Design Exploration and Experimental Validation of Abstract Requirements

Roozbeh Farahbod[1]

Vincenzo Gervasi[2]

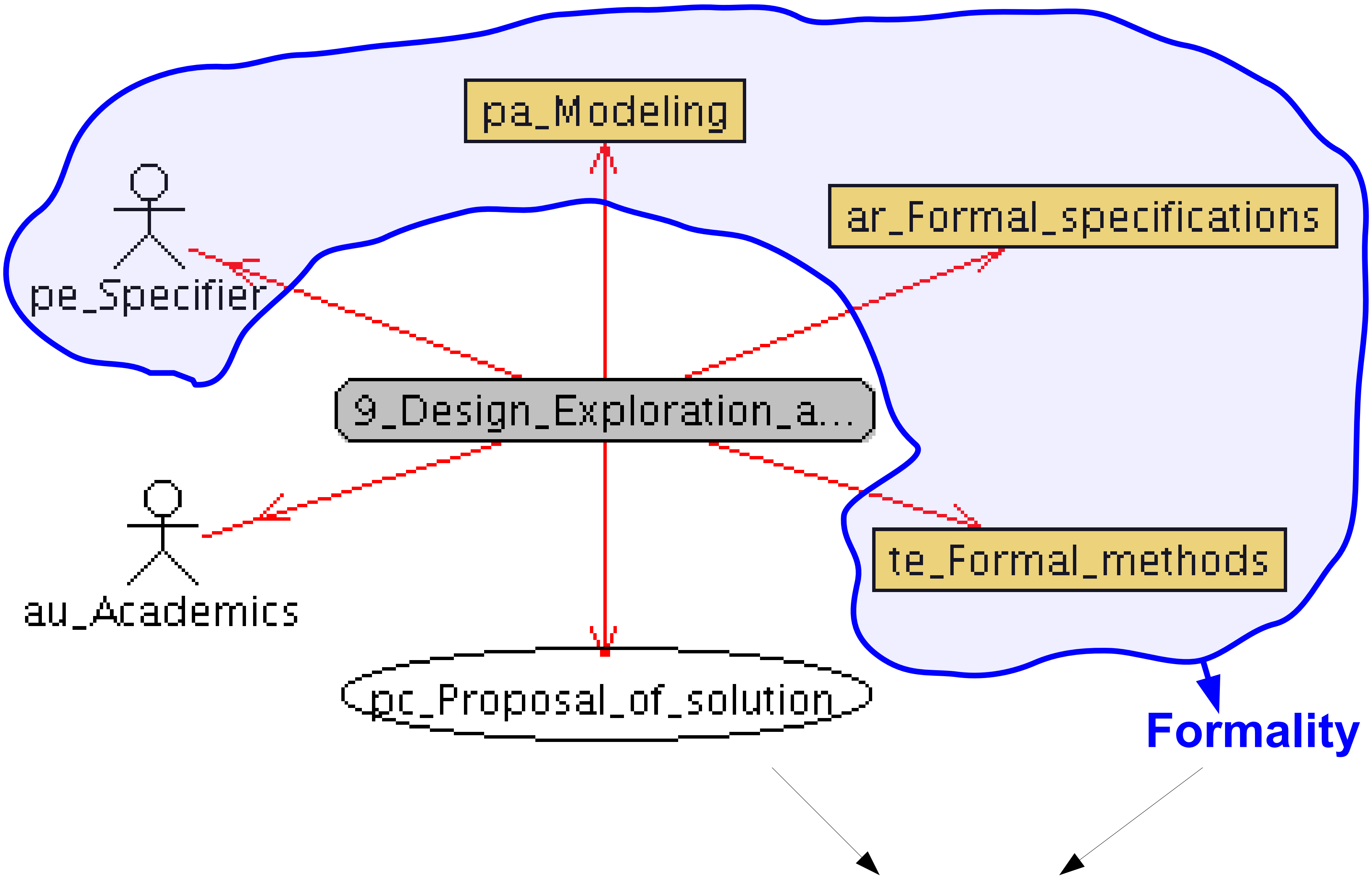
Uwe Glaesser[1]

Mashaal Memon[1]

[1] Simon Fraser University, Vancouver, BC

[2] University of Pisa, Italy

First slide



... but with controllable costs!

