

Executing ASM Specifications with CoreASM

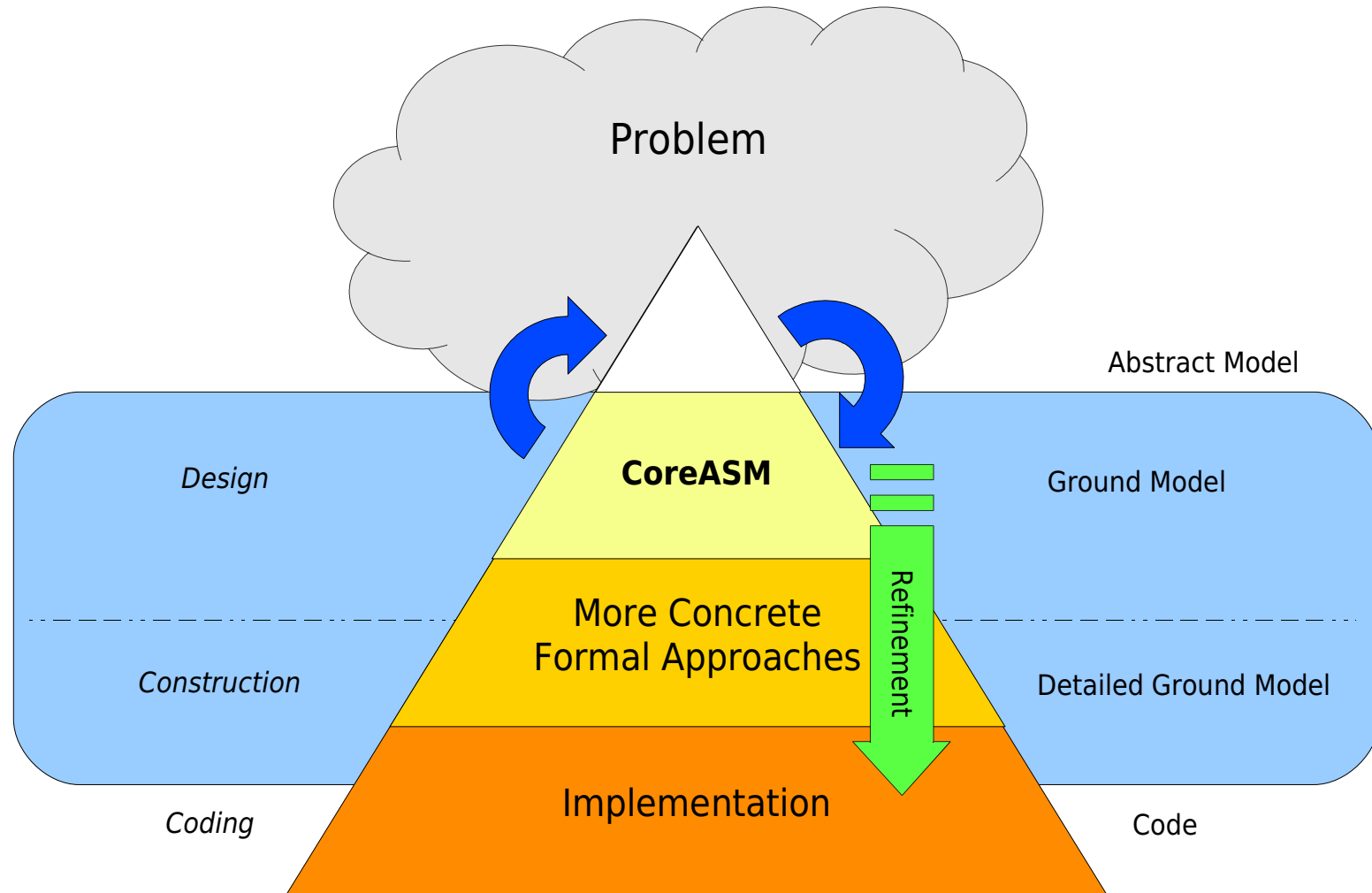
Part 2: Extensibility and Applications

Roozbeh Farahbod

Software Technology Lab
Simon Fraser University
Burnaby, Canada

A joint work with
Vincenzo Gervasi, Uwe Glässer, George Ma, and Mashaah Memon

CoreASM: The Idea



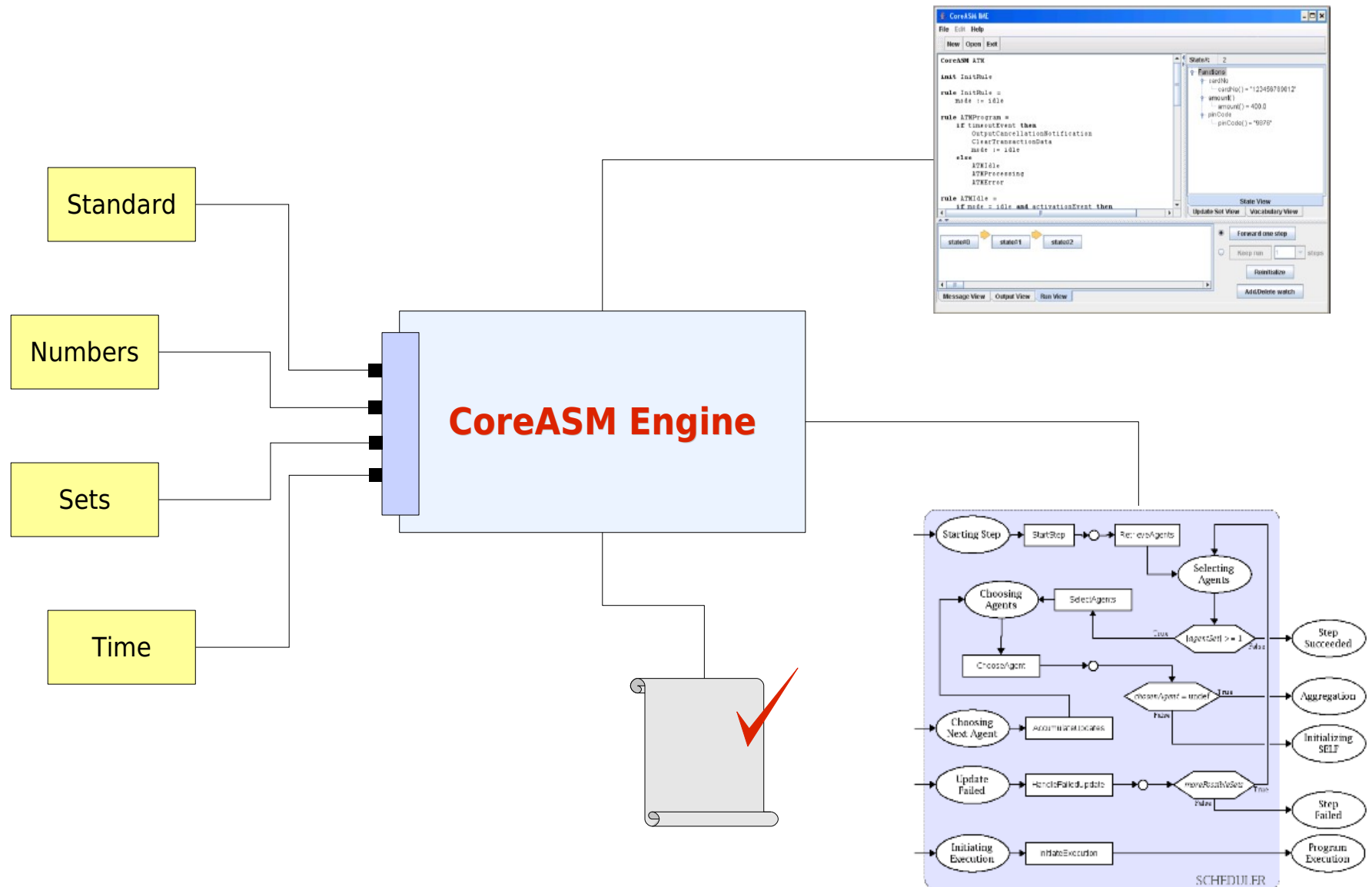
CoreASM Goals

- A *lean, executable, and extensible* ASM language which is faithful to its mathematical definition
- An *extensible, platform-independent* execution engine
- A supporting *tool environment* for
 - Design exploration
 - Experimental validation, fast prototyping
 - Formal verification

CoreASM – architecture features

- We want to **reduce the cost** of writing a spec
- Hence, we have to reduce the cost of **encoding** (from domain concepts to language concepts)
- Hence, we want to offer a **domain-specific** language – for all domains...
- Hence, we designed an **extensible language**, which can be adapted to several domains
- Net result: **plug-in architecture**

