mathsf{loc}_1 := \mathsf{val}_1 \,||\, \mathsf{loc}_2 := \mathsf{val}_2$$

Symbolic Representation of State Transitions

Generalization of explicit substitutions

Elementary update (val has no side effects)

$$\mathsf{loc} := \mathsf{val}$$

Same meaning as an assignment

Parallel update

$$\mathsf{loc}_1 := \mathsf{val}_1 \,||\, \mathsf{loc}_2 := \mathsf{val}_2$$

Update application

on term: $\{\mathcal{U}\}t$ on formula: $\{\mathcal{U}\}\phi$

Symbolic Representation of State Transitions

Generalization of explicit substitutions

Elementary update (val has no side effects)

$$\mathsf{loc} := \mathsf{val}$$

Same meaning as an assignment

Parallel update

$$\mathsf{loc}_1 := \mathsf{val}_1 \,||\, \mathsf{loc}_2 := \mathsf{val}_2$$

Update application

$$\text{on term: } \{\mathcal{U}\}t \qquad \text{on formula: } \{\mathcal{U}\}\phi$$

Example (Update simplification)

$\{\mathsf{i} := \mathsf{j} \| \mathsf{j} := \mathsf{i}\}(\mathsf{i} = \mathsf{i}_{old}) \rightsquigarrow$

ABS-DL: Dynamic Logic and Updates

Symbolic Representation of State Transitions

Generalization of explicit substitutions

Elementary update (val has no side effects)

$$\mathsf{loc} := \mathsf{val}$$

Same meaning as an assignment

Parallel update

$$\mathsf{loc}_1 := \mathsf{val}_1 \,||\, \mathsf{loc}_2 := \mathsf{val}_2$$

Update application

$$\text{on term: } \{\mathcal{U}\}t \qquad \text{on formula: } \{\mathcal{U}\}\phi$$

Example (Update simplification)

$$\{\mathsf{i} := \mathsf{j} \| \mathsf{j} := \mathsf{i}\}(\mathsf{i} = \mathsf{i}_{old}) \rightsquigarrow$$
$$(\{\mathsf{i} := \mathsf{j} \| \mathsf{j} := \mathsf{i}\}\mathsf{i}) = (\{\mathsf{i} := \mathsf{j} \| \mathsf{j} := \mathsf{i}\}\mathsf{i}_{old}) \rightsquigarrow$$

Symbolic Representation of State Transitions

Generalization of explicit substitutions

Elementary update (val has no side effects)

$$\mathsf{loc} := \mathsf{val}$$

Same meaning as an assignment

Parallel update

$$\mathsf{loc}_1 := \mathsf{val}_1 \,||\, \mathsf{loc}_2 := \mathsf{val}_2$$

Update application

$$\text{on term: } \{\mathcal{U}\}t \qquad \text{on formula: } \{\mathcal{U}\}\phi$$

Example (Update simplification)

$$\{\mathsf{i} := \mathsf{j} \| \mathsf{j} := \mathsf{i}\}(\mathsf{i} = \mathsf{i}_{old}) \rightsquigarrow$$
$$(\{\mathsf{i} := \mathsf{j} \| \mathsf{j} := \mathsf{i}\}\mathsf{i}) = (\{\mathsf{i} := \mathsf{j} \| \mathsf{j} := \mathsf{i}\}\mathsf{i}_{old}) \rightsquigarrow \mathsf{j} = \mathsf{i}_{old}$$

ABS-DL: Support for Imperative ABS

Assignment Rule

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

e side effect free (pure) expression

ABS-DL: Support for Imperative ABS

Assignment Rule

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

e side effect free (pure) expression

Conditional Rule

$$\frac{\Gamma, \{\mathcal{U}\}e \Rightarrow \{\mathcal{U}\}[p;\omega]\phi, \Delta \qquad \Gamma, \{\mathcal{U}\}\neg e \Rightarrow \{\mathcal{U}\}[q;\omega]\phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathrm{if}\ (e)\ \{p\}\ \mathrm{else}\ \{q\}\ \omega]\phi, \Delta}$$

