MIT-CSAIL-TR-2015-007 March 3, 2015

On the Formal Semantics of the Cognitive Middleware AWDRAT

Muhammad Taimoor Khan , Dimitrios Serpanos, and Howard Shrobe*

On the Formal Semantics of the Cognitive Middleware AWDRAT

Muhammad Taimoor Khan[†], Dimitrios Serpanos[†] and Howard Shrobe[*]

[†]{mtkhan, dserpanos}@qf.org.qa
[*]hes@csail.mit.edu
[†]QCRI, Qatar
[*]CSAIL, MIT, USA

February 24, 2015

Abstract

The purpose of this work is two fold: on one hand we want to formalize the behavior of critical components of the self generating and adapting cognitive middleware AWDRAT such that the formalism not only helps to understand the semantics and technical details of the middleware but also opens an opportunity to extend the middleware to support other complex application domains of cybersecurity; on the other hand, the formalism serves as a pre-requisite for our proof of the behavioral correctness of the critical components to ensure the safety of the middleware itself. However, here we focus only on the core and critical component of the middleware, i.e. Execution Monitor which is a part of the module "Architectural Differencer" of AWDRAT. The role of the execution monitor is to identify inconsistencies between runtime *observations* of the target system and *predictions* of the System Architectural Model. Therefore, to achieve this goal, we first define the formal (denotational) semantics of the *observations* (runtime events) and *predictions* (executable specifications as of System Architectural Model); then based on the aforementioned formal semantices, we formalize the behavior of the "Execution Monitor" of the middleware.

Contents

1 Calculus of AWDRAT

Defending systems against cyber attack requires us to be able to rapidly and accurately detect that an attack has occurred. Today's detection systems are woefully inadequate suffering from both high false positive and false negative rates. There are two key reasons for this: First, the systems do not understand the complete behavior of the system they are protecting. The second is that they do not understand what an attacker is trying to achieve. Most such systems, in fact, are retrospective, that is they understand some surface signatures of previous attacks and attempt to recognize the same signature in current traffic. Furthermore, they are passive in character, they sit back and wait for something similar to what has already happened to recur. Attackers, of course, respond by varying their attacks so as to avoid detection.

AWDRAT [5] is a representative of a new class of protection systems that employ a different, active form of perception, one that is informed both by knowledge of what the protected application is trying to do and by knowledge of how attackers think. It employs both bottom-up reasoning (going from sensors data to conclusions about what attacks might be in progress) as well as top-down reasoning (given a set of hypotheses about what attacks might be in progress, in focuses its attention to those events most likely to significantly help in discerning the ground truth).

There are two dimensions along which detection systems can be characterized. The first is the distinction between profile and model based approaches. The other dimension is the distinction between looking for matches to bad behavior or deviations from good. This gives four quadrants, each with unique strengths and weaknesses. For example, the bulk of our sensors are model-based and look for matches to bad behavior; signature based systems are in this category. The advantage is that when a match occurs, you know what has happened; i.e. these systems have high diagnostic resolution. But they also lack robustness; if they don't have a model of an attack, and there are always novel attacks, then they will fail to detect it. On the other hand, there are a class of detectors that use employ machine learning techniques on labeled training data to build statistical profiles of attacks. These systems tend to be a bit more robust than model based systems, since the machine learning techniques tend to generalize from the data presented. However, they make up for this by a loss of diagnostic resolution. The third quadrant involves building a statistical profile of normal behavior, detecting deviations from the profile. Such anomaly detectors are yet more robust, since they don't depend on prior knowledge of the form of the attack, but they afford even less diagnostic resolution. When things go wrong, all you know is that something out of the ordinary has happened; whether that something is malicious or not isn't known.

AWDRAT sits in the fourth quadrant: It has a model of normal behavior; when the application deviates from the behavior prescribed by that model, it employs diagnostic reasoning techniques [6] to further isolate and characterize the failure. It has both greater robustness and higher diagnostic resolution. But it achieves this only through the construction of a far more complex model.

AWDRAT has an active model of normal behavior, namely an executable specification (aka System Architectural Model) of the application [5]. This executable specification consists of a decomposition into sub-modules and pre- and post-conditions for each sub-module. In addition, data-flow and control-flow links connect the sub-modules, specifying the expected flow of values and of control. The pre- and post-conditions are arbitrary first-order statements about the set of data values that flow into and out of the sub-modules.

AWDRAT runs this executable specification in parallel with the actual application code, comparing their results at the granularity and abstraction level of the executable specification. (This is therefore a special case of the standard fault tolerance technique of running multiple versions of the same code and comparing their results.) The executable specification is hierarchical, allowing flexibility in the granularity of the monitoring. When threats are not expected, the executable specification is run at a high level of abstraction, incurring less overhead, but requiring more diagnostic reasoning should the program diverge from the prescribed behavior of the executable specification. In times of heightened threat, the executable specification can be elaborated to a greater degree, incurring more overhead, but providing more containment.

Optionally, the model can also include models for suspected incorrect behaviors of a component, allowing the diagnostic reasoning to characterize the way in which a component might have misbehaved. A diagnosis is then a selection of behavioral modes for each component of the specification such that the specification predicts the observed misbehavior of the system.

The rest of the paper is organized as follows: in Section 2 we discuss syntax of the System Architectural Model followed by the formalization of semantics of the critical syntactic domains model in Section 3. Section 5 formalizes the semantics of the execution monitor. Finally, we conclude in Section 6. Appendices A and B give the formal syntactic grammar and an example System Architectural Model respectively.

2 Syntax of System Architectural Model

An AWDRAT model is built from several related following forms which represent corresponding high-level syntactical domains of the model. Note, we only discuss selected domains here, for complete syntactic domains and their elements, please see Appendix A.

1. A description of a component type consists of

   (a) its interface

   - a list of inputs
   - a list of its outputs
   - a list of the resources it uses (e.g. files it reads, the code in memory that represents this component, etc)
   - list of subcomponents required for the execution of the subject component
   - a list of events that represent entry into the component (usually just one)
   - a list of events that represent exit from the component (usually just one)
   - a list of events that are allowed to occur during any execution of this component
   - a set of conditional probabilities between the possible modes of the resources and the possible modes of the whole component
   - a list of known vulnerabilities occurred to the component

   (b) and a structural model which is a list of sub-components some of which might be splits or joins of

   - data-flows between linking ports of the sub-components (outputs of one to inputs of another)
   - control-flow links between cases of a branch and a component that will be enabled if that branch is taken

   The description of the component type is represented by syntactical domain "StrMod" which is defined as follows:

   StrMod ::= **define-ensemble** CompName
   :entry-events :auto | (EvntSeq)
   :exit-events (EvntSeq)
   :allowable-events (EvntSeq)
   :inputs (ObjNameSeq)
   :outputs (ObjNameSeq)
   :components (CompSeq)
   :controlflows (CtrlFlowSeq)
   :splits (SpltCFSeq)
   :joins (JoinCFSeq)
   :dataflows (DataFlowSeq)
   :resources (ResSeq)
   :resource-mapping (ResMapSeq)

:model-mappings (ModMapSeq)
:vulnerabilities (VulnrabltySeq)

Example 1: The specification of the component `maf-editor` is given below. In detail, the specification says that the component

- is top level component and hence starts automatically and thus requires no `entry-event`,
- requires no `inputs`
- results in `the-model` as an output
- has four subcomponents, i.e. `startup`, `create-model`, `create-events` and `save` which have corresponding types and also have both `normal` and `compromised` behaviors
- has control and data flows as described
- has an access to two resources, i.e. `imagery` and `code-files` which have corresponding probabilities of being in a `normal` and `hacked` mode
- has model mappings of the above resources to the subcomponents as described in `model-mappings` and
- has two vulnerabilities, i.e. `reads-complex-imagery` and `loads-code` for the resources `imagery` and `code-files` respectively.

```
(define-ensemble maf-editor
    :entry-events :auto
    :inputs ()
    :outputs (the-model)
    :components
    ((startup :type maf-startup :models (normal compromised))
     (create-model :type maf-create-model :models (normal compromised))
     (create-events :type maf-create-events :models (normal compromised))
     (save :type maf-save :models (normal compromised)))

    :controlflows ((before maf-editor before startup)
     (after startup before create-model))

    :dataflows ((the-model create-model the-model create-events)
     (the-model create-events the-model save)
     (the-model save the-model maf-save-model))

    :resources ((imagery image-file (normal .7) (hacked .3))
     (code-files loadable-files (normal .8) (hacked .2)))

    :resource-mappings ((startup imagery)
     (create-model code-files)
     (create-events code-files)
     (save-model code-files))


    :model-mappings ((startup normal ((imagery normal)) .99)
     (startup compromised ((imagery normal)) .01)
     (startup normal ((imagery hacked)) .9)
     (startup compromised ((imagery hacked)) .1)

     (create-model normal ((code-files normal)) .99)
     (create-model compromised ((code-files normal)) .01)
     (create-model normal ((code-files hacked)) .9)
     (create-model compromised ((code-files hacked)) .1)

     (create-events normal ((code-files normal)) .99)
     (create-events compromised ((code-files normal)) .01)
     (create-events  normal ((code-files hacked)) .9)
     (create-events compromised ((code-files hacked)) .1)

     (save normal ((code-files normal)) .99)
     (save compromised ((code-files normal)) .001)
     (save normal ((code-files hacked)) .01)
     (save compromised ((code-files hacked)) .999))

    :vulnerabilities ((imagery reads-complex-imagery)
      (code-files loads-code)
      ))
```

2. Behavioral specification of a component (a component type may have one normal behavioral specification and many abnormal behavioral specifications, each one representing some failure mode) which has

   - inputs and outputs
   - preconditions on the inputs (logical expressions involving one or more of the inputs)
   - postconditions (logical expressions involving one or more of the outputs and the inputs)
   - allowable events during the execution in this mode

   The behavioral specification of a component is represented by a corresponding syntactical domain "BehMod" as follows:

   BehMod ::= **defbehavior-model** (CompName **normal** | **compromised**)

:inputs (ObjNameSeq)
:outputs (ObjNameSeq)
:allowable-events (EvntSeq)
:prerequisites (BehCondSeq)
:post-conditions (BehCondSeq)

Example 2: In the following first we give the structure of a component `maf-create-model` (which is one of the submodule as stated in the previous specification example) and then give the behavioral specification of the component. The structure of the component is defined as follows:

```
(define-ensemble maf-create-model
    :entry-events (create-mission-action-action-performed)
    :exit-events (mission-builder-submit)
    :allowable-events (create-mission-builder-with-client-panel
       create-mission-builder
       create-mission-builder-with-hash-table
       mission-builder-submit
       (set-initial-info exit (the-model nil))
       create-mission-action-action-performed
       retrieve-info
       create-mission-action-action-performed
       (set-initial-info entry)
       )
    :inputs ()
    :outputs (the-model))
```

In the following we define the legal and illegal (compromised) behaviors of the component. For example, the specification of a legal (normal) behavior of the component says that as a normal behavior the component

- requires no input as specified by the clause `inputs`
- has `the-model` output and also
- no `prerequisite` of the component but
- guarantees that the object `mission-builder` of `the-model` are consistent.

The corresponding normal behavior is defined as:

```
(defbehavior-model (maf-create-model normal)
    :inputs ()
    :outputs (the-model)
    :prerequisites ()
```

```
        :post-conditions ([dscs ?the-model mission-builder good])
        )

    (defbehavior-model (maf-create-model compromised)
        :inputs ()
        :outputs (the-model)
        :prerequisites ()
        :post-conditions ([not [dscs ?the-model mission-builder good]])
        )
```

Similarly, the compromised behavior of the component is also described above. For further details on the behavioral specification of the other components, please see Appendix A.

3. Model of a resource type contains

   - possible modes
   - prior probabilities of being in each mode
   - attack types to which it is vulnerable

   The syntactical domain "ResModMap" represents the model of a resource type

   ResModMap ::= ResName **normal** | **hacked** FVal
   | ((ResName **normal** | **hacked**)) FVal

   where "FVal" represents the float values for probabilities.

   The trust model of the resources is specified in example 1 above by the clauses `:resources` and `:resource-mappings`.

