

Executing ASM Models with CoreASM

Roozbeh Farahbod and Vincenzo Gervasi

Software Technology Lab, SFU, Canada
Dipartimento di Informatica, Università di Pisa, Italy

Joint work with Uwe Glässer, George Ma, and Mashaah Memon

Agenda

- Introduction
 - Goals and philosophy
- Architecture
- Extensibility
 - Input and output
 - JASMine
- Eclipse IDE
- Demo
- Model Checking
- Work in progress
 - Literate programming
 - Verification & Validation
- Conclusions

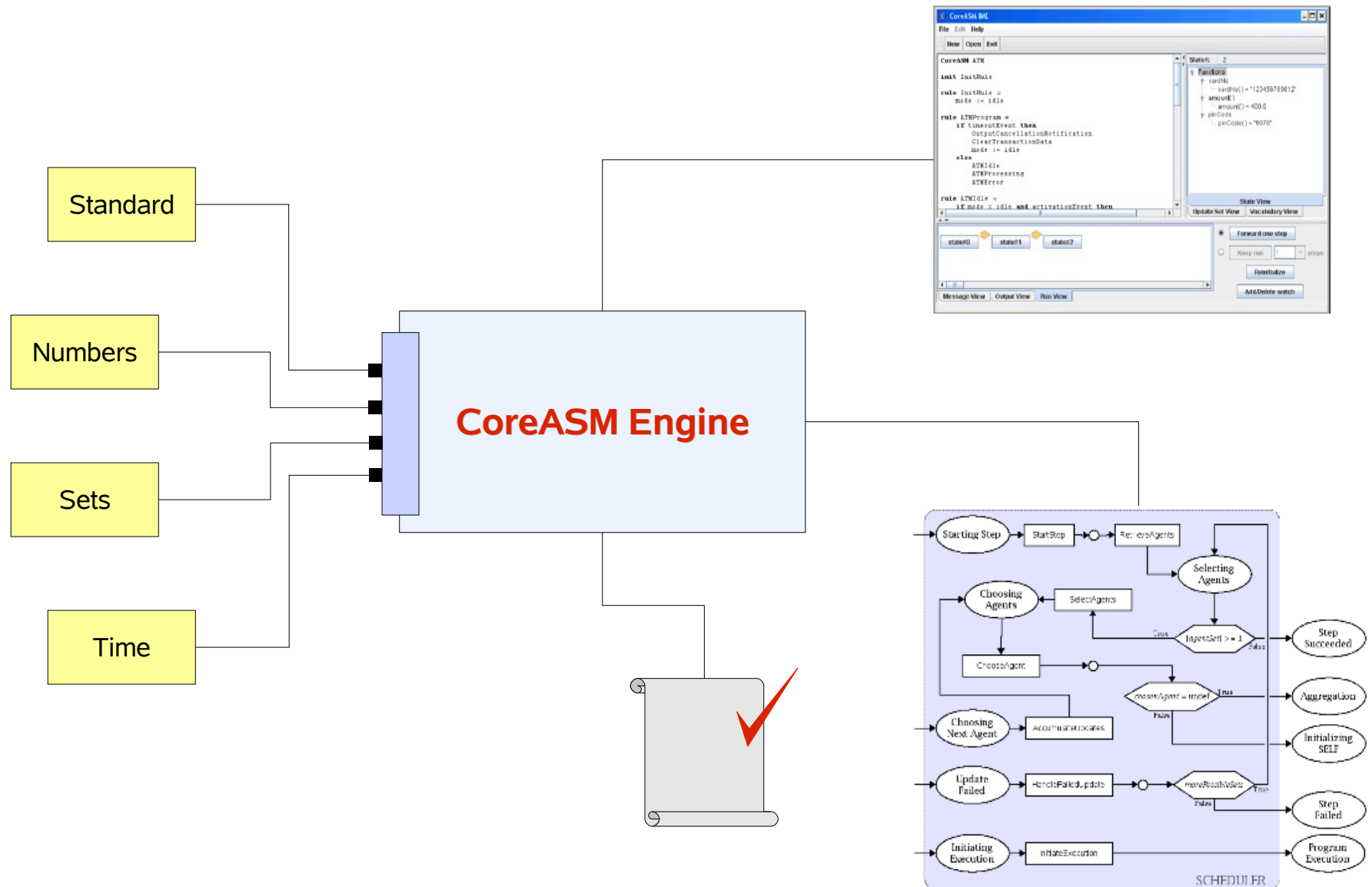
Introduction – Goals

- 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
 - High-level design
 - Experimental validation, fast prototyping
 - Formal verification