- *e* pure expression
- Rule splits proof in a then and else branch

ABS-DL: Heap Modelling

Explicit Heap Model

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

$$\mathrm{select}(\underbrace{\mathrm{store}(\mathit{heap}, \mathit{this}, \mathit{field}, 5)}_{\mathit{this}.\mathit{field} = 5;}, \mathit{this}, \mathit{field})$$

Advantage

Easy to assign all fields an unknown value

- anonymization of field values at control release

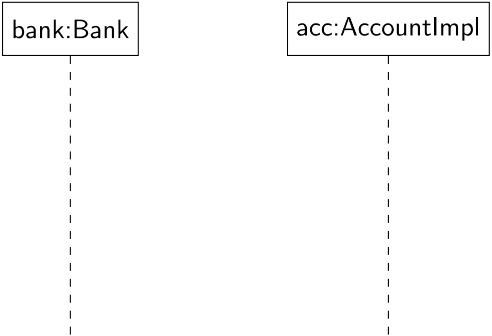
ABS Dynamic Logic (DL): History Events

Verification of distributed systems achieved by sequential means

ABS Dynamic Logic (DL): History Events

Verification of distributed systems achieved by sequential means

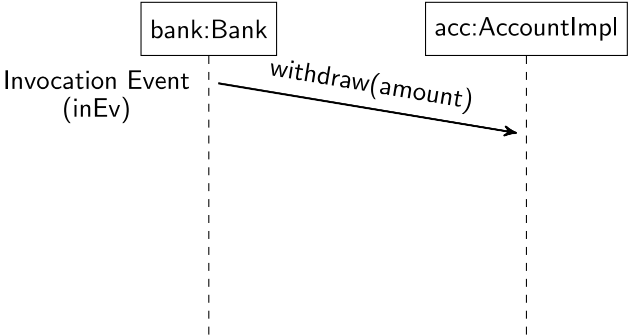
⇒ Event Histories: Sequences of Messages [Din et al.]



ABS Dynamic Logic (DL): History Events

Verification of distributed systems achieved by sequential means

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



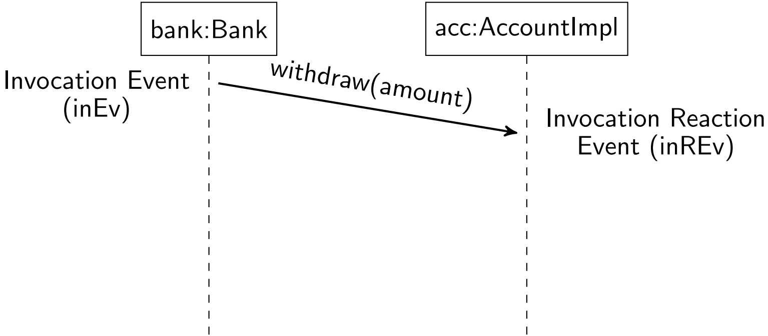
History Event: **inEv**(bank, acc, fut, withdraw, (*amount*))

- fut: newly created future for asynchronous message
- $(\cdots)$: list of arguments

ABS Dynamic Logic (DL): History Events

Verification of distributed systems achieved by sequential means

$\Rightarrow$ Event Histories: Sequences of Messages [Din et al.]



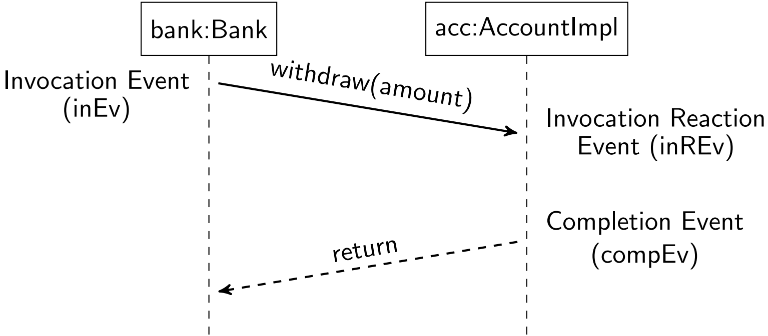
History Event: **inREv**(bank, acc, fut, withdraw, (*amount*))

- event created upon execution of method
- fut: future to be resolved upon completion