Kernel of a Novel Environment

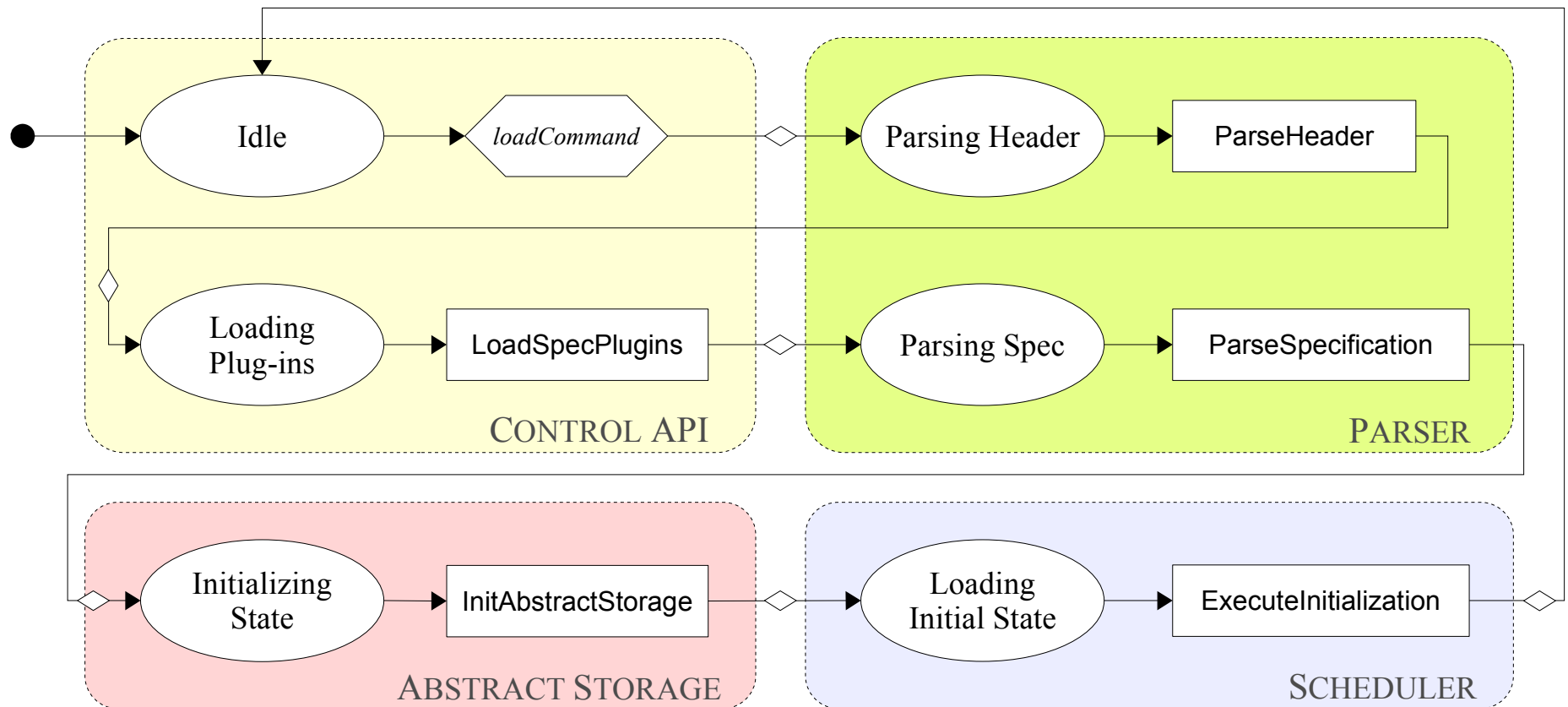


Extensible Engine

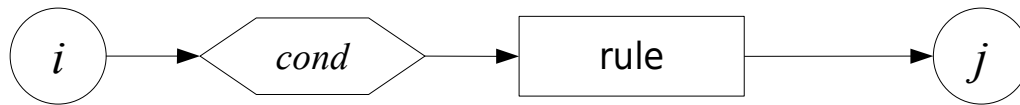
- Plug-ins can extend and alter the engine in three dimensions:
 1. Data Structures
 - new backgrounds
 2. Control Structures
 - new rule forms
 3. Execution Model:
 - new scheduling policies
 - preprocessing the abstract syntax tree
 - monitoring updates

Specification of the Engine

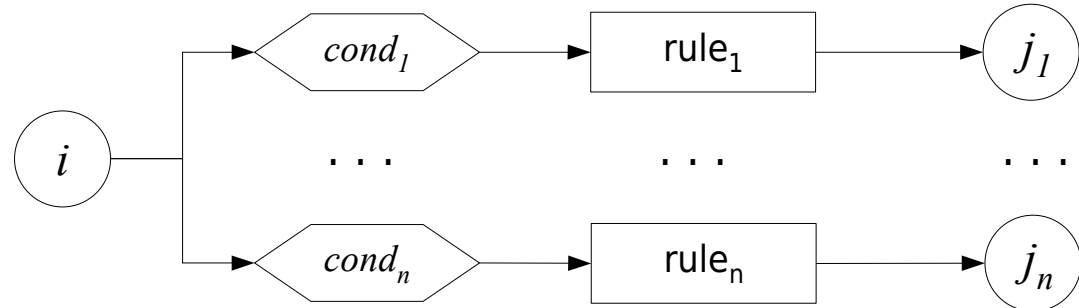
Loading Specifications



Extensible Control State ASMs (1)

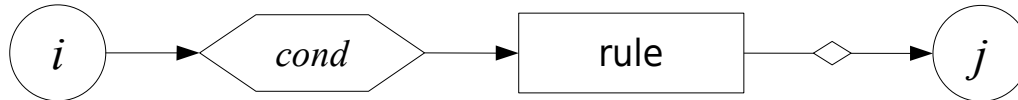


$\text{FSM}(i, \text{if } cond \text{ then } rule, j) \equiv$
 $\text{if } ctl_state = i \text{ and } cond \text{ then}$
 $rule$
 $ctl_state := j$



