Acknowledgments

Chapter 1

requires a language that emphasizes freedom of experimentation by minimizing the need for encoding in mapping the problem space to a formal model. This can be achieved by

- *reducing the cost of encoding domain concepts to language concepts*

The ASM framework comes with a sound and powerful notion of step-wise re nement that helps the designer to structure the design of a system into

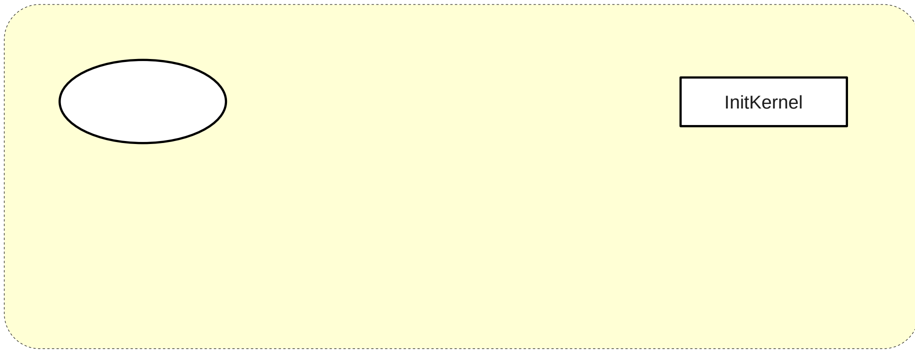


Figure 1.1: An Example of a Control State ASM

1.2 The **CoreASM**

Based on such experience, a second generation of more mature ASM tools and tool environments was developed: *AsmL* (ASM Language) [66] and the *Xasm (Extensible ASM) language* [2, 3] are both based on compilers, while the *ASM Workbench* [21], *AsmGofer* [69], and *Asmeta* [39] provide ASM interpreters.

All the above languages build on predefined type concepts rather than the untyped language underlying the theoretical model of ASMs. The most prominent of these languages are Asmeta and AsmL. The Asmeta language, called AsmetaL, implements all the constructs of basic, structured, and multi-agent ASMs as defined in [20], but it is a fully typed ASM language with limited extensibility features. AsmL is a strongly typed language based on the concepts of ASMs but also incorporates

S_0 s_0

7. *Choose rule:* **choose** x **with** **do**

set may change dynamically over runs of M^D, as required to model a varying number of computational resources. Agents of M^D normally interact with one another, and typically also with the operational environment of M^D, by reading and writing shared locations of a global machine state.[4]

A DASM M^D performs a computation step whenever one of its agents performs a computation step. In general, one or more agents may participate in the same computation step of M^D. A single computation step of an individual agent is called a *move*

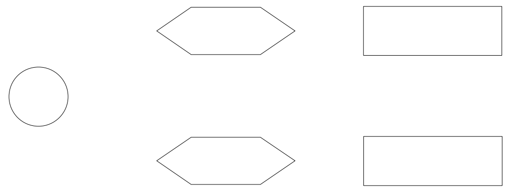


Figure 2.1: Control State ASMs

Thus, the control state ASM of Figure 2.1 can be formulated as a parallel composition of the following FSM rules:

FSM(*i;* if $cond_1$ then

```
rule ClearDeadline(x) =
    if trackStatus(x) = empty and deadline(x) < infinity then
        deadline(x) := infinity

rule SignalOpen =
    if gateSignal = close and safeToOpen then
        gateSignal := open
```

The predicate *safeToOpen*, used in the SignalOpen rule, can be defined as follows

$$safeToOpen \equiv \forall t \in \mathtt{Track}\ trackStatus = empty \vee deadline(t) > now + d_{open}$$

which is defined in CoreASM as

```
derived safeToOpen = forall t in Track525(Track525(Track525(Track525(Track52J/F15rld0g0G0g02
```

```
init InitRule

rule InitRule = {
    forall t in Track do {
        trackStatus(t) := empty
        deadline(t) := infinity
    }
    gateState:= opened
    dmin:= 5000
    dmax:= 10000
    dopen:= 2000
    dclose:= 2000
    startTime:= now

    program(trackController) := @TrackControl
    program(gateController) := @GateControl
    program(observer) := @ObserverProgram
    program(environment) := @EnvironmentProgram
    program(self) := undef
}
```

The Simulation

Finally, we have everything in place to execute the model in CoreASM and validate the behavior of the gate controller (see Appendix B.1 for the full specification). The execution provides a printout of the states of the system. The output shows that the controller keeps the gate open while there is no train on the tracks and keeps it closed as long as there is at least one train crossing the intersection. Figure 2.2 shows parts of the output of one particular run of the system. As a result of the non-deterministic behavior of the environment, different runs of the model most likely provide different outputs.