ABS Dynamic Logic (DL): History Events

Verification of distributed systems achieved by sequential means

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



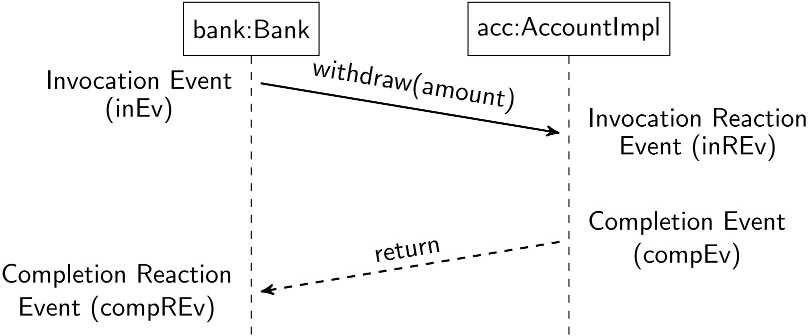
History Event: **compEv**(acc, fut, withdraw, r)

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

ABS Dynamic Logic (DL): History Events

Verification of distributed systems achieved by sequential means

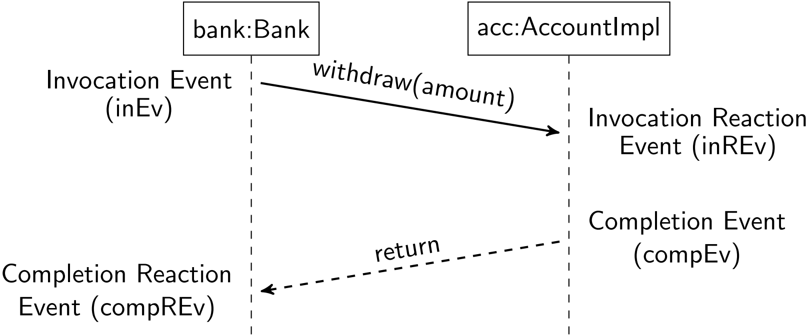
⇒ Event Histories: Sequences of Messages [Din et al.]



History Event: **compREv**(bank, fut, r)

- event created upon resolving fut

ABS Dynamic Logic (DL): History Events



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

ABS-DL: Concurrent Constructs

Messages: Structured Labels/Events

$$\mathrm{inEv}(\mathit{caller}, \mathit{callee}, \mathit{future}, \mathit{method}, \overline{\mathit{args}})$$
$$\mathrm{inREv}(\mathit{caller}, \mathit{callee}, \mathit{future}, \mathit{method}, \overline{\mathit{args}})$$
$$\mathrm{compEv}(\mathit{callee}, \mathit{future}, \mathit{method}, \mathit{return\ value})$$
$$\mathrm{compREv}(\mathit{receiver}, \mathit{future}, \mathit{return\ value})$$

ABS-DL: Concurrent Constructs

Messages: Structured Labels/Events

$$\mathrm{inEv}(\mathit{caller}, \mathit{callee}, \mathit{future}, \mathit{method}, \overline{\mathit{args}})$$
$$\mathrm{inREv}(\mathit{caller}, \mathit{callee}, \mathit{future}, \mathit{method}, \overline{\mathit{args}})$$
$$\mathrm{compEv}(\mathit{callee}, \mathit{future}, \mathit{method}, \mathit{return\ value})$$
$$\mathrm{compREv}(\mathit{receiver}, \mathit{future}, \mathit{return\ value})$$

History

Represented as program variable of type History

- Standard sequent axiomatisation
- Specification predicates and functions:
  - *wfHist*, *isCompletionEv*, *isInvocationEv*
  - *getResultFrom*(*label*), etc.

ABS-DL: Concurrent Constructs

Messages: Structured Labels/Events

$$\begin{array}{l}
\mathrm{inEv}(\mathit{caller}, \mathit{callee}, \mathit{future}, \mathit{method}, \overline{\mathit{args}}) \\
\mathrm{inREv}(\mathit{caller}, \mathit{callee}, \mathit{future}, \mathit{method}, \overline{\mathit{args}}) \\
\mathrm{compEv}(\mathit{callee}, \mathit{future}, \mathit{method}, \mathit{return\ value}) \\
\mathrm{compREv}(\mathit{receiver}, \mathit{future}, \mathit{return\ value})
\end{array}$$

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

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

Example: Invariants of class Account

We can now express:

