A formal semantics for RTEdge™

Technical report

Ernesto Posse

School of Computing
Queen's University
Kingston, Ontario, Canada

August 2, 2013

Abstract

This report proposes a formal semantics for RTEdge™ [1], a language for modelling real-time embedded systems based on AADL [3] and developed by Edgewater Computer Systems Inc. We define a formal abstract syntax of RTEdge™and define the formal operational semantics of a model as a Labelled Transition System (LTS) as is customary for many reactive, concurrent languages. The LTS of a model is defined inductively as in Plotkin-style Structural Operational Semantics (SOS) [2]. This enables the use of well-known proof techniques to establish properties of models.

It should be noted that the semantics proposed here does not cover some RTEdge™ concepts such as "flows" and "transactions". This is because these concepts play a role in causality (flow) analysis and schedulability analysis but correspond to a more concrete level of execution, while the semantics presented here is abstract in that it captures the possible executions of a model without committing to particular scheduling policies.

This semantics doesn't define some language features such as Data Access Points (DAPs: scoped shared variables), services (one-to-many connectors), or timers, which rely on a services.

This report describes the semantics of RTEdge™ as implemented in the RTEdge™ platform release 1.3. This document is intended as a reference and thus, it does not provide examples or detailed explanations, and it does not develop a meta-theory for the language.

Contents

List of Tables **iii**

1 The RTEdge™ language, an informal description **1**

2 Syntax of RTEdge™ models **3**
2.1 Types . . . 3
2.2 Values . . . 4
2.3 Protocols . . . 5
2.4 Interfaces . . . 5
2.5 State machines . . . 6
2.6 Atomic capsules . . . 7
2.7 Composite capsules . . . 8
2.8 Proxy capsules . . . 9
2.9 External task capsules . . . 9
2.10 Applications . . . 10
2.11 Extended interfaces . . . 10
2.12 Extended composite capsules . . . 10

3 Semantics of RTEdge™ models **11**
3.1 Labelled Transition Systems . . . 11
3.2 Semantics of atomic capsules . . . 12
3.3 Semantics of composite capsules . . . 14
3.4 Semantics of extended, proxy and external task capsules and applications . . . 16

References **16**

Index **17**

A A textual syntax for RTEdge™ **18**

List of Tables

1 Correspondence between mathematical notation and ASCII notation for RTEdge^{TM}. 19
2 Correspondence between mathematical notation and ASCII notation for RTEdge^{TM}. 20

List of Symbols

General

Names	The set of all possible names	3

Types

$\mathbb{T}_B$	The set of basic types	3
$\mathbb{T}_S$	The set of struct types	3
$\mathbb{T}_A$	The set of array types	3
$\mathbb{T}_E$	The set of enum types	3
Types	The set of all types	3
memtypes	Direct member types of a (structured) type	3
memtypes*	Direct or indirect member types of a (structured) type	3
Σ	Variable signature	4
$type(x)$	Type of a variable x in a signature	4

Values

Values	The set of all possible values	4
$\perp$	The null value	4
$\mathbb{B}$	The set of boolean values {`true`, `false`}	4
$\mathbb{C}$	The set of 8-bit characters	4
$\mathbb{Z}^{(n)}$	The set of n-bit integers	4
$\mathbb{N}^{(n)}$	The set of n-bit unsigned integers	4
$\mathbb{D}^{(n)}$	The set of n-bit floating point numbers	4
$\mathbb{V}_S$	The set of struct values	4
$\mathbb{V}_A$	The set of array values	4
$\mathbb{V}_E$	The set of enum values	4
tdom(t)	The type domain (set of values) of type t	4

Protocols (syntax)

$type(e)$	The type of signal e	5
Protocols	The set of all possible protocols	5
isignals(R)	The set of input signals of protocol RP	5
osignals(R)	The set of output signals of protocol R	5
typmap(P)	The typing function of protocol R	5

Interfaces (syntax)

$prot(p)$	The protocol of port p	5

$kind(p)$	The kind (base or conjugate) of port p	5
base	The base kind of a port	5
conj	The conjugate kind of a port	5
Interfaces	The set of all possible interfaces	5
ports(F)	The set of ports of interface F	5
protocols(F)	The set of protocols of interface F	5
protmap(F)	The map from ports to protocols of interface F	5
kindmap(F)	The map from ports to port kinds of interface F	5
itrigs(F)	The input trigger alphabet of interface F	5
ostmts(F, Σ)	The output statement alphabet of interface F	6
$p.e$	Input (external event) trigger: signal e on port p	5
$p.e(x)$	Output statement: signal e with data from variable x on port p	6

Data stores, activities, triggers and output statements

σ	Data store	6
$\sigma(x)$	Value of variable x in data store σ	6
Stores$_\Sigma$	Data stores over a variable signature Σ	6
$\emptyset$	The empty store	6
initstore(Σ)	The initial store (all variables assigned the null value)	6
$\sigma[x := v]$	Store update assigning value v to variable x in store σ	6
Activities$_\Sigma$	The set of all possible activities over variables in signature Σ	6
ext(η)	External event trigger, where η is a trigger $p.e$ with port p and signal e	6
int(v)	Internal event trigger, or activity completion, where $v \in$ **Values**	6
Triggers$_F$	The set of possible triggers of an interface F	6
out(η)	Output statement where η is of the form $p.e(x)$ where p is a port, e is a signal and x is a variable	6
$\perp$	Null output statement (no output)	6
OutStmts$_F$	The set of all possible output statements for an interface F	6

State machines (syntax)

$def(s)$	The set of deferred ports in state s	6
$act(s)$	The activity associated to transient state s	7
$s \xrightarrow[\gamma]{\beta} s'$ or $s \xrightarrow{\beta/\gamma} s'$	Transition from state s to state s' with trigger β and output statement γ	7
StateMachines	The set of all possible state machines	7
locations(M)	The set of locations (states) of state machine M	7

initial(M) The initial location (state) of state machine M 7
stables(M) The set of stable locations (states) of state machine M 7
transients(M) The set of transient locations (states) of state machine M 7
interface(M) The interface of state machine M 7
vars(M) The variable signature of state machine M 7
activities(M) The set of activities of state machine M 7
activitiesmap(M) The map of transient states to activities of state machine M 7
transitions(M) The set of transitions of state machine M 7

Atomic capsules (syntax)

Atomic The set of all atomic capsules 7
ports(K) The set of ports of atomic capsule K 7
interface(K) The interface of atomic capsule K 7
attributes(K) The set of attributes of atomic capsule K 8
activities(K) The set of activities of atomic capsule K 8
statemachine(K) The statemachine of atomic capsule K 8

Composite capsules (syntax)

self Reserved role name of a capsule used when refered inside itself 8
role(r) The part or capsule of role r 8
connpts(K) The set of connection points of capsule K 8
$r.p$ Connection point: port p of role (part) r 8
Composite The set of all composite capsules 8
Capsules The set of all capsules 8
ports(K) The set of ports of capsule K 9
interface(K) The interface of capsule K 9
parts(K) The set of parts (subcapsules) of capsule K 9
roles(K) The set of roles of capsule K 9
connectors(K) The set of connectors of capsule K 9
links(K) The set of links of capsule K 9

Proxy capsules (syntax)

Proxies The set of all proxies 9
ports(K) The set of ports of proxy K 9
rtports(K) The set of (normal) RT ports of proxy K 9
osports(K) The set of OS ports of proxy K 9

interface(K)	The interface of proxy K	9
attributes(K)	The set of attributes of proxy K	9
activities(K)	The set of activities of proxy K	9
statemachine(K)	The statemachine of proxy K	9
Labelled transition systems		
$s \xrightarrow{\alpha} s'$	Transition from state s to state s' with label α (shorthand for $(s, \alpha, s') \in \rightarrow$)	11
$s \xrightarrow{\alpha_1\alpha_2\cdots\alpha_{n-1}} s_n$	Execution fragment s to state s_n with event trace $\alpha_1\alpha_2\cdots\alpha_{n-1}$	11
$\text{beh}_T(s)$	The set of all execution fragments (transition sequences) starting in s	11
$\text{evtrace}_T(\rho)$	Event (or action) trace of execution fragment ρ	11
$\text{sttrace}_T(\rho)$	State trace of execution fragment ρ	11
$\text{evtraces}_T(s)$	The set of all event traces starting from state s	11
$\text{sttraces}_T(s)$	The set of all state traces starting from state s	11
Atomic and composite capsules (semantics)		
$\mathbf{Queues}_X$	The set of all queues over elements in X	12
front(q)	The front element of queue q	12
enqueue(q, x)	The queue resulting from appending x to queue q	12
dequeue(q)	The pair (x, q') resulting from removing x from queue q	12
emptyqueue	The empty queue	12
inpalpha(F)	The input alphabet of interface F	12
outalpha(F)	Output alphabet of interface F	12
$p.e(v)$	Input message on port p with signal e carrying value v	12
$p.e(v)$	Output message on port p with signal e carrying value v	12
$\mathbf{AtomCapAct}_K$	The set of all possible actions of atomic capsule K	12
$\mathbf{CompCapAct}_K$	The set of all possible actions of composite capsule K	14
$\mathsf{recv}(\eta)$	Message reception action where $\eta = p.e(v)$ is an input message.	12
$\mathsf{cons}(\eta/\theta)$	Message consumption action where $\eta = p.e(v)$ is an input message and θ is either an output message $p'.e'(v')$ or $\perp$	12
$\mathsf{exec}(\eta/\theta)$	Activity execution action where $\eta = p.e(v)$ is an input message and θ is either an output message $p'.e'(v')$ or $\perp$	12
$\mathsf{msg}(u_1 \rightarrow u_2 \vert e(v))$	Message transmission action from connection point u_1 to connection point u_2 with signal e carrying value v	14
$\mathbf{AtomCapConfigs}_K$	The set of all configurations of atomic capsule K	13
AtomCapConfigs	The set of all possible atomic capsule configurations	13
$\mathbf{CompCapConfigs}_K$	The set of all configurations of composite capsule K	15