Talk outline

- Motivations
- Abstract State Machines in a nutshell
- CoreASM: an executable ASM language
- The role of CoreASM in RE
  - Features of the language relevant for RE
  - Features of the architecture relevant for RE
- Current state and future work
- Conclusions

Motivations

- **Abstract State Machines** (ASM) are known to be effective in *specifying* and *modeling* a variety of systems:
  - Languages, protocols, reactive/embedded systems, web services, information systems, social behavior, CPUs and other hardware, ...
  - Several books and hundreds of papers published with examples (many of them quite large)
- Several compilers and interpreters for various ASM dialect exist
  - All of them targeted at detailed specification

Motivations

- Research question:
  What does it take to profitably use ASMs at the requirements or early design stages?
- Our answer:
  - **Design**, **specify** and **implement** a **language** and related **tools** optimized for high-level design
  - Make **rapid prototyping** of **abstract specifications** possible, enhance freedom of **experimentation**
  - Provide all the advantages of **executable specifications** (incl. **validation**)

ASM in a nutshell

- A ***signature*** Σ is a finite collection of ***function*** names f
  - Each function name has an arity
  - Nullary functions are called *constants*
  - The constants *true*, *false*, *undef* are always defined
- A ***state*** A for Σ is a non-empty set X (the superuniverse of A) together with an interpretation f^A for each function name f in Σ
  - If f is an n-ary function name of Σ, then f^A: $X^n \rightarrow X$
  - If c is a constant of Σ, then $c^A \in X$
- Functions can be *static* or *dynamic*
  - The value of a dynamic function can change from state to state

ASM in a nutshell

- A ***location*** is a pair $l=(f,(a_1,\ldots,a_n))$
  - The contents of l in A are $f^A(a_1,\ldots,a_n)$
- Locations can be ***updated***
  - **Update** $u=(l,v)$
  - **Update set** U is a set of updates
  - An update set is ***consistent*** if there are no clashing updates to the same location
- Firing of updates moves from one state to the next:

$$(A+U)(l)=\begin{cases} v & \text{if } (l,v)\in U \\ A(l) & \text{otherwise} \end{cases}$$

ASM in a nutshell

- ASM specifications describe through updates how the state of the specified system evolves over time
- Important: *values* here are totally general mathematical structures (abstraction)
- Rules:
  - Updates: f**(**a1**,**...**,**an**):=**v
  - Conditional: **if** b **then** P **else** Q
  - Sequence and Parallel: P **seq** Q, P **par** Q
  - Parallelism and nondeterminism: **forall** and **choose**