Balance of class Account always non-negative

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

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

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

```
\invariants(Seq theHistory, Heap theHeap, ABSAnyInterface self) {
 amountOfDepositNonNegative : AccountImpl {
 \forall Event ev; \forall int i; ( i >= 0 & i < seqLen(theHistory) ->
    ( ev = Event::seqGet(theHistory, i) &
      ( isInvocationEv(ev) | isInvocationREv(ev)) &
      getMethod(ev) = Account::deposit ->
                  int::seqGet(getArguments(ev), 0) >= 0 ) ) ) };
```

Asynchronous Method Invocation

Asynchronous Method Call

$$\frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}(o \not\doteq \mathsf{null} \land \mathit{wfHist}(\mathcal{H})), \Delta \\ \Gamma \Rightarrow \{\mathcal{U}\}(\mathit{futureUnused}(\mathit{frc}, \mathcal{H}) \rightarrow \\ \{\mathit{fr} := \mathit{frc} \,||\, \mathcal{H} := \mathcal{H} \circ \mathit{inEv}(\mathit{this}, o, \mathit{frc}, m, \overline{\mathit{args}})\}(R_I(\mathcal{H}, o) \land [\omega]\phi)), \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[r = o!\mathrm{m}(\overline{\mathit{args}}); \omega]\phi, \Delta}$$

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 ϕ holds after execution of remaining program

Awaiting Completion

$$
\frac{\begin{array}{l}\Gamma \Rightarrow \mathit{CInv}(C)(\mathit{heap}, \mathcal{H}, \mathit{this}), \Delta \\ \Gamma \Rightarrow \{\mathit{heap} := \mathit{newHeap} \mid\mid \mathcal{H} := \mathcal{H} \circ \mathcal{A}_{\mathcal{H}} \circ \mathit{compREv}(\ldots)\} \\ \qquad\qquad (\mathit{CInv}(C)(\mathit{heap}, \mathcal{H}, \mathit{this}) \wedge E_I(\mathcal{H}, \mathit{this}) \wedge \mathit{wfHist}(\mathcal{H}) \rightarrow [\omega]\phi), \Delta\end{array}}{\Gamma \Rightarrow [\mathrm{await}\ r?; \omega]\phi, \Delta}
$$

Asynchronous Method Invocation

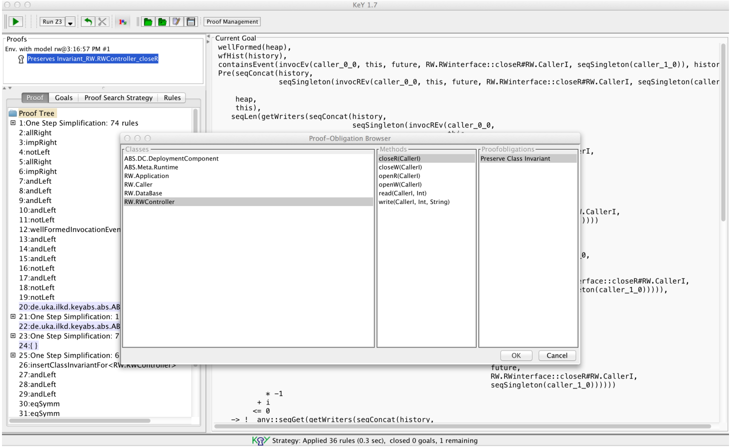
Awaiting Completion

$$\frac{\begin{array}{l}\Gamma \Rightarrow \mathit{CInv}(C)(\mathit{heap}, \mathcal{H}, \mathit{this}), \Delta \\ \Gamma \Rightarrow \{\mathit{heap} := \mathit{newHeap} \mid\mid \mathcal{H} := \mathcal{H} \circ \mathcal{A}_{\mathcal{H}} \circ \mathit{compREv}(\ldots)\} \\ \qquad\qquad (\mathit{CInv}(C)(\mathit{heap}, \mathcal{H}, \mathit{this}) \wedge E_I(\mathcal{H}, \mathit{this}) \wedge \mathit{wfHist}(\mathcal{H}) \rightarrow [\omega]\phi), \Delta\end{array}}{\Gamma \Rightarrow [\mathrm{await}\ r?; \omega]\phi, \Delta}$$

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



Tool

- 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