4. Attack Model

   - a list of types of attacks that are being anticipated and the prior probability of each
   - a list describing how each attack type can effect that mode of a resource
   - a set of logical rules expressing the conditional probabilities between attack types and resource modes

   The attack models are presented by the syntactic domain "AtkMod" while the corresponding attack rules are specified by the syntactic domain "AtkRule" as given below respectively:

AtkMod ::= **define-attack-model** AtkModName
:attack-types (AtkTypeSeq)
:vulnerability-mapping (AtkVulnrabltyMapSeq)

AtkRule ::= **defrule** AtkRulName (**:forward**)
if AtkCondSeq
then AtkConsSeq

Example 3: The example attack model `maf-attacks` specifies the two attacks `hacked-image-file-attack` and `hacked-code-file-attack` with some probabilities as specified in the following.

```
(define-attack-model maf-attacks
    :attack-types ( (hacked-image-file-attack .3)
     (hacked-code-file-attack .5))
    :vulnerability-mapping
     ((reads-complex-imagery hacked-image-file-attack)
     (loads-code hacked-code-file-attack)))
```

Furthermore, the two attacks are mapped to the corresponding vulnerabilities `reads-complex-imagery` and `loads-code` respectively.
Additionally, the corresponding one attack rule `bad-image-file-takeover` says that `if` we have the contextual resource `?ensemble` and `type-of-resource` is `image-file` and the `resource-might-have-been-attacked` with `hacked-image-file-attack` `then` it is highly probable (.9) that the resource has been `hacked` by `hacked-image-file-attack` as given below:

```
(defrule bad-image-file-takeover (:forward)
  if [and [resource ?ensemble ?resource-name ?resource]
  [resource-type-of ?resource image-file]
  [resource-might-have-been-attacked ?resource
   hacked-image-file-attack]]

  then [and [attack-implies-compromised-mode
   hacked-image-file-attack ?resource hacked .9 ]
    [attack-implies-compromised-mode
     hacked-image-file-attack ?resource normal .1 ]])
```

Further details on the example of the model, please see Appendix B. However, the corresponding syntactic details of the elements of the above syntactic domains are explained in the corresponding subsections of the next section. However, for the general syntax of the domain, please see Appendix A.

3 Semantics of System Architectural Model

In this section, we first give the definition of semantic algebras and then discuss informal description and the formal semantics of the core constructs of the System Architectural Model.

3.1 Semantic Algebras

The definition of a formal denotational semantics is based on a collection of data structures. *Semantic domains* represent set of elements that share some common properties. A semantic domain is accompanied by a set of operations as functions over the domain. A domain and its operations together form a *semantic algebra* [4]. In the following we enlist the semantic domains and their corresponding operations. Some operations are defined and some are just declared for the purpose of completeness of this document.

3.1.1 Truth Values

The truth values are represented by the semantic domain "Bool" which is defined as follows:

Domain: Bool
Operations:

- `true`: Bool
- `false`: Bool
- and: Bool $\times$ Bool $\rightarrow$ Bool
- or: Bool $\times$ Bool $\rightarrow$ Bool
- not: Bool $\times$ Bool $\rightarrow$ Bool

3.1.2 Numeral Values

Here we consider typical domains to represent integer and float values (e.g. $\mathbb{Q}, \mathbb{N}$).

3.1.3 Environment Values

The domain *Environment* holds the environment values of the System Architectural Model. *Environment* is formalized as a tuple of domains *Context* and *Space*. The domain *Context* is a mapping of identifiers to the environment values (*Variable*, *Component*, *Resource*, *RTEvent* and *Function*), while the domain *Space* models the memory space.

Domain: Environment
Environment := Context $\times$ Space
Context := Identifier $\rightarrow$ EnvValue

EnvValue := Variable + Component + Resource + RTEvent
+ Function + AtkModel

Space := $\mathbb{P}$(Variable)
Variable := n, where n $\in \mathbb{N}$ represents locations
Operations:

- space: Environment $\rightarrow$ Space
  space(<c,s>) = s

- context: Environment $\rightarrow$ Context
  context(<c,s>) = c

- environment: Context $\times$ Space $\rightarrow$ Environment
  environment(c,s) = <c, s>

- take: Space $\rightarrow$ Identifier $\times$ Space
  take(s) = LET x = SUCH x: x $\in$ s IN <x, s$\setminus\{$x$\}$>

- push: Environment $\times$ Identifier $\rightarrow$ Environment

  push(e, I) = LET <x, s'>= take(space(e)) IN
  environment(context(e)[I $\mapsto$ inVariable(x)], s')

- push: Environment $\times$ Identifier $\times$ Component $\rightarrow$ Environment

  push(e, I, c) = LET <x, s'>= take(space(e)) IN
  environment(context(e)[I $\mapsto$ inComponent(c)], s')

- push: Environment $\times$ Identifier $\times$ AtkModel $\rightarrow$ Environment

  push(e, I, m) = LET <x, s'>= take(space(e)) IN
  environment(context(e)[I $\mapsto$ inAtkModel(m)], s')

3.1.4 State Values

This section defines the domain for the *State* of the execution of program. A *Store* is the most important part of the state and holds for every *Variable* a *Value*. The value can be read and modified. The *Data* of the state is a tuple of a *Flag* that represents the current status of the state and a *Mode* to represent the current mode of execution of the state respectively component.

Domain: State
State := Store $\times$ Data
Store := Variable $\rightarrow$ Value
Data := Flag $\times$ Mode
Flag := {running, ready, completed}
Mode := {normal, compromised}
Operations:

- state: Store $\times$ Flag $\rightarrow$ State
  state(s,f) = <s,f>
- store: State $\rightarrow$ Store
  store(<s,f>) = s
- data: State $\rightarrow$ Data
  data(<s,d>) = d
- flag: Data $\rightarrow$ Flag
  flag(<f,m>) = f
- mode: Data $\rightarrow$ Mode
  mode(<f,m>) = m
- setFlag: State $\times$ Flag $\rightarrow$ State
  setFlag(s, f) = LET d = <f, mode(data(s))>IN <s, d>
- setMode: State $\times$ Mode $\rightarrow$ State
  setMode(s, m) = LET d = <flag(data(s)), m>IN <s, d>
- eqFlag: State $\times$ Flag $\rightarrow$ Bool
  eqFlag(s, f) = IF equals(flag(data(s)), f) THEN `true` ELSE `false` END
- eqMode: State $\times$ Mode $\rightarrow$ Bool
  eqMode(s, m) = IF equals(mode(data(s)), m) THEN `true` ELSE `false` END
- update: State $\times$ Variable $\times$ Value $\rightarrow$ State
  update(s, var, val) = state(store(s)[var $\mapsto$ val], flag(s))

3.1.5 Semantic Values

Value is a disjunctive union domain and note that the domain *Value* is a recursive domain, e.g. *List* is defined by *Value** as discussed in the next section.

Domain: Value

Value := Event + ObsEvent + RTEvent + Function + Component + Split + Resource + AtkModel + String + List + ... + Value^*

Operations:

- equals: Value $\times$ Value $\rightarrow$ Bool

3.1.6 Character String Values

Character strings are defined as a semantic domain *String*.

3.1.7 Lifted Values

The evaluation of some semantic domains might result as unsafe. To address these unsafe evaluations we lifted the domains of *State* and *Value* to domains $\text{State}_\perp$ and $\text{Value}_\perp$, which are disjoint sums of the basic domains and the domain $\perp$.

In order to capture different kind of events we need different semantic domain to model each of them. The three kind of events are:

1. *Registered Events* are the events of interest for monitoring. These events are defined by the user at the top of the System Architectural Model by the syntactic domain "RegModSeq" as discussed in Appendix A. AWDRAT register these events for the monitoring purposes.
2. *Observed Events* are the entry, exit and allowable events as defined by the syntactic domain "Event" of the System Architectural Model.
3. *Run Time Events* are the runtime events that are generated by the monitor from the target system. These events are also called *observations*.

In the following, we give definitions of the corresponding semantic domains respectively.

3.1.8 (Registered) Event Values

The semantics domain *Event* defines the registered events as a predicate over a sequence of input values, sequence of output values, a pre-state and a corresponding post-state as follows:

$$\text{Event} := \mathbb{P}(\text{Value}^* \times \text{Value}^* \times \text{State} \times \text{State}_\perp)$$

3.1.9 (Observed) Event Values

The semantics domain *ObsEvent* formalizes the observed events of System Architectural Model. An *ObsEvent* is defined as a predicate over a sequence of input values, a pre-state and a post-state as follows:

$$\text{ObsEvent} := \mathbb{P}(\text{Value}^* \times \text{State} \times \text{State}_\perp)$$

Note that the observed events do not capture output values because they just work as placeholders for runtime and registered events.

3.1.10 (Runtime) Event Values

The runtime events of System Architectural Model are formalized with the help of a semantic domain *RTEvent*. The semantic domain *RTEvent* is defined as a predicate over a sequence of input values, sequence of output values, a pre-state, post-state and event data as follows::

$$\text{RTEvent} := \mathbb{P}(\text{Value}^*{}_\perp \times \text{Value}^* \times \text{State} \times \text{State}_\perp \times \text{EventData})$$

where
EventData := Tag $\times$ TimeStamp $\times$ ProcessID
Tag := {entry, exit}
TimeStamp := date and time of the event execution
ProcessID := operating system process id for the event

An *EventData* captures the type of an event, time of event generation and an operating system level process id for this event. Note, process identification provides more low-level information about the event which is helpful to detect any misbehavior of the event correspondingly component.

3.1.11 Resource Values

The semantic domain *Resource* is one of the complex domains because semantically this domain depends on the runtime behavior of an associated components as well. The semantics domain *Resource* formalizes different kind of resources used by computational modules of System Architectural Model and is also defined as a predicate over a

- map which is further a predicate over
  - a mode,
  - its likelihood value being in normal mode,
  - corresponding likelihood being in hacked mode and
  - an associated vulnerability,
- current mode of the resource,

- probability of the resource being in the current mode,
- name of the running component associated with the resource,
- mode of the running associated component,
- a pre-state and a post-state of the program.

The predicate *Resource* is mathematically defined as:

$$\text{Resource} := \mathbb{P}(\text{ModeMap} \times \text{Mode} \times \text{FVal} \times \text{I} \times \text{Mode} \times \text{State} \times \text{State}_{\perp})$$

where
$\text{Mode} := \{\text{normal, compromised}\}$
$\text{ModeMap} := \mathbb{P}(\text{Mode} \times \text{Fval} \times \text{FVal} \times \text{Vulnerability})$

3.1.12 Function Values

The semantics domain *Function* defines and formalizes a specification function of System Architectural Model and can be defined mathematically as:

$$Function = \bigcup_{n \in \mathbb{N}} Function^n$$

where

$$Function^n = Value^n \rightarrow Value$$

3.1.13 Component Values

The semantics domain *Component* formalizes the model of the components of the target system which are specified by the corresponding behaviors in the System Architectural Model. A *Component* is defined as a predicate over a structural behavior of the component, a normal behavior of the component, its corresponding compromised behavior, a pre-state and a post-state of the program as follows:

$$\text{Component} = \mathbb{P}(\text{SBehavior} \times \text{NBehavior} \times \text{CBehavior} \times \text{State} \times \text{State}_{\perp})$$

where

$\text{SBehavior} := \mathbb{P}(\text{Value}^* \times \text{Value}^* \times \text{Value}^* \times \text{State} \times \text{State}_{\perp})$
$\text{NBehavior} = \text{CBehavior} := \mathbb{P}(\text{Value}^* \times \text{Value}^* \times \text{State} \times \text{State}_{\perp})$

Furthermore, a structural behavior is defined as a predicate over a sequence of input values, sequence of output value, sequence of allowable values (as a consequence of allowable events), a pre-state and a post-state of the behavior. Also, a normal (functional) behavior and corresponding compromised behavior are defined as a predicates *NBehavior* and *CBehavior* over a sequence of input values, sequence of output values, a pre-state and a corresponding post-state respectively. Note that the two predicates are the valuation functions of corresponding syntactic domains.

3.1.14 Split Values

The semantics domain *Split* formalizes the control flow behavior of a certain unit of a computational module of System Architectural Model and is defined as a predicate over a sequence of parameter values of the split as follows:

$$\text{Split} := \mathbb{P}(\text{Value}^*)$$

3.1.15 Attack Values

The semantics domain *AtkModel* formalizes the attack model and is defined as a predicate over an attack name, probability of the attack and the corresponding vulnerability causing the attack; the attack model is formulated as follows:

$$\text{AtkModel} := \mathbb{P}(\text{Identifier} \times \text{FVal} \times \text{Vulnerability})$$

These values are the result of the valuation function for the corresponding syntactic domain.

3.2 Signatures of Valuation Functions

A valuation function defines a mapping of a language's abstract syntax structures to its corresponding meanings (semantic algebras) [4]. A valuation function VF for a syntax domain VF is usually formalized by a set of equations, one per alternative in the corresponding BNF Register for each syntactic domain of specification expression.

We define the result of valuation function as a predicate. In this section we first give the definitions of various relations and functions that are used in the definition of valuation functions. For example the behavioral relation (BehRelation) is defined as a predicate over an environment, a pre-state and a post-state. The corresponding relation is defined as follows:

$$\text{BehRelation} := \mathbb{P}(\text{Environment} \times \text{State} \times \text{State}_\perp)$$

3.2.1 System Architectural Model

The valuation function for the abstract syntax domain system architectural model values of SAM is defined as follows:

$$[\![\text{SAM}]\!]\text{: Environment} \rightarrow \text{BehRelation}$$

3.2.2 Behavioral Models

The valuation functions for abstract syntax domains of Register, structural, behavioral and split model values (RuleMod, StrMod, BehMod and SpltMod respectively) are the same and can be defined similarly; however in the following we give only the signature of valuation function for the behavioral model:

$$[\![\text{BehMod}]\!]: \text{Environment} \rightarrow \text{BehRelation}$$

In the following section we define the auxiliary functions and predicates used in the formal semantics of the specification language (and associated domains).