It is worth to emphasize that although the ability to execute the model and to observe its behavior enables us to validate the model by experiment, satisfying results of such experiments by no means guarantee the "correctness" of the model. Section 6.3.2 offers a brief discussion on this subject.

```
Time: 0.131 seconds
Track track2 is empty
Track track1 is empty
Gate is opened

...

Time: 4.531 seconds
Track track2 is coming track2 i2 2 2 2 2 2 2 2 2 2 2 2 2 2 B s (
```

Figure 2.2: Output of the Railroad Crossing Example in CoreASM

Chapter 3

CoreASM: Architectural Overview

The CoreASM language and supporting tool architecture focus on early phases of

3.

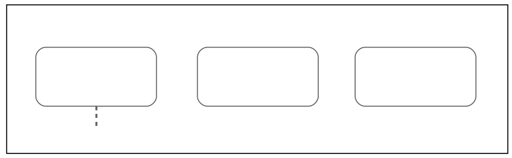


Figure 3.2: Overall Architecture of CoreASM

   (d) Initializing the abstract storage
   (e) Setting up the initial state[5]

3. Execution of the speci cation
   (a) Execute a single step
   (b) If termination condition is not met, repeat from 3a.

The execution process of a single step in the

(plugins that are basically a set of other plugins) are expanded and their enclosed plugins are added to the list of required plugins. In the next step, plugins are loaded one by one according to their loading priority.

Control API

LoadSpecPlugins

```
seq
// 1. expanding package plugins
  forall p in specPlugins do
    if isPackagePlugin(p) then
      forall p' in enclosedPlugins(p) do
        add p' to specPlugins
next
// 2. loading plugins with the maximum load priority first
  while |specPlugins\loadedPlugins| > 0 do
```

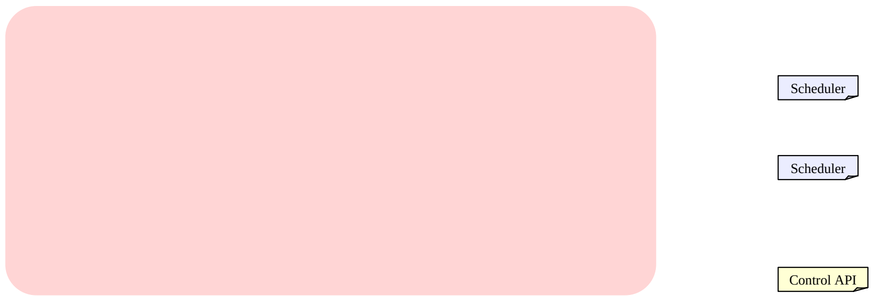


Figure 3.8: Control State ASM of a

the ASM framework to include ASM rules (programs) as elements of the state; i.e.

In the earlier versions of CoreASM [27], if an inconsistent set of updates would be generated in a step, the HandleFailedUpdate rule in the scheduler module would prepare a di erent subset of agents for execution, and the step would be re-initiated.

Figure 3.10: Revised Control State ASM of a *step*

there would be no way of specifying how the state should evolve, and that **import** has a special role in introducing new elements to the state. All other rule forms (e.g., **if**, **choose**, **forall**

Chapter 4

providing that

Function Elements

Functions de ned in a CoreASM

ments, they hold the same values.[4] For all $f_1; f_2$ 2 FunctionElement

4.

of a node, if any, by orderly assigning *pos* accordingly; when all needed subtrees are evaluated, we compute the resulting location, updates or value and assign it to $[\![pos]\!]$, thus implicitly returning control back to our parent. As exemplified in Table 4.2, our notation allows us to clearly visualize this process by the progressive substitution of evaluated u nodes for unevaluated $\tilde{r}$ nodes, and of v or l nodes for unevaluated $\tilde{e}$ nodes. Notice that identifiers do not have to be evaluated, hence we do not need a "boxed" version of x.

