The Abstract Behavioral Specification Language

Frank S. de Boer

CWI
frb@cwi.nl

Scientific Meeting
CWI, November 29, 2013

How It All Started?

- Simula (Ole Johan Dahl, Turing award 2001)
- Credo FP6 project (Modeling and analysis of evolutionary structures for distributed services)
- HATS FP7 project (Highly Adaptable and Trustworthy Software using Formal Models)

Main Challenges: Cloud and Multicore

Resource-aware programming models

- Monitoring resources
- Resource management

Executable deployment models

- Simulation
- Analysis

Approach

Abstract behavioral specification language

for leveraging resources and their dynamic management to the abstraction level of models.

Abstraction level

matches that of a high-level programming language, in order to be easily usable and accessible to software developers.

Formal methods

for the analysis of resource needs and deployment scenarios at the design stage.

Our Goal

Full development cycle

- Modeling
- Analysis
- Code generation

The Abstract Behavioral Specification Language (ABS)

Main features

- User-defined Algebraic Data Types
- Concurrent Objects
  - Encapsulation
  - Cooperative multitasking
    (multiple tasks originating from asynchronous calls)

Encapsulation: Classes and Interfaces

Interfaces

- Provide reference types of objects (implementation abstraction)
- Subinterfaces allowed
- Multiple inheritance allowed
- Reference types may occur in data types, but:
  no method calls in function definitions (possible side effects)

Classes

- Only for object construction
- Class name is not a type
- Classes can implement several interfaces
- Fields are private

Class Initialization, Active Classes

Class Initialization

- Optional class parameters = fields = constructor signature
- Fields with primitive types must be initialized when declared
- Optional `init` block executed first

Active Classes

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

Example:

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

Asynchronous Method Calls

Syntax and Semantics

- `target ! methodName(arg1, arg2, ...)`
- Sends an asynchronous message to the target object
- Creates new task
- Caller continues execution and allocates a future to store the result
  - `Fut<T> v = o!m(e);`

Scheduling

Unconditional Scheduling

- `suspend` command yields control to other task (in the object)

Conditional Scheduling

- `await g`, where `g` is a polling guard
- Guards are inductively defined as:
  - `b` - where `b` is a side-effect-free boolean expression
  - `f?` - future guards
  - `g & g` - conjunction (not Boolean operator)
- Yields task execution until guard is true
  (continue immediately if guard is true already)

Synchronization and Blocking

Reading Futures

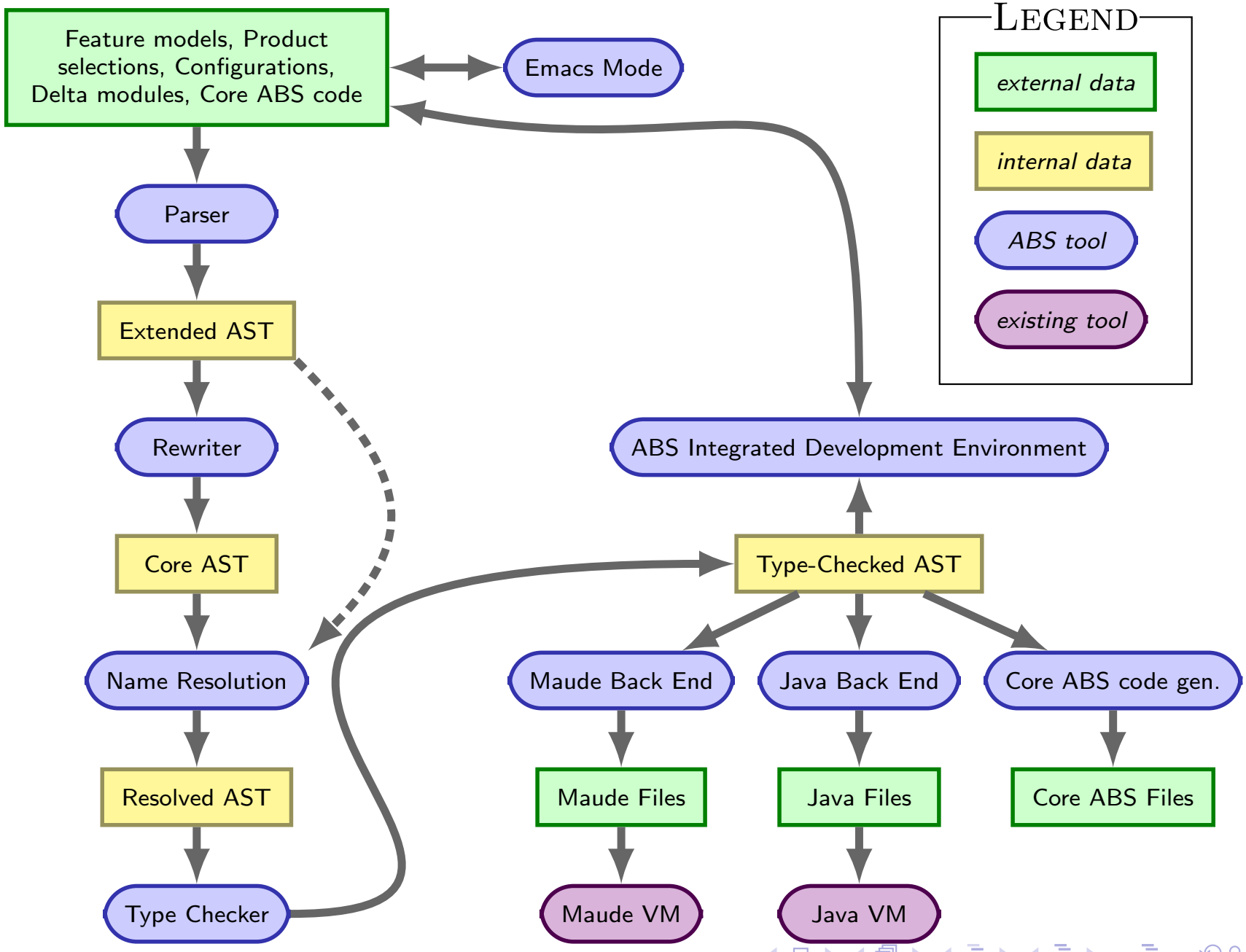
- `f.get` - reads future `f` and blocks execution until result is available
- Deadlocks possible (use static analyzer for detection)
- Programming idiom: use `await f?` to prevent blocking (safe access)
  - `Fut<T> v = o!m(e);...; await v?; r = v.get;`