3.3 Auxiliary Predicates and Functions

In the following subsections auxiliary functions and predicates for the use in semantics definition of sequence, binding and special expressions are defined.

- **monitors** $\subset \mathbb{N} \times$ **RTEvent** $\times$ **Component** $\times$ **Environment*** $\times$ **Environment*** $\times$ **State*** $\times$ **State**$^*_\perp$

```
monitors(i, ⟦rte⟧, ⟦c⟧, e, e', s, s') $\Leftrightarrow$
( eqMode(s(i), "running") $\vee$ eqMode(s(i), "ready") ) $\wedge$ ⟦c⟧(e(i))(e'(i), s(i), s'(i)) $\wedge$
  $\exists$ oe $\in$ ObEvent: equals(rte, store(⟦name(rte)⟧)(e(i))) $\wedge$
  IF entryEvent(oe, c) THEN
      data(c, s(i), s'(i)) $\wedge$
      ( preconditions(c, e(i), e'(i), s(i), s'(i), "compromised") $\Rightarrow$
      equals(s(i+1), s(i)) $\wedge$ equals(s'(i+1), s(i+1)) $\wedge$
      setFlag(inState(s'(i+1)), "compromised") ) $\vee$
      ( preconditions(c, e(i), e'(i), s(i), s'(i), "normal")
      $\Rightarrow$ setMode(s(i), "running") $\wedge$
      LET cseq = components(c) IN
            equals(s(i+1), s'(i)) $\wedge$ equals(e(i+1), e'(i)) $\wedge$
            $\forall$ $c_1$ $\in$ cseq, $rte_1$ $\in$ RTEvent:
               arrives($rte_1$, s(i+1)) $\wedge$
               monitor(i+1, $rte_1$, $c_1$, e(i+1), e'(i+1), s(i+1), s'(i+1))
      END )
  ELSE IF exitEvent(oe, c) THEN
            data(c, s(i), s'(i)) $\wedge$ eqMode(inState(s'(i)), "completed") $\wedge$
            ( postconditions(c, e(i), e'(i), s(i), s'(i), "compromised") $\Rightarrow$
            equals(s(i+1), s(i)) $\wedge$ equals(s'(i+1), s(i+1)) $\wedge$
            setFlag(inState(s'(i+1)), "compromised") ) $\vee$
            ( postconditions(c, e(i), e'(i), s(i), s'(i), "normal")
                    $\Rightarrow$ equals(s(i+1), s'(i)) $\wedge$ equals(e(i+1), e'(i)) $\wedge$
                       setMode(inState(s'(i+1), "completed") )
  ELSE IF allowableEvent(oe, c) THEN
            equals(s(i+1), s'(i)) $\wedge$ equals(e(i+1), e'(i))
  ELSE
         equals(s(i+1), s(i)) $\wedge$ equals(s'(i+1), s(i+1)) $\wedge$
         setFlag(inState(s'(i+1)), "compromised")
  END
```

The predicate "monitors" captures the core semantics of the monitor which is defined as a relation on

- number of observation i with respect to iteration of a component,
- an observation (runtime event) rte,
- corresponding component c under observation,
- a sequence of pre-environments e,
- a sequence of post-environments e',
- a sequence of pre-states s and
- a sequence of post-states s'.

The predicate *monitors* is defined such that, at any arbitrary observation if the current execution state $s(i)$ of component c is "ready" or "running" and behavior of component c has been evaluated and there is a *prediction oe* which is semantically equal to an *observation rte* and any of the following can happen:

- either the *prediction* respectively *observation* is an entry event of the component c, then it waits until the complete data for the component c arrives, if so, then
  - either preconditions of "normal" behavior of the component hold; if so then, the subnetwork of the component is initiated and the components in the subnetwork are monitored iteratively with the corresponding arrival of the *observation*
  - or preconditions of "compromised" behavior of the component hold, in this case the state is marked to "compromised" and returns
- or the *observation* is an exit event and after the completion of data arrival the postconditions hold and the resulting state is marked as "completed"
- or the *observation* is an allowable event and just continues the execution
- or the *observation* is an unexpected event (or any of the above does not hold), then the state is marked as "compromised" and returns.

The predicate *monitors* is used later in the semantics of the Execution Monitor.

- **entryEvent** $\subset$ **ObEvent** $\times$ **Component**: returns `true` only if the given event is in a set of entry events of the given component.
- **exitEvent** $\subset$ **ObEvent** $\times$ **Component**: returns `true` only if the given event is in a set of exit events of the given component.
- **allowableEvent** $\subset$ **ObEvent** $\times$ **Component**: returns `true` only if the given event is in a set of allowable events of the given component.

- **data** $\subset$ **Component** $\times$ **State** $\times$ **State**$_\perp$: returns `true` only if all the data for the given component is received by transforming a given pre-state (former) into a corresponding given post-state (latter).
- **arrives** $\subset$ **RTEvent** $\times$ **State**: returns `true` only if the given runtime event (observation) arrives in a given state.
- **preconditions** $\subset$ **Component** $\times$ **Environment** $\times$ **Environment** $\times$ **State** $\times$ **State**$_\perp$ $\times$ **Mode**

  returns `true` only if all the preconditions of the given component hold in a given pair of pre- and post-environments, a pair of pre- and post-states and in a given mode.
- **postconditions** $\subset$ **Component** $\times$ **Environment** $\times$ **Environment** $\times$ **State** $\times$ **State**$_\perp$ $\times$ **Mode**

  returns `true` only if all the postconditions of the given component hold in a given pair of pre- and post-environments, a pair of pre- and post-states and in a given mode.
- **startup** $\subset$ **State** $\times$ **Target_System**: returns `true` only if the given state is an initial state of the execution of the given target system.
- **isTop** $\subset$ **Component** $\times$ **(Environment** $\rightarrow$ **BehRelation)**: returns `true` only if the given component is a top level component of the given semantics of a System Architectural Model.
- **enableDiagnosis: Environment** $\rightarrow$ $\mathbb{P}$(**State** $\times$ **Value**): results in a given recovered state and a boolean value (`true`, if recovered safely, `false` otherwise) from a given environment.
- **respectsOrder** $\subset$ **Identifier_Sequence** $\times$ **Identifier_Sequence**

  returns `true` only if the identifiers in the latter sequence has the same order as the identifiers in the former sequence.
- **buildEnv** $\subset$ **Environment** $\times$ **List*** $\times$ **List*** $\times$ **List*** $\times$ **List*** $\rightarrow$ **Environment**

  builds the resulting environment by updating the given environment with given sequences of values of

  - resources
  - resource mappings
  - model mappings and
  - vulnerabilities

  respectively.

3.4 Definition of Valuation Functions

In this section we give the definition of the formal semantics of the interesting syntactic domains (and associated domains) of the specification language, e.g. system architectural model, Register mode, behavioral model and split model. The semantics of other domains of the specification language are very simple and can be easily rehearsed.

3.4.1 System Architectural Model

The System Architectural Model is give by the syntactic domain SAM such that

- the Register model (syntactic domain *RegModSeq*) is defined at the top of the System Architectural Model which gives the registered events to be monitored at runtime
- followed by
  - a hierarchical structural behavior (syntactic domain *StrModSeq*) of components,
  - a normal respectively compromised behavior (syntactic domain *BehModSeq*) and
  - corresponding split behaviors (syntactic domain *SplModSeq*) occurring in any of the structural behavior of the components.

For further details on the syntax of the model, please see Appendix A.

Semantically, an overall (system architectural) model holds (`true`) in a given environment e such that it produces a new environment e' and a post-state e' when executed in a pre-state s as defined below.

$[\![\text{SAM}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow$
$\forall\, e_1, e_2, e_3 \in \text{Environment},\, s_1, s_2, s_3 \in \text{State}$:
$[\![\text{RegModSeq}]\!](\text{e})(\text{e}_1, \text{s}, \text{inState}_\perp(\text{s}_1)) \wedge [\![\text{StrModSeq}]\!](\text{e}_1)(\text{e}_2, \text{s}_1, \text{inState}_\perp(\text{s}_2)) \wedge$
$[\![\text{BehModSeq}]\!](\text{e}_2)\ (\text{e}_3, \text{s}_2, \text{inState}_\perp(\text{s}_3)) \wedge [\![\text{SpltModSeq}]\!](\text{e}_3)(\text{e'}, \text{s}_3, \text{s'})$

In detail, the semantics of the System Architectural Model SAM holds in a given environment e resulting in an environment e' by transforming a pre-state s into post-state s' and

- the evaluation of the registered events in a given environment e results in environment e_1 transforming a pre-state s into a post-state s_1 and in principle
  - the structural behavior of components hold in environment e_1 (with some auxiliary transformations) and

– the functional behavior of components hold in environment e_2 (with some auxiliary transformations) and finally

– the split behavior of components hold in e_3 resulting in given environment s' and transforming a pre-state s_3 into a given post-state s'.

In the following, first we define the semantics of unit elements *RegMod*, *StrMod*, *BehMod* and *SplModSeq* and then define corresponding sequence domains *RegModSeq*, *StrModSeq*, *BehModSeq* and *SplModSeq* respectively.

3.4.2 Register Model

The syntactic domain (RegMod) defines a registered event as follows:

```
RegMod ::= register-event 'EvntName JavClaName JavMetName '(JavParamSeq)
                          [ :static ObjName ]
                          [ :output-type JavParam ]
                          [ :bypass ObjNameStr ]
                          [ :EvntName ObjName]
```

Though the domain represents language independent event registration, in this document we focus only on the the Java based target system. The syntactic phrase *RegMod* states that a registered event can be represented by a name (EvntName) whose source is a Java method (JavMetName) with parameters (JavParmSeq) of corresponding class (JavClaName). The other sub-clauses introduce further characterization of the method, e.g. the clause **:output-type** represents the return type of the method.

A monitoring machinery of Architectural Differencer of the middleware AWDRAT is based on these registered events.

In the following we define the semantics of a registered event such that the evaluation of a registered event in a given environment e results in an environment e' transforming a pre-state s into a post-state s'.

```
[[RegMod]](e)(e', s, s') ⇔
 ∀ e_j ∈ JTypeEnvironment, s_j, s_j' ∈ JState:
  typeCheck(JavClaName)(e_j)(s_j, s'_j) ∧ equals(s, s_j) ∧ equals(e, e_j)
  ∧ (∃ p ∈ JProcedure: equals(p(valseq, val), store(s_j')([[JavMetName]](e_j)))
        ∧ equals(valseq, store(s_j')([[JavParamSeq]](e_j)))
        ∧ equals(val, store(s_j')([[JavParam]](e_j))))
  ∧ isStatic(...) ∧ byPass(...) ∧ otherEvents(...)
  ∧ e' = push(e, EvntName)
  ∧ LET ev(valseq, val, s, s') ∈ Event IN
          s' = update(s, [[EvntName]](e'), ev)
    END
```

In detail, semantically, the Java class (JavClaName) is well-defined respectively type checked in an arbitrary environment e_j transforming an arbitrary pre-state s_j into a corresponding arbitrary post-state s_j' while e_j and s_j are semantically equivalent to s and e respectively and

- there is some Java procedure $p(valseq, val)$ which we get by evaluating "JavMetName" with given environment e_j such that the sequence of input values *valseq* equals the evaluation of "JavParamSeq" in given environment e_j and the return value *val* of procedure p equals the evaluation of "JavParam" in environment e_j and
- finally we get e' by pushing "EvntName" in given environment e and
- s' is produces by updating the value *ev* for an identifier "EvntName" in the given pre-state s.