4.2.2 Kernel Expression Interpreter

As previously described, the kernel interpreter rules implement the Boolean domain

4.

Interpreter: Kernel Rules

Limport x do e

$uiVal : \mathrm{Update} \mapsto \mathrm{Element}$
returns the value associated with the given update instruction.

$uiAction : \mathrm{Update} \mapsto \mathrm{Action}$

set. When called for aggregation, an aggregator plugin aggregates all update instructions for which it is responsible and ags them as either successful or failed. It is important to note that the order in which plugins are called to perform aggregation should not a ect the resultant updates produced. Also note that the failure

4. CoreASM

4. CoreASM

to equip the engine with a fast parser generator capable of generating parsers with

Plugin Interface	Extends	Description

for an example. Rules of extensible control state ASMs are formulated in textual form

4.

Chapter 5

CoreASM: The Plugins

Most of the functionalities of CoreASM and its language constructs are provided through plugins to the CoreASM kernel. In this chapter we present the speci cation of those plugins that are currently available as part the CoreASM project. Most of these plugins are part of the standard library of CoreASM and can be loaded by simply loading the `Standard` package plugin.

Here, we divide the plugins into four categories: plugins that extend the CoreASM

5.

5.1.6 The forall-rule Plugin

The semantic definition of **forall**-rule is similar to that of **choose**-rule with the difference that **all** the elements of the given enumerable element that satisfy the optional guard are given a chance to be the free variable in the **do**-rule. Here, we present the semantics of **forall**-rule with a guard. The semantics of **forall** with no guard is presented in Appendix A.5.2.

Forall Rule

$\llbracket$ **forall** x **in** $\overset{e}{e}_1$ **with** $\overset{e}{e}_2$ **do** $\overset{e}{r}$ $\rrbracket$ $\rightarrow$ $pos :=$

$\llbracket pos \rrbracket := (undef, \{\}, undef)$

To evaluate this rule, the case condition will be evaluated rst and then all the

SeqRule

$[\![\tilde{r}_1 \text{ seq } \tilde{r}_2]\!] \rightarrow$ *pos* :=

$[\![u_1 \text{ seq } \tilde{r}_2]\!] \rightarrow$ **let** *uSet* = Aggregate(u_1) **in**
  if *isConsistent*(*uSet*) $\wedge$ *aggregationConsistent*(u_1) **then**
    PushState
    Apply(*uSet*)
    pos :=
  else
    $[\![pos]\!]$:= ($undef; u_1; undef$)

$[\![u_1 \text{ seq } u_2]\!] \rightarrow$

Iterate Rule

Literate	e r M !	PushState *composedUpdates*(*pos*) := fjg *pos* :=

While Rule

L while (e e)

Local Rule

$z := R(a$

The Number plugin also provides the following relational operators de ned on Number elements:

\>" : greater-than binary operator (precedence level: 650)

\>=" : greater-than or equal-to binary operator (precedence level: 650)

\<"

$$\mathit{stringValue} : \mathrm{StringElement} \mapsto \mathrm{List(Character)}$$

Modifiable Collections

The Collection plugin introduces a modifiable-collection attribute on elements, defined by the following function:

$$\mathit{isModifiableCollection} : \mathrm{Element} \mapsto \mathrm{Boolean}$$

The modifiability attribute set on an element indicates that generic collection modifications (at this point limited to addition and removal of an element) can be applied to the element. Plugins that provide modifiable collection elements (such as sets and list) must also provide the semantics of such modifications through two functions of the form

computeAddUpdate

The Plugins

Set Plugin: Set Enumeration

L *f*

semantic definition is provided in Appendix A.5.4

L e ? \ e ? M [675] ! choose 2 f ; g with : *evaluated*( )
pos :=
ifnone
if *8x 2 fl; rg* SetElement(*x*) _ *x* = $undef_e$ then
if *l* = $undef_e$ = $undef_e$
= undef = (

Set Plugin

Aggregate$_{Set}$(*uMset*)
  local *resultantUpdate* in
  seq
    result := *fg*
  next
    forall *l* 2 *locsToAggregate* f9.6 0 Td [(f9.6m107.5in) [(next)]TJ 9if.9626 Tf 9.409 0 Td1055 451(esulta

state.