CompCapConfigs The set of all possible composite capsule configurations 15

Configs$_K$ The set of all possible capsule configurations of capsule K 15

Configs The set of all possible capsule configurations 15

(l, σ, q, x) Atomic capsule configuration with location l, data store σ, port queues q and last input x 12

$\{r_1 \mapsto s_1, ..., r_n \mapsto s_n\}$ Composite capsule configuration 15

1 The RTEdge^{TM} language, an informal description

RTEdge^{TM} is a language that can be used to describe concurrent, reactive, real-time systems. In RTEdge^{TM}, a system is a collection of interconnected *components* or *processes* called *capsules*. Each capsule is an *active object* with attributes and reactive behaviour. A capsule executes concurrently with the other components in the system. Capsules interact with other components by sending messages or *signals* over connections (also called *connectors*). Each capsule has a well-defined *interface* which consists of a set of ports through which signals are sent and received. Connectors link ports between different capsules. The reactive behaviour of capsules is defined by a certain kind of state machines. Communication is *asynchronous*: the sending of a message is non-blocking, so the sender doesn't wait for the message to be delivered. Capsules can be composed and grouped together to define a hierarchical structure.

The core elements of the RTEdge^{TM} language are:

- Protocols
- Interfaces
- Atomic Capsules with State Machines
- Composite Capsules
- Proxy Capsules
- External Task Capsules
- Timers
- Applications

Informally a *protocol* defines a set of input and output signals which may be transmitted between capsules.

An *interface* defines a collection of *named ports*, each of which has a protocol and can be either a *base port* or a *conjugate port*. In a base port, input and output signals of the corresponding protocol, are treated by the owining capsule as inputs and outputs respectively, whereas in a conjugate port the roles are flipped: input signals of the protocol are treated by the capsule as outputs and output signals in the protocol are treated as inputs.

An *atomic capsule* defines a process or active object with data attributes and a behaviour and has a specific interface. The behaviour is defined by a *state machine*. RTEdge^{TM} state machines are flat (no hierarchical states) and divide the states into two groups: stable states and transient states. Stable states are states where the capsule is at rest waiting for external input signals on its ports. Hence transitions emmanating from stable states are annotated with *input or external event triggers*. Transient states are intermediate states which may have actions or activities associated to them. These activities are written in an underlying *action language*, which in the case of the RTEdge^{TM} platform is C++. Actions are parametrized with the data attributes of the capsule object and with the last message received. Transitions emmanating from a transient state can be labelled with a value or *action completion code*, which is used as an *activity completion trigger*, allowing internal choice. Transitions can be annotated with *output statements*, which send output signals through the capsule ports to other capsules. These state machines have a *run-to-completion semantics*: when the capsule is on a stable state, the arrival of an input on a port results in a chain of transitions being followed according to the transition triggers, possibly going through transient states and ending in a stable state. If an input signal arrives and the capsule is not on a stable state, the signal will be queued in its port until the capsule can handle it. Thus, each port has its own FIFO queue. A port might be marked as *deferred* in a stable state. If an input arrives on that port when the system is in the stable state that defers it, the message will not be processed, and it will simply remain in its queue until it can be consumed in another stable state. If input arrives at a port which is not deferred and the current stable state doesn't have a transition with a trigger for that port and signal, there are two possible behaviours: the signal can be ignored, or an error can be issued. If more than one transition is enabled in a stable state, the tie is broken by the relative priorities of the signals. These priorities are assigned statically by RTEdge^{TM} during schedulability analysis, to ensure that required deadlines are met.

A *composite capsule* defines a group of interconnected capsules (atomic or composite) and has an interface. It serves a the basic structuring construct in the language providing a hiding and encapsulation construct so that the only way to access the composite capsule's sub-components is through its interface. Informally, the capsules within a composite capsule execute concurrently, although the platform implementation may schedule the transitions within the same thread. Since ports are queues, the basic communication mechanism is asynchronous message passing.

A *proxy capsule* is a special kind of atomic capsule which has "OS ports", this is, ports that allow the capsule to interact with software outside of the application.

An *external task capsule* is also a special kind of capsule and it doesn't represent a component within the application, but rather it is used to represent external components with which the application may interact. External capsules can only be connected to proxy capsules.

A *timer* is a special kind of primitive component that issues a given signal periodically. This signal is received by every capsule which has subscribed to the timer's *service*.

An *application* is the top level component of the language. It can be seen as a special composite capsule which groups together all components and elements.

2 Syntax of RTEdge$^{\mathbf{TM}}$ models

Notation 1. Let **Names** denote the set of all possible *names*. Let $\mathbb{N}$ denote the set of natural numbers and $\mathbb{Z}$ denote the set of integers.

2.1 Types

Definition 1. (Basic types) Let $\mathbb{T}_B$ denote the set of *basic types*, defined as:

$$\mathbb{T}_B \stackrel{def}{=} \{\texttt{bool}, \texttt{char}, \texttt{int8}, \texttt{int16}, \texttt{int32}, \texttt{int64}, \texttt{uint8}, \texttt{uint16}, \texttt{uint32}, \texttt{uint64}, \texttt{float32}, \texttt{float64}, \texttt{void}\}$$

Definition 2. (Struct types) A *struct type* is a triple $(l, M, type)$ where:

- $l \in$ **Names** is the *name* of the struct type,
- $M \subseteq$ **Names** is a (finite) set of names of struct *members*,
- $type : M \to$ **Types** is an assignment of types to members, where the set **Types** of types is defined below in Definition 5.

We denote $\mathbb{T}_S$ the set of all possible struct types.

Definition 3. (Array types) An *array type* is a triple (l, t, n) where

- $l \in$ **Names** is the *name* of the array type,
- $t \in$ **Types** is the type of the array items, where the set **Types** of types is defined below in Definition 5,
- $n \in \mathbb{N}$ is the size of the array type, with $n > 0$.

We denote $\mathbb{T}_A$ the set of all possible array types.

Definition 4. (Enum types) An *enum type* is a pair (l, J) where

- $l \in$ **Names** is the *name* of the enum type,
- $J \subseteq$ **Names** is a finite set of *enumerators*.

We denote $\mathbb{T}_E$ the set of all possible enum types.

Definition 5. (Types) Let **Types** denote the set of all *types*, defined as:

$$\textbf{Types} \stackrel{def}{=} \mathbb{T}_B \cup \mathbb{T}_S \cup \mathbb{T}_A \cup \mathbb{T}_E$$

Remark 1. The definitions in this section are mutually recursive. In particular Definition 2 and Definition 3 are given in terms of Definition 5 and vice-versa. To ensure that the set of all types is well-defined and no circularities arise we need some further definitions.

Definition 6. (Member types) Let $t = (l, M, type)$ be a struct type. We define the set of *member types* of t as

$$\text{memtypes}(t) \stackrel{def}{=} \{type(m) \,|\, m \in M\}$$

Let $t = (l, t', n)$ be an array type. We define the set of *member types* of t as

$$\text{memtypes}(t) \stackrel{def}{=} \{t'\}$$

Let t be a basic type or an enum type. Then

$$\text{memtypes}(t) \stackrel{def}{=} \emptyset$$

The set of *all direct or indirect member types* of a type t is defined as

$$\text{memtypes}^*(t) \stackrel{def}{=} \text{memtypes}(t) \cup \bigcup_{t' \in \text{memtypes}(t)} \text{memtypes}^*(t')$$

Definition 7. (Legal types) A type t is said to be *legal* or *well-defined* if $t \notin \text{memtypes}^*(t)$.

In the rest of this document we assume all types to be well-defined.