$\text{FSM}(i, \text{if } cond_1 \text{ then } rule_1, j_1)$
 $\text{FSM}(i, \text{if } cond_2 \text{ then } rule_2, j_2)$
 \dots
 $\text{FSM}(i, \text{if } cond_n \text{ then } rule_n, j_n)$

Extensible Control State ASMs (2)

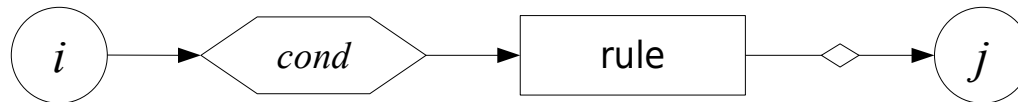


EFSM(*i*, if *cond* then *rule*, *j*) \equiv
if *ctl_state* = *i* and *cond* then
rule seq Proceed(*i*, *j*)

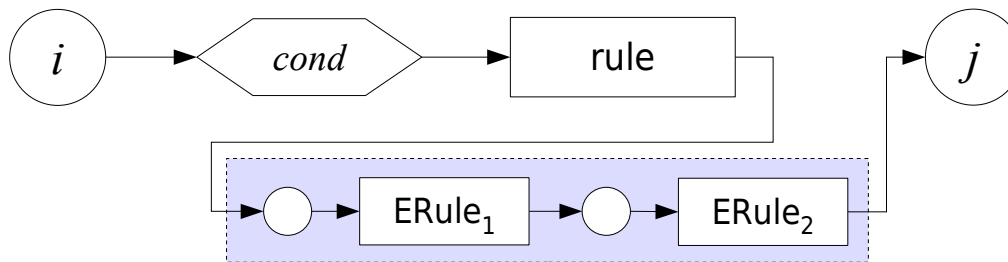
Proceed(*i*, *j*) \equiv
forall *p* \in *extensionPointPlugins* do
 marked(*p*) := *isRegistered*(*p*, *i*, *j*)
 seq
 iterate
 choose *p* \in *extensionPointPlugins* with *marked*(*p*) do
 marked(*p*) := *false*
 let *R* = *extensionRule*(*p*) in
 R(*i*, *j*)
 seq
ctl_state := *j*

Extensible Control State ASMs (3)