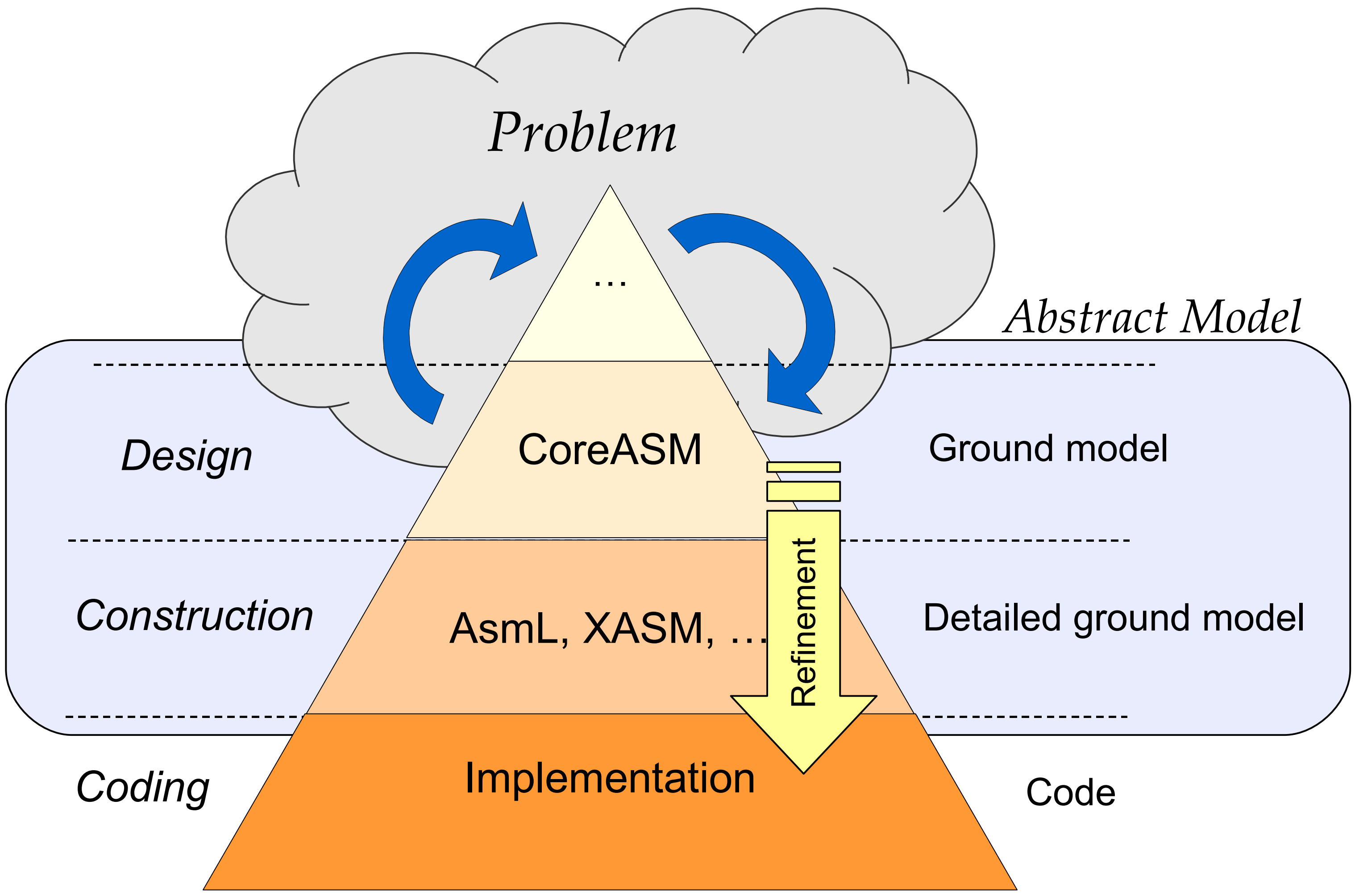
An example

- A fragment from a published ASM spec of the Broy-Lamport problem (modeling RPC calls):

```
if CallName(Me)=read then
    if MemLocs(First(CallArgs(Me)))=false then RETURN(exception, BadArg)
    elseif Fail then RETURN (exception, MemFailure)
    else RETURN(normal, Memory(First(CallArgs(Me))))
    endif
elseif CallName(Me)=write then
    if MemLocs(First(CallArgs(Me)))=false or MemVals(Second(CallArgs(Me)))=false then
        RETURN(exception, BadArg)
    elseif Fail then RETURN(exception, MemFailure)
    else
        Memory(First(CallArgs(Me))) := Second(CallArgs(Me))
        if Succeed then RETURN(normal, Ok) endif
    endif
endif
```

- *ASM* = ***Pseudo-code*** *over* ***abstract data***

CoreASM: The very idea



The CoreASM Project

- 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
  - Formal verification

ASMs in RE

- **Executability** is a useful feature to have in RE
  - Animation, tracing, validation, model checking, etc.
- But most executable specification languages are costly
- CoreASM tries to *change the economics* and make writing executable high-level specifications convenient through
  - Features of the language
  - Features of the architecture

CoreASM – language features

- CoreASM is an ***untyped*** language
  - Types *can* be declared and if they are, the spec will be type checked
  - But they are not compulsory
  - Even better, *partial typing* is possible
  - *Spontaneous casts* (e.g., from “12” to 12) as needed
  - Same spirit as scripting languages
- Makes writing “quick&dirty” specs possible
  - Encourages **experimentation**,
  - avoids **early commitment**

CoreASM – language features

- Non-determinism expressed through **choose** clauses
- Abstraction expressed through:
  - *Oracle functions* (e.g., value input by user)
  - *Abstract macros* (e.g., executed symbolically)
- Both are **explicitly** marked
  - No confusion between *abstraction* and *ambiguity*
- Distributed systems modeled by multi-agent ASMs
  - Scheduling policy can be left arbitrary or specified