Definition 8. (Variable signature) A *variable signature* Σ is a pair $(V, type)$ where $V \subseteq \textbf{Names}$ is a finite set of *variables* and $type : V \to \textbf{Types}$ is a *typing* function, assigning a type to each variable.

2.2 Values

Definition 9. (Data values) Let **Values** denote the set of all possible *data values*, defined as:

$$\textbf{Values} \stackrel{def}{=} \{\bot\} \cup \mathbb{B} \cup \mathbb{C} \cup \mathbb{Z}^{(8)} \cup \mathbb{Z}^{(16)} \cup \mathbb{Z}^{(32)} \cup \mathbb{Z}^{(64)} \cup \mathbb{N}^{(8)} \cup \mathbb{N}^{(16)} \cup \mathbb{N}^{(32)} \cup \mathbb{N}^{(64)} \cup \mathbb{D}^{(32)} \cup \mathbb{D}^{(64)} \cup \mathbb{V}_S \cup \mathbb{V}_A \cup \mathbb{V}_E$$

where

- $\bot$ is called the *null value*,
- $\mathbb{B} \stackrel{def}{=} \{\texttt{true}, \texttt{false}\}$,
- $\mathbb{C}$ is the set of 8-bit characters. We write character literals as $\texttt{'a'}, \texttt{'b'}, \texttt{'c'}, ...$ etc.,
- $\mathbb{Z}^{(n)}$ is the set of n-bit signed integers,
- $\mathbb{N}^{(n)}$ is the set of n-bit unsigned integers,
- $\mathbb{D}^{(n)}$ is the set of n-bit floating-point decimals,
- $\mathbb{V}_S \stackrel{def}{=} \textbf{Names} \rightharpoonup \textbf{Values}$ is the set of all possible *struct values*, (partial functions) assigning values to struct members,
- $\mathbb{V}_A \stackrel{def}{=} \cup_{n \in \mathbb{N}} [J_n \to \textbf{Values}]$ is the set of all possible *array values*, *i.e.*, functions from a finite index set $J_n = \{0, 1, 2, ..., n-1\}$ to the set of values,
- $\mathbb{V}_E \stackrel{def}{=} \textbf{Names}$

We also define the family of value sets as:

$$\mathcal{V} \stackrel{def}{=} \{\{\bot\}, \mathbb{B}, \mathbb{C}, \mathbb{Z}^{(8)}, \mathbb{Z}^{(16)}, \mathbb{Z}^{(32)}, \mathbb{Z}^{(64)}, \mathbb{N}^{(8)}, \mathbb{N}^{(16)}, \mathbb{N}^{(32)}, \mathbb{N}^{(64)}, \mathbb{D}^{(32)}, \mathbb{D}^{(64)}, \mathbb{V}_S, \mathbb{V}_A, \mathbb{V}_E\}$$

Definition 10. (Type domains) The function $\text{tdom} : \textbf{Types} \to \mathcal{V}$ associates each type to a set of values, called its *type domain*, and defined by:

$$\begin{aligned}
\text{tdom}(\texttt{void}) &\stackrel{def}{=} \{\bot\} \\
\text{tdom}(\texttt{bool}) &\stackrel{def}{=} \mathbb{B} \\
\text{tdom}(\texttt{char}) &\stackrel{def}{=} \mathbb{C} \\
\text{tdom}(\texttt{int}n) &\stackrel{def}{=} \mathbb{Z}^{(n)} \\
\text{tdom}(\texttt{uint}n) &\stackrel{def}{=} \mathbb{N}^{(n)} \\
\text{tdom}(\texttt{float}n) &\stackrel{def}{=} \mathbb{D}^{(n)} \\
\text{tdom}(\mathbb{T}_S) &\stackrel{def}{=} \mathbb{V}_S \\
\text{tdom}(\mathbb{T}_A) &\stackrel{def}{=} \mathbb{V}_A \\
\text{tdom}(\mathbb{T}_E) &\stackrel{def}{=} \mathbb{V}_E
\end{aligned}$$

We extend this function to define the domain of values of structured types as follows:

Let $\text{tdom} : \mathbb{T}_S \to \mathbb{V}_S$ be defined for each struct type $t = (l, M, type) \in \mathbb{T}_S$ by

$$\text{tdom}(t) \stackrel{def}{=} \{v : \textbf{Names} \rightharpoonup \textbf{Values} \mid \text{dom}(v) = M,\ \forall m \in M.\ v(m) \in \text{tdom}(type(m))\}$$

in other words, the type domain of the struct type t is the set of all struct values, or assignments v of values to struct members such that the value $v(m)$ of each member $m \in M$ is in the type domain of m's type.

Let $\text{tdom} : \mathbb{T}_A \to \mathbb{V}_A$ be defined for each array type $t = (l, t', n) \in \mathbb{T}_A$ by

$$\text{tdom}(t) \stackrel{def}{=} \{v : J_n \to \textbf{Values} \mid \forall i \in J_n.\, v(i) \in \text{tdom}(t')\}$$

where $J_n = \{0, 1, 2, ..., n-1\}$ is the *index set* of t. In other words, the type domain of t is the set of all array values, or assignments v of values to array slots such that for all indices $i \in J_n$, the value $v(i)$ is in the type domain of t's member type t'.

2.3 Protocols

Definition 11. (Protocols) A *protocol* is a tuple (I, O, type) where:

- $I \subseteq$ **Names** is a set of *input signals*
- $O \subseteq$ **Names** is a set of *output signals*
- $I \cap O = \emptyset$
- $\mathit{type} : I \cup O \to$ **Types** is a typing function, assigning a type to each signal

We call **Protocols** the set of all possible protocols. Given a protocol $R = (I, O, \mathit{type})$ we define

- $\text{isignals}(R) \stackrel{def}{=} I$
- $\text{osignals}(R) \stackrel{def}{=} O$
- $\text{typmap}(R) \stackrel{def}{=} \mathit{type}$

Remark 2. A protocol can be seen as a variable signature $(I \uplus O, \mathit{type})$ (cf. Definition 8).

2.4 Interfaces

Definition 12. (Interfaces) An *interface* F is a tuple $(P, L, \mathit{prot}, \mathit{kind})$ where

- $P \subseteq$ **Names** is a set of *port names*,
- $L \subseteq$ **Protocols** is a set of protocols,
- $\mathit{prot} : P \to L$ is an assignment of protocols to ports,
- $\mathit{kind} : P \to \{\mathsf{base}, \mathsf{conj}\}$ is an assignment of kinds to ports

We call **Interfaces** the set of all possible interfaces. We define

- $\text{ports}(F) \stackrel{def}{=} P$
- $\text{protocols}(F) \stackrel{def}{=} L$
- $\text{protmap}(F) \stackrel{def}{=} \mathit{prot}$
- $\text{kindmap}(F) \stackrel{def}{=} \mathit{kind}$

Definition 13. (Input trigger alphabet of an interface) The *input trigger alphabet of an interface* $F = (P, L, \mathit{prot}, \mathit{kind})$, denoted $\text{itrigs}(F)$, is defined as

$$\begin{aligned}\text{itrigs}(F) \stackrel{def}{=} &\{(p, e) \mid p \in P,\, e \in \text{isignals}(\mathit{prot}(p)),\, \mathit{kind}(p) = \mathsf{base}\} \\ &\cup \{(p, e) \mid p \in P,\, e \in \text{osignals}(\mathit{prot}(p)),\, \mathit{kind}(p) = \mathsf{conj}\}\end{aligned}$$

We write $p.e$ for the element $(p, e) \in \text{itrigs}(F)$.

Definition 14. (Output statement alphabet of an interface) The *output statement alphabet of an interface* $F = (P, L, prot, kind)$ for a given variable signature $\Sigma = (V, type)$, denoted $\text{ostmts}(F, \Sigma)$, is defined as

$$\begin{aligned}\text{ostmts}(F,\Sigma) \stackrel{def}{=} &\{(p,e,x) \mid p \in P,\, e \in \text{osignals}(prot(p)),\, kind(p) = \mathsf{base},\, x \in V,\, type(x) = \text{typmap}(prot(p))(e)\} \\ &\cup \{(p,e,x) \mid p \in P,\, e \in \text{isignals}(prot(p)),\, kind(p) = \mathsf{conj},\, x \in V,\, type(x) = \text{typmap}(prot(p))(e)\}\end{aligned}$$

We write $p.e(x)$ for the element $(p, e, x) \in \text{ostmts}(F, \Sigma)$.

2.5 State machines

Definition 15. (Data stores) A *data store* over a variable signature $\Sigma = (V, type)$ is a function $\sigma : V \to \textbf{Values}$ assigning a value to each variable such that for each variable $x \in V$, $\sigma(x) \in \text{tdom}(type(x)) \uplus \{\perp\}$.