3.4.3 Register Model Sequence

The syntax of the syntactic domain RegModSeq is defined as follows:

RegModSeq := EMPTY | (RegMod) RegModSeq

Semantically, when an EMPTY sub-phrase is evaluated in a given environment e then simply the resulting environment e' equals e and a post-state s' equals the given pre-state s as defined below:

Case: EMPTY

$[\![\text{EMPTY}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow \text{e'} = \text{e} \wedge \text{s'} = \text{inState}_{\perp}(\text{s})$

While in the second alternate of the domain "RegModSeq", first semantics of the phrase "RegMod" in a given environment e produce an environment e'' transforming a pre-state s into a post-state s'', then the evaluation of the phrase "RegModSeq" in environment e'' results in a given environment e' and transforms the pre-state s'' into a given post-state s'. The semantics of the second alternative is formalized as follows:

Case: (RegMod) RegModSeq

$[\![\text{(RegMod) RegModSeq}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow$
$\forall \text{ e"} \in \text{Environment, s"} \in \text{State:}$
$[\![\text{RegMod}]\!](\text{e})(\text{e"}, \text{s}, \text{inState}_{\perp}(\text{s"})) \wedge [\![\text{RegModSeq}]\!](\text{e"})(\text{e'}, \text{s"}, \text{s'})$

3.4.4 Structural Model

The structural behavior of the system is defined by the syntactic phrase "StrMod" which represents a corresponding hierarchical model of the components. The syntax for the overall structural behavior of the component

"CompName" is defined by the syntactic phrase "StrMod" where different clauses define three logical parts of the behavior as follows:

1. *signals* specify global control behavior of the component, e.g.
   - the clauses **:entry-events** and **:exit-events** models the entry and exit events of the component respectively and
   - the other allowable events (while execution of the component) are modeled with the clause **:allowable-events**
2. *signature* of the component consists of
   - the sequences of objects for the clauses **:inputs** and **:outputs** respectively
3. *body* of the component is modeled as a sub-network which involves different components as represented by the **:components** clause. These components are connected through various nodes and links as follows:
   (a) the control flows **:controlflows** which further have corresponding splits **:splits** and joins **:joins** and
   (b) the propagation of data among the components (via control flows) is represented by the clause **:dataflows**.
   (c) while the execution of the body, various computing resources **:resources** (each with a name, its type and its probabilities of being in normal and hacked modes respectively) are involved which further requires
   (d) the resource mappings **:resource-mappings** (where each resource is mapped to a component that uses it) in addition to
   (e) the model mappings **:model-mappings** (where the conditional probability between the compromises and misbehaviors for each of the component is given) and
   (f) the vulnerabilities **:vulnerabilities** such that each resource is mapped to a corresponding (possible) vulnerability (which is assumed to be defined as the part of an attack plan that is beyond the scope of this document).

The syntactic domain of for the structural behavioral model (StrMod) is defined as follows:

StrMod ::= **define-ensemble** CompName
:entry-events **:auto** | (EvntSeq$_1$)
:exit-events (EvntSeq$_2$)
:allowable-events (EvntSeq$_3$)
:inputs (ObjNameSeq$_1$)

```
                    :outputs (ObjNameSeq$_2$)
                    :components (CompSeq)
                    :controlflows (CtrlFlowSeq)
                    :splits (SpltCFSeq)
                    :joins (JoinCFSeq)
                    :dataflows (DataFlowSeq)
                    :resources (ResSeq)
                    :resource-mapping (ResMapSeq)
                    :model-mappings (ModMapSeq)
                    :vulnerabilities (VulnrabltySeq)
```

The semantics of the structural behavioral model in a given environment e results in an environment e' transforming a pre-state s into a post-state s' as defined below:

```
$[\![$StrMod$]\!]$(e)(e', s, s') $\Leftrightarrow$
 $\forall$ $e, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8 \in$ Environment, $s, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8 \in$ State,
   oeseq, oeseq$_1$, aeseq $\in$ ObsEvent*, anameseq, enameseq, enameseq$_1$ $\in$ EvntNameSeq:
   ( eqFlag(s, "running") $\wedge$
   $[\![$EvntSeq$_3]\!]$(e)(e$_1$, s, inState$_\perp$(s$_1$), enameseq, oeseq) $\wedge$
   $\forall$ ename $\in$ enameseq:
     $\exists$ se $\in$ Event, rte $\in$ RTEvent, oe $\in$ oeseq:
     IF equals(se, oe) THEN
         LET rte = store(s$_1$)(ename) IN
               IF equals(rte, se) THEN
                   true
               ELSE
                   s$_1$ = enableDiagnosis(e$_1$)(s$_1$, inBool(true))
               END
         END
     ELSE
         s$_1$ = enableDiagnosis(e$_1$)(s$_1$, inBool(true))
     END )
   $\vee$
   ( eqFlag(s, "running") $\vee$ eqFlag(s, "ready") $\wedge$
   $[\![$EvntSeq$_1]\!]$(e)(e$_1$, s, inState$_\perp$(s$_1$), enameseq, oeseq) $\wedge$
   $\forall$ ename $\in$ enameseq, oe $\in$ oeseq:
     $\exists$ se $\in$ Event, rte $\in$ RTEvent:
     equals(se, store(s$_1$)(ename)) $\wedge$ equals(se[1], oe[1]) $\wedge$
     LET rte = store(s$_1$)(ename) IN
           IF equals(rte[5][1], "entry" ) THEN
               equals(rte[1], se[1])
           ELSE equals(rte[2], se[2])
           END
```

```
  END $\wedge$
  $\forall$ inseq $\in$ Value$^*$, c $\in$ Component:
    $[\![$ObjNameSeq$_1]\!]$($e_1$)(inState$_\perp$($s_1$), inseq) $\wedge$ $[\![$CompName$]\!]$($e_1$)(inValue(c)) $\wedge$
    IF equals(c[2][1], inseq) THEN
        eqMode($s_1$, “normal”)
    ELSE
        $s_1$ = enableDiagnosis($e_1$)($s_1$, inBool(true))
    END $\wedge$
    IF equals(c[3][1], inseq) THEN
        eqMode($s_1$, “compromised”) $\wedge$ $s_1$ = enableDiagnosis($e_1$)($s_1$, inBool(true))
    ELSE true
    END )
$\Rightarrow$ eqFlag($s_1$, “running”) $\wedge$
   $\forall$ compseq $\in$ Component*: $[\![$CompSeq$]\!]$($e_2$)($e_3$, $s_2$, inState$_\perp$($s_3$), compseq) $\wedge$
   $\forall$ rmseq, crmapseq, cpmapseq, vbltyseq $\in$ List*:
    $[\![$ResSeq$]\!]$($e_3$)($s_3$, inState$_\perp$($s_4$), rmseq) $\wedge$
    $[\![$ResMapSeq$]\!]$($e_3$)($s_3$, inState$_\perp$($s_4$), crmapseq) $\wedge$
    $[\![$ModMapSeq$]\!]$($e_3$)($s_3$, inState$_\perp$($s_4$), cpmapseq) $\wedge$
    $[\![$VulnrabltySeq$]\!]$($e_3$)($s_3$, inState$_\perp$($s_4$), vbltyseq) $\wedge$
    $e_4$ = buildEnv($e_3$, rmseq, crmapseq, cpmapseq, vbltyseq) $\wedge$
    $[\![$CtrlFlowSeq$]\!]$($e_4$)($e_5$, $s_4$, inState$_\perp$($s_5$)) $\wedge$
    $[\![$SpltCFSeq$]\!]$($e_5$)($e_6$, $s_5$, inState$_\perp$($s_6$)) $\wedge$
    $[\![$JoinCFSeq$]\!]$($e_6$)($e_7$, $s_6$, inState$_\perp$($s_7$)) $\wedge$
    $[\![$DataFlowSeq$]\!]$($e_7$)($e_8$, $s_7$, inState$_\perp$($s_8$)) $\wedge$
    $[\![$EvntSeq$_2]\!]$($e_8$)($e_9$, $s_8$, s’, enameseq$_1$, oeseq$_1$) $\wedge$
    $\forall$ ename $\in$ enameseq, oe $\in$ oeseq:
      $\exists$ se $\in$ Event, rte $\in$ RTEvent:
      equals(se, store(inState(s’))(ename)) $\wedge$ equals(se[1], oe[1]) $\wedge$
      LET rte = store(inState(s’))(ename) IN
           IF equals(rte[5][1], “entry” ) THEN
              equals(rte[1], se[1])
           ELSE equals(rte[2], se[2])
           END
      END
      $\Rightarrow$
        $\forall$ outseq $\in$ Value$^*$, c $\in$ Component:
          $[\![$ObjNameSeq$_2]\!]$($e_9$)(s’, outseq) $\wedge$ $[\![$CompName$]\!]$($e_9$)(inValue(c)) $\wedge$
          ( ( IF equals(c[2][2], outseq) THEN
                 eqMode(inState(s’), “normal”)
          ELSE
                 s’ = enableDiagnosis($e_9$)(inState(s’), inBool(true))
          END ) $\vee$
          ( IF equals(c[3][2], outseq) THEN
               eqMode(inState(s’), “compromised”) $\wedge$
               s' = enableDiagnosis(e_9)(inState(s'), inBool(true))
          END ) ) ∧
          eqMode(inState(s'), "normal") ∧
          eqFlag(inState(s'), "completed") ∧
          LET sbeh = <inseq, outseq, s, s'>, nbeh = c[2], cbeh = c[3] IN
                e' = push(e_9, store(inState(s'))(⟦CompName⟧(e_9))
                               , c(sbeh, nbeh, cbeh, s, s'))
          END
```

In general, the semantics is defined as a big logical implication, where the premise is a disjunction of two formulas as explained below:

1. either the current state *s* of the component is "running" and it receives allowable events "EvntSeq$_3$" and for every event *oe* in the allowable event sequence *oeseq* there is a corresponding equivalent registered event *se* for which we receive an equivalent runtime event *rte* such that *rte* is one of the under observation (legal) event *se*, if not then the runtime event is a result of the misbehavior of the component so the diagnosis component of AWDRAT is activated by calling "enableDiagnosis(...)" which successfully recovers the compromised state s_1

2. or the current state *s* of the component is either "running" or "ready" and it receives the entry events "EvntSeq$_1$" (evaluating to *oeseq*) and for every event *oe* in the sequence of entry events *oeseq* there is a corresponding registered event *se* and the received runtime event *rte* (equals *se* depending on its type "entry" or "exit") is the monitored event and

    - if the sequence of input values *inseq* satisfies the pre-conditions of the "normal" behavior (c[2][1])) of the component (c) then the resulting state s_1 is in "normal" mode
    - otherwise (when pre-conditions are not satisfied) then the diagnosis component is activated which recovers the compromised state s_1 and
    - if the sequence of input values *inseq* satisfies the pre-conditions of the already "compromised" behavior (c[3][1]) of the component (c) then the resulting state s_1 is "compromised" state and we restore it by enabling diagnostic engine
    - otherwise the system is safe `true` to start executing component respectively *body*/sub-network.

Semantically, if any of the above two holds then the

- the current state s_1 is "running" and the components of the sub-network evaluate to *compseq* and

- a new environment e_4 is constructed based on the evaluation of the resources, resource mappings, model mappings and vulnerabilities of the sub-network to *rmseq*, *crmapseq*, *cpmapseq*, and *vbltyseq* respectively (such that all the trust model of the components is known before the actual execution of the *body* starts) in which

- the execution blocks are evaluated (such that the evaluation of the control flows their respective splits and joins and associated data flows results in an environment e_8 and a post-state s_8) to complete the executional behavior and

- once all the sub-network is executed (recursively), then the receiving exit events (EvntSeq_2) evaluate to $oeseq_1$ and if for every event *oe* in $oeseq_1$ there is an equivalent registered event *se* and a runtime event *rte* then

  – either the sequence of output values *outseq* satisfies the post-conditions of the "normal" behavior (c[2][2]) of the component (c) then the post-state s' is in "normal" mode otherwise the diagnosis component restores the post-state s' and

  – or the sequence of output values *outseq* satisfies the post-conditions of the misbehavior (c[3][2]) of the component (c) then the post-state must be in "compromised" mode and corresponding diagnosis component is enabled to recover the state back and

- the given and transformed final post-state s' must be in "normal" mode with "completed" flag and

- finally the resulting environment e' is build with the evaluated behavior of the component of the current component.

3.4.5 Structural Model Sequence

The syntactic domain StrModSeq is:

StrModSeq := EMPTY | (StrMod) StrModSeq

Case: EMPTY

$[\![\text{EMPTY}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow \text{e'} = \text{e} \wedge \text{s'} = \text{inState}_\perp(\text{s})$

Case: (StrMod) StrModSeq

$[\![\text{(StrMod) StrModSeq}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow$
$\forall$ e" $\in$ Environment, s" $\in$ State:
$[\![\text{StrMod}]\!](\text{e})(\text{e''}, \text{s}, \text{inState}_\perp(\text{s''})) \wedge [\![\text{StrModSeq}]\!](\text{e''})(\text{e'}, \text{s''}, \text{s'})$

The semantics of the domain “StrModSeq” of structural model sequence are similar to the semantics of the domain “RegModSeq” as discussed above in the corresponding section. Similarly, the semantics of the syntactic domains of “BehModSeq” and “SplModSeq” can be exercised which are discussed in the corresponding sections later in this document.

3.4.6 Behavioral Model

The behavioral model represents the functional behavior of a component, which can be either “normal” or known “compromised” one. The functional behavior of the component “CompName” consists of the following elements:

1. the inputs of the component as given by the clause **:inputs**,
2. the outputs of the component as represented by the corresponding clause **:outputs**,
3. the allowable events **:allowable-events** represents the auxiliary communication of the component,
4. the pre-conditions of the component are specified in the clause **:prerequisites** while
5. the corresponding post-conditions are specified by the **:postconditions** clause.

Note that the “compromised” behavior is used to model already known misbehaviors of the component (e.g. some attack) and needs corresponding diagnosis which in this case is already known.