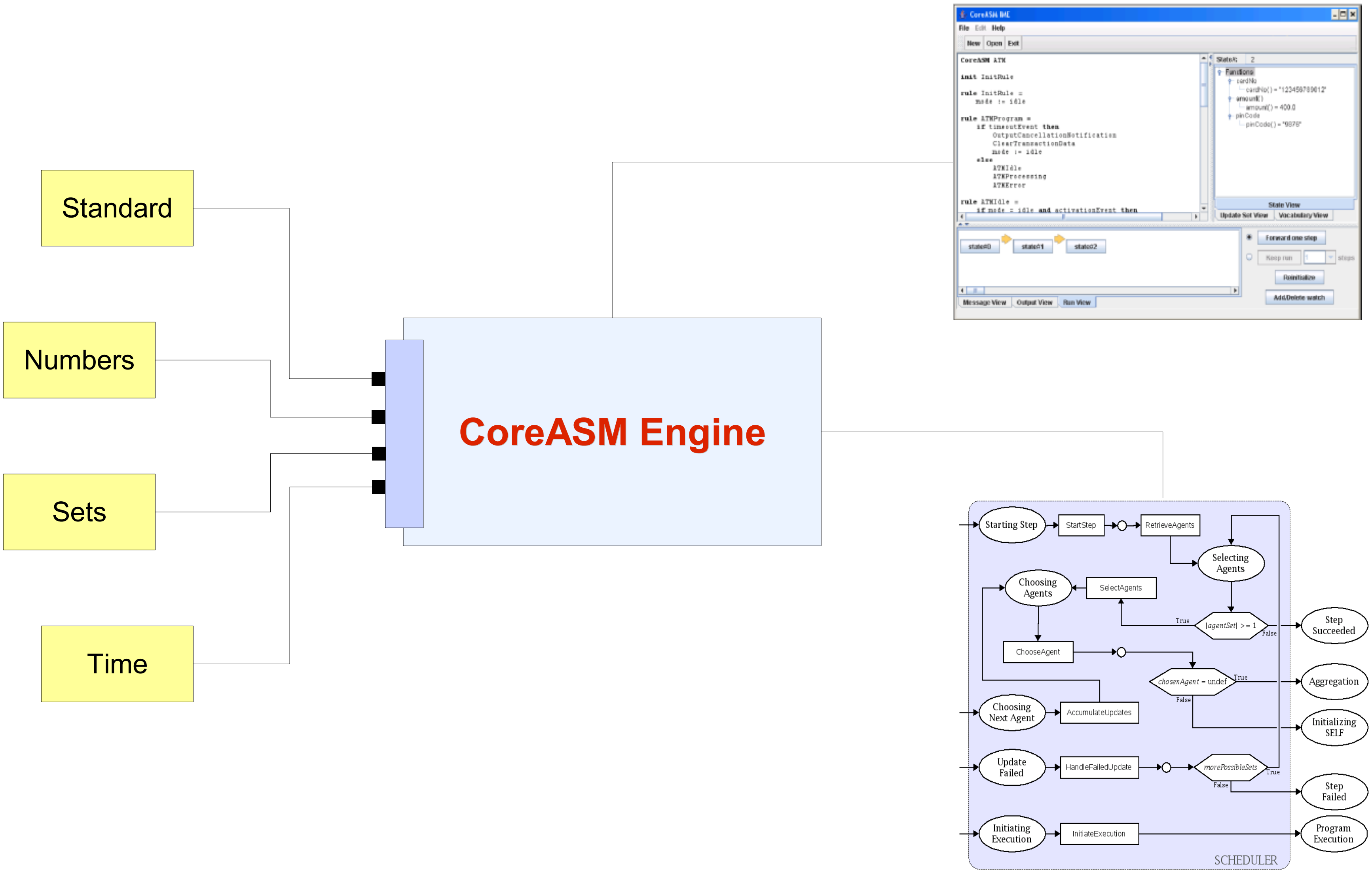
CoreASM – architecture features

- 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**

CoreASM – architecture features

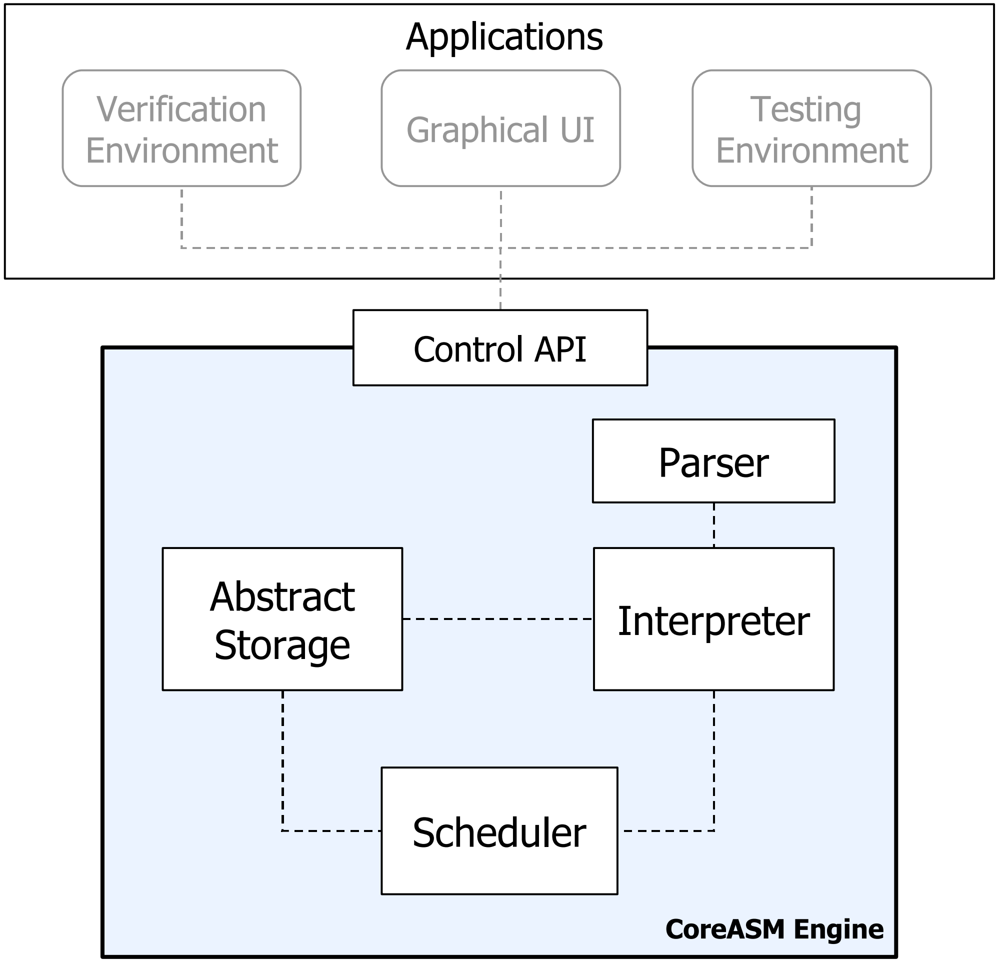
- Plug-ins provide:
  - New *backgrounds*
    - Data types with operations, constants, literals and notation, e.g.: trees
    - Static or derived functions, e.g.: *now* for timed ASMs
  - New *rule forms*
    - Syntax and semantics to simplify writing, e.g.: **signal** *agent* **with** *value* for communications
  - New scheduling and choosing *policies*
    - e.g.: priority-based agent scheduling
  - *Extensions* to the execution cycle
    - e.g.: preprocessing of source specs, or monitoring updates