We call $\textbf{Stores}_\Sigma$ the set of all possible data stores over Σ. If σ is a store, x is a variable and v is a value, we write $\sigma[x := v]$ for the store that results from updating the value of variable x to v. Formally,

$$\sigma[x := v](y) \stackrel{def}{=} \begin{cases} v & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

We extend this notation to multiple updates:

$$\sigma[x_1 := v_1, x_2 := v_2, ..., x_n := v_n] \stackrel{def}{=} \sigma[x_1 := v_1][x_2 := v_2] \cdots [x_n := v_n]$$

We write $\emptyset$ for the *empty store*, and $\text{initstore}(\Sigma) \stackrel{def}{=} \{x \mapsto \perp \mid x \in V\}$ for the *initial store* over Σ, which assigns the null value to each variable.

Definition 16. (Activities) An *activity* over a variable signature Σ is a function $f : \textbf{Stores}_\Sigma \times (\textbf{Values} \uplus \{\perp\}) \to \textbf{Stores}_\Sigma \times \textbf{Values}$, *i.e.*, activities are functions that receive as argument a data store (see Definition 15) and an input value (or $\perp$) and return an updated store and a return value. We call $\textbf{Activities}_\Sigma$ the set of all actions over Σ.

Definition 17. (Triggers) Given an interface F, *the set of possible triggers over* F, is defined as $\textbf{Triggers}_F \stackrel{def}{=} \{\mathsf{ext}(p.e) \mid p.e \in \text{itrigs}(F)\} \cup \{\mathsf{int}(v) \mid v \in \textbf{Values}\}$. A trigger of the form $\mathsf{ext}(p.e)$ is called an *external event trigger*, and a trigger of the form $\mathsf{int}(v)$ is called an *activity completion trigger*.

Definition 18. (Output statements) Given an interface F, *the set of possible output statements over* F, is defined as $\textbf{OutStmts}_F \stackrel{def}{=} \{\mathsf{out}(p.e(x)) \mid p.e(x) \in \text{ostmts}(F, \Sigma)\} \cup \{\perp\}$. A term of the form $\mathsf{out}(p.e(x))$ is called an *output statement*. The term $\perp$ denotes the *null output*, *i.e.*, the absence of output.

Definition 19. (State machines) A *state machine* M is a tuple $(S, i, Q, F, def, \Sigma, A, act, T)$ where

- S is a finite set of *states*, also called *locations*,
- $i \in S$ is a distinguished *initial state*,
- $Q \subseteq S$ is a subset of *stable states*; the elements of the set $S \backslash Q$ of states not in Q are called *transient states*,
- F is an *interface*,
- $def : Q \to 2^{\text{ports}(F)}$ is a map, assigning a set of ports to each stable state; $def(s)$ is called the set of *deferred ports* of s, and must be such that for each $s \in Q$, there is a port $p \in \text{ports}(F)$ such that $p \notin def(s)$, (*i.e.*, no state can defer all ports),
- $\Sigma = (V, type)$ is variable signature (see Definition 8),
- $A \subseteq \textbf{Activities}_\Sigma$ is a set of of *activities*,

- $act : S\backslash Q \to A$ is a map assigning an activity to each transient state,
- $T \subseteq S \times \mathbf{Triggers}_F \times \mathbf{OutStmts}_F \times S$ is a *transition relation*, satisfying the following conditions:
  - for all $(s, \beta, \gamma, s') \in T$, written $s \xrightarrow[\gamma]{\beta} s'$ or $s \xrightarrow{\beta/\gamma} s'$
    - β is of the form $\mathsf{ext}(p.e)$ if and only if $s \in Q$ (*i.e.*, transitions from stable states must have input event triggers) and $p \notin def(s)$ (*i.e.*, port p must not be deferred in state s)
    - β is of the form $\mathsf{int}(v)$ if and only if $s \in S\backslash Q$ (*i.e.*, transitions from transient states must have values as condition)
  - for all $s \in S\backslash Q$ there is a finite sequence of transitions $s \xrightarrow{\beta_1/\gamma_1} s_1 \xrightarrow{\beta_2/\gamma_2} s_2 \cdots \xrightarrow{\beta_n/\gamma_n} s_n$ such that $s_n \in Q$; *i.e.*, any sequence of transitions from a transient state must lead to a stable state,
  - for all $s \in S$, if $s \xrightarrow{\beta'/\gamma'} s'$ and $s \xrightarrow{\beta''/\gamma''} s''$ then $\beta' = \beta''$, $\gamma' = \gamma''$ and $s' = s''$, that is, there is one and only one transition from s for each input event β of the form $\mathsf{ext}(x)$ if s is stable, and for each value β of the form $\mathsf{int}(v)$ if s is transient (*i.e.*, the machine is deterministic on input events and values).

For a state machine $M = (S, i, Q, F, def, \Sigma, A, act, T)$ we define the following functions:

- $\text{locations}(M) \stackrel{def}{=} S$
- $\text{initial}(M) \stackrel{def}{=} i$
- $\text{stables}(M) \stackrel{def}{=} Q$
- $\text{transients}(M) \stackrel{def}{=} S\backslash Q$
- $\text{interface}(M) \stackrel{def}{=} F$
- $\text{vars}(M) \stackrel{def}{=} \Sigma$
- $\text{activities}(M) \stackrel{def}{=} A$
- $\text{activitiesmap}(M) \stackrel{def}{=} act$
- $\text{transitions}(M) \stackrel{def}{=} T$

We call **StateMachines** the set of all possible state machines.

2.6 Atomic capsules

Definition 20. (Atomic capsules) An *atomic capsule* K is a tuple (F, Σ, A, M) where

- $F = (P, L, prot, kind) \in \mathbf{Interfaces}$ is an *interface* (see Definition 12),
- $\Sigma = (V, type)$ is variable signature where the variables are called *attributes*,
- $A \subseteq \mathbf{Activities}_\Sigma$ is a of of *actions* or *activities*,
- $M = (S, i, Q, F, def, \Sigma, A, act, T) \in \mathbf{StateMachines}$ is a state machine (see Definition 19)

We call **Atomic** the set of all possible atomic capsules. We define:

- $\text{ports}(K) \stackrel{def}{=} \text{ports}(F) = P$
- $\text{interface}(K) \stackrel{def}{=} F$

- attributes$(K) \stackrel{def}{=} \Sigma$
- activities$(K) \stackrel{def}{=} A$
- statemachine$(K) \stackrel{def}{=} M$

2.7 Composite capsules

Definition 21. (Composite capsules) A *composite capsule* K is a tuple $(F, H, R, role, C, link)$ where

- $F = (P, L, prot, kind) \in$ **Interfaces** is an *interface* (see Definition 12),
- $H \subseteq$ **Capsules** is a finite set of (atomic or composite) capsules called *parts*, subject to the condition that $K \notin H$ and K is not a sub-part (transitively) of any of its parts,
- $R \subseteq$ **Names** is a finite set of *role names*, such that $\mathsf{self} \notin R$,
- $role : R \uplus \{\mathsf{self}\} \to H \uplus \{K\}$ is a map associating each role with a capsule, where $role(\mathsf{self}) \stackrel{def}{=} K$, and for all $r \in R$, $role(r) \neq K$,
- $C \subseteq$ **Names** is a finite set of *connector names*,
- $link : C \to \text{connpts}(K) \times \text{connpts}(K)$ is map assigning each connector name to a *link* $(b_1, b_2) \in \text{connpts}(K) \times \text{connpts}(K)$ where connpts(K) denotes the set of all *connection points* of K and is defined as
$$\text{connpts}(K) \stackrel{def}{=} \{(\mathsf{self}, p) \mid p \in P\} \cup \bigcup_{r \in R} \{(r, p) \mid p \in \text{ports}(role(r))\}$$
We write $r.p$ for a connection point (r, p). For any connector $c \in C$, $link(c)$ must satisfy the following conditions:
  - for any connection point $r.p$, $(r.p, r.p) \notin link(c)$ (*i.e.*, $link(c)$ must be *irreflexive*, a port cannot be connected to itself),
  - for any connection point $r_1.p_1$, there is at most one connection point $r_2.p_2$ such that $(r_1.p_1, r_2.p_2) \in link(c)$ (i.e., links(c) must be a *partial* or *total function*)
  - for any connection point $r_2.p_2$, there is at most one connection point $r_1.p_1$ such that $(r_1.p_1, r_2.p_2) \in link(c)$ (i.e., $link(c)$ must be a *one-to-one mapping*)
  - whenever $link(c) = (r_1.p_1, r_2.p_2)$ such that $r_1 \neq \mathsf{self}$ and $r_2 \neq \mathsf{self}$, $prot_1(p_1) = prot_2(p_2)$ and either
    - $kind_1(p_1) = \mathsf{base}$ and $kind_2(p_2) = \mathsf{conj}$ or
    - $kind_1(p_1) = \mathsf{conj}$ and $kind_2(p_2) = \mathsf{base}$

    where $prot_i = \text{protmap}(\text{interface}(role(r_i)))$ and $kind_i = \text{kindmap}(\text{interface}(role(r_i)))$ for $i \in \{1, 2\}$ (*i.e.*, a connection between internal parts can only be between a base port and a conjugated port)
  - whenever $link(c) = (r_1.p_1, r_2.p_2)$ such that $r_1 = \mathsf{self}$ and $r_2 \neq \mathsf{self}$ or $r_1 \neq \mathsf{self}$ and $r_2 = \mathsf{self}$, $prot_1(p_1) = prot_2(p_2)$ and either
    - $kind_1(p_1) = \mathsf{base}$ and $kind_2(p_2) = \mathsf{base}$ or
    - $kind_1(p_1) = \mathsf{conj}$ and $kind_2(p_2) = \mathsf{conj}$

    where $prot_i = \text{protmap}(\text{interface}(role(r_i)))$ and $kind_i = \text{kindmap}(\text{interface}(role(r_i)))$ for $i \in \{1, 2\}$ (*i.e.*, a connection between a port of the composite capsule and a port of a sub-capsule must be of the same kind)