The syntactic domain “BehMod” for the behavioral model is defined as follows:

BehMod ::= **defbehavior-model** (CompName **normal** | **compromised**)
:inputs (ObjNameSeq_1)
:outputs (ObjNameSeq_2)
:allowable-events (EvntSeq)
:prerequisites (BehCondSeq_1)
:postconditions (BehCondSeq_2)

Semantically, normal and compromised behavioral models results in modifying the corresponding elements of the environment value “Component” as defined below:

$[\![\text{BehMod}]\!](e)(e', s, s') \Leftrightarrow$
$\forall\, e_1 \in \text{Environment}, \text{nseq} \in \text{EvntNameSeq}, \text{eseq} \in \text{ObsEvent}^*, \text{inseq}, \text{outseq} \in \text{Value}^*:$
$[\![\text{ObjNameSeq}_1]\!](e)(\text{inState}_\perp(s), \text{inseq}) \wedge [\![\text{BehCondSeq}_1]\!](e)\ (\text{inState}_\perp(s)) \wedge$
$[\![\text{EvntSeq}]\!](e)\ (e_1, s, s', \text{nseq}, \text{eseq})$

```
⟦ObjNameSeq$_2$⟧(e$_1$)(s', outseq) $\wedge$ ⟦BehCondSeq$_2$⟧(e$_1$) (s') $\wedge$
$\exists$ c $\in$ Component: ⟦CompName⟧(e$_1$)(inValue(c)) $\wedge$
IF eqMode(inState$_\perp$(s'), "normal") THEN
  LET sbeh = c[1], nbeh = <inseq, outseq, s, s'>, cbeh = c[3] IN
      e' = push(e$_1$, store(inState(s'))(⟦CompName⟧(e$_1$))
                , c(sbeh, nbeh, cbeh, s, s'))
  END
ELSE
  LET sbeh = c[1], nbeh = c[2], cbeh = <inseq, outseq, s, s'>  IN
      e' = push(e$_1$, store(inState(s'))(⟦CompName⟧(e$_1$))
                , c(sbeh, nbeh, cbeh, s, s'))
  END
END
```

In detail, if the semantics of of syntactic domain "BehMod" holds in a given environment e resulting in environment e' and transforming a pre-state s into corresponding post-state s' then

- the inputs "ObjNameSeq$_1$" evaluates to a sequence of values *inseq* in a given environment e and a given state s which satisfies the corresponding pre-conditions "BehCondSeq$_1$" in the same e and s and
- the allowable events happens whose evaluation results in new environment e_1 and given post-state s' with some auxiliary sequences *nseq* and *eseq* and
- the outputs "ObjNameSeq$_2$" evaluates to a sequence of values *outseq* in an environment e_1 and given post-state s' which satisfies the corresponding post-conditions "BehCondSeq$_2$" in the same environment e_1 and state s' and the given environment e' can be constructed such that
  - if the post-state is "normal" then e' is an update to the normal behavior "nbeh" of the component "CompName" in environment e_1
  - otherwise e' is an update to the compromised behavior "cbeh" of the component.

In the construction of the environment e' the rest of the semantics of the component do not change as represented in the corresponding LET-IN constructs.

3.4.7 Behavioral Model Sequence

The syntactic domain BehModSeq is:

BehModSeq := EMPTY | (BehMod) BehModSeq

Case: EMPTY

$[\![\text{EMPTY}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow \text{e'} = \text{e} \wedge \text{s'} = \text{inState}_{\perp}(\text{s})$

Case: (BehMod) BehModSeq

$[\![\text{(BehMod) BehModSeq}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow$
$\forall$ e" $\in$ Environment, s" $\in$ State:
$[\![\text{BehMod}]\!](\text{e})(\text{e''}, \text{s}, \text{inState}_{\perp}(\text{s''})) \wedge [\![\text{BehModSeq}]\!](\text{e''})(\text{e'}, \text{s''}, \text{s'})$

3.4.8 Split Model

Though the splits of control flows are declared in the "StrBeh" domain but their corresponding definitions are given with the help of the domain "SplMod" which consists of its

- name "SpltModName",
- required sequence of parameters "SpltParamSeq" which are used by the various branches of the split as defined in
- the split condition branches "SpltCondSeq".

The syntax of the domain "SplMod" is given as follow:

SplMod ::= **defsplit** SpltModName**?** (SpltParamSeq) SpltCondSeq)

If the semantics of the split model "SplMod" in a given environment e results in environment e' and transforms a pre-state s into post-state s' then

- first the parameters are evaluated in a given environment e which results in an environment e_1 and sequence of values $vseq$ transforming a given pre-state s into post-state s_1 and
- the split conditions "SpltCondSeq" hold in environment e_1 producing environment e_2 and given post-state s' and finally
- given environment e' is a result of a push operation on environment e_2 updating the value of the split "SpltModName" with the one constructed by the computed values $vseq$.

The semantics of the split behavior is formalized as follows:

$[\![\text{SplMod}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow$
$\forall e_1, e_2 \in$ Environment, $s_1 \in$ State, vseq $\in$ Value*:
$[\![\text{SpltParamSeq}]\!](\text{e})(\text{e}_1, \text{s}, \text{inState}_{\perp}(\text{s}_1), \text{vseq}) \wedge$
$[\![\text{SpltCondSeq}]\!](\text{e}_1)\ (\text{e}_2, \text{s}_1, \text{s'}) \wedge$
LET s $\in$ Split IN
e' = push(e$_2$, store(inState(s'))($[\![\text{SpltModName}]\!](\text{e}_2)$), s(vseq))
END

3.4.9 Split Model Sequence

The syntactic domain SplModSeq is:

SplModSeq := EMPTY | (SplMod) SplModSeq

Case: EMPTY

$[\![\text{EMPTY}]\!](e)(e', s, s') \Leftrightarrow e' = e \wedge s' = \text{inState}_{\perp}(s)$

Case: (SplMod) SplModSeq

$[\![\text{(SplMod) SplModSeq}]\!](e)(e', s, s') \Leftrightarrow$
$\forall$ e" $\in$ Environment, s" $\in$ State:
$[\![\text{SplMod}]\!](e)(e'', s, \text{inState}_{\perp}(s'')) \wedge [\![\text{SplModSeq}]\!](e'')(e', s'', s')$

3.4.10 Attack Model

The attack model represents the different types of known/hypothetical attack, their corresponding probabilities and the respective vulnerabilities causing the attack types. The attack model "AtkModName" has:

1. types of attack and their conditional probabilities as specified by the clause :attack-types and
2. mapping between the types of attack and vulnerabilities as described by the corresponding clause **:vulnerability-mapping**.

Additionally, the attack model is extended by the rules which map conditional probabilities of the attacks and vulnerabilities. The attack rule "AtkRulName" has

1. a sequence of attack conditions which describe the attack situation as specified by the clause **if**
2. and the attack consequences which map the probabilities of attacks and vulnerabilities; the maps are represented by the clause **then**.

Note that the attack models can be used in the following ways:

- the models are already known attacks and thus already know the corresponding diagnosis
- or the models can be hypothetical attacks which can be used to generate rigorous monitors for the system.

The syntactic domain "AtkMod" for the attack model is defined as follows:

AtkMod ::= **define-attack-model** AtkModName
:attack-types (AtkTypeSeq)
:vulnerability-mapping (AtkVulnrabltyMapSeq)

While the syntactic domain "AtkRule" for defining attack rules is defined as follows:

AtkRule ::= **defrule** AtkRulName (**:forward**)
if AtkCondSeq
then AtkConsSeq

Semantically, an attack model results in the environment value "AtkModel" as defined below:

$[\![$AtkMod$]\!]$(e)(e', s, s') $\Leftrightarrow$
$\forall$ s" $\in$ State, aseq, aseq', vnseq $\in$ ISeq, apseq $\in$ Value*:
$[\![$AtkTypeSeq$]\!]$(e)(s, inState$_\perp$(s"), aseq, apseq) $\wedge$
$[\![$AtkVulnrabltyMapSeq$]\!]$(e) (s", s', aseq', vnseq) $\wedge$ respectsOrder(aseq, aseq') $\wedge$
LET amod $\in$ AtkModel IN
e' = push(e, store(inState(s'))($[\![$AtkModName$]\!]$(e)), amod(aseq, apseq, vnseq)))
END

In detail, the semantics of the syntactic domain "AtkMod" updates the environment e with a attack semantic value *amod* such that

- in a given environment e and state s, the evaluation of "AtkTypeSeq" results in a post-state s'', a sequence of attack types *aseq* and a sequence of values (conditional probabilities) *apseq* and
- in a given environment e and state s, the evaluation of "AtkVulnrabltyMapSeq" results in post-state s', a sequence of attack types *aseq'* and a sequence of vulnerabilities *vnseq* and
- the environment e' is an update of environment e with the semantic value *amod* which is a triple of
  1. a sequence of attack types,
  2. a sequence of corresponding probabilities and
  3. a sequence of vulnerabilities causing the attack types, respectively.

However, if the semantics of the syntactic domain "AtkRule" holds in an environment e, then

- there is some resource r such that (as given in "AtkCondSeq" respective "AtkCond")
  1. the resource name is "?resource-name" and

2. the resource type is "ResType" and
3. if the resource has been compromised by an attack "AtkType-Name", then

- the resource r (and its associated component c) has behavior as specified by the evaluation of consequences "AtkCodSeq" in an environment e and state s.

Formally, the semantics of the syntactic domain "AtkRule" is defined as:

$[\![\text{AtkRule}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow$
$\exists$ r $\in$ Resource, c $\in$ Component: $[\![\text{AtkCondSeq}]\!](\text{e})(\text{s}, \text{s'}, \text{r}, \text{c}) \wedge$
$[\![\text{AtkConsSeq}]\!](\text{e})$ (s, s', r, c) $\wedge$ e' = e

3.4.11 Attack Model Sequence

The syntactic domain AtkModSeq is:

AtkModSeq := EMPTY | (AtkMod) AtkModSeq

Case: EMPTY

$[\![\text{EMPTY}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow \text{e'} = \text{e} \wedge \text{s'} = \text{inState}_{\perp}(\text{s})$

Case: (AtkMod) AtkModSeq

$[\![(\text{AtkMod})\ \text{AtkModSeq}]\!](\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow$
$\forall$ e" $\in$ Environment, s" $\in$ State:
$[\![\text{AtkMod}]\!](\text{e})(\text{e"}, \text{s}, \text{inState}_{\perp}(\text{s"})) \wedge [\![\text{AtkModSeq}]\!](\text{e"})(\text{e'}, \text{s"}, \text{s'})$

4 Execution Monitor

In principle, Architectural Differencer synthesizes both the wrappers and the execution monitor where the wrappers traces the execution of the target system by creating an event stream (these traces are also called *observations*); while the role of an execution monitor is to interpret the stream against the system (Architectural Model) specification (the execution of the specification is also called *predictions*) by detecting inconsistencies between *observations* and the *predictions*, if there are any.

We have already discussed the formal syntax and semantics of the *predictions* in the previous sections, now we first give the formal syntax of the *observations* in this section and the corresponding formal semantics in the following section.

4.1 Observation Model

Each runtime event (*observation*) consists of

- a name "EvntName",
- its type, i.e. **entry** or **exit**,
- depending on the type of event
  - either sequence of event parameters (if **entry** event)
  - or a parameter representing return value of the event (if **exit** event)
- a numeric value "Numeral" representing an operating system level process id, which can be used later to get more information about the event to detect any system level threats and other technical dependencies and
- a time "TimeStamp" of the event which later can be used to detect inconsistencies in the sequence of events.

The syntax of the runtime event is defined by the syntactic domain "Obsrv" as follow:

```
Obsrv := EvntName entry | exit
                  EvntParamSeq
                  Numeral
                  TimeStamp
```

If the semantics of an observation "Obsrv" in a given environment e results in environment e' and transforms a pre-state s into post-state s' then

- first the parameters are evaluated in a given environment e which results in an environment e_1 and sequence of values *pseq* transforming a given pre-state s into post-state s_1 and
- evaluation of the numeric value "Numeral" results in a value n in environment e_1 and state s_1 and
- also time stamp "TimeStamp" evaluates to a value t in environment e_1 and state s_1 and finally
  - if the observation is "entry" event the resulting environment e' is a result of a push operation on environment e_2 updating the value of the observation "EvntName" with the semantic value of the observation, i.e. of type "RTEvent" which is constructed with the help of computed input values *pseq*, process id n and time value t and

– if the observation is "exit" event the resulting environment e' is a result of a push operation on environment e_2 updating the value of the observation "EvntName" with the semantic value of the observation, i.e. of type "RTEvent" which is constructed with the help of computed output values *pseq* (sequence with a single value), process id n and time value t.