An EFSM of the form

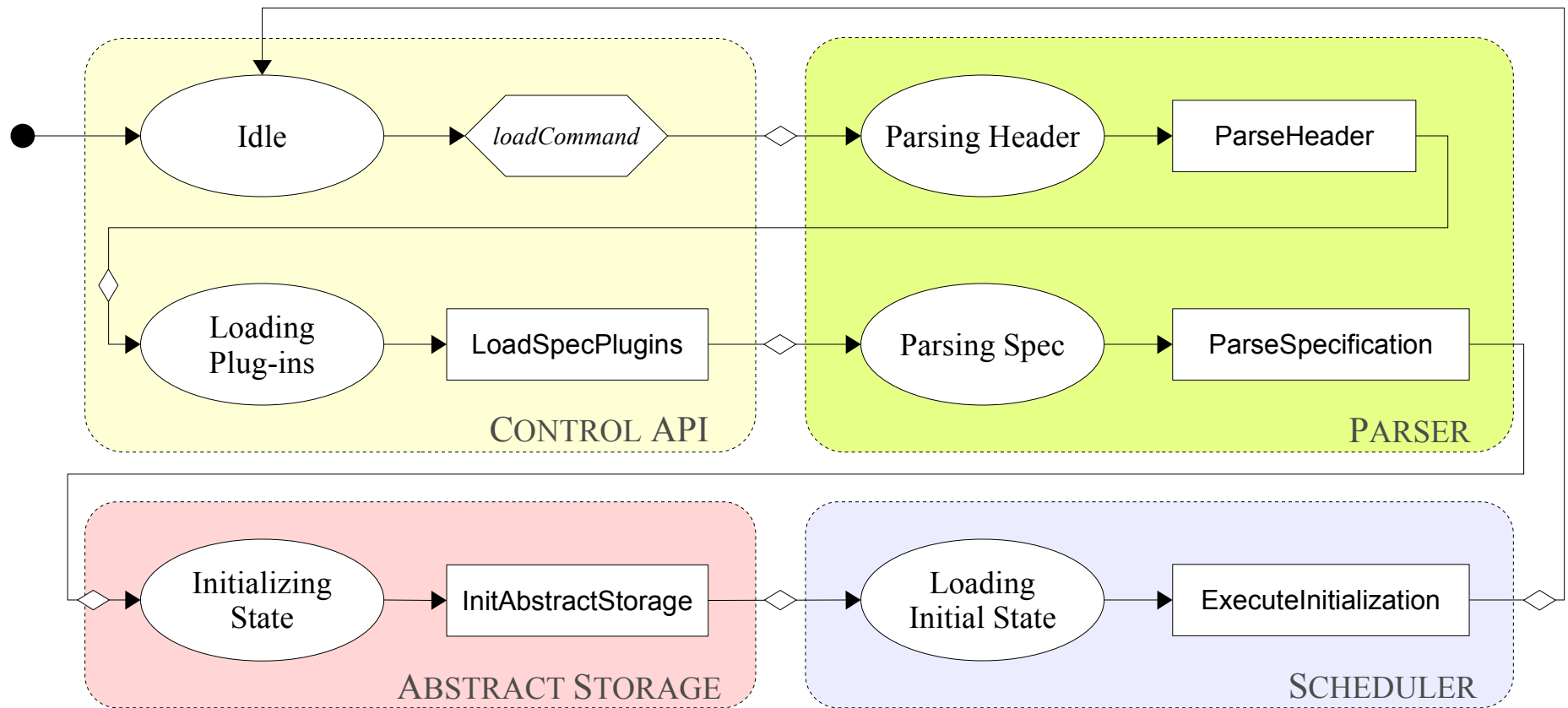


with two extensions, would expand to



Specification of the Engine

Loading Specifications



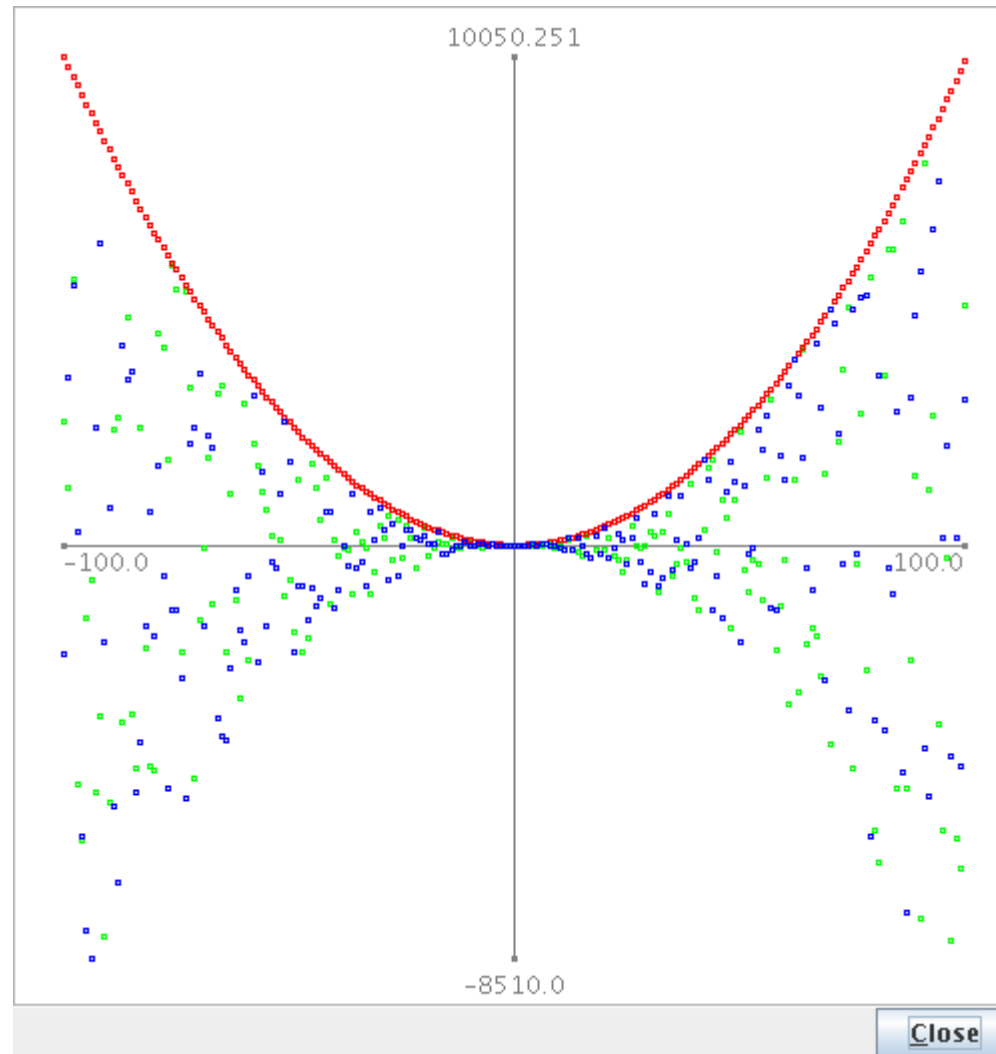
Plotter Plugin

- Biological Reactor Specification
 - by V. Gervasi, A. Cisternino, and D. Mazzei
 - Controlling the PH level of a biological reactor
 - Spraying CO2 when the PH level is high
 - Modelling the Controller and the Environment
- How should we see the output?
 - Print the results and export it to Excel
 - Why not having a **plot** rule?