We call **Composite** the set of all possible composite capsules and **Capsules** $\stackrel{def}{=}$ **Atomic** $\cup$ **Composite** the set of all capsules. We define

- ports$(K) \stackrel{def}{=}$ ports$(F) = P$
- interface$(K) \stackrel{def}{=} F$
- parts$(K) \stackrel{def}{=} H$
- roles$(K) \stackrel{def}{=} R$
- connectors$(K) \stackrel{def}{=} C$
- links$(K) \stackrel{def}{=} (\bigcup_{c \in C} link(c)) \cup (\bigcup_{c \in C} \{(r_2.p_2, r_1.p_1) \mid (r_1.p_1, r_2.p_2) \in link(c)\})$

2.8 Proxy capsules

Definition 22. (Proxy capsules) A *proxy capsule* is an atomic capsule $K = (F, \Sigma, A, M)$ where the set of ports ports$(F) = P \uplus P_{\mathsf{os}}$ is partitioned in two subsets: a set P of *normal-* or *RT- ports* and a set P_{os} of *OS-ports*.

We call **Proxies** the set of all possible atomic capsules. We define:

- ports$(K) \stackrel{def}{=}$ ports(F)
- rtports$(K) \stackrel{def}{=} P$
- osports$(K) \stackrel{def}{=} P_{\mathsf{os}}$
- interface$(K) \stackrel{def}{=} F$
- attributes$(K) \stackrel{def}{=} \Sigma$
- activities$(K) \stackrel{def}{=} A$
- statemachine$(K) \stackrel{def}{=} M$

2.9 External task capsules

External task capsules are used in RTEdge$^{\mathrm{TM}}$ to describe the behaviour of processes external to the application. In the RTEdge$^{\mathrm{TM}}$ platform they can be associated to a state machine for the purpose of relating outgoing system signals to incoming system signals. This is used in causality (flow) analysis and schedulability analysis, but it doesn't represent part of the behaviour of the model. For this reason, to define the semantics of the model, we do not include the semantics of external tasks, which by definition, lie outside of the model. So for the purpose of defining the semantics of a model we only need to consider the interface of an external task.

Definition 23. (External task capsules) An *external task capsule* is a triple $E = (F, S, isimap)$ where:

- $F = (P_{\mathsf{os}}, L, prot, kind) \in$ **Interfaces** is an *interface* (see Definition 12), such that P_{os} is a set of *OS ports*, and
- S is a set of *independent system inputs*,
- $isimap : S \to P_{\mathsf{os}} \times \bigcup_{R \in L}(\text{isignals}(R) \cup \text{osignals}(R))$ is a bijective function such that for all $s \in S$, if $isimap(s) = (p, x)$ then
  - if $kind(p) =$ base then $x \in$ isignals$(prot(p))$, and
  - if $kind(p) =$ conj then $x \in$ osignals$(prot(p))$

We call **ExtTasks** the set of all external task capsules.

2.10 Applications

An application groups together a set of capsules, proxies and external tasks. It can be seen as a composite capsule containing all normal atomic and composite capsules as well as proxy capsules, and whose interface consists of all the OS ports of the proxy capsules. Syntactically, an application also includes the external task capsules,

Definition 24. (Applications) An *application* is a composite capsule $K = (F, H, R, role, C)$ where

- $H = \mathcal{K} \uplus \mathcal{P} \uplus \mathcal{E}$ is a set of capsules partitioned into *normal capsules* ($\mathcal{K} \subseteq$ **Capsules**), proxy capsules ($\mathcal{P} \subseteq$ **Proxies**) and external task capsules ($\mathcal{E} \subseteq$ **ExtTasks**),
- The interface F is defined such that the following hold, where $R' \stackrel{def}{=} \{r \in R \mid role(r) \in \mathcal{P}\}$:
  - $\text{ports}(F) = \bigcup_{r \in R'} \text{osports}(role(r))$ (where we assume the sets port names are mutually disjoint for all capsules)
  - $\text{protocols}(F) = \bigcup_{r \in R'} \text{protocols}(\text{interface}(role(r)))$
  - $\text{protmap}(F) = \bigcup_{r \in R'} \text{protmap}(\text{interface}(role(r)))$ (where we take the union of functions considered as sets)
  - $\text{kindmap}(F) = \bigcup_{r \in R'} \text{kindmap}(\text{interface}(role(r)))$ (where we take the union of functions considered as sets)

2.11 Extended interfaces

Definition 25. (Extended interfaces) An *extended interface* is a tuple $F = (P, L, prot, kind, F')$ where $F_{ext} = (P, L, prot, kind)$ is an interface (see Definition 12) and F' is also an interface called its *parent interface*, such that $P \cap \text{ports}(F') = \emptyset$ (*i.e.*, the set of ports must be disjoint).

An extended interface $F = (P, L, prot, kind, F')$ with parent $F' = (P', L', prot', kind')$ is the same as the interface $(P \cup P', L \cup L', prot \cup prot', kind \cup kind')$.[1] This is, whenever we talk about an extended interface, we really mean the equivalent simple interface.

2.12 Extended composite capsules

Definition 26. (Extended composite capsule) An *extended composite capsule* K is a tuple $(F, H, R, role, C, K')$ where $K' = (F', H', R', role', C')$ is a composite capsule and where $K_{ext} = (F_{ext}, H_{ext}, R_{ext}, role_{ext}, C_{ext})$ is also a composite capsule (see Definition 21) with:

- $F_{ext} = (\text{ports}(F) \cup \text{ports}(F'), \text{protocols}(F) \cup \text{protocols}(F'), \text{protmap}(F) \cup \text{protmap}(F'), \text{kindmap}(F) \cup \text{kindmap}(F'))$ is an interface, where $\text{ports}(F) \cap \text{ports}(F') = \emptyset$
- $H_{ext} = H \cup H'$
- $R_{ext} = R \cup R'$ where $R \cap R' = \emptyset$
- $role_{ext} = role \cup role'$
- $C_{ext} = C \cup C'$

Whenever we talk about an extended composite capsule K we mean the equivalent "base" composite capsule K_{ext}.

[1] Note that the unions of *prot* and *kind* are well defined because their domains are disjoint.

3 Semantics of RTEdge^{TM} models

The semantics of capsules is given in terms of labelled transition systems.

3.1 Labelled Transition Systems

Definition 27. (Labelled Transition Systems) A *labelled transition system* or *LTS* for short is a triple $(\mathcal{S}, \mathcal{L}, \rightarrow)$ where

- $\mathcal{S}$ is a set of *(concrete) states*,
- $\mathcal{L}$ is a set of *(action) labels*,
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is a *transition relation*,

A *rooted LTS* is an LTS $(\mathcal{S}, s_0, \mathcal{L}, \rightarrow)$ with a distinguished initial state $s_0 \in S$. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$.

Definition 28. (Execution fragment/behaviour) An *execution fragment* or *behaviour* ρ of an LTS $T = (\mathcal{S}, \mathcal{L}, \rightarrow)$ is a sequence of transitions $t_1 t_2 \cdots t_n$ where for each $i \in \{1, 2, ..., n-1\}$, $t_i = (s_i, \alpha_i, s_{i+1}) \in \rightarrow$. We also write the execution fragment as

$$s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} s_n$$

or as

$$s_1 \xrightarrow{\alpha_1 \alpha_1 \cdots \alpha_{n-1}} s_n$$

We call $\text{beh}_T(s)$ the *set of all execution fragments/behaviours* from state s in LTS T.

Given two states s and s' and a sequence of events $\tilde{\alpha} = \alpha_1 \alpha_2 \cdots \alpha_{n-1}$ we write $s \xrightarrow{\tilde{\alpha}} s'$ if there is a sequence of states $s_1 s_2 \cdots s_n$ such that $\rho = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} s_n$ is an execution fragment where $s_1 = s$ and $s_n = s'$.

