

The CoreASM Project

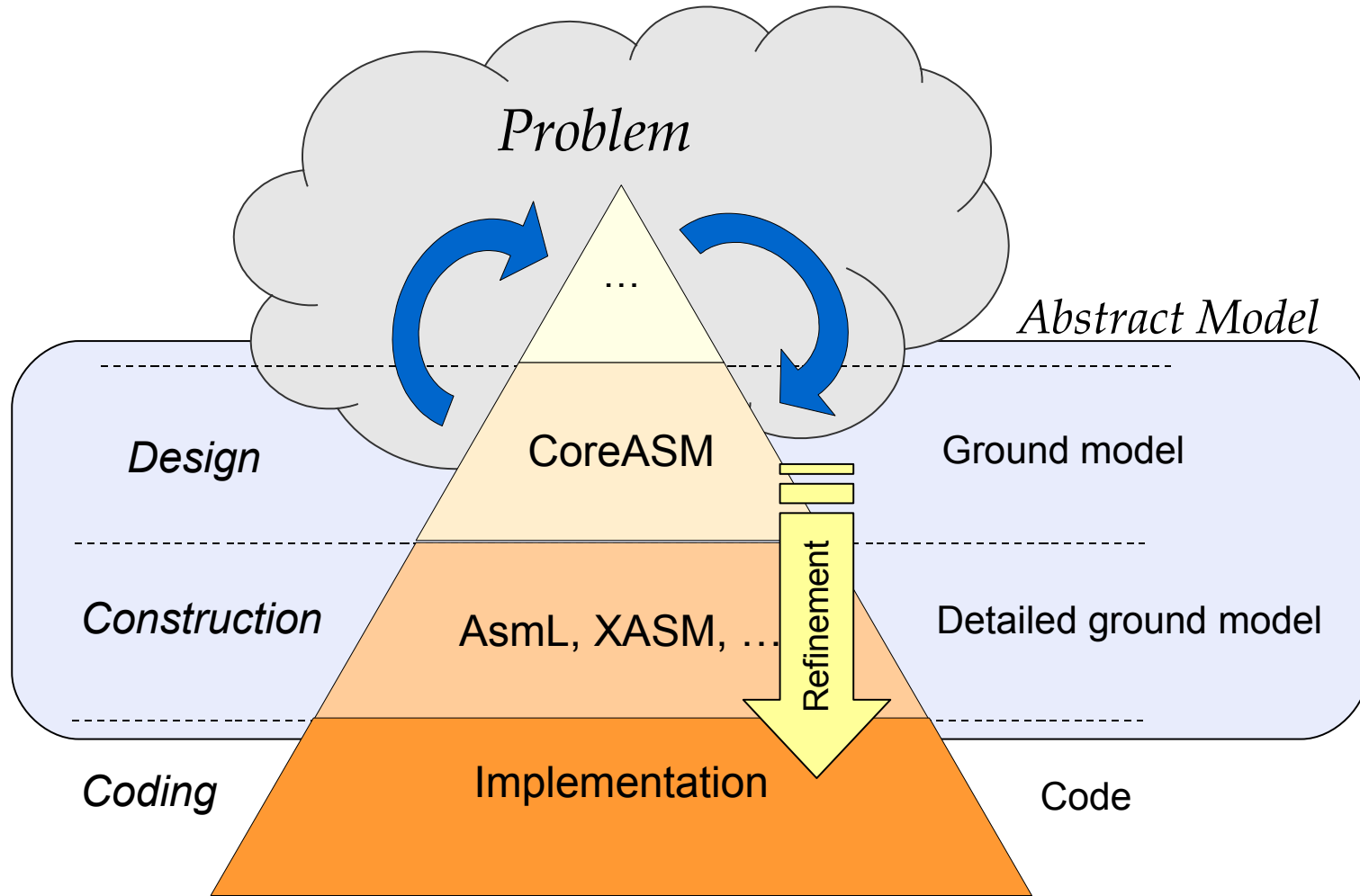
An Extensible ASM Execution Engine

Roozbeh Farahbod

Software Technology Lab
School of Computing Science
Simon Fraser University
Canada