CoreASM Guiding Principles

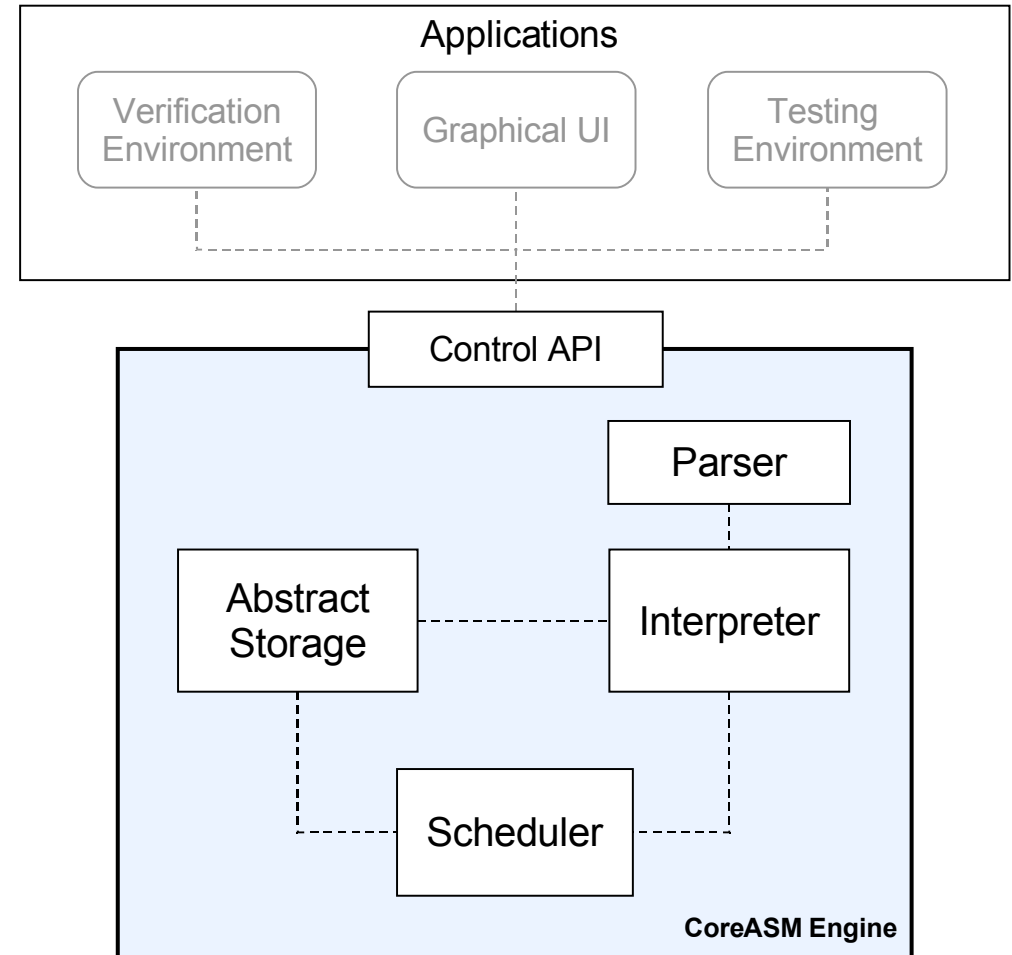
- Preservation of pure ASM semantics
- Ensuring freedom through extensibility
- Support experimentation with high-level specifications
 - “rapid prototyping” of specifications
 - design exploration
- Provisions for the many roles of ASM specs
 - descriptions for humans to read
 - modeling existing systems to prove properties
 - stubs for unimplemented components of a system
 - drivers for implemented components (test harness)