Kernel of a full environment



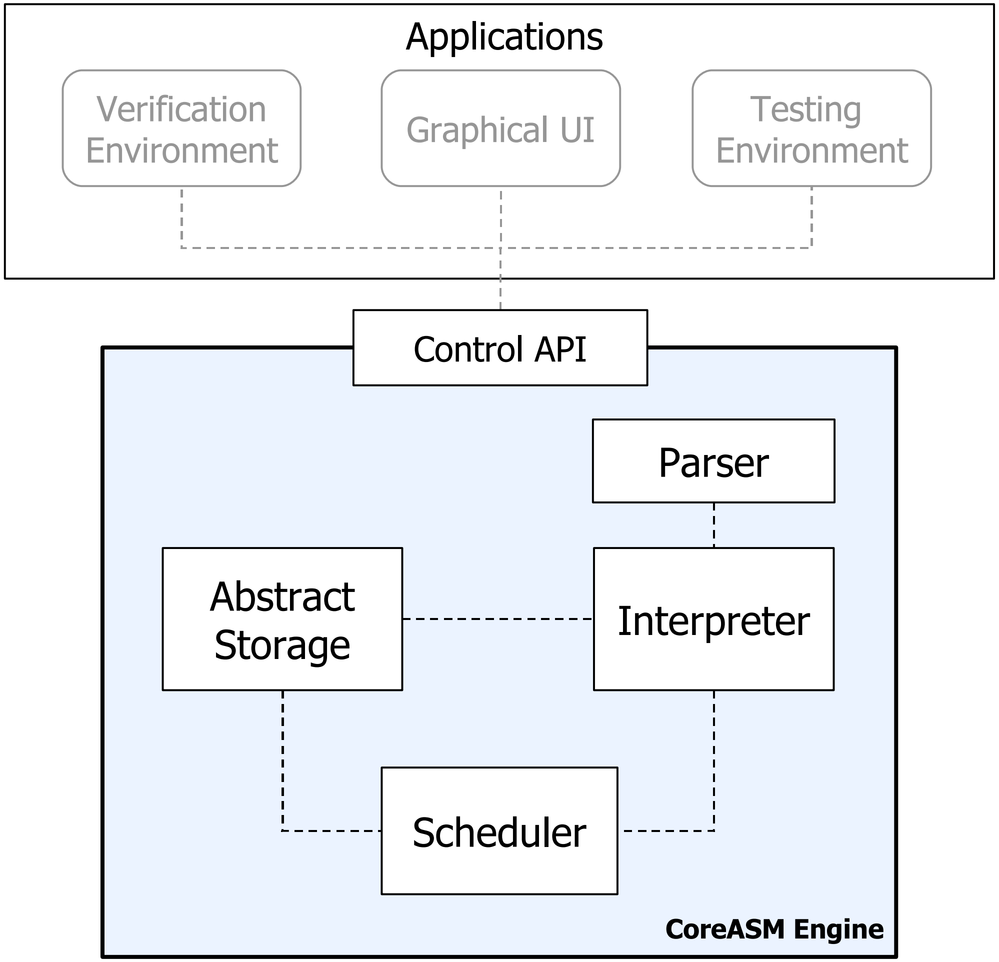
The architecture

- 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

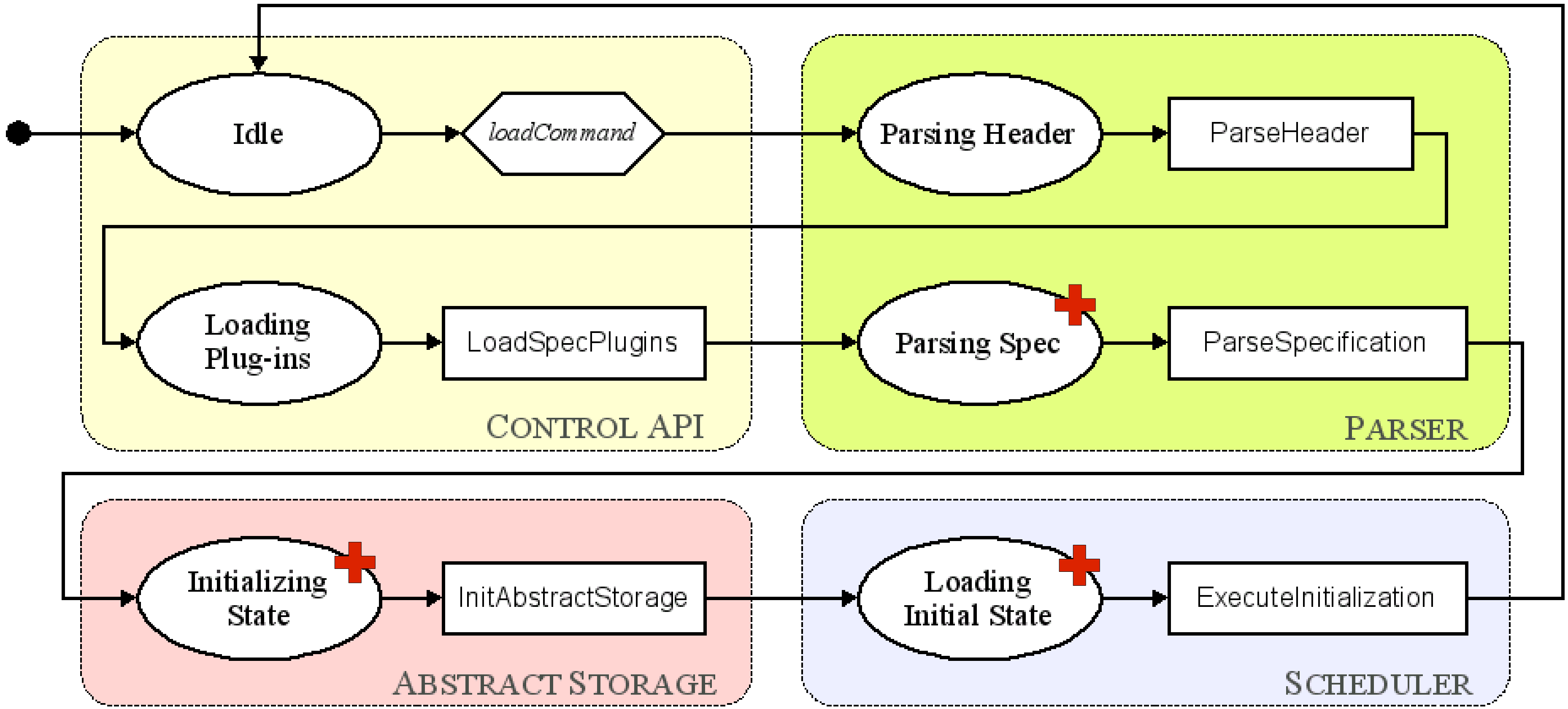


A micro-kernel approach

- A micro-kernel approach
  - Kernel provides the bare minimum structure
    - Updates, true, false, undef, etc.
  - Other language elements are provided by plug-ins
    - Integers, sets, strings, etc.
    - If-rule, choose-rule, block-rule, etc.
  - Standard ASM features are provided by plug-ins in the standard library
  - Custom extensions can be implemented by custom plug-ins