Joint work with Vincenzo Gervasi, Uwe Glässer, and Mashaal Memon

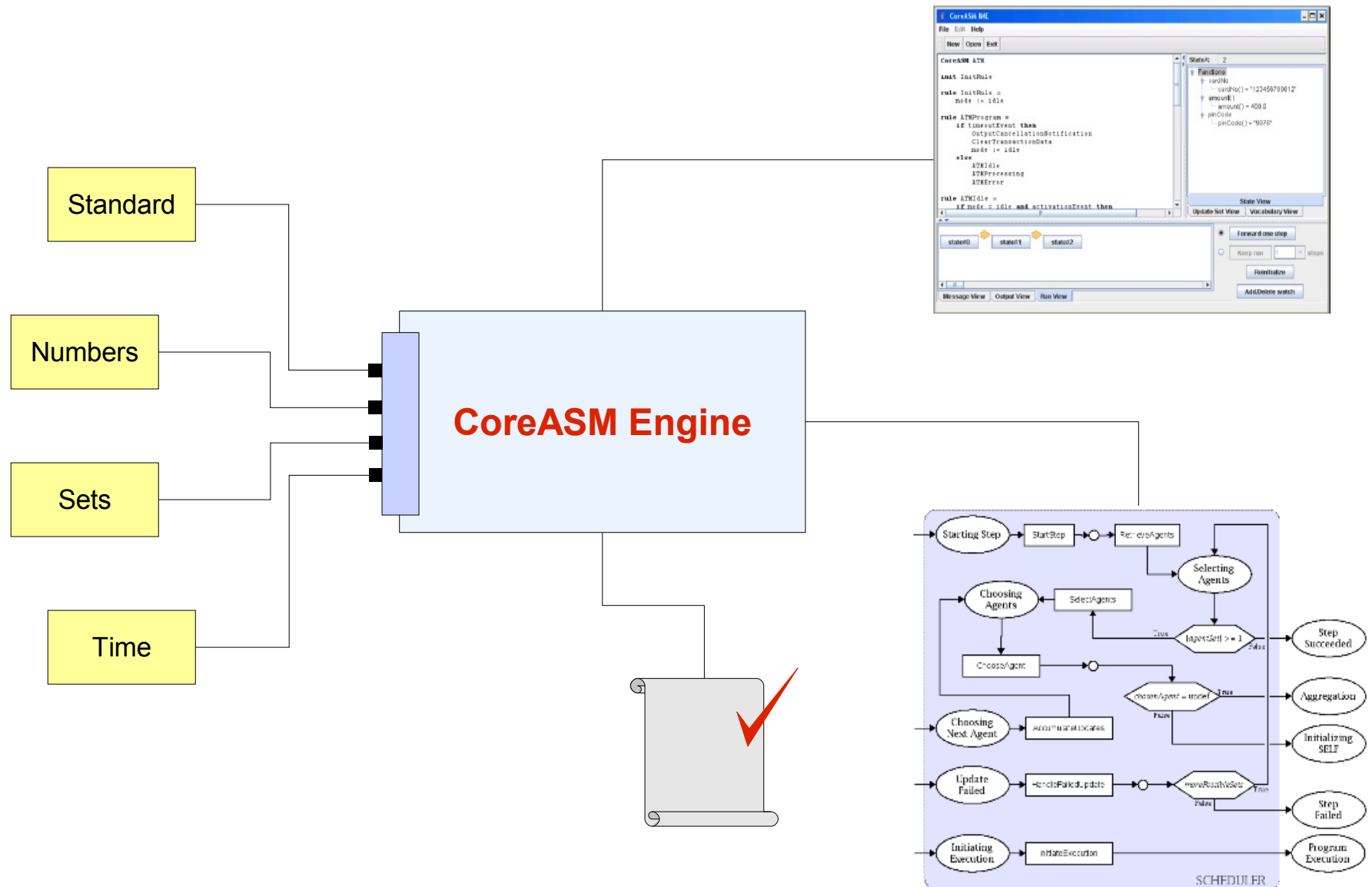
The Idea



The CoreASM Project

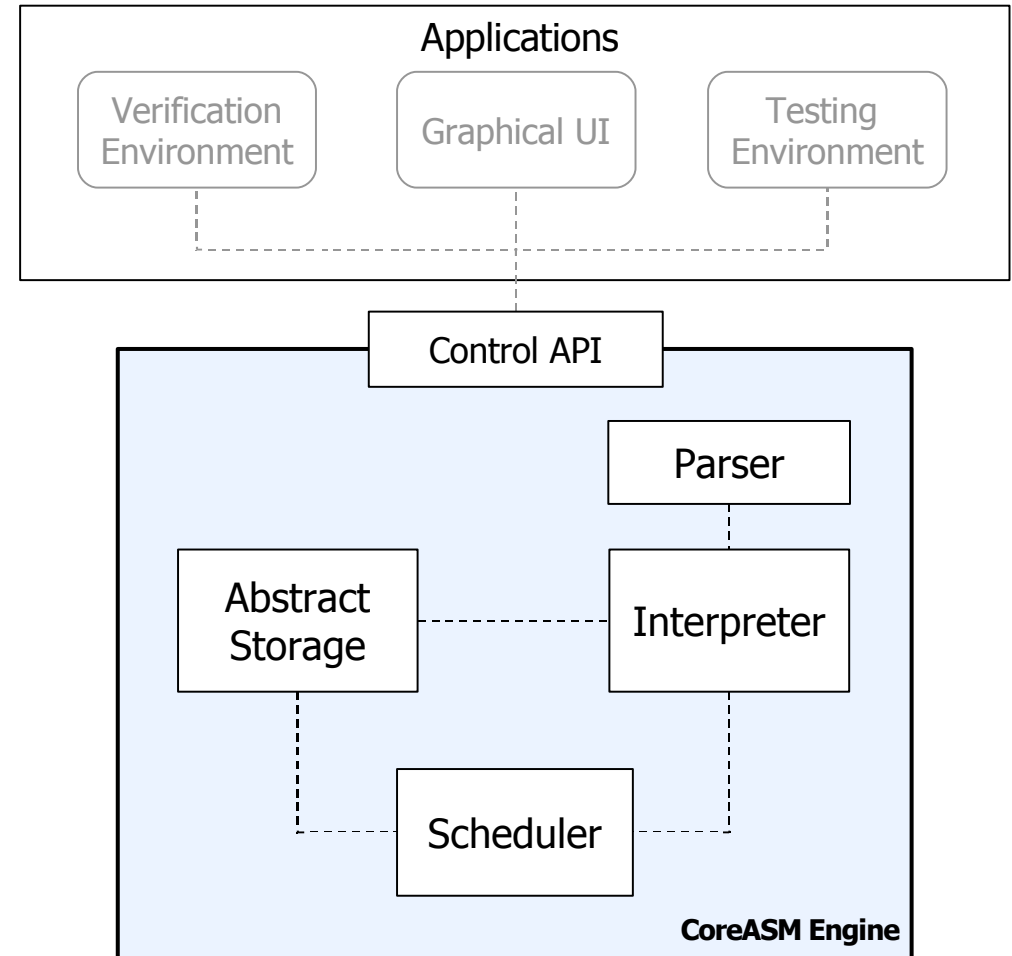
- A *lean, executable, and extensible* ASM language
- An extensible platform-independent engine
- A supporting tool environment for
 - High-level design
 - Experimental validation
 - Formal verification