Kernel of a Novel Environment



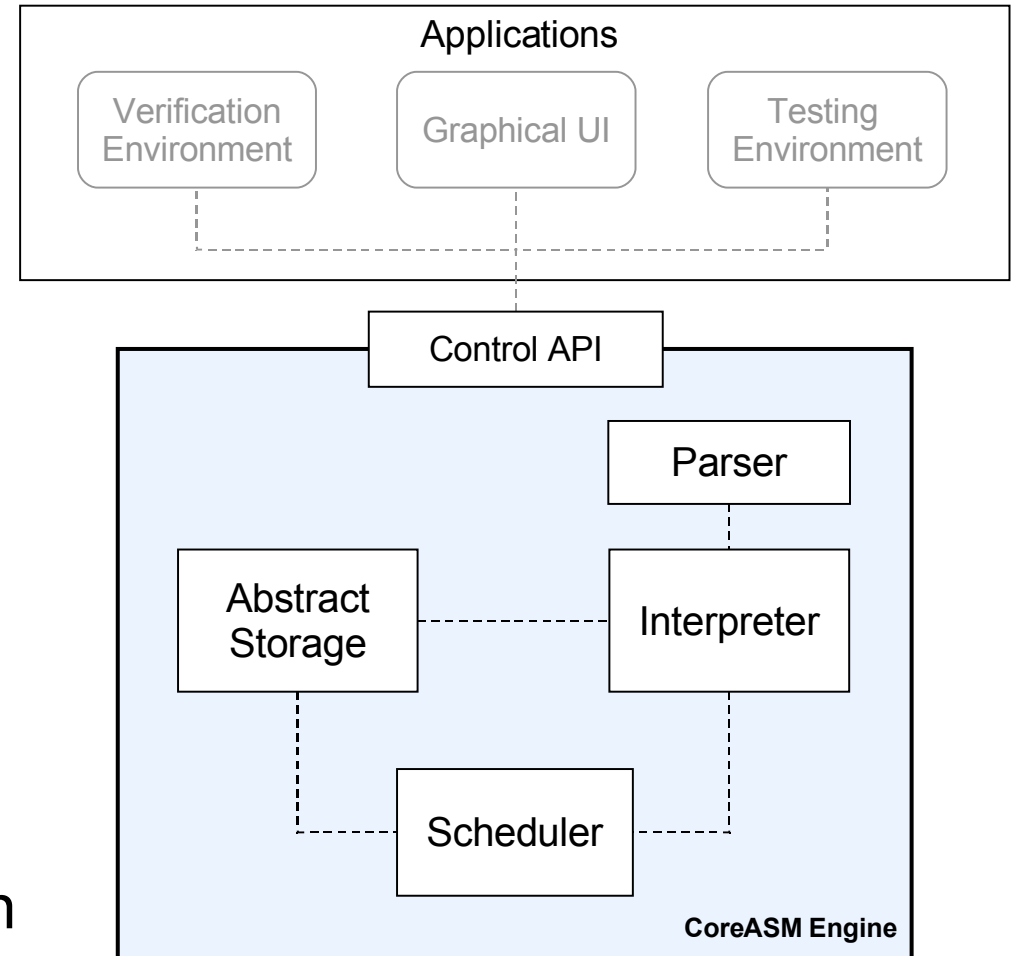
The Architecture

- Control API:
 - interface to the environment
 - interface to the engine
- Parser
 - builds an annotated Abstract Syntax Tree
 - based on grammar fragments contributed by plug-ins



The Architecture

- Abstract Storage
 - a representation of the current state
- Interpreter
 - generates an update set, given an AST and the current state
- Scheduler
 - Orchestrates every computation step
 - Organizes the execution of agents