Plotter Plugin

- Plotter Plugin implements
 - parser, interpreter, extension-points, and vocabulary extender plug-in interfaces
- Provides two rule forms:
 - **plot** func
 - **plot** func **in** window
- Extends the execution of the engine:
 - after every successful step looks into the state and plots the functions marked by the plot rule

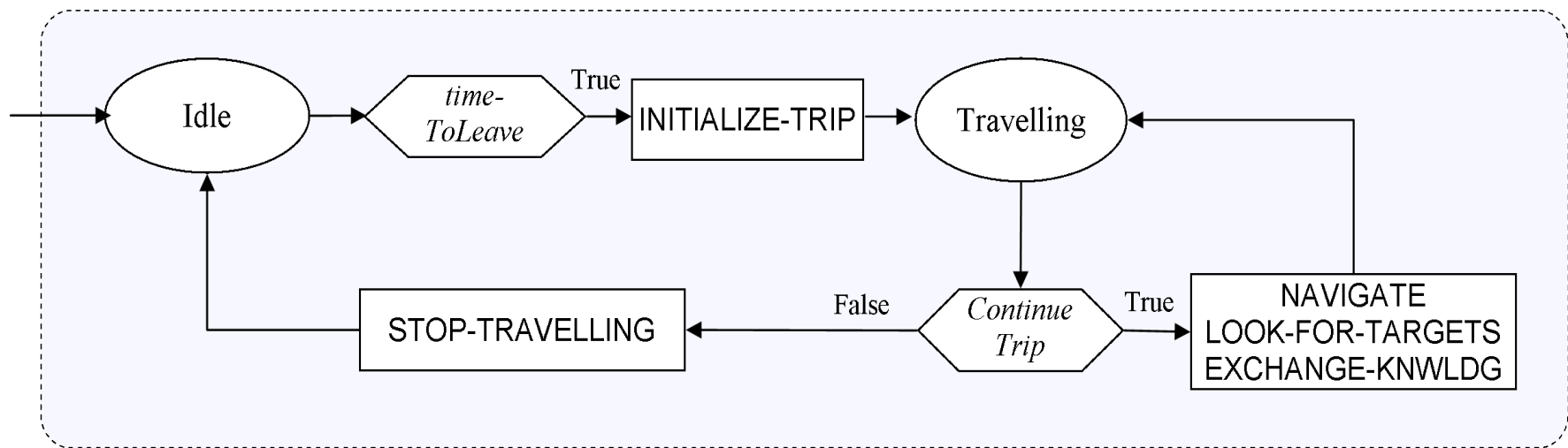
Plotter Plugin



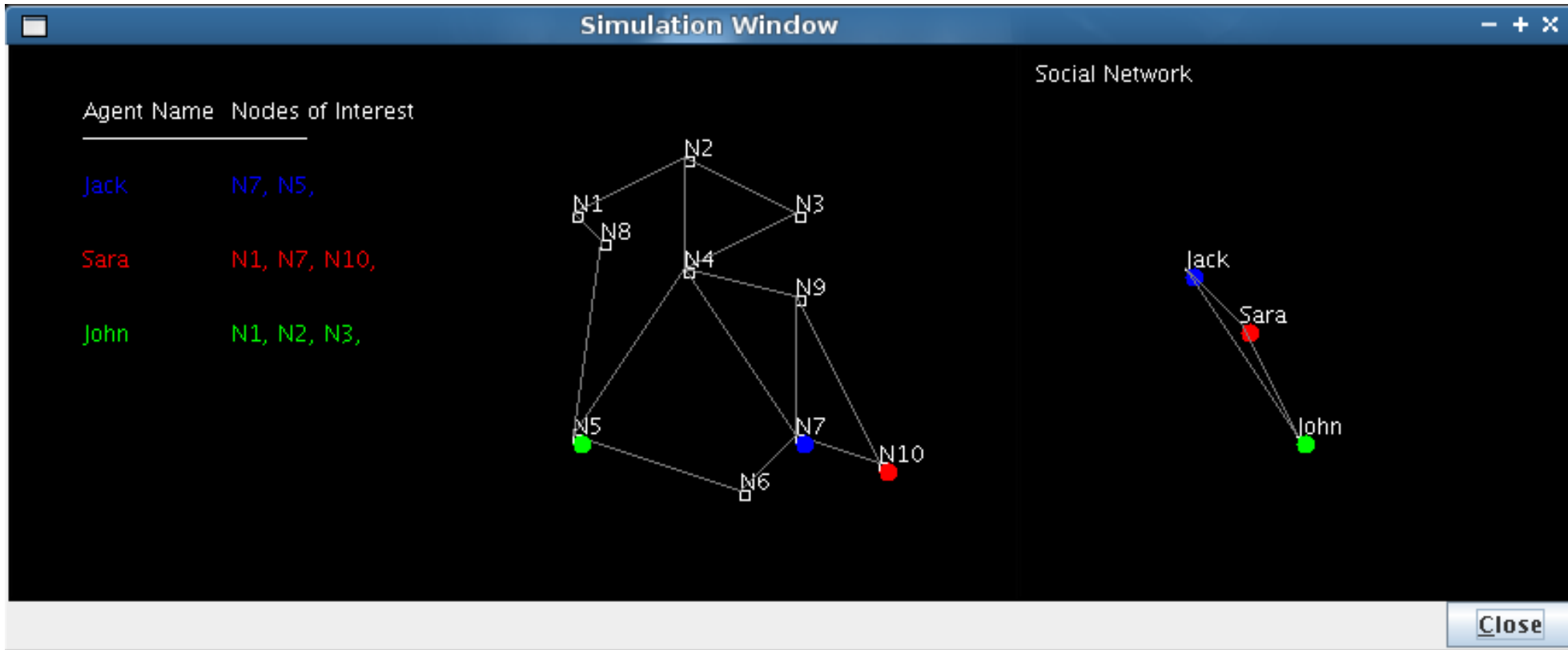
Mastermind

- An interdisciplinary research project in Computational Criminology
- Motivation:
 - Unifying framework for integrating crime theories
 - Computational modeling and simulation in studying crime
- Mastermind and ASM
 - A well-defined semantic foundation
 - Communication Problem
 - Experimental validation

Person Agents in Mastermind



Mastermind Plugin



IO Plug-in

- Provides a simple input and output
- Extends the state of the simulated machine by providing the following functions:
 - *input*: `STRING` → `STRING`
 - *output*: `STRING`
- It provides an extension point to the environment through which the environment can provide input to the IO Plug-in.

IO Plug-in

- IO Plug-in extends the *Interpreter* and the *Parser* to provide the following rule form:
 - **print** *value*
- IO Plug-in implements the *Aggregator* interface to aggregate **print** updates
- It also extends the control state ASM of the engine to keep a history of the updates

```
if input("Name:") = "John" then  
    print "Hello " + input("Name:") + "!"
```

JASMine

- A new background whose elements are Java objects
- New rule forms to create new instances, reading/writing object fields, invoking methods, etc.
- Java world seen as part of the environment; standard semantics of ASM step preserved
- Useful to
 - access Java libraries
 - implement complex algorithms in Java
 - write ASM test drivers for Java implementations