Set Plugin

BuildResultantUpdate($l; uMset$)
local *newSet* [*newSet* := $\{\}$] in
seq
forall $e \in$ *enumerate*(

Set Plugin

Compose$_{Set}$(*uMset*$_1$, *uMset*$_2$)
  seq
   result := $\{\}$
  next
   forall *l* $\in$ *locsAffected* do
    if *locHasAddRemove*(*uMset*$_1$) $\wedge$ $\neg$ *locUpdated*(*uMset*$_2$) then
     forall *ui* $\in$ *uMset*$_1$ with *uiLoc*(*ui*) = *l* do
      add *ui* to result
    else if $\neg$ *locUpdated*(*uMset*$_1$) $\wedge$ *locHasAddRemove*(*uMset*$_2$) then
     forall *ui* $\in$ *uMset*$_2$ with *uiLoc*(*ui*) = *l* do
      add *ui* to result
    else if *locHasAddRemove*(*uMset*$_2$) $\wedge$ *locRegularUpdate*(*uMset*$_2$) then
     forall *ui* $\in$ *uMset*$_2$ with *uiLoc*(*ui*) = *l* do
      add *ui* to result
    else if *locHasAddRemove*(*uMset*$_2$) $\wedge$ *locRegularUpdate*(*uMset*$_1$) then
     add

$bagElement$: Multiset(Element) $\mapsto$ BagElement

Since incremental updates on bags do not come with much constraints as for sets (due to multiplicity of elements), instead of using di erent update actions for adding/removing elements to/from bags, Bag plugin uses a more general action, *bagUpdateAction*, with

listElement : List(Element) 7! ListElement
returns a list element representing the given sequence of elements.

listValue : ListElement 7! List(Element)
returns the sequence of elements that are represented by the given list element,

$head_{le}$: ListElement 7! Element
$last_{le}$: ListElement 7! Element
return the rst and last elements of the list, or $undef_e$ if the list is empty.

$tail_{le}$: ListElement 7! ListElement

5.

List Plugin : Search

L indexof $\overset{e}{e}$ in $\overset{e}{e}$ M $!$ choose 2 f

Rule Forms

The List plugin extends the interpreter of the engine to provide the following rule T.rms

L pop e / from e / M !

$bkg(m) = $ "Map"

$\forall m' \in \text{MapElement} \quad equal_{Map}(m; m'$

Signature Plugin

CreateEnumeration(*name; members*)
let *b* = *new*(EnumerationBackground) in
add (*name; b*) to *pluginBackgrounds*(

Signature Plugin

CheckUpdateSetForTypes
if *engineProperties*("TypeChecking") = "strict" then
forall $\langle loc, val, act \rangle \in updateSet$ do
let $f = stateFunction(state, name_{lc}(loc)), sig_f = signature(f)$ in
if $sig_f =$

5. CoreASM

R_1 **step** R_2

if *uniqueCtlState*(@R_1) *2 ctl_state* **then**

R_1

seq

if *89 cs 2 ctl*

seq

```
        print "sum( 1, 2, 100, @a ) = " + sum(f1, 2, 100g, @a)
        print "powerset(1, 2, 3) = " + powerset(f1, 2, 3g)
        print "2, 3 memberof powerset(1, 2, 3 = "
                    + (f2, 3g memberof powerset(f1,2,3g))
        choose x in powerset(f1, 2, 3, 4g) do
            if x memberof powerset(f1, 2, 3g) then
                print x + " is a member of powerset(1, 2, 3)"
            else
                print x + " is not a member of powerset(1, 2, 3)"
        ifnone
            print powerset(f1, 2, 3g)
    g
```

As an example, the output of the execution of Program ?? is the following:

```
sum( {1, 2, 100} ) = 103
min(51, 43) = 43
asin(0.5) = 30
powerset({1, 2, 3}) = {{}, {3}, {2}, {3, 2}, {1}, {3, 1}, {2, 1}, {3, 2, 1}}
{2, 3} memberof powerset({1, 2, 3} = true
log(e) = 1
{3, 2, 4} is not a member of powerset({1, 2, 3})
sum( {1, 2, 100}, @a ) = 515
'e' = 2.718281828459045
sin(30) = 0.5
```

5.4.7 The Time Plugin

To introduce the notion of time in CoreASM may be possible to extend the Property plugin to check simple global assertions). Correctness properties are only applicable during model checking, and are translated by our CoreASM to Promela translator.

where *newJObject*() returns a new `JObject` element pointing to a new Java object.