A Micro-kernel Approach

- A micro-kernel approach
 - Kernel provides the bare minimum structure
 - All advanced features are provided by plug-ins
 - Standard ASM features are provided by plug-ins in the standard library
 - Custom extensions can be implemented by custom plug-ins; e.g.,
 - Custom Backgrounds
 - Special Rule forms

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

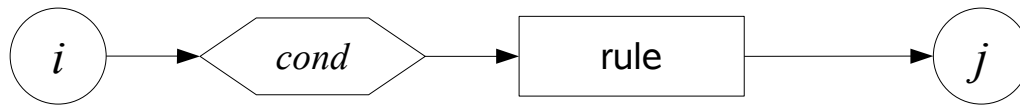
CoreASM Plug-in Framework

- Supports two extension mechanisms:
 - Extending the functionality of specific components
 - Extending the control state ASM of the engine
- Thus, plug-ins:
 - provide new or extend the existing grammar rules
 - provide interpretation
 - extend the vocabulary of the engine
 - extend the engine's execution cycle

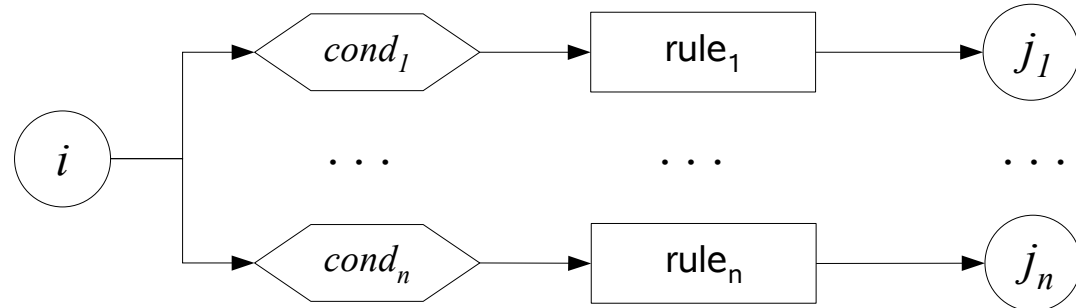
CoreASM Plug-in Interfaces

Plug-in Interface	Extends	Description
<i>Parser Plug-in</i>	Parser	provides additional grammar rules to the parser
<i>Interpreter Plug-in</i>	Interpreter	provides new semantics to the interpreter
<i>Operator Provider</i>	Parser, Interpreter	provides grammar rules for new operators along with their precedence levels and semantics
<i>Vocabulary Extender</i>	Abstract Storage	extends the state with additional functions, universes, and backgrounds
<i>Aggregator</i>	Abstract Storage	aggregates partial updates into basic updates
<i>Scheduler Plugin</i>	Scheduler	provides new scheduling policies for multi-agent ASMs
<i>Extension Point Plugin</i>	all components	extends the control state model of the engine

Extensible Control State ASMs (1)

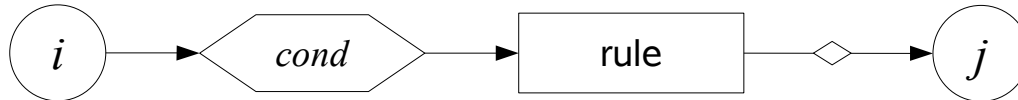


$\text{FSM}(i, \text{if } cond \text{ then } rule, j) \equiv$
 $\text{if } ctl_state = i \text{ and } cond \text{ then}$
 $rule$
 $ctl_state := j$



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

Extensible Control State ASMs (2)

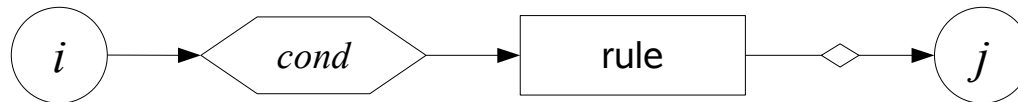


EFSM(i , if $cond$ then $rule$, j) \equiv
if $ctl_state = i$ and $cond$ then
 $rule$ seq Proceed(i , j)