The semantics of the observation is formalized as follows:

```
⟦Obsrv⟧(e)(e', s, s') ⇔
 ∀ $e_1, e_2$ ∈ Environment, $s_1$ ∈ State, pseq ∈ Value*, n, t ∈ Value:
   ⟦EvntParamSeq⟧(e)($e_1$, s, $\text{inState}_\perp$(s'), pseq) ∧
   ⟦Numeral⟧($e_1$) (inState(s'), n) ∧ ⟦TimeStamp⟧($e_1$) (inState(s'), t) ∧
   LET rte ∈ RTEvent IN
        IF isEntry(Obsrv) THEN
            e' = push($e_2$, store(inState(s'))(⟦EvntName⟧($e_2$))
                          , rte(pseq, EMPTY, s, s', <"entry", t, n>))
        ELSE
            e' = push($e_2$, store(inState(s'))(⟦EvntName⟧($e_2$))
                          , rte(EMPTY, pseq, s, s', <"exit", t, n>))
        END
   END
```

4.1.1 Observations

The event respectively observation stream is a sequence of observations, which is modeled by corresponding syntactic domain ObsrvSeq as follows:

ObsrvSeq := EMPTY | (Obsrv) ObsrvSeq

The semantics of the observation sequence are similar to the other syntactic sequences discussed earlier in this document.

Case: EMPTY

⟦EMPTY⟧(e)(e', s, s') ⇔ e' = e ∧ s' = $\text{inState}_\perp$(s)

Case: (Obsrv) ObsrvSeq

⟦(Obsrv) ObsrvSeq⟧(e)(e', s, s') ⇔
 ∀ e" ∈ Environment, s" ∈ State:
   ⟦Obsrv⟧(e)(e", s, $\text{inState}_\perp$(s")) ∧ ⟦ObsrvSeq⟧(e")(e', s", s')

5 Semantics of the Execution Monitor

Though the technical details of the operation of the execution monitor are discussed in [5], in the following we give their informal semantics.

We presume that a reasonable fine grained level behavior of the target system is specified in the corresponding System Architectural Model. When the target system starts execution, an initial "startup" event is generated and dispatched to the top level component (module) of the system which transforms the execution state of the component into "running" mode. The component instantiates its subnetwork (of components, if there is one) and also propagates the data along its data links by enabling the corresponding control links (if involved). When the data arrives on the input port of the component, the execution monitor checks if it is complete; if so, the execution monitor checks the preconditions of the component for the data and if they succeed, it transform the state of the component into "ready" mode. In case, any of the preconditions fails, it enables diagnosis engine.

After the above startup of the target system, the execution monitor starts monitoring the arrival of every *observation* (runtime event) as follows:

1. If the event is a "method entry", then the execution monitor checks if this is one of the "entry events" of the corresponding component in the "ready" state; if so, then after receiving the data and the respective preconditions are checked; if they succeed, then the data is applied on the input port of the component and the mode of the execution state is changed to "running".

2. If the event is a "method exit", then the execution monitor checks if this one of the "exit events" of the component in the "running" state; if so, it changes its state into "completed" mode and collects the data from the output port of the component and checks for the corresponding postconditions. Should the checks fail, the execution monitor enables the diagnosis engine.

3. If the event is one of the "allowable events" of the component, it continues execution and finally

4. if the event is an unexpected event, i.e. it is neither an "entry event", nor an "exit event" and also not in "allowable events", then the execution monitor starts diagnosis.

Based on the above behavioral description of the execution monitor, we have formalized the corresponding semantics of the execution monitor by a relation *monitor* which has signatures as follows:

$$\begin{aligned} \mathit{monitor} \subset\ & \text{Target\_System} \times \text{System\_Architectural\_Model} \\ & \rightarrow \text{Environment} \rightarrow \text{State} \times \text{State}_{\perp} \end{aligned}$$

and which is defined as follows:

$\mathit{monitor}(\text{app}, \text{sam})(e)(s, s') \Leftrightarrow$

$$
\begin{array}{l}
\forall \text{ c} \in \text{Component, t, t'} \in \text{State}_s\text{, d, d'} \in \text{Environment}_s\text{, e'} \in \text{Environment, rte} \in \text{RTEvent:} \\
\quad [\![\text{sam}]\!](\text{d})(\text{d', t, t'}) \wedge [\![\text{app}]\!](\text{e})(\text{e', s, s'}) \wedge \text{startup(s, app)} \wedge \text{isTop(c, } [\![\text{app}]\!](\text{e})(\text{e', s, s'})) \wedge \\
\quad \text{setMode(s, ``running'')} \wedge \text{arrives(rte, s)} \wedge \text{equals(t, s)} \wedge \text{equals(d, e)} \\
\quad \Rightarrow \\
\qquad \forall \text{ p, p'} \in \text{Environment}^*\text{, m, n} \in \text{State}_\perp^*\text{:} \\
\qquad\quad \text{equals(m(0), s)} \wedge \text{equals(p(0), e)} \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists \text{ k} \in \mathbb{N}\text{:} \\
\qquad\qquad\quad \forall \text{ i} \in \mathbb{N}_k : \text{monitors(i, rte, c, p, p', m, n)} \wedge \\
\qquad\qquad\qquad \text{(eqMode(n(k), ``completed'')} \wedge \text{eqFlag(n(k), ``normal'')} \wedge \text{equals(s', n(k))} \\
\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad \text{eqFlag(n(k), ``compromised'')} \\
\qquad\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad\qquad \text{enableDiagnosis(p'(k))(n(k), inBool(}\texttt{true}\text{))} \wedge \neg \text{ equals(s', n(k)))}
\end{array}
$$

In detail, given a target system "app" and its specification"sam" and their semantices are defined such that their corresponding pre-states are equivalent. Furthermore, if the application starts "startup(...)", and an arbitrary *c* is a top-level component "isTop(...)", then the current state of the component is marked as "running" and when an observation "rte" arrives in this state, then the monitor starts monitoring the event stream/sequence and thus, here, we have formalized the corresponding semantics of the monitor by the two sequences of pre- and post-states [3] and their respective sequences of the pre- and post-environments. Both the former and later sequences are constructed from their corresponding pre- and post objects. The arrival and monitoring of the *ith* observation (event) transforms state $pre(i)$ into state $post(i + 1)$ from which the state $pre(i + 1)$ is constructed and the same repeats for the construction of the corresponding environments. No event can be accepted in an *Error* state and the corresponding monitoring terminates either when the application has terminated with "normal" mode or when there is some misbehavior is detected as indicated by the respective "compromised" state. This semantics is formalized with the help of predicate "monitor", for details please see Subsection 3.3.

Finally, when there are sequences of states and environments for which the predicate "monitor" holds, then either the given post-state s' is equal to the "monitor"ed post-state "n(k)" which is in "completed" mode and has a "normal" flag or post-state "n(k)" is "compromised" and in this case diagnosis is enabled which successfully transforms the compromised state into a normal state which results in the given post-state s'.

6 Conclusions and Future Work

In this report, we gave the formal definition of the syntax and semantices of the System Architectural Model and the Execution Monitor of AWDRAT.

These definitions help to understand internal behavior of the corresponding components on one hand, and also serves as a formal basis for ADWRAT to extend the current system on the other hand. Based on this formalism, we are currently working on the formal reliability (soundness) analysis of the Execution Monitor of AWDRAT.

In future, we plan to extend AWDRAT such that a target system behavior is specified using Abstract State Machine (ASM) [1] based formalism which then will automatically translate into a semantically equivalent System Architectural Model. This will allow to already check the inconsistencies in the system behavior with existing ASM supported tools [2].

Acknowledgment

The authors cordially thank Adam Chilpala for his valuable and constructive remarks and suggestions.

References

[1] E. Borger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[2] Jean-Baptiste Jeannin, Guido de Caso, Juan Chen, Yuri Gurevich, Prasad Naldurg, and Nikhil Swamy. Dkal*: Constructing executable specifications of authorization protocols. Technical Report MSR-TR-2013-19, March 2013.

[3] Muhammad Taimoor Khan. On the Formal Semantics of MiniMaple and Its Specification Language. In *Proceedings of the 2012 10th International Conference on Frontiers of Information Technology*, FIT '12, pages 169–174, Washington, DC, USA, 2012. IEEE Computer Society.

[4] Schmidt, David A. *Denotational Semantics: a methodology for language development.* William C. Brown Publishers, Dubuque, IA, USA, 1986.

[5] Howard Shrobe, Robert Laddaga, Bob Balzer, Neil Goldman, Dave Wile, Marcelo Tallis, Tim Hollebeek, and Alexander Egyed. AWDRAT: A Cognitive Middleware System for Information Survivability'. In *Proceedings of the 18th Conference on Innovative Applications of Artificial Intelligence - Volume 2*, IAAI'06, pages 1836–1843. AAAI Press, 2006.

[6] Shrobe, Howard E. Dependency Directed Reasoning for Complex Program Understanding. Technical report, 1979.

Appendices

Appendix A gives the formal abstract syntax (language grammar) for the specification language "system architectural model" of AWDRAT.

A Formal Syntax of System Architectural Model

A.1 Declaration of Syntactic Domains

/* top level syntactic domains */
SAM $\in$ System_Architectural_Model
RegModSeq $\in$ Register_Model_Sequence
StrModSeq $\in$ Structural_Model_Sequence
BehModSeq $\in$ Behavioral_Model_Sequence
SplModSeq $\in$ Split_Model_Sequence
AtkModSeq $\in$ Attack_Model_Sequence
AtkRuleSeq $\in$ Attack_Rule_Sequence

/* top level syntactic sub-domains */
RegMod $\in$ Register_Model
StrMod $\in$ Structural_Model
BehMod $\in$ Behavioral_Model
SplMod $\in$ Split_Model
AtkMod $\in$ Attack_Model
AtkRule $\in$ Attack_Rule

/* event related syntactic domains */
Evnt $\in$ Event
EvntSeq $\in$ Event_Sequence
EvntName $\in$ Event_Name
EvntNameSeq $\in$ Event_Name_Sequence
EvntParamSeq $\in$ Event_Parameter_Sequence

/* java related syntactic domains */
JavClaName $\in$ Java_Class_Name
JavMetName $\in$ Java_Method_Name
JavParam $\in$ Java_Parameter
JavParamSeq $\in$ Java_Parameter_Sequence
JavParamName $\in$ Java_Parameter_Name
JavParamType $\in$ Java_Parameter_Type

/* object related syntactic domains */
ObjName $\in$ Object_Name
ObjtNameStr $\in$ Object_Name_String
ObjNameSeq $\in$ Object_Name_Sequence

ObjType ∈ Object_Type
ObjComp ∈ Object_Component
ObjCompSeq ∈ Object_Component_Sequence

/* behavioral condition, parameter and situation related syntactic domains */
BehCond ∈ Behavioral_Condition
BehCondSeq ∈ Behavioral_Condition_Sequence
BehCondMode ∈ Behavioral_Condition_Mode
BehParam ∈ Behavioral_Parameter
BehParamSeq ∈ Behavioral_Parameter_Sequence
BehSit ∈ Behavioral_Situation

/* branch related syntactic domains */
BrnchName ∈ Branch_Name
BrnchNameSeq ∈ Branch_Name_Sequence
BrnchCond ∈ Branch_Condition

/* component related syntactic domains */
Comp ∈ Component
CompSeq ∈ Component_Sequence
CompName ∈ Component_Name
CompType ∈ Component_Type

/* control flow related syntactic domains */
CtrlFlow ∈ Control_Flow
CtrlFlowSeq ∈ Control_Flow_Sequence

/* function related syntactic domains */
FuncName ∈ Function_Name
FuncParam ∈ Function_Parameter
FuncParamSeq ∈ Function_Parameter_Sequence

/* split related syntactic domains */
SpltCF ∈ Split
SpltCFSeq ∈ Split_Sequence
SpltName ∈ Split_Name
SpltModName ∈ Split_Model_Name
SpltParamSeq ∈ Split_Parameter_Sequence
SpltCond ∈ Split_Condition
SpltCondSeq ∈ Split_Condition_Sequence

/* join related syntactic domains */
JoinCF ∈ Join
JoinCFSeq ∈ Join_Sequence
JoinName ∈ Join_Name

JoinParamSeq $\in$ Join_Parameter_Sequence

/* data flow related syntactic domains */
DataFlow $\in$ Data_Flow
DataFlowSeq $\in$ Data_Flow_Sequence

/* resource related syntactic domains */
Res $\in$ Resource
ResSeq $\in$ Resource_Sequence
ResName $\in$ Resource_Name
ResType $\in$ Resource_Type
ResMap $\in$ Resource_Mapping
ResMapSeq $\in$ Resource_Mapping_Sequence
ResModMap $\in$ Resource_Model_Mapping

/* model mapping syntactic domains */
ModMap $\in$ Model_Mapping
ModMapSeq $\in$ Model_Mapping_Sequence

/* vulnerability related syntactic domains */
Vulnrablty $\in$ Vulnerability
VulnrabltyName $\in$ Vulnerability_Name
VulnrabltySeq $\in$ Vulnerability_Sequence

/* attack related syntactic domains */
AtkType $\in$ Attack_Type
AtkTypeSeq $\in$ Attack_Type_Sequence
AtkModName $\in$ Attack_Model_Name
AtkCond $\in$ Attack_Condition
AtkCondSeq $\in$ Attack_Condition_Sequence
AtkCons $\in$ Attack_Consequence
AtkConsSeq $\in$ Attack_Consequence_Sequence
AtkTypeName $\in$ Attack_Type_Name
AtkRulName $\in$ Attack_Rule_Name
AtkVulnrabltyMap $\in$ Attack_Vulnerability_Mapping
AtkVulnrabltyMapSeq $\in$ Attack_Vulnerability_Mapping_Sequence