In formal terms, using the notation described above, creation of a new Java object is accomplished as follows:

CreationRules

L import native e into e / M ! *pos* :=

L import native *x* into / M ! if *if*! *if*! *if* related update instructions that are performed during a step, whereas the DefUpd macro produces an encoding of its parameters, suitable for later execution of the relevant update.

While the subject will be discussed more fully in the following, it is worthwhile to remark here that this strategy ensures that any action that can perturb the environment (e.g., instantiation of a new Java object) will only be taken if the step turns out to be effective, i.e. if no conflicting updates are generated in that step.

Access to Fields of Java Objects

Reading a field in a Java object does not have side effects and thus can be treated as a pure expression as far as the ASM computation cycle is concerned[9]. In particular, the value in the field can be computed immediately at expression evaluation time. In

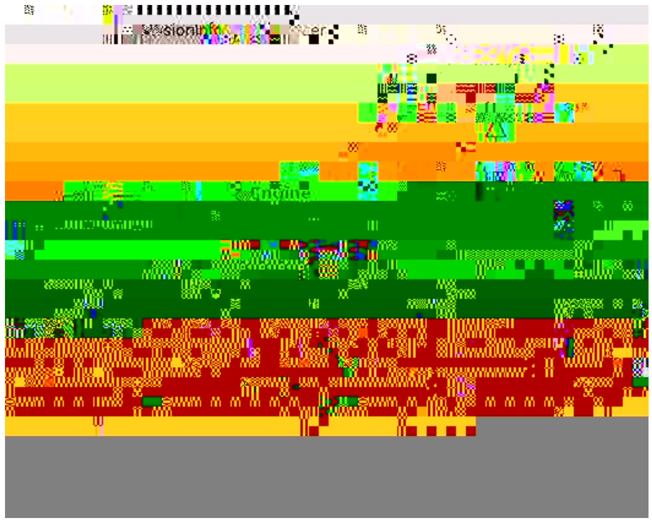
5. CoreASM

2. If multiple STORE

Chapter 6

Implementing **CoreASM**

As we addressed in Section 1.2, one of the requirements of the CoreASM modeling environment is that it should be implemented as an open framework, under an open



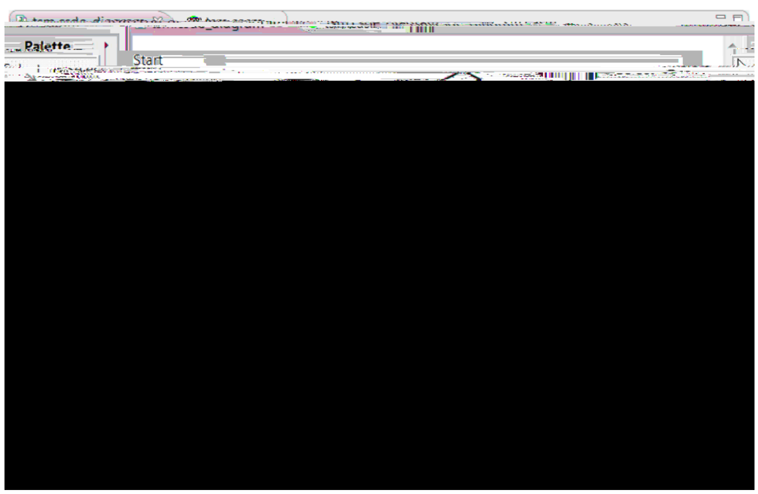
(instance of `AbstractUniverse`), thereby implementing contents of CoreASM state as defined in Section 4.1:

$$\mathit{stateFunction} : \mathrm{State}$$

Eclipse Plugin passes the speci cation to the CoreASM engine and gets the set of plugins that are used by the speci cation. The editor then asks the plugins for the set of keywords, functions, universes and backgrounds they provide and uses this information to o er a dynamic syntax highlighting of the speci cation.