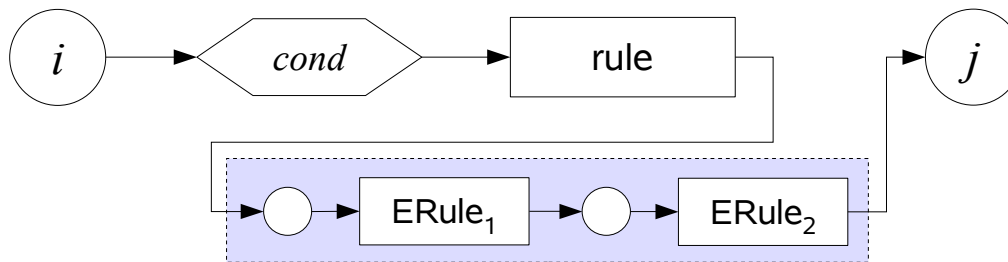
Proceed(i , j) \equiv
forall $p \in extensionPointPlugins$ do
 $marked(p) := isRegistered(p, i, j)$
 seq
 iterate
 choose $p \in extensionPointPlugins$ with $marked(p)$ do
 $marked(p) := false$
 let $R = extensionRule(p)$ in
 $R(i, j)$
 seq
 $ctl_state := j$

Extensible Control State ASMs (3)