Blocking vs. Suspension

Suspension Lets other task in same object continue (if any)

Blocking No task in the same object can continue until future resolved

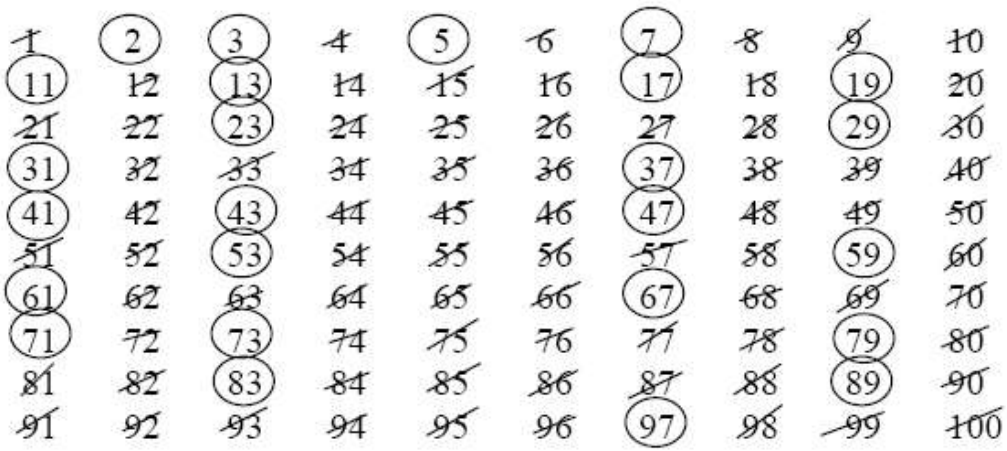
The ABS Basic Tool Chain



Capabilities of the ABS Tool Set

- ABS IDE (Eclipse-based), parser, compiler, type checker
- Maude, Java and Erlang code generation
- Execution visualization
- Monitor inlining
- Deployment components with timing constraints
- Run-time assertion checking
- Deadlock analysis
- Automated resource (time, space) analysis
- Automated test case generation

Sieve of Eratosthenes



A Parallel Sieve of Eratosthenes

```
interface IPrime {
    Unit divide (Int n);
}
```

```
class Generator(Int b) {
    Int n = 3 ;
    IPrime two;
    {
        two = new Prime(2);
    }
    Unit run() {
        while (n < b) {
            two!divide(n) ;
            n = n + 1;
        }
    }
}
```

```
class Prime (Int p) implements IPrime {
    IPrime next;
    Unit divide (Int n) {
        if ( (n % p) != 0) {
            if (next != null) {
                next!divide (n);
            }
            else {
                next = new Prime(n);
            }
        }
    }
}
```

Main

```
{
    new Generator(read());
}
```