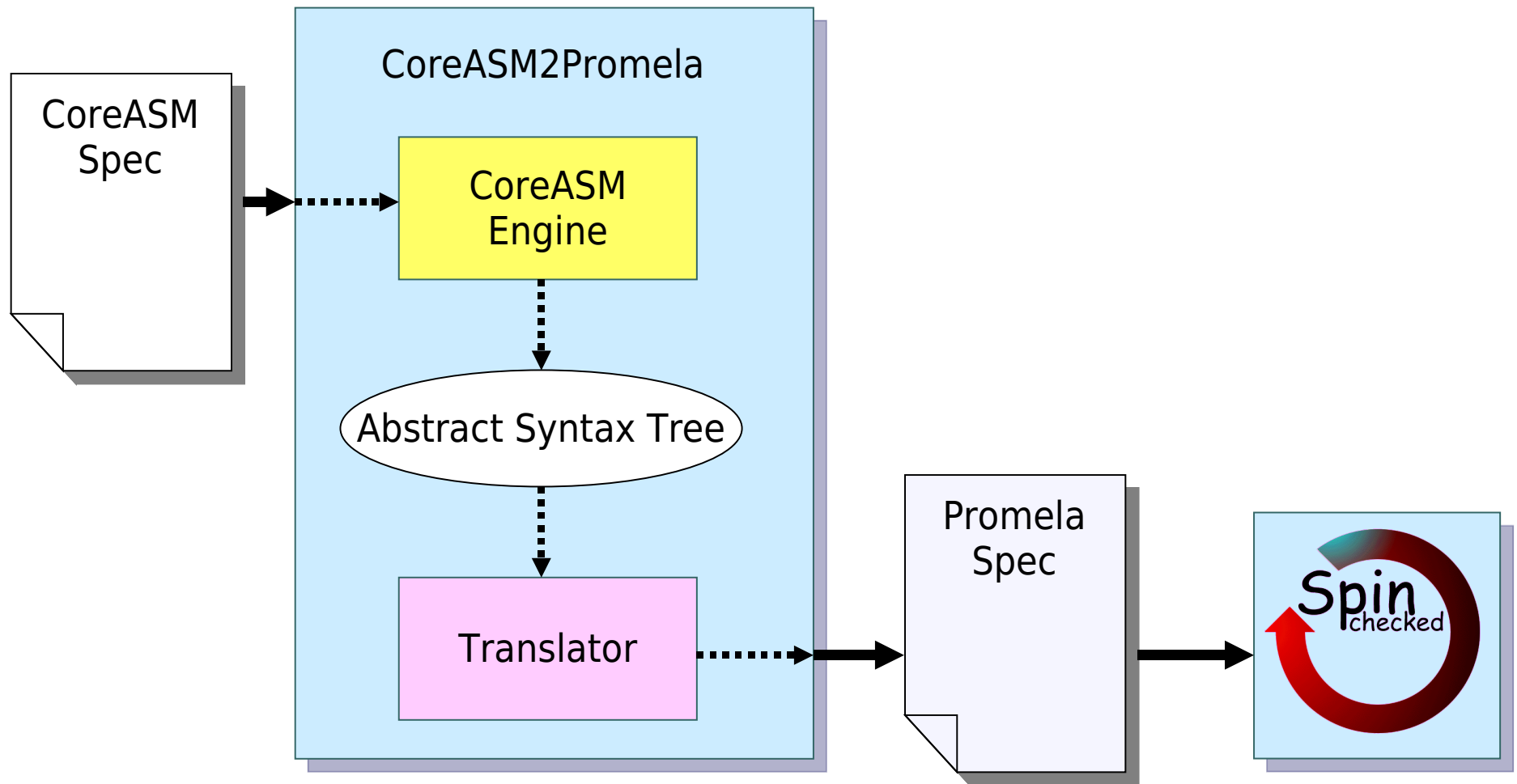
JASMine

- JASMine rule forms blend into the rest of the language

```
CoreASM JASMineExample
use StandardPlugins
use JASMinePlugin           // uses the JASMine plug-in
init InitRule

rule InitRule =
  if mode = undef then
    mode := 1
    import native org.jasmine.example.Foo into foo    // new Foo()
  else if mode = 1 then
    mode := 2
    invoke foo->setMsg("How are you?")                // foo.setMsg(...)
    invoke foo->getTime() result into t                // t=foo.getTime()
  endif
```

Model Checking CoreASM Specs



Extending the CoreASM Engine

- We needed to extend the CoreASM engine:
 - Adding type information to CoreASM
 - An untyped language is good for initial specifications
 - Type information and function signatures:
 - improve understandability
 - allow for runtime type checking
 - facilitate model checking
 - Support documenting correctness properties in CoreASM specifications

Signature Plug-in

- Signature Plug-in adds type information to CoreASM functions.
- Universe declaration

```
universe People = {Bob, Sue, Tom}  
enum DIRECTION = {North, East, South, West}
```

- Function declarations

```
function owner :Resource -> Agent  
function dropped :-> BOOLEAN  
function messageCount :-> NUMBER initially 0
```

Property Plug-in

- Allows LTL properties to be included in CoreASM specifications

```
property G(request → F response)
```

- Improves the usability of Spin
 - In Spin, properties are defined by describing the behavior of a property automaton
 - You can define more than one property

Model Checking Experiments

- Termination Detection Algorithm
 - ASM model by Robert Eschbach (1999)
- FLASH Cache Coherence Protocol
 - Coordinates sharing of memory among the processing nodes of the Stanford FLASH multiprocessor
- i-Protocol
 - An optimized sliding window protocol used in GNU Unix to Unix Copy (UUCP)

Current State

- ASM specification of
 - The Kernel and basic ASM and Turbo ASM rules
 - Various background plug-ins
 - JASMine Plugin
- Java implementation of
 - The Kernel