Similarly, given a pair of states s and s' we write $s \xrightarrow{\tilde{\alpha}} s$ if there is a sequence of states $s_1 s_2 \cdots s_n$ and a sequence of events and a sequence of events $\tilde{\alpha} = \alpha_1 \alpha_2 \cdots \alpha_{n-1}$ such that $\rho = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} s_n$ is an execution fragment where $s_1 = s$ and $s_n = s'$.

Definition 29. (Event and state traces) Given an LTS $T = (\mathcal{S}, \mathcal{L}, \rightarrow)$ and an execution

$$\rho = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} s_n$$

we call the *event trace* $\text{evtrace}_T(\rho)$ of ρ the sequence of labels:

$$\text{evtrace}_T(\rho) = \alpha_1 \alpha_2 \cdots \alpha_{n-1}$$

and the *state trace* $\text{sttrace}_T(\rho)$ of ρ the sequence of states:

$$\text{sttrace}_T(\rho) = s_1 s_2 \cdots s_n$$

We call $\text{evtraces}_T(s)$ the set of *all event traces* of execution fragments beginning in state s, and $\text{sttraces}_(s)$ the set of all state traces of execution fragments beginning in s. This is

$$\text{evtraces}_T(s) \stackrel{def}{=} \{\alpha_1 \alpha_2 \cdots \alpha_{n-1} \,|\, \alpha_1 \alpha_2 \cdots \alpha_{n-1} \in \text{evtraces}_T(\rho),\, \rho = s \xrightarrow{\alpha_1 \alpha_2 \cdots \alpha_{n-1}} s'\}$$

and

$$\text{sttraces}_T(s) \stackrel{def}{=} \{s_1 s_2 \cdots s_n \,|\, s_1 s_2 \cdots s_n \in \text{sttraces}_T(\rho),\, s_1 = s\}$$

3.2 Semantics of atomic capsules

Definition 30. (Queues) Let X be a set. We write $\mathbf{Queues}_X$ for the set of queues over elements of X. Let $q \in \mathbf{Queues}_X$. Then

- front(q) is the front element x of q,
- enqueue(q, x) is the queue that results from adding x at the back,
- dequeue(q) is the pair (x, q') where x was the front element of q and q' is the queue that results from removing x from q.
- emptyqueue denotes the empty queue.

Definition 31. (Input alphabet of an interface) The *input alphabet of an interface* $F = (P, L, prot, kind)$, denoted inpalpha(F), is defined as

$$\begin{aligned}\text{inpalpha}(F) \stackrel{def}{=} &\{(p, e, v) \mid p \in P,\ e \in \text{isignals}(prot(p)),\ kind(p) = \mathsf{base},\ v \in \text{tdom}(\text{typmap}(prot(p))(e))\} \\ &\cup \{(p, e, v) \mid p \in P,\ e \in \text{osignals}(prot(p)),\ kind(p) = \mathsf{conj},\ v \in \text{tdom}(\text{typmap}(prot(p))(e))\}\end{aligned}$$

We write $p.e(v)$ for the element $(p, e, v) \in \text{inpalpha}(F)$.

Remark 3. The difference between the input alphabet of an interface and the input trigger alphabet of an interface (cf. Definition 13) is that the latter is part of the syntax of RTEdge$^{\text{TM}}$ state machines (used in the labels of transitions), whereas the former represents the set of actual run-time input events which carry a value of the type associated with the signal received.

Definition 32. (Output alphabet of an interface) The *output alphabet of an interface* $F = (P, L, prot, kind)$, denoted outalpha(F), is defined as

$$\begin{aligned}\text{outalpha}(F) \stackrel{def}{=} &\{(p, e, v) \mid p \in P,\ e \in \text{osignals}(prot(p)),\ kind(p) = \mathsf{base},\ v \in \text{tdom}(\text{typmap}(prot(p))(e))\} \\ &\cup \{(p, e, v) \mid p \in P,\ e \in \text{isignals}(prot(p)),\ kind(p) = \mathsf{conj},\ v \in \text{tdom}(\text{typmap}(prot(p))(e))\}\end{aligned}$$

We write $p.e(v)$ for the element $(p, e, v) \in \text{outalpha}(F)$.

Remark 4. The difference between the output alphabet of an interface and the output statement alphabet of an interface (cf. Definition 14) is that the later is part of the syntax of RTEdge$^{\text{TM}}$ state machines (used in the labels of transitions), whereas the former represents the set of actual run-time output events which carry a value of the type associated with the signal sent.

Definition 33. (Atomic capsule actions) Let $K = (F, V, A, M)$ be an atomic capsule. An *atomic capsule action* of K is one of the following:

- $\mathsf{recv}(\eta)$ (message reception) where $\eta \in \text{inpalpha}(F)$ (see Definition 31); η represents an input event received by the capsule to be added on a port's queue;
- $\mathsf{cons}(\eta/\theta)$ (message consumption) where $\eta \in \text{inpalpha}(F)$ and $\theta \in \text{outalpha}(F) \uplus \{\perp\}$ (see Definition 32); η is an event at the *front* of some port's queue which is to be consumed, and θ represents the output on some port, or $\perp$ if there is no output;
- $\mathsf{exec}(\eta/\theta)$ (activity execution) where $\eta \in \text{inpalpha}(F)$ and $\theta \in \text{outalpha}(F) \uplus \{\perp\}$; η represents the *last input event consumed*, which can be used by a transient state action, and θ represents the output on some port, or $\perp$ if there is no output;

The set of *possible actions* of K is denoted $\mathbf{AtomCapAct}_K$.

Definition 34. (Atomic capsule configurations) Let $K = (F, \Sigma, A, M)$ be an atomic capsule. A *configuration* or *concrete state* of K is a tuple (l, σ, q, x) where:

- $l \in \text{locations}(M)$ is a location of the state machine M,

- $\sigma \in \mathbf{Stores}_\Sigma$ is a data store over the variable signature Σ
- $q : \text{ports}(F) \to \mathbf{Queues}_{\text{inpalpha}(F)}$ is a map associating a queue of input events to each port so that $q(p)$ is the queue of port p,
- $x \in \text{inpalpha}(F) \uplus \{\bot\}$ is the last message processed or $\bot$ when there was no message before.

We call $\mathbf{AtomCapConfigs}_K = \text{locations}(M) \times \mathbf{Stores}_\Sigma \times [\text{ports}(F) \to \mathbf{Queues}_{\text{inpalpha}(F)}] \times (\text{inpalpha}(F) \uplus \{\bot\})$ the *set of all atomic capsule configurations of* K and **AtomCapConfigs** the *set of all atomic capsule configurations (of all possible atomic capsules).*

Definition 35. (Non-deterministic small-step LTS of an atomic capsule) Let $K = (F, \Sigma, A, M)$ be an atomic capsule. Then $\mathcal{T}_s[\![K]\!]$ denotes the rooted LTS $(\mathcal{S}, s_0, \mathcal{L}, \to_s)$ where

- $\mathcal{S} \stackrel{def}{=} \mathbf{AtomCapConfigs}_K$
- $s_0 \stackrel{def}{=} (\text{initial}(M), \text{initstore}(\Sigma), q_0, \bot)$ where $q_0(p) \stackrel{def}{=} \text{emptyqueue}$ for each $p \in \text{ports}(F)$
- $\mathcal{L} \stackrel{def}{=} \mathbf{AtomCapAct}_K$
- $\to_s \subseteq \mathbf{AtomCapConfigs}_K \times \mathbf{AtomCapAct}_K \times \mathbf{AtomCapConfigs}_K$ is defined as the smallest relation satisfying the following:
  - (MESSAGE RECEPTION)
    $(l, \sigma, q, x) \xrightarrow{\mathsf{recv}(p.e(v))}_s (l', \sigma', q', x')$ if
    - $l' = l$
    - $\sigma' = \sigma$
    - $q'(p) = \text{enqueue}(q(p), p.e(v))$
    - for all $p' \in \text{ports}(F)$ such that $p' \neq p$, $q'(p') = q(p')$
    - $x' = x$
  - (MESSAGE CONSUMPTION)
    $(l, \sigma, q, x) \xrightarrow{\mathsf{cons}(p.e(v)/\theta)}_s (l', \sigma', q', x')$ if
    - $l \in \text{stables}(M)$
    - $l \xrightarrow{\mathsf{ext}(p.e)/\mathsf{out}(p'.e'(z))} l' \in \text{transitions}(M)$ and $\theta = p'.e'(\sigma(z))$
      or $l \xrightarrow{\mathsf{ext}(p.e)/\bot} l' \in \text{transitions}(M)$ and $\theta = \bot$
    - $\sigma' = \sigma$
    - $(x', q'(p)) = \text{dequeue}(q(p))$
    - for all $p'' \in \text{ports}(F)$ such that $p'' \neq p$, $q'(p'') = q(p'')$
    - $x' = p.e(v)$
  - (ACTIVITY EXECUTION)
    $(l, \sigma, q, x) \xrightarrow{\mathsf{exec}(x/\theta)}_s (l', \sigma', q', x')$ if
    - $l \in \text{transients}(M)$
    - $x = p.e(v)$
    - $l \xrightarrow{\mathsf{int}(v')/\mathsf{out}(p'.e'(z))} l' \in \text{transitions}(M)$ and $\theta = p'.e'(\sigma'(z))$
      or $l \xrightarrow{\mathsf{int}(v')/\bot} l' \in \text{transitions}(M)$ and $\theta = \bot$
    - $f(\sigma, v) = (\sigma', v')$ where $f = \text{activitiesmap}(M)(l)$
    - $q' = q$
    - $x' = x$