Figure 6.5(a)

(a) CoreASM Eclipse Plugin



parsed by the CoreASM

Chapter 7

Conclusions and Perspectives

Extensible Language and Architecture

The most significant feature of CoreASM is the extensibility of its language and modeling environment. To reduce the cost of writing specifications, one has to minimize the need for encoding in mapping the problem space to a formal model. This approach usually leads to the design of domain-specific languages. The CoreASM extensibility

CoreASM Integrated Development Environment

Appendix A

Supplementary De nitions

A.1 Abstract Storage

PushState puts the current state in the stack. We assume that $stack_{state}$ is empty in the initial state.

PushState
Push($stack_{state}$; $state$)

HandleUnde nedIdenti er(

agentSet : Set(Element)
is the set of all the available agents in the current state retrieved from the Abstract Storage at the beginning of every computation step.

engineProperties : Name $\mapsto$ Name

Choose Rule

L choose x

Set Plugin : Set Comprehension variant 3

$\llbracket f \;\; x \text{ is } \tilde{e} \mid \tilde{}_1 x_1 \text{ in } \tilde{}_1 ?_1, \dots, \tilde{}_n x_n \text{ in } \tilde{}_n ?_n \text{ with } \tilde{?} g \rrbracket$!

if n 1 then
  if $\forall j \in [1..n],\; x \neq x_j$ then
    choose $j \in [1..n]$ with $value(\;_j) = undef$ do
      $pos := \;_j$
    ifnone
      if $sameNameTwoConstVar$ then
        Error(`No two constrainer variables may have the same name')
      else if $\exists c \in [1..n]$; CoreASM

$L f \quad x$ is $v \, j \;\; {}^{1}x_1$ in ${}^{1}v_1; : : : ; \;\; {}^{n}x_n$ in ${}^{n}v_n$ with vgM $!$

seq
  add *value*( ) to *newSet*(*pos*)
next
  if OtherCombosToConsider then
    ChooseNextCombo
    ClearTree( )
    ClearTree( )
    pos :=
  else
    DestroyConsideredCombos
    ⟦*pos*⟧ := (*undef*

A.5.5 Math Plugin

Most of the functions provided by the Math plugin are equivalent of their Java counterparts defined in the Java library package `java.lang.Math`. For such functions, we use the descriptions provided by the *Java 2 Platform Standard Edition 5.0 API Specification* [72].

Constants

`MathE` returns the Number element that is closer in value than any other to e, the base of the natural logarithms.

`sum({v1,...,vn})` returns the sum of a collection of numbers. If there is one non-number in the collection, it returns *undef*.

`sum({v1,...,vn}, @f)` returns the sum of a collection of numbers, after applying function `f` to the values in the collection. If there is one non-number in the collection, it returns *undef*.

`powerset({e1,...,en})` returns the powerset of the given set of elements.

Appendix B

CoreASM Examples

B.1 The Railroad Crossing Example

```
CoreASM RailRoadCrossing

use
```

B. CoreASM Examples

g

```
// --- Auxiliary Rules ---
```

```
            + (2 * random * bearingError(self) - bearingError(self))
          ) in
    Move(dir)
```

```
derived range(a) =
    sqrt( pow(posX(a) - posX( self )<5s0a) pow(pYsX(a) - pYsX( self
        derived[QRR!T%+%0PmE%5%0%0PmEU%P(( powTJ0.500.3rg0<3rg0120                    dean52+5
```

Appendix C

Change List

Since August 2009

semantics of the operators o ered by the following plugins is revised such that in binary operators if both operands are *undef* or one is *undef* and the other is a relevant value (depending on the plugin), the evaluation results in *undef*. In unary operators if the operand is *undef* the result of the operation will be *undef*. Of course, if other plugins evaluate the operation to a non-*undef* value, the *undef* value is ignored and the non-*undef* value will be considered as the value of the operation.

{

Bibliography

[1] M. Altenhofen, A. Friesen, and J. Lemcke. Asms in service oriented architectures. *Journal of Universal Computer Science*, 14(12):2034{2058, 2008.

[2]

[11]

[25]

[56] Olav Jensen, Raymond Koteng, Kjetil Monge, and Andreas Prinz. Abstraction using ASM Tools. In A. Prinz, editor, *Proceedings of the 14th International ASM Workshop (ASM'07)*, 2007.

[57] C. W. Johnson. Literate speci cations. *Software Engineering Journal*