Kernel of a Novel Environment



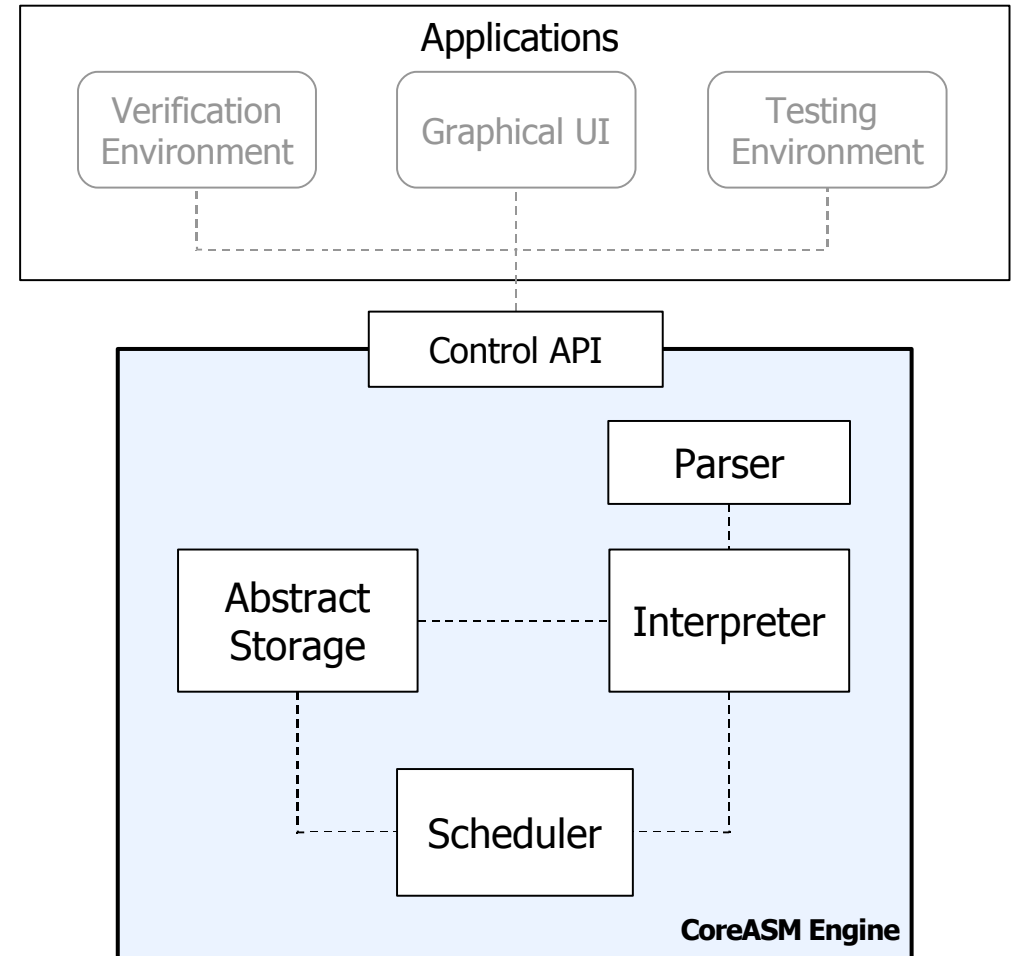
The Architecture

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



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 plugins
 - Standard ASM features are provided by plugins in the standard library
 - Custom extensions can be implemented by custom plugins; e.g.,
 - Custom Backgrounds
 - Special Rule forms

The Kernel

- Abstract Storage
 - Element, Function, Rule, Universe, and Background
 - Aggregation and composition interfaces
 - Boolean values
 - *Agents, program(a), self*
- Interpreter
 - Expressions: Boolean values and function terms
 - Operators: equality
 - Rules: assignment, macro call, and import

Pattern Rules

- The interpreter is specified by a set of rules of the form

$$(\textit{pattern}) \rightarrow \textit{actions}$$

- which are interpreted as

if *conditions* **then** *actions*

- where *conditions* are derived from the *pattern*.

Pattern Rules: Example

- Evaluating a function with arguments

$$\langle \langle \alpha x(\lambda_1 \boxed{?}_1, \dots, \lambda_n \boxed{?}_n) \rangle \rangle$$
 \rightarrow

```
if isFunctionName(x) then
  choose  $i \in [1..n]$  with  $\neg \text{evaluated}(\lambda_i)$ 
   $pos := \lambda_i$ 
ifnone
  let  $l = (x, \langle value(\lambda_1), \dots, value(\lambda_n) \rangle)$  in
     $[[pos]] := (l, undef, getValue(l))$ 
if undefined(x) then
  HandleUndefinedIdentifier(x,  $\langle \lambda_1, \dots, \lambda_n \rangle$ )
```

where

$$undefined(x) \equiv \nexists e \in \text{ELEMENT} : name(e) = x$$
$$isFunctionName(x) \equiv \exists e \in \text{ELEMENT} : name(e) = x \wedge isFunction(e)$$