Definition 36. (Acceptable executions) The set of *acceptable executions of an atomic capsule* K from a state s, is the largest set $\text{accexe}_T(s) \subseteq \text{beh}_T(s)$, where $T = \mathcal{T}_s[\![K]\!]$ (see Definition 35) such that for every execution $\rho = s \xrightarrow{\beta_1/\gamma_1}_s s_1 \xrightarrow{\beta_2/\gamma_2}_s \cdots \in \text{accexe}_T(s)$, there is an i such that β_i is of the form $\mathsf{cons}(\eta/\theta)$. This is, along every execution there is at least one message consumption.

Definition 37. (Priority assignment) A *priority assignment* is a tuple $(S, P, \leqslant, pri)$ where

- S is some set
- P is a set of *priorities*
- $\leqslant \subseteq P \times P$ is a partial order over priorities (*i.e.*, a reflexive, anti-symmetric and transitive relation over P) such that every non-empty subset $P' \subseteq P$ has a (unique) maximum element which we denote $\max(P')$ (*i.e.*, $\max(P') \in P'$ and $\forall p \in P'.\, p \leqslant \max(P')$)
- $pri : S \to P$ is an assignment of priorities to elements of S

Notation 2. We overload the notation for maximum elements and write $\max(S')$ for $\max(\{pri(x) \mid x \in S'\})$ for any $S' \subseteq S$.

Definition 38. (Prioritized small-step LTS of an atomic capsule) Let $K = (F, \Sigma, A, M)$ be an atomic capsule, $\mathcal{T}_s[\![K]\!] = (\mathcal{S}, s_0, \mathcal{L}, \to_s)$ the LTS defined for K in Definition 35 and $O = (\text{inpalpha}(F), P, \leqslant, pri)$ be some priority assignment of input events. Then $\mathcal{T}_{sp}[\![K]\!]_O$ denotes the rooted LTS $(\mathcal{S}, s_0, \mathcal{L}, \to_{sp})$ where:

- (MESSAGE RECEPTION)
  $(l, \sigma, q, x) \xrightarrow{\mathsf{recv}(p.e(v))}_{sp} (l', \sigma', q', x')$ if $(l, \sigma, q, x) \xrightarrow{\mathsf{recv}(p.e(v))}_{s} (l', \sigma', q', x')$
- (MESSAGE CONSUMPTION)
  $(l, \sigma, q, x) \xrightarrow{\mathsf{cons}(p.e(v)/\theta)}_{sp} (l', \sigma', q', x')$ if
  - $(l, \sigma, q, x) \xrightarrow{\mathsf{cons}(p.e(v)/\theta)}_{s} (l', \sigma', q', x')$, and
  - for all transitions $(l, \sigma, q, x) \xrightarrow{\mathsf{cons}(p'.e'(v')/\theta)}_{s} (l'', \sigma'', q'', x'')$, $pri(p'.e'(v')) \leqslant pri(p.e(v))$
- (ACTIVITY EXECUTION)
  $(l, \sigma, q, x) \xrightarrow{\mathsf{exec}(x/\theta)}_{sp} (l', \sigma', q', x')$ if $(l, \sigma, q, x) \xrightarrow{\mathsf{exec}(x/\theta)}_{s} (l', \sigma', q', x')$

3.3 Semantics of composite capsules

Definition 39. (Composite capsule actions) Let$K = (F, H, R, role, C, \text{links})$ be a composite capsule. A *composite capsule action* of K is one of the following:

- $\mathsf{recv}(\eta)$ (message reception) where $\eta \in \text{inpalpha}(F)$ (see Definition 31)
- $\mathsf{cons}(\eta/\theta)$ (message consumption) where $\eta \in \text{inpalpha}(F)$ and $\theta \in \text{outalpha}(F) \uplus \{\bot\}$ (see Definition 32)
- $\mathsf{exec}(\eta/\theta)$ (activity execution) where $\eta \in \text{inpalpha}(F)$ and $\theta \in \text{outalpha}(F) \uplus \{\bot\}$
- $\mathsf{msg}(u_1 \to u_2 | e(v))$ (message passing) where e is a signal name, v is a value, and $u_i \in \mathbf{Names}^*$ for $i \in \{1, 2\}$ such that each u_i is of the form $r_1.r_2.\cdots.r_n.p_k$ with p_k being a port name and each r_j being a role name with $r_{j+1} \in \text{roles}(K_j)$ (*i.e.*, u_i is a qualified path to a connection point, according to the nesting hierarchy of K)

The set of *possible actions* of K is denoted $\mathbf{CompCapAct}_K$.

Definition 40. (Composite capsule configurations) Let $K = (F, H, R, role, C, \text{links})$ be a composite capsule. A *configuration* or *concrete state* of K is an R-indexed set of configurations, this is, a function $s : R \to \mathbf{Configs}$ of the form

$$s = \{r_1 \mapsto s_1, ..., r_n \mapsto s_n\}$$

such that for all $r_i \in R$, $s_i = s(r_i) \in \mathbf{Configs}_{role(r_i)}$ is the configuration of the part (capsule) in role r_i where $\mathbf{Configs}_K$ denotes the *set of all possible configurations* of K and $\mathbf{Configs} \stackrel{def}{=} \mathbf{AtomCapConfigs} \cup \mathbf{CompCapConfigs}$ is the *set of all possible capsule configurations*, with $\mathbf{CompCapConfigs}$ being the *set of all composite capsule configurations.* [2] We call $\mathbf{CompCapConfigs}_K$ the *set of all composite capsule configurations* of K. Hence,

$$\mathbf{Configs}_K \stackrel{def}{=} \begin{cases} \mathbf{AtomCapConfigs}_K & \text{if } K \in \mathbf{Atomic} \\ \mathbf{CompCapConfigs}_K & \text{if } K \in \mathbf{Composite} \end{cases}$$

Definition 41. (Non-deterministic small-step LTS of a composite capsule) Let $K = (F, H, R, role, C, \text{links})$ be a composite capsule. Then $\mathcal{T}_s[\![K]\!]$ denotes the LTS $(\mathcal{S}, s_0, \mathcal{L}, \to_s)$ where

- $\mathcal{S} \stackrel{def}{=} \mathbf{CompCapConfigs}_K$
- $\mathcal{L} \stackrel{def}{=} \mathbf{CompCapAct}_K$
- $\to_s \subseteq \mathbf{CompCapConfigs}_K \times \mathbf{CompCapAct}_K \times \mathbf{CompCapConfigs}_K$ is defined as the smallest relation satisfying the following:
  - (INTER-CAPSULE COMMUNICATION 1)
    for any $r_1, r_2 \in \text{roles}(K)$, $s \xrightarrow{\mathsf{msg}(r_1.p_1 \to r_2.p_2 | e(v))}_{\text{s}} s'$ if
    - $(r_1.p_1, r_2.p_2) \in \text{links}(K)$,
    - $s(r_1) \xrightarrow{\mathsf{exec}(\eta / p_1.e(v))}_{\text{s}} s'(r_1)$ and
    - $s(r_2) \xrightarrow{\mathsf{recv}(p_2.e(v))}_{\text{s}} s'(r_2)$
  - (INTER-CAPSULE COMMUNICATION 2)
    for any $r_1, r_2 \in \text{roles}(K)$, $s \xrightarrow{\mathsf{msg}(r_1.p_1 \to r_2.p_2 | e(v))}_{\text{s}} s'$ if
    - $(r_1.p_1, r_2.p_2) \in \text{links}(K)$,
    - $s(r_1) \xrightarrow{\mathsf{cons}(\eta / p_1.e(v))}_{\text{s}} s'(r_1)$ and
    - $s(r_2) \xrightarrow{\mathsf{recv}(p_2.e)}_{\text{s}} s'(r_2)$
  - (INCOMING MESSAGE)
    for any $r_1 \in \text{roles}(K)$, $s \xrightarrow{\mathsf{recv}(p.e(v))}_{\text{s}} s'$ if
    - $(\mathsf{self}.p, r_1.p_1) \in \text{links}(K)$ and
    - $s(r_1) \xrightarrow{\mathsf{recv}(p_1.e(v))}_{\text{s}} s'(r_1)$
  - (OUTGOING MESSAGE 1)
    for any $r_1 \in \text{roles}(K)$, $s \xrightarrow{\mathsf{exec}(\eta / p.e(v))}_{\text{s}} s'$ if
    - $(r_1.p_1, \mathsf{self}.p) \in \text{links}(K)$ and
    - $s(r_1) \xrightarrow{\mathsf{exec}(\eta / p_1.e(v))}_{\text{s}} s'(r_1)$
  - (OUTGOING MESSAGE 2)
    for any $r_1 \in \text{roles}(K)$, $s \xrightarrow{\mathsf{cons}(\eta / p.e(v))}_{\text{s}} s'$ if