/* other syntactic domains */
MembName $\in$ Member_Name
ParamName $\in$ Parameter_Name
DSCond $\in$ Data_Structure_Condition
ISeq $\in$ Identifier_Sequence

A.2 Grammar

Based on the declarations of various syntactic domains, in this section we discuss the grammar rules for the domains.

```
/* top level syntactic domains */
SAM ::= RegModSeq StrModSeq BehModSeq SplModSeq
RegModSeq ::= EMPTY
            | (RegMod) RegModSeq
StrModSeq ::= EMPTY
            | (StrMod) StrModSeq
BehModSeq ::= EMPTY
            | (BehMod) BehModSeq
SplModSeq ::= EMPTY
            | (SplMod) SplModSeq
AtkModSeq ::= EMPTY
            | (AtkMod) AtkModSeq
AtkRuleSeq ::= EMPTY
            | (AtkRule) AtkRuleSeq

/* top level syntactic sub-domains */
RegMod ::= register-event 'EvntName JavClaName JavMetName '(JavParamSeq)
                          [ :static ObjName ]
                          [ :output-type JavParam ]
                          [ :bypass ObjNameStr ]
                          [ :EvntName ObjName]
StrMod ::= define-ensemble CompName
                           :entry-events :auto | (EvntSeq)
                           :exit-events (EvntSeq)
                           :allowable-events (EvntSeq)
                           :inputs (ObjNameSeq)
                           :outputs (ObjNameSeq)
                           :components (CompSeq)
                           :controlflows (CtrlFlowSeq)
                           :splits (SpltCFSeq)
                           :joins (JoinCFSeq)
                           :dataflows (DataFlowSeq)
                           :resources (ResSeq)
                           :resource-mapping (ResMapSeq)
                           :model-mappings (ModMapSeq)
                           :vulnerabilities (VulnrabltySeq)
BehMod ::= defbehavior-model (CompName normal | compromised)
                             :inputs (ObjNameSeq)
                             :outputs (ObjNameSeq)
                             :allowable-events (EvntSeq)
```

```
                                   :prerequisites (BehCondSeq)
                                   :postconditions (BehCondSeq)
SplMod ::= defsplit SpltModName? (SpltParamSeq) SpltCondSeq)
AtkMod ::= define-attack-model AtkModName
                                   :attack-types (AtkTypeSeq)
                                   :vulnerability-mapping (AtkVulnrabltyMapSeq)
AtkRule ::= defrule AtkRulName (:forward)
                    if AtkCondSeq
                    then AtkConsSeq

/* event related syntactic domains */
Evnt ::= EvntName | (EvntName [entry | exit] (EvntParamSeq))
EvntSeq ::= EMPTY
          | Evnt EvntSeq
EvntParam ::= I Iseq
            | nil I Iseq
            | I ISeq nil
EvntParamSeq ::= EMPTY
               | EvntParam EvntParamSeq

/* java related syntactic domains */
JavClaName ::= "I"
JavMetName ::= "ID" | "<I>"
JavParam ::= (JavParamType JavParamName)
JavParamSeq ::= EMPTY
              | JavParam JavaParamSeq
JavParamName ::= "I"
JavParamType ::= "ID" | "ID[]"

/* object related syntactic domains */
ObjComp ::= (ObjName CompName?)
ObjCompSeq ::= EMPTY
             | ObjComp ObjCompSeq
ObjNameStr ::= ObjName | "ObjName"

/* behavioral condition related syntactic domains */
BehCond ::= [ DSCond ObjName ObjType BehCondMode ]
          | [ and BehCond ]
          | [ or BehCond ]
          | [ not BehCond ]
          | [ SpecFuncName BehParamSeq BehSit ]
BehCondSeq ::= EMPTY
             | BehCond BehCondSeq
BehCondMode ::= EMPTY | good
BehParam ::= ?ObjName | (MembName ?ObjName)
```

BehParamSeq ::= EMPTY
                | BehParam BehParamSeq
BehSit ::= **?before-**CompName | **?after-**CompName

/* branch condition syntactic domain */
BrnchCond ::= FuncName FuncParamSeq

/* component related syntactic domains */
Comp ::= (CompName **:type** CompType **:models** (**normal** [**compromised**]))
CompSeq ::= EMPTY
            | Comp CompSeq

/* control flow related syntactic domains */
CtrlFlow ::= (**before** | **after** CompName[**?-**BrnchName])
CtrlFlowSeq ::= EMPTY
                | CtrlFlow CtrlFlowSeq

/* function related syntactic domains */
FuncParam ::= **?**ParamName | 'ParamName | ParamName**?** | **not** (ParamName)
FuncParamSeq ::= EMPTY
                 | FuncParam FuncParamSeq

/* split of control flow related syntactic domains */
SpltCF ::= (SpltName**?** SpltModName**?** [(SpltParamSeq)] (BrnchNameSeq))
SpltCFSeq ::= EMPTY
              | SpltCF SpltCFSeq
SpltCondSeq ::= EMPTY
                | SpltCond SplitCondSeq
SpltCond ::= (BrnchName (BrnchCond))

/* join of control flow related syntactic domains */
JoinCF ::= (JoinName**?** [(JoinParamSeq)] (BrnchNameSeq))
JoinCFSeq ::= EMPTY
              | JoinCF JoinCFSeq

/* data flow related syntactic domains */
DataFlow ::= (ObjCompSeq)
DataFlowSeq ::= EMPTY
                | DataFlow DataFlowSeq

/* resource related syntactic domains */
Res ::= (ResName ResType [(**normal** | **hacked** FVal)]+)
ResSeq ::= EMPTY
           | Res ResSeq
ResType ::= File | Port | Mem

ResMap ::= (CompName ResName)
ResMapSeq ::= EMPTY
                  | ResMap ResMapSeq
ResModMap ::= ResName **normal** | **hacked** FVal
                  | ((ResName **normal** | **hacked**)) FVal

/* model mapping related syntactic domains */
ModMap ::= (CompName **noromal** | **compromised** ResModMap)
ModMapSeq ::= EMPTY
                  | ModMap ModMapSeq

/* vulnerability related syntactic domains */
Vulnrablty ::= (ResName VulnrabltyName)
VulnrabltySeq ::= EMPTY
                  | Vulnrablty VulnrabltySeq

/* attack related syntactic domains */
AtkType ::= (AtkTypeName FVal)
AtkTypeSeq ::= EMPTY
                  | AtkType AtkTypeSeq
AtkCond ::= [**resource** **?**ensemble **?**ResName **?**Res]
              [**resource-type-of** **?**Res ResType]
              [**resource-might-have-been-attacked** **?**Res AtkTypeName]
              | [**and** AtkCond]
              | [**or** AtkCond]
              | [**not** AtkCond]
AtkCondSeq ::= EMPTY
                  | AtkCond AtkCondSeq
AtkCons ::= [**attack-implies-compromised-mode** AtkTypeName **?**Res
                                        normal | **compromised** FVal]
              | [**and** AtkCons]
              | [**or** AtkCons]
              | [**not** AtkCons]
AtkConsSeq ::= EMPTY
                  | AtkCons AtkConsSeq
AtkVulnrabltyMap ::= (VulnrabltyName AtkTypeName)
AtkVulnrabltyMapSeq ::= EMPTY
                              | AtkVulnrabltyMap AtkVulnrabltyMapSeq

/* syntactic domains of various names and types */
CompName, CompType, FuncName, ObjName, ObjType,
              EvntName, SpltName, SpltModName ::= I
JoinName, ResName, BrnchName, VulnrabltyName, SpecFuncName
              , ParamName, MembName ::= I
AtkModName, AtkTypeName, AtkRulName ::= I

/* syntactic domains of various sequences */
ObjNameSeq, SpltParamSeq, JoinParamSeq, BrnchNameSeq ::= ISeq

/* other syntactic domains */
DSCond ::= EMPTY | **dscs**
ISeq :: = EMPTY
        | I ISeq
I ::= any valid LISP system name
FVal ::= a sequence of decimal digits prefixed by a period (valid float value)

B An Example of a System Architectural Model

In this section, we give the syntax of an example System Architectural Model of MAF editor system which is discussed in detail in [5]. In the following, we give a brief detail on how to read the example, i.e. `maf-editor` is the top level component of the application whose structural behavior is specified at first. Every sentence of the specification is self-explanatory. In principle, the behavior of every component in the subnetwork of a parent component has to be specified separately with two corresponding parts, e.g. a component `maf-startup` (which is in the subnetwork of top-level component as mentioned in `:components` clause) has

1. structural behavior as specified by clause

   ```
   define-ensemble maf-startup
   ```

2. normal behavior as specified by the clause

   ```
   defbehavior-model (maf-startup norml)
   ```

3. and a corresponding compromised behavior is specified by the clause

   ```
   defbehavior-model (maf-startup compromised)
   ```

The former part corresponds to the *control* level of the specification while the latter two corresponds to the *behavioral* level of the specification of the component.

Furthermore, as explained in Section 2, the split behavior of the component `maf-create-events` is further specified with the corresponding clauses, e.g.

```
defsplit maf-more-events?
```

Also, any pre/postcondition that is followed by the `dscs` specifies the data structure consistency property.