Extensible Engine

- A micro-kernel approach
 - Kernel provides the bare minimum structure
- Language extensions
 - plugins extend the kernel providing
 - Backgrounds
 - Rule forms
- Extension Points
 - Facilitates extensions to the execution cycle
 - Preprocessing the AST
 - Monitoring updates

The Plugin Architecture

- Plugins
 - provide new or extend the existing grammar rules
 - provide interpretation
 - extend the vocabulary of the engine
 - extend the engine's execution cycle
- Plugin interface
 - Abstract Plugin
 - Interpreter Plugin
 - Parser Plugin
 - Scheduler Plugin
 - Vocabulary Plugin
 - Aggregator Plugin
 - Extension Point Plugin

Example: Block-Rule Plugin

- The Block-Rule plugin extends both the parser and the interpreter
 - extends the grammar of ASM rules
 - provides the semantics of block-rules
- The plugin is specified as

$$\langle \{ \lambda_1 \square; \dots; \lambda_n \square \} \rangle \rightarrow \text{choose } i \in [1..n] \text{ with } \neg \text{evaluated}(\lambda_i)$$
$$pos := \lambda_i$$

ifnone

$$\llbracket pos \rrbracket := (undef, \bigcup_{i \in [1..n]} \text{updates}(\lambda_i), undef)$$

Example: Block-Rule Plugin

```
public Node interpret(Node pos) {
    String token = pos.getToken();

    if ((token != null) && (token.equals("par"))) {
        Node currentRule = pos.getFirst();

        while (currentRule != null) {
            if (!currentRule.isEvaluated()) {
                return currentRule;
            }
            currentRule = currentRule.getNext();
        }

        currentRule = pos.getFirst();
        UpdateMultiset updates = new UpdateMultiset();

        while (currentRule != null) {
            updates.addAll(currentRule.getUpdates());
            currentRule = currentRule.getNext();
        }

        pos.setNode(null, updates, null);
        return pos;
    }
    else {
        return null;
    }
}
```

Example: SET Plugin

- The SET plugin provides
 - the *background* of **Set**
 - extending the abstract storage with a new background
 - extending the parser and the interpreter with new syntax and semantics for *set literals*, *set enumeration*, and *set comprehension*
 - set-related *operations*
 - Union, intersection, difference, ...
 - definition of set specific rule forms
 - *add-to* and *remove-from*
 - set-specific aggregation and composition

SET Plugin: Definitions

- $setBack \in \text{BACKGROUND}$
- $name(setBack) = \text{"Set"}$
- $newValue(setBack) =$ a SETELEMENT representing the empty set
- $\forall s \in \text{SETELEMENT}, bkg(s) = \text{"Set"}$.
- $setMember : \text{SETELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$

$(\{ \lambda_1 [?]_1, \dots, \lambda_n [?]_n \})$

\rightarrow

```
choose  $i \in [1..n]$  with  $\neg evaluated(\lambda_i)$   
   $pos := \lambda_i$   
ifnone  
  let  $newSet = newValue(setBack)$  in  
    forall  $i \in [1..n]$   
       $setMember(newSet, value(\lambda_i)) := true$   
   $[[pos]] := (undef, undef, newSet)$ 
```

Updates and Update Instructions

- In ASM, we can have elements with complex structures
- How should we handle *incremental changes* to complex structures?
- *Update instructions* instead of regular updates
 - Based on the idea of partial updates by Dr. Gurevich

Updates and Update Instructions

- Replace updates of the form

$\langle \text{LOC}, \text{ELEMENT} \rangle$

- with update instructions of the form

$\langle \text{LOC}, \text{ELEMENT}, \text{ACTION} \rangle$

- where *action* represents the intended incremental modification to the location.
- Finally, **aggregate** all the actions on locations to produce the *update set*.

Aggregation

- Each plugin is responsible for the aggregation of the actions it provides.