An EFSM of the form

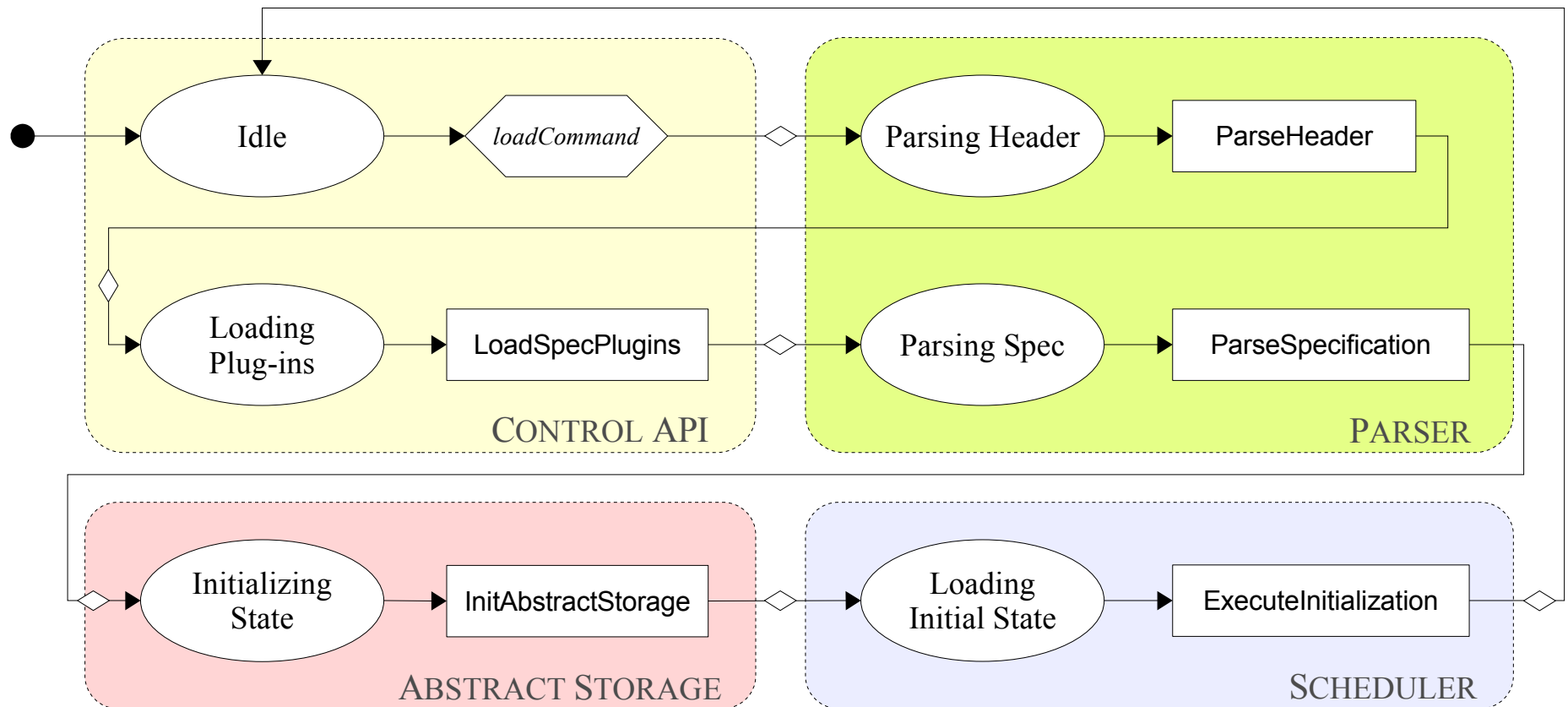


with two extensions, would expand to



Extension Points

Example: Loading Specifications



Example: IO Plug-in

- IO Plug-in provides a simple input and output mechanism using *monitored* and *out* functions.
- It extends the state of the simulated machine by providing the following functions:
 - *input*: `STRING` \rightarrow `STRING`
 - *output*: `STRING`
- It provides an extension point to the environment through which the environment can provide input to the IO Plug-in.

Example: 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:") + "!"
```

Example: 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

Example: 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
```

Eclipse Plug-in for CoreASM

- An Eclipse plug-in implementing an Integrated Development Environment for CoreASM
- Supports the specification writer
 - syntax highlighting
- Interactive execution of specifications
- CoreASM and Java side-to-side in the same GUI

Demo Time!

Carma, Eclipse Plug-in, IO Plugin, Plotter,...