Extension points

Example: Loading Specifications



Example: Tabbed Block Rules

- A simple parallel block rule plugin may require **par** and **endpar**

  if flag **par** a:=1; b:=2 **endpar** **else** **c**:=3

- It doesn't look nice? Indentation looks better?

```
if flag
    a:=1
    b:=2
else
    c:=3
```

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

Example: Spec of a language

- A fragment of the actual specification **of** CoreASM (the language), showing domain-specific constructs and use of abstraction

$$\llparenthesis\, {}^{\alpha}x({}^{\lambda_1}\boxed{?}_1, \ldots, {}^{\lambda_n}\boxed{?}_n) \,\rrparenthesis \quad \rightarrow$$

$$\begin{array}{l}
\textbf{if } \mathit{isFunctionName}(x) \textbf{ then} \\
\quad \textbf{choose } i \in [1..n] \textbf{ with } \neg \mathit{evaluated}(\lambda_i) \\
\quad\quad \mathit{pos} := \lambda_i \\
\quad \textbf{ifnone} \\
\quad\quad \textbf{let } l = (x, \langle \mathit{value}(\lambda_1), \ldots, \mathit{value}(\lambda_n) \rangle) \textbf{ in} \\
\quad\quad\quad [\![\mathit{pos}]\!] := (l, \mathit{undef}, \mathit{getValue}(l)) \\
\textbf{if } \mathit{undefined}(x) \textbf{ then} \\
\quad \mathsf{HandleUndefinedIdentifier}(x, \langle \lambda_1, \ldots, \lambda_n \rangle)
\end{array}$$

where

$$\mathit{undefined}(x) \equiv \nexists e \in \text{ELEMENT} : \mathit{name}(e) = x$$

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

Example: Integration with Java

- For testing and verification purposes, it is useful to have the formal specification interact with the implementation
- A plugin provides integration with Java
  - Instantiation of objects (**create** o **as** JavaClass)
  - Calling methods, accessing fields (**invoke** o->m(...))
  - Marshalling and unmarshalling (as spontaneous casts) of basic types
  - Marshalling and unmarshalling of Collection and String (treated as significant special cases)