 - Major rule forms: *choose, forall, let, iterate,...*
 - Set, List, Stack, Queue, Number, and String plug-ins
 - IO plug-in, Signature plug-in, Math Plug-in,...

Ongoing Work

- Release: CoreASM engine version *0.9.1-beta*
 - Next release by the end of August 2007
- Many standard plug-ins can be improved.
- CoreASM user interface is still simple
 - A nice Integrated Modeling Environment
- Model checking tool can be improved
- There are lots of ideas for new plug-ins!
- Literate Programming
- ...

Literate Programming

- Extracting CoreASM specification fragments from
 - OpenOffice (ODF) documents (**this is done!**)
 - LaTeX source code
- Executable specification and documentation extracted from the same compound document
 - “Executable papers”
 - Specification as an *explanation* of how a system work

Verification and validation

- Verification rule forms implemented by a verification plug-in
- **assert** - checks that a property holds in a specific state (when **assert** is executed)
- **invariant** - checks that a property holds in all the states traversed by a computation
 - ' a, a' ' notation to refer to the value of a in the previous and next states
- **precondition** and **postcondition** for rule execution

Verification and validation

- Validation performed by executability
 - full traceability: states, updates, rules executed can be logged at each step and analyzed
 - simulation: I/O instructions allow interaction with a user, exploring the behavior of the specification in different cases
 - test drivers: the CoreASM engine can be called from Java code
 - e.g., integration in a JUnit test suite

Open Issues

- **Debugging:** traditional debugging models do not suite ASMs
 - no current instruction, no stepping, etc.
 - new UI paradigms need to be investigated
- **Typing:** CoreASM typing model is not just constructive; plug-ins can provide *more* than type combinators
- **Development cycle:** integration with model-based development or RAD / eXtreme Programming practices

visit us

www.coreasm.org





visit us

www.coreasm.org