B.1 MAF Editor Model

```
(define-ensemble maf-editor
    :entry-events :auto
    :inputs ()
    :outputs (the-model)
    :components ((startup :type maf-startup :models (normal compromised))
     (create-model :type maf-create-model :models (normal compromised))
     (create-events :type maf-create-events :models (normal compromised))
     (save :type maf-save :models (normal compromised)))

    :controlflows ((before maf-editor before startup)
     (after startup before create-model))

    :dataflows ((the-model create-model the-model create-events)
     (the-model create-events the-model save)
     (the-model save the-model maf-save-model))

    :resources ((imagery image-file (normal .7) (hacked .3))
     (code-files loadable-files (normal .8) (hacked .2)))

    :resource-mappings ((startup imagery)
     (create-model code-files)
     (create-events code-files)
     (save-model code-files))


    :model-mappings ((startup normal ((imagery normal)) .99)
     (startup compromised ((imagery normal)) .01)
     (startup normal ((imagery hacked)) .9)
     (startup compromised ((imagery hacked)) .1)

     (create-model normal ((code-files normal)) .99)
     (create-model compromised ((code-files normal)) .01)
     (create-model normal ((code-files hacked)) .9)
     (create-model compromised ((code-files hacked)) .1)

     (create-events normal ((code-files normal)) .99)
     (create-events compromised ((code-files normal)) .01)
     (create-events  normal ((code-files hacked)) .9)
     (create-events compromised ((code-files hacked)) .1)

     (save normal ((code-files normal)) .99)
     (save compromised ((code-files normal)) .001)
     (save normal ((code-files hacked)) .01)
     (save compromised ((code-files hacked)) .999))

    :vulnerabilities ((imagery reads-complex-imagery)
      (code-files loads-code)
      ))

(define-ensemble maf-startup
    :entry-events (startup)
    :exit-events (startup)
    :allowable-events (post-validate create-client-frame
     center-action load-image)
    :inputs ()
    :outputs ())

(defbehavior-model (maf-startup normal)
    :inputs ()
    :outputs ()
    :prerequisites ()
    :post-conditions ())

(defbehavior-model (maf-startup compromised)
    :inputs ()
    :outputs ()
    :prerequisites ()
    :post-conditions ())

;;; Need defbehaviors for each of these even if its empty

(define-ensemble maf-create-model
    :entry-events (create-mission-action-action-performed)
    :exit-events (mission-builder-submit)
    :allowable-events (create-mission-builder-with-client-panel
       create-mission-builder
       create-mission-builder-with-hash-table
       mission-builder-submit
       (set-initial-info exit (the-model nil))
       create-mission-action-action-performed
       retrieve-info
       create-mission-action-action-performed
       (set-initial-info entry)
       )
    :inputs ()
    :outputs (the-model))
```

```
(defbehavior-model (maf-create-model normal)
    :inputs ()
    :outputs (the-model)
    :prerequisites ()
    :post-conditions ([dscs ?the-model mission-builder good])
    )

(defbehavior-model (maf-create-model compromised)
    :inputs ()
    :outputs (the-model)
    :prerequisites ()
    :post-conditions ([not [dscs ?the-model mission-builder good]])
    )

(define-ensemble maf-create-events
    :entry-events :auto
    :exit-events ()
    :allowable-events ()
    :inputs (the-model)
    :outputs (the-model)
    :components ((get-next-cmd :type maf-get-next-cmd :models (normal))
 (get-event-info :type maf-get-event-info :models (normal compromised))
 (add-event-to-model :type maf-add-event-to-model :models
  (normal compromised))
 (get-leg :type maf-get-leg :models (normal compromised))
 (get-movement :type maf-get-movement :models (normal compromised))
 (get-sortie :type maf-get-sortie :models (normal compromised))
 (add-additional-info-to-model :type maf-add-additional-info :models
  (normal compromised))
 (continue :type maf-create-events :models (normal compromised)))

    :dataflows ((the-model maf-create-events the-model join-exit-exit)
(the-model maf-create-events the-model add-event-to-model)
(the-cmd get-next-cmd cmd more-events?)
(the-event get-event-info the-event add-event-to-model)
(the-model add-event-to-model the-model join-events-non-take-off)
(the-event get-event-info event takeoff?)
(the-leg get-leg the-leg add-additional-info-to-model)
(lms-event-counter get-leg event-number add-additional-info-to-model)
(the-movement get-movement the-movement add-additional-info-to-model)
(the-sortie get-sortie the-sortie add-additional-info-to-model)
(the-model add-event-to-model the-model add-additional-info-to-model)
(the-model add-additional-info-to-model the-model join-events-take-off)
```

```
(the-model join-events the-model continue)
(the-model continue the-model join-exit-recur)
(the-model join-exit the-model maf-create-events)
)

   :controlflows ((after more-events?-build-event before add-event-to-model)
    (after more-events?-exit before join-exit-exit)
    (after takeoff?-get-additional-info before get-leg)
    (after takeoff?-get-additional-info before get-movement)
    (after takeoff?-get-additional-info before get-sortie)
    (after takeoff?-exit before join-events-non-take-off))

    :splits ((more-events? maf-more-events? (cmd) (build-event exit))
     (takeoff? maf-takeoff? (event) (get-additional-info exit)))

    :joins ((join-events (the-model) (take-off non-take-off))
    (join-exit (the-model) (recur exit)))

    :resources ((code-files loadable-files (normal .8) (hacked .2)))

    :resource-mappings ((get-event-info  code-files)
     (add-event-to-model code-files)
     (get-leg code-files)
     (get-movement code-files)
     (get-sortie code-files)
     (add-additional-info-to-model code-files)
     (continue code-files))

    :model-mappings ((get-event-info normal code-files normal .99)
     (get-event-info compromised code-files normal .01)
     (get-event-info normal code-files hacked .9)
     (get-event-info compromised code-files hacked .1)

     (add-event-to-model normal code-files normal .99)
     (add-event-to-model compromised code-files normal .01)
     (add-event-to-model  normal code-files hacked .9)
     (add-event-to-model compromised code-files hacked .1)

     (get-leg normal code-files normal .99)
     (get-leg compromised code-files normal .001)
     (get-leg normal code-files hacked .01)
     (get-leg compromised code-files hacked .999)

     (get-movement normal code-files normal .99)
     (get-movement compromised code-files normal .001)
     (get-movement normal code-files hacked .01)
     (get-movement compromised code-files hacked .999)

     (get-sortie normal code-files normal .99)
     (get-sortie compromised code-files normal .001)
     (get-sortie normal code-files hacked .01)
     (get-sortie compromised code-files hacked .999)

     (add-additional-info-to-model normal code-files normal .99)
     (add-additional-info-to-model compromised code-files normal .001)
     (add-additional-info-to-model normal code-files hacked .01)
     (add-additional-info-to-model compromised code-files hacked .999)

     (continue normal code-files normal .99)
     (continue compromised code-files normal .001)
     (continue normal code-files hacked .01)
     (continue compromised code-files hacked .999))

    :vulnerabilities ((code-files loads-code))
    )

(defbehavior-model (maf-create-events normal)
    :inputs (the-model)
    :outputs (the-model)
    :prerequisites ([dscs ?the-model mission-builder good])
    :post-conditions ([dscs ?the-model mission-builder good])
    )

(defbehavior-model (maf-create-events compromised)
    :inputs (the-model)
    :outputs (the-model)
    :prerequisites ([dscs ?the-model mission-builder good])
    :post-conditions ([not [dscs ?the-model mission-builder good]])
    )

(define-ensemble maf-get-next-cmd
    :entry-events (next-cmd)
    :exit-events ((next-cmd exit (the-cmd)))
    :inputs ()
    :outputs (the-cmd))

(defbehavior-model (maf-get-next-cmd normal)
    :inputs ()
    :outputs (the-cmd)
    :prerequisites ()
    :post-conditions ())

(define-ensemble maf-get-event-info
    :entry-events (create-mission-event-point)
    :allowable-events (set-current-point
       (create-mission-event-point exit)
       create-mission-event-object
       meo-set-information
       mpl-action-performed
       close-form
       add-new-event-internal)
    :exit-events ((got-event-info exit (the-event)))
    :inputs ()
    :outputs (the-event))

(defbehavior-model (maf-get-event-info normal)
    :inputs ()
    :outputs (the-event)
    :prerequisites ()
    :post-conditions ([dscs ?the-event event good]))

(defbehavior-model (maf-get-event-info compromised)
    :inputs ()
    :outputs (the-event)
    :prerequisites ()
    :post-conditions ([not [dscs ?the-event event good]]))

(define-ensemble maf-add-event-to-model
    :entry-events (update-msn-evt)
    :allowable-events
    ((update-msn-evt exit (mb event-number event))
     add-new-event-internal
     create-new-additional-mission-info-panel
     )
    :exit-events (mpl-action-performed)
    :inputs (the-event the-model)
    :outputs (the-model event-number))

(defbehavior-model (maf-add-event-to-model normal)
    :inputs (the-event the-model)
    :outputs (the-model event-number)
    :prerequisites ([dscs ?the-event event good]
    [dscs ?the-model mission-builder good])
    :post-conditions
    ([add-to-map (events ?the-model)?event-number ?the-event
 ?before-maf-add-event-to-model]
     [dscs ?the-model mission-builder good]))

(defbehavior-model (maf-add-event-to-model compromised)
    :inputs (the-event the-model)
    :outputs (the-model event-number)
    :prerequisites ([not [dscs ?the-event event good]]
    [not [dscs ?the-model mission-builder good]])
    :post-conditions
    ([dscs ?the-model mission-builder good]))

(define-ensemble maf-get-leg
    :entry-events (retrieve-leg)
    :exit-events ((retrieve-leg exit (nil the-leg lms-event-counter)))
    :allowable-events (create-mission-leg-object mlo-set-information)
    :inputs ()
    :outputs (the-leg lms-event-counter))

(defbehavior-model (maf-get-leg normal)
    :inputs ()
    :outputs (the-leg lms-event-counter)
    :prerequisites ()
    :post-conditions ([dscs ?the-leg leg good]))

(defbehavior-model (maf-get-leg compromised)
    :inputs ()
    :outputs (the-leg lms-event-counter)
    :prerequisites ()
    :post-conditions ([not [dscs ?the-leg leg good]]))

(define-ensemble maf-get-movement
    :entry-events (retrieve-movement)
    :exit-events ((retrieve-movement exit (nil the-movement)))
    :allowable-events
    (create-mission-movement-object mmo-set-information)
    :inputs ()
    :outputs (the-movement))

(defbehavior-model (maf-get-movement normal)
    :inputs ()
    :outputs (the-movement)
    :prerequisites ()
    :post-conditions ([dscs ?the-movement movement good]))

(defbehavior-model (maf-get-movement compromised)
    :inputs ()
    :outputs (the-movement)
    :prerequisites ()
    :post-conditions ([not [dscs ?the-movement movement good]]))

(define-ensemble maf-get-sortie
    :entry-events (retrieve-sortie)
    :exit-events ((retrieve-sortie exit (nil the-sortie)))
    :allowable-events
    (create-mission-sortie-object mso-set-information)
    :inputs ()
    :outputs (the-sortie))

(defbehavior-model (maf-get-sortie normal)
    :inputs ()
    :outputs (the-sortie)
    :prerequisites ()
    :post-conditions ([dscs ?the-sortie sortie good]))

(defbehavior-model (maf-get-sortie compromised)
    :inputs ()
    :outputs (the-sortie)
    :prerequisites ()
    :post-conditions ([not [dscs ?the-sortie sortie good]]))

(define-ensemble maf-add-additional-info
    :entry-events ((retrieve-sortie exit))
    :exit-events (Mission-builder-add-info)
    :inputs (the-model the-leg the-movement the-sortie event-number)
    :outputs (the-model))

(defbehavior-model (maf-add-additional-info normal)
    :inputs (the-model the-leg the-movement the-sortie event-number)
    :outputs (the-model)
    :prerequisites  ([dscs ?the-leg leg good]
     [dscs ?the-movement movement good]
     [dscs ?the-sortie sortie good]
     [dscs ?the-model mission-builder good])
    :post-conditions ([add-to-map (legs ?the-model) ?event-number ?the-leg
  ?before-maf-add-additional-info]
```

```
        [add-to-map (sorties ?the-model) ?event-number ?the-sortie
    ?before-maf-add-additional-info]
        [add-to-map (movements ?the-model) ?event-number ?the-movement
    ?before-maf-add-additional-info]
        [dscs ?the-model mission-builder good]))

(defbehavior-model (maf-add-additional-info compromised)
    :inputs (the-model the-leg the-movement the-sortie event-number)
    :outputs (the-model)
    :prerequisites  ([dscs ?the-leg leg good]
     [dscs ?the-movement movement good]
     [dscs ?the-sortie sortie good]
     [dscs ?the-model mission-builder good])
    :post-conditions ([not [dscs ?the-model mission-builder good]]))

(defsplit maf-more-events? (cmd)
  (build-event (equal ?cmd 'new-event))
  (exit (equal ?cmd 'save-mission)))

(defsplit maf-takeoff? (event)
  (get-additional-info (take-off-event? ?event))
  (exit (not (take-off-event? ?event))))

(define-ensemble maf-save
    :inputs (the-model)
    :outputs ())

(defbehavior-model (maf-save normal)
    :inputs (the-model)
    :outputs ()
    :prerequisites ([dscs ?the-model mission-builder good])
    :post-conditions ([dscs ?the-model mission-builder good]))

(defbehavior-model (maf-save compromised)
    :inputs (the-model)
    :outputs ()
    :prerequisites ([dscs ?the-model mission-builder good])
    :post-conditions ([not [dscs ?the-model mission-builder good]]))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; attack models
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define-attack-model maf-attacks
    :attack-types ((hacked-image-file-attack .3) (hacked-code-file-attack .5))
    :vulnerability-mapping ((reads-complex-imagery hacked-image-file-attack)
    (loads-code hacked-code-file-attack)))

;;; rules mapping conditional probabilities of vulnerability and attacks

(defrule bad-image-file-takeover (:forward)
  if [and [resource ?ensemble ?resource-name ?resource]
  [resource-type-of ?resource image-file]
  [resource-might-have-been-attacked ?resource hacked-image-file-attack]]
  then [and [attack-implies-compromised-mode hacked-image-file-attack
   ?resource hacked .9 ]
    [attack-implies-compromised-mode hacked-image-file-attack
     ?resource normal .1 ]])

(defrule bad-image-file-takeover-2 (:forward)
  if [and [resource ?ensemble ?resource-name ?resource]
  [resource-type-of ?resource code-memory-image]
  [resource-might-have-been-attacked ?resource hacked-image-file-attack]]
  then [and [attack-implies-compromised-mode hacked-image-file-attack
   ?resource hacked .9 ]
    [attack-implies-compromised-mode hacked-image-file-attack
     ?resource normal .1 ]])

(defrule hacked-code-file-takeover (:forward)
  if [and [resource ?ensemble ?resource-name ?resource]
  [resource-type-of ?resource loadable-files]
  [resource-might-have-been-attacked ?resource hacked-code-file-attack]]
  then [and [attack-implies-compromised-mode hacked-code-file-attack
   ?resource hacked .9 ]
    [attack-implies-compromised-mode hacked-code-file-attack
     ?resource normal .1 ]])

(defrule hacked-code-file-takeover-2 (:forward)
  if [and [resource ?ensemble ?resource-name ?resource]
  [resource-type-of ?resource loadable-files]
  [resource-might-have-been-attacked ?resource hacked-code-file-attack]]
  then [and [attack-implies-compromised-mode hacked-code-file-attack
   ?resource hacked .9 ]
    [attack-implies-compromised-mode hacked-code-file-attack
     ?resource normal .1 ]])
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;;;;;      Hacked Code file attacks
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defrule bad-code-file-takeover (:forward)
  if [and [resource ?ensemble ?resource-name ?resource]
  [resource-type-of ?resource code-file]
  [resource-might-have-been-attacked ?resource hacked-code-file-attack]]
  then [and [attack-implies-compromised-mode hacked-code-file-attack
   ?resource hacked .9 ]
    [attack-implies-compromised-mode hacked-code-file-attack
     ?resource normal .1 ]])

(defrule bad-code-file-takeover-2 (:forward)
  if [and [resource ?ensemble ?resource-name ?resource]
  [resource-type-of ?resource code-memory-image]
  [resource-might-have-been-attacked ?resource hacked-code-file-attack]]
  then [and [attack-implies-compromised-mode hacked-code-file-attack
   ?resource hacked .9 ]
    [attack-implies-compromised-mode hacked-code-file-attack
     ?resource normal .1 ]])
```