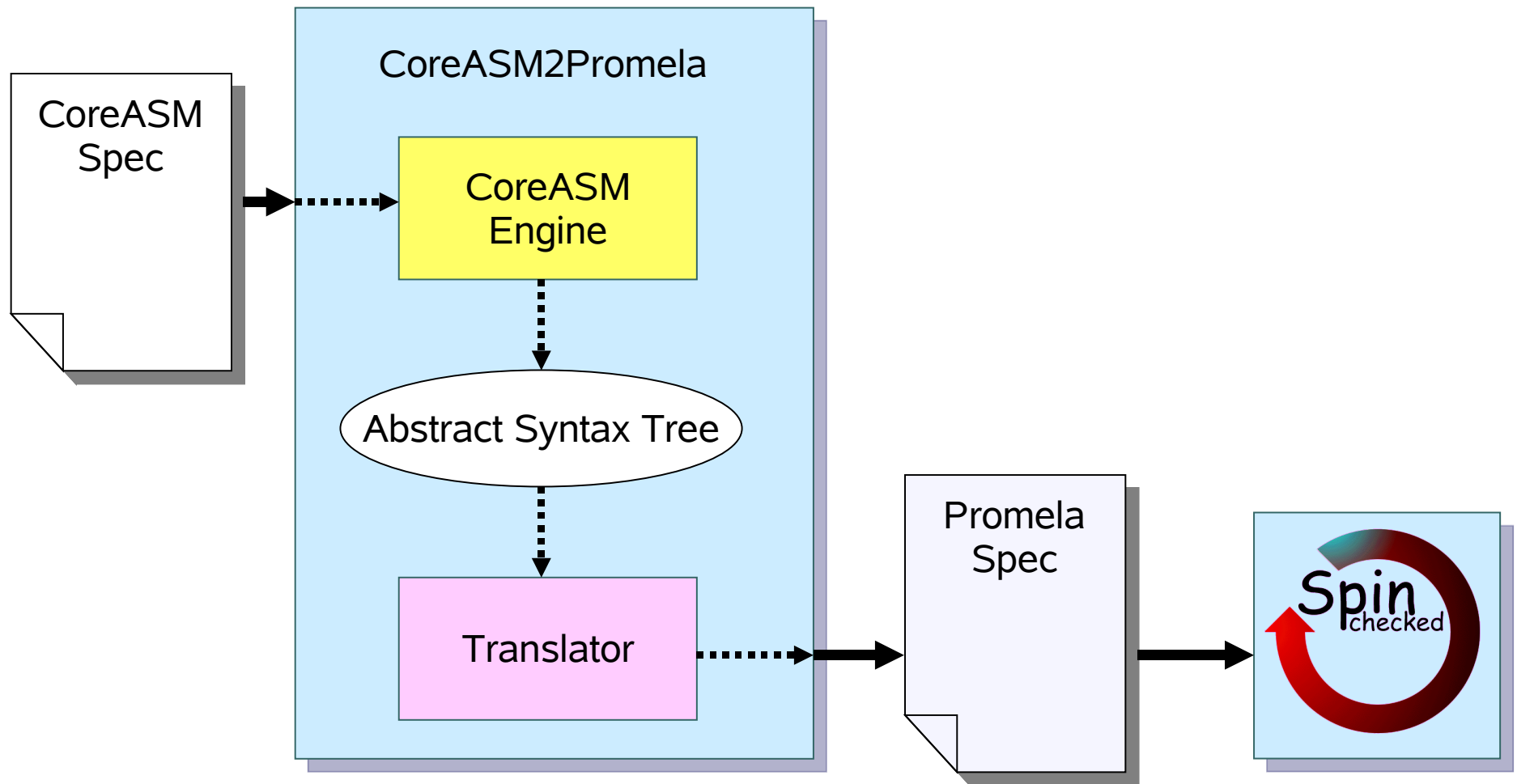
CoreASM to Promela

- Model check CoreASM specifications using SPIN
- Extends Gargantini's translation scheme to support more rule forms, n-ary functions, and distributed ASMs.
- Properties to be checked are part of the CoreASM spec.

Why SPIN?

- SPIN has been used to model asynchronous software systems.
- Promela, SPIN's input language, has high level programming constructs, which makes translation easier.
- DASM agents can be modeled as SPIN processes.

Translation Process



Backgrounds and Functions

- Backgrounds and Universes
 - Support for backgrounds (static universes) of finite size.
 - Elements are mapped to integers.
- Functions
 - Nullary functions are translated to integer variables.
 - N-ary functions are translated to multi-dimensional arrays
 - Each controlled function is translated to two Promela variables, one for the current state and one for the next state.

Rules and Macros

- Supported Basic Rules:
 - assignment, block, conditional, forall, and choose
- Named Rules:
 - Agent program rules are translated as Promela *proctypes*.
 - Macro rules are translated as Promela in-lines.

Simulating CoreASM Runs

```
init {  
    atomic {  
        functionInit();  
        run_program_1();  
        ...  
        run_program_n();  
    };  
    do ::  
    atomic {  
        updateMonitoredFunctions();  
        if  
        :: program_1_chan!start;  
        ...  
        :: program_n_chan!start;  
        fi;  
        fireUpdates();  
    };  
    od;  
}
```

- Agent interleaving is simulated using non-deterministic Promela “if” statement

Current State

- ASM specification of
 - The Kernel and basic ASM and Turbo ASM rules
 - Set Plugin and Number Plugin
- Java implementation of
 - The Kernel
 - Major rule forms: *choose*, *forall*, *let*, *iterate*, *while*,...
 - Set, Number, and String plug-ins
 - IO plug-in, Signature plug-in (basic version), ...
- Graphical User Interface (currently disjoint)

Ongoing Work

- CoreASM engine version 0.9.1-beta is released.
- Many standard plug-ins can be improved.
- CoreASM user interface is very simple
 - A nice Integrated Modeling Environment
- Model checking tool is a work under progress
- There are a lot of ideas for new plug-ins!
- Literate Programming
- ...

Literate Programming

- Extracting CoreASM specification fragments from
 - OpenOffice (ODF) documents
 - 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

Final Remarks

- CoreASM guiding principles:
 - Preservation of pure ASM semantics
 - Ensuring freedom through extensibility
- Model-based engineering of abstract requirements in early phases of design
- A platform-independent open source project

www.coreasm.org