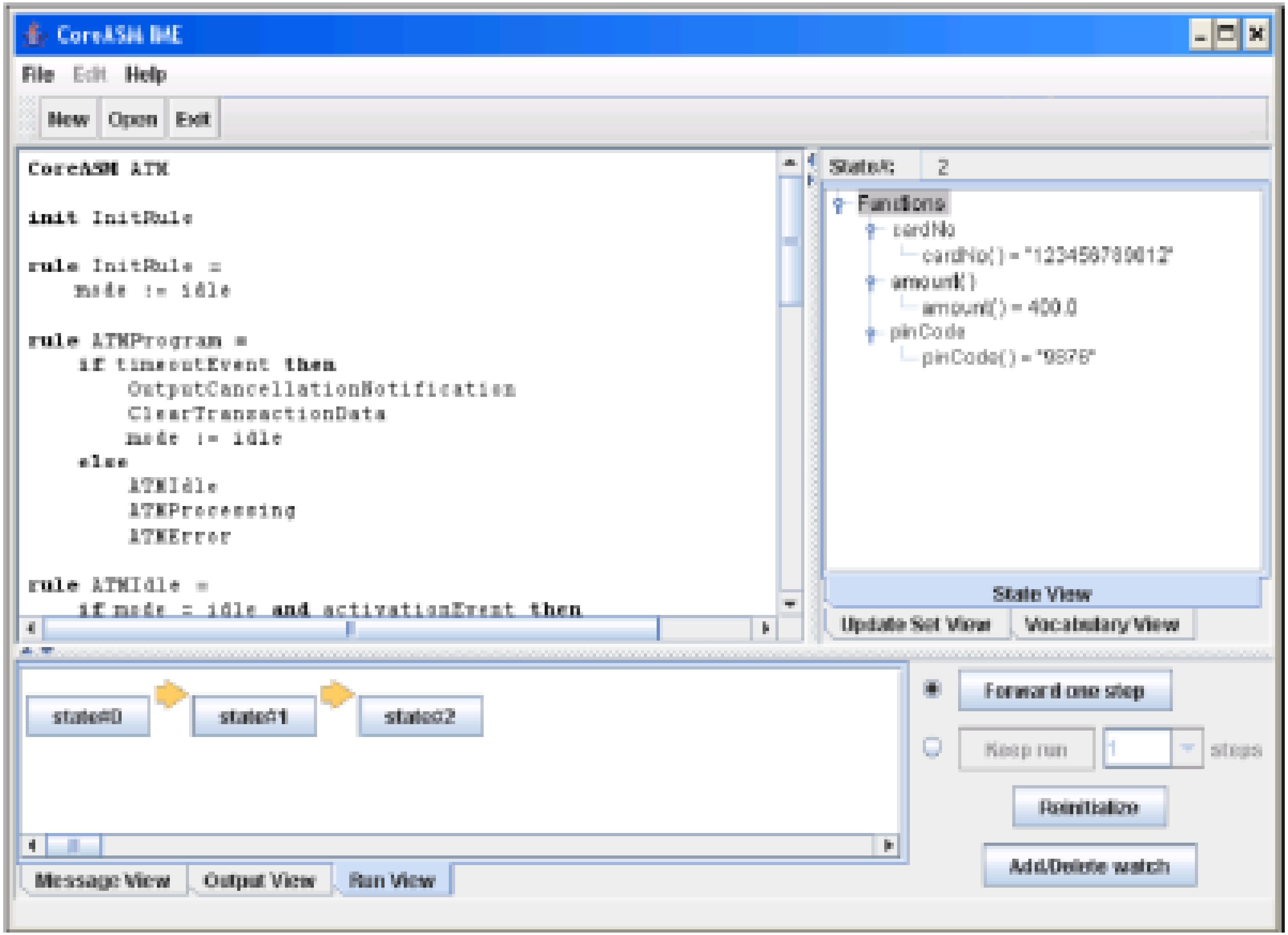
Example: Integration with Java

- Typical uses:
  - Running self-checking, side-to-side parallel runs to specification and implementation
  - Accessing special OS interfaces from CoreASM (e.g., sockets)
  - Adding GUIs or GUI mock-ups to specifications
- Moreover:
  - CoreASM engine can be called **from** Java
  - Two-way interaction possible

Current state

- ASM specification of
  - The kernel
  - Basic ASM and Turbo ASM rule forms
  - Numbers and Sets
- Working implementation of
  - The kernel (minus a few low-priority functions)
  - Most rule forms
  - Numbers, Sets, Strings, etc.
  - GUI (still rough edges, though)

GUI



CoreASM IDE
File
Edit
Help
New
Open
Exit
CoreASM ATM
init InitRule
rule InitRule =
mode := idle
rule ATMProgram =
if timeoutEvent then
OutputCancellationNotification
ClearTransactionData
mode := idle
else
ATMIdle
ATMProcessing
ATMError
rule ATMIdle =
if mode = idle and activationEvent then
State#:
2
Functions
cardNo
cardNo() = "123456789012"
amount()
amount() = 400.0
pinCode
pinCode() = "9876"
State View
Update Set View
Vocabulary View
state#0
state#1
state#2
Forward one step
Keep run
1
steps
Reinitialize
Add/Delete watch
Message View
Output View
Run View

Future work

- Complete implementation of the kernel
- Implementation of more sophisticated data types as plugins
- Implementation of type checking, assertions, invariants as custom plugin
  - These do not exist in traditional ASMs
- Under consideration: rewrite the GUI as an Eclipse plug-in
  - Integration with modeling and development environment

Conclusions

- Bringing RE concerns into formal language design
- 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

http://www.coreasm.org

Last slide

- **Which quality features are addressed by the paper?**
  - *Validation* and *verification* through *executable specifications*
- **What is the main novelty/contribution of the paper?**
  - A formal specification method which is designed to be *low-cost* and *executable*, yet *scalable* to full-fledged formality
- **How will this novelty/contribution improve RE practice or RE research?**
  - *Support adoption* of ASMs in industry
  - Make formal methods *practical* in RE context
- **What are the main problems with the novelty/contribution and/or with the paper?**
  - Work in progress, <u>effectiveness unproven</u>
  - Risk of loosing advantages of hard FMs if too much “hardness” is removed
- **Can the proposed approach be expected to scale to real-life problems?**
  - ASMs are known to scale well (they have been used for large real-life problems)
  - Scalability of investment and extensibility <u>unproven</u>, but apparently possible