AggregateUpdates \equiv

$updateSet \leftarrow \text{Aggregate}(updateInstructions)$

Aggregate($uMset : \text{UPDATEMULTISET}$) \equiv

let $ap = \{a \mid a \in \text{PLUGIN} \wedge aggregator(a)\}$ **in**

seq

forall $p \in ap$ **do**

$resultantUpdates(p, uMset) \leftarrow \text{InvokeAggregation}(p, uMset)$

result $:= \bigcup_{p \in ap} resultantUpdates(p, uMset)$

Set Aggregation: Example

```
CoreASM SetExamples
```

```
use ConditionalRulePlugin
use BlockRulePlugin
use TabBlocksPlugin
use NumberPlugin
use SetPlugin
```

```
init InitRule
```

```
rule InitRule =
  a := {1, 4, 7}
  program(self) := ruleelement Main
```

```
rule Main =
  if setCardinality(a) = 3 then
    add 5 to a
    remove 1 from a
  endif
```

Set Aggregation: Example

- Initial state

Backgrounds:

- BOOLEAN
- FUNCTION
- NUMBER
- RULE
- SET

Universes:

- Agents: {Element1000015}

Functions:

- a
 - $a() = \{7.0, 1.0, 4.0\}$
- program
 - $\text{program}(\text{Element1000015}) = \text{rule-element Main}$
- self
 - $\text{self}() = \text{Element1000015}$
- setCardinality

Set Aggregation: Example

- After the first step

update instructions: $\{ \{ ((a, []), 5.0, \text{setAddAction}), ((a, []), 1.0, \text{setRemoveAction}) \} \}$

update set is $\{ ((a, []), \{5.0, 7.0, 4.0\}, \text{updateAction}) \}$

Universes:

- Agents: $\{\text{Element1000015}\}$

Functions:

- a

$a() = \{5.0, 7.0, 4.0\}$

- program

$\text{program}(\text{Element1000015}) = \text{rule-element Main}$

- self

$\text{self}() = \text{Element1000015}$

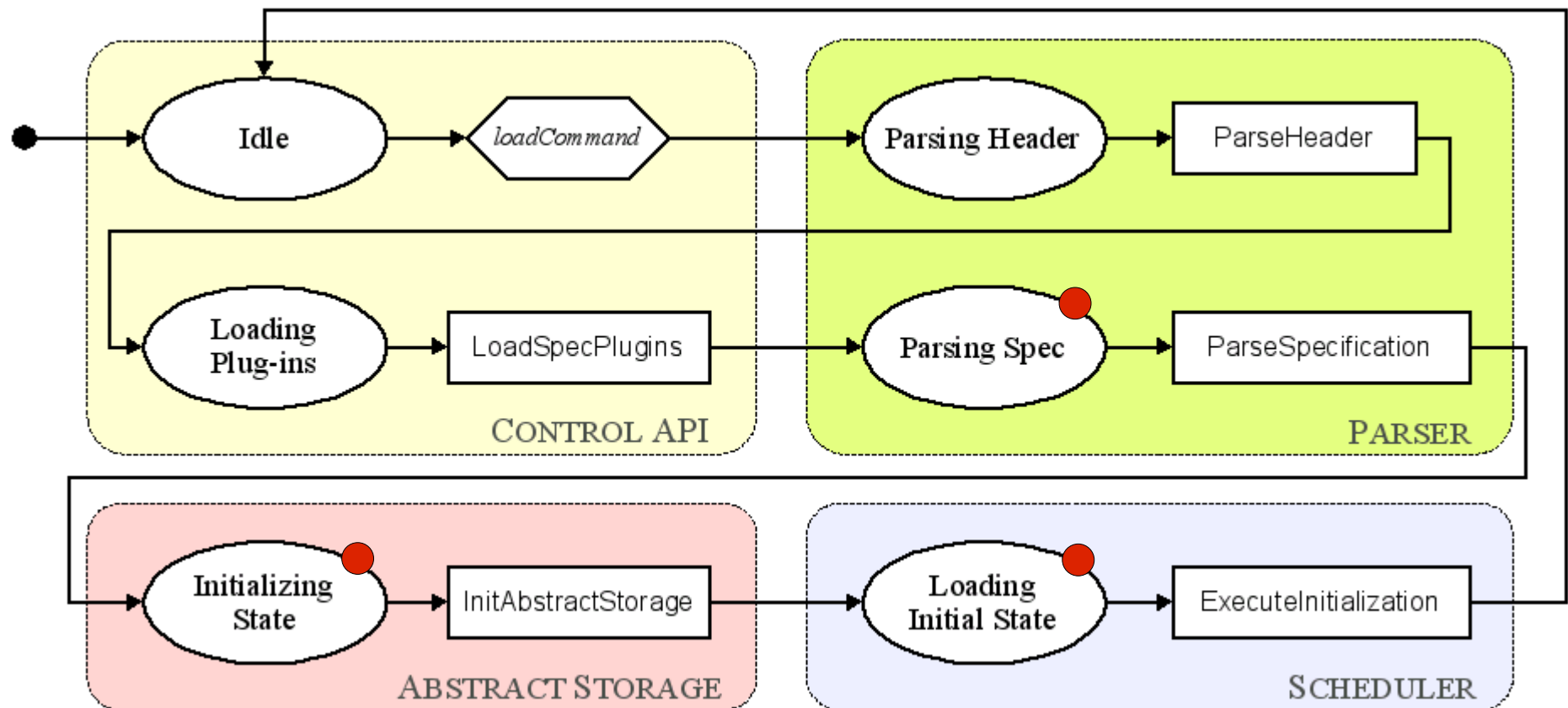
- setCardinality

Extensible Engine

- A micro-kernel approach
 - Kernel provides the bare minimum structure
- Language extensions
 - Plugins extend the kernel providing
 - Backgrounds
 - Rule forms
- Extension Points
 - Facilitates extensions to the execution cycle
 - Preprocessing the AST
 - Monitoring updates

Extension Points

Example: Loading Specifications



Extension Points

- Plugin Interface

$pluginSourceModes : \text{PLUGIN} \rightarrow \text{ENGINEMODE-SET}$

$pluginTargetModes : \text{PLUGIN} \rightarrow \text{ENGINEMODE-SET}$

rule FirePluginOnModeTransition($p, fromMode, toMode$)

- Mode Switching of the engine

Next($newMode : \text{ENGINEMODE}$) \equiv

seq

forall $p \in registeredPlugins(engineMode, newMode)$

 FirePluginOnModeTransition($p, engineMode, newMode$)

$engineMode := newMode$

$registeredPlugins : \text{ENGINEMODE} \times \text{ENGINEMODE} \rightarrow \text{PLUGIN-SET}$

$registeredPlugins(src, trg) =$

$\{p \mid p \in \text{PLUGIN} \wedge (src \in pluginSourceModes(p) \vee trg \in pluginTargetModes(p))\}$

Extension Point Plugin

```
public interface ExtensionPointPlugin {  
  
    public abstract Set<ControlAPI.EngineMode> getTargetModes ();  
  
    public abstract Set<ControlAPI.EngineMode> getSourceModes ();  
  
    public abstract void fireOnModeTransition (EngineMode source,  
                                               EngineMode target);  
  
}
```

Example: Tabbed Block Rules

- A simple block rule plugin may require **par** and **endpar**; e.g.,

```
par
  a := 1
  b := 2
endpar
```

- Using the extension points, a plugin can
 - register itself to be called before the parsing mode
 - read the indentation and convert them to **par-endpar** blocks

Current State

- ASM specification of
 - The Kernel
 - Basic ASM and Turbo ASM Rule forms
 - Set Plugin and Number Plugin
- Java implementation of
 - The Kernel (minus some low-priority functions)
 - Block rule and conditional rule forms
 - Set and Number plugins (only the main features)
- Graphical User Interface (currently disjoint)

Ongoing Work

- The Engine
 - Complete implementation of the Kernel
 - Implementation of Set and Number plugins
 - Implementation of other rule forms
 - forall, choose, seq, iterate, ...
 - Signature Plugin providing type-checking
- Applications
 - Integrated Modeling Environment
 - Model Checking Tool

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