[2] A composite capsule configuration has a tree structure, where the leaves are atomic capsule configurations and in each node $s = \{r_1 \mapsto s_1, ..., r_n \mapsto s_n\}$, $s(r_i) = s_i \in \mathbf{Configs}_{role(r_i)}$ is a configuration of the capsule in role r_i. This definition is recursive, but it is well-founded, since composite capsules are finite and therefore have a finite height (nesting depth). Thus, the tree itself is finite.

  - $(r_1.p_1, \mathsf{self}.p) \in \text{links}(K)$ and
  - $s(r_1) \xrightarrow{\mathsf{cons}(\eta/p_1.e(v))}_{\text{s}} s'(r_1)$

- (INTERNAL MESSAGE)
  for any $r_i \in \text{roles}(K)$, $s \xrightarrow{\mathsf{msg}(r_i.u_1 \to r_i.u_2|e(v))}_{\text{s}} s'$ if
  - $s(r_i) \xrightarrow{\mathsf{msg}(u_1 \to u_2|e(v))}_{\text{s}} s'(r_i)$

3.4 Semantics of extended, proxy and external task capsules and applications

An extended composite capsule K is identified with its equivalent "base" composite capsule K_{ext} as defined in Subsection 2.12, therefore its semantics is the same according to Subsection 3.3.

The semantics of a proxy capsule K is given according to Subsection 3.2 with K viewed as an atomic capsule.

External task capsules do not have a semantics within RTEdge™, and are simply provided as a means to connect the application to the external environment.

The semantics of a application is given by the semantics in Subsection 3.3 where the application is viewed as a composite capsule with the proviso that the behaviour of external task capsules is not defined. Alternatively one can take the behaviour of external task capsules to include all possible behaviours involving independent system inputs. This can be defined as an LTS with a single state and for each possible signal, a transition from the state to itself. The language of such an LTS would include all possible sequences of external signals. Defined as such, the semantics of an application would contain all possible behaviours for all possible sequences of external signals. Yet another alternative is to view the application as a composite capsule without the external tasks in it.

References

[1] S. Gheorghe. Integration of Formal Model Checking with the RTEdge™ AADL Microkernel. Tech. Report 2011-01-2531, SAE International/Edgewater Computer Systems Inc., 18 October 2011.

[2] G. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Dept. of Computer Science, Aarhus University, 1981.

[3] SAE International. Architecture Analysis & Design Language (AADL). SAE Standard AS5506b, 10 September 2012.

Index

acceptable executions, 14
action completion code, 1
activities, 7
activity, 6
activity completion trigger, 1, 6
application, 2, 10
array type, 3
array values, 4
asynchronous communication, 1
atomic capsule, 1, 7
atomic capsule action, 12
atomic capsule configuration, 12
attributes, 7

base port, 1
basic types, 3
behaviour, 11

capsules, 1
composite capsule, 2, 8
composite capsule action, 14
composite capsule configuration, 15
concrete state of a composite capsule, 15
concrete state of an atomic capsule, 12
conjugate port, 1
connection points, 8
connector, 8
connectors, 1

data store, 6
data values, 4
deferred port, 1
deferred ports, 6

empty store, 6
enum type, 3
event trace, 11
execution fragment, 11
extended composite capsule, 10
extended interface, 10
external event trigger, 6
external event triggers, 1
external task capsule, 2, 9

independent system inputs, 9
initial state, 6
initial store, 6
input alphabet of an interface, 12
input signals, 5
input trigger alphabet of an interface, 5
interface, 1, 5

labelled transition system, 11
legal type, 4
link, 8
locations, 6
LTS, 11

member types, 3

names, 3
null output, 6
null value, 4

output alphabet of an interface, 12
output signals, 5
output statement, 6
output statement alphabet of an interface, 6
output statements, 1

parent interface, 10
parts, 8
port, 5
ports, 1
possible actions of an atomic capsule, 12
possible actions of an composite capsule, 14
priority assignment, 14
protocol, 1, 5
proxy capsule, 2, 9

roles, 8
rooted LTS, 11
run-to-completion semantics, 1

signals, 1
stable states, 6
state machine, 1, 6
state trace, 11
states, 6
struct type, 3
struct values, 4

the set of possible output statements, 6
the set of possible triggers, 6
timer, 2
transient states, 6
transition relation, 7
type domain, 4
types, 3
typing, 4

variable signature, 4

well-defined type, 4

A A textual syntax for RTEdge^{TM}

Table 1 on page 19 and Table 2 on page 20 show a sample of the textual (ASCII) notation corresponding to the mathematical notation for RTEdge^{TM} elements.

Textual ASCII notation	Mathematical notation

`protocol Prot1 {`
`  in signal a : bool;`
`  out signal b : char;`
`}`
	$Prot_1 = (I_1, O_1, type)$ where:
	• $I_1 = \{a\}$
	• $O_1 = \{b\}$
	• $type_1 = \{a \mapsto \texttt{bool}, b \mapsto \texttt{char}\}$

`interface F1 {`
`  base port p1 : Prot1;`
`  conj port p2 : Prot2;`
`}`
	$F_1 = (P_1, L_1, prot_1, kind_1)$ where
	• $P_1 = \{p_1, p_2\}$
	• $L_1 = \{r_1, r_2\}$
	• $prot_1 = \{p_1 \mapsto Prot_1, p_2 \mapsto Prot_2\}$
	• $kind_1 = \{p_1 \mapsto \mathsf{base}, p_2 \mapsto \mathsf{conj}\}$

`state machine M1 {`
`  stable states s0, s1, s3;`
`  transient states s2(m1);`
`  initial s0;`
`  attributes d : int32;`
`  transition t1 from s0 to s1 on q1.a;`
`  transition t2 from s0 to s2 on q2.c`
`  with output q3.b(0);`
`  transition t3 from s2 to s3 if true`
`  with output q3.b(1);`
`  transition t4 from s2 to s1 if false`
`  with output q2.a(1);`
`}`
	$M_1 = (S_1, i_1, Q_1, F_2, \Sigma_1, A_1, act_1, T_1)$ where
	• $S_1 = \{s_0, s_1, s_2, s_3\}$
	• $i_1 = s_0$
	• $Q_1 = \{s_0, s_1, s_3\}$
	• $F_2 = (P_2, L_2, prot_2, kind_2)$ where
	– $P_2 = \{q_1, q_2, q_3\}$
	– $L_2 = \{Prot_2, Prot_3\}$ with
	∗ $Prot_2 = (\{a\}, \{b\}, \{a \mapsto \texttt{void}, b \mapsto \texttt{int8}\})$
	∗ $Prot_3 = (\{a\}, \{c\}, \{a \mapsto \texttt{int8}, c \mapsto \texttt{void}\})$
	– $prot_2 = \{q_1 \mapsto Prot_2, q_2 \mapsto Prot_3, q_3 \mapsto Prot_2)$
	– $kind_2 = \{q_1 \mapsto \mathsf{base}, q_2 \mapsto \mathsf{conj}, q_3 \mapsto \mathsf{base}\}$
	• $\Sigma_1 = (\{d\}, \{d \mapsto \texttt{int32}\})$
	• $A_1 = \{m_1\}$
	• $act_1 = \{s_2 \mapsto m_1\}$
	• $T_1 = \{t_1, t_2, t_3, t_4\}$ where
	– $t_1 = s_0 \xrightarrow[\bot]{\mathsf{ext}(q_1.a)} s_1$
	– $t_2 = s_0 \xrightarrow{\mathsf{ext}(q_2.c)/\mathsf{out}(q_3.b(0))} s_2$
	– $t_3 = s_2 \xrightarrow{\mathsf{int}(\texttt{true})/\mathsf{out}(q_3.b(1))} s_3$
	– $t_4 = s_2 \xrightarrow{\mathsf{int}(\texttt{false})/\mathsf{out}(q_2.a(1))} s_1$

Table 1: Correspondence between mathematical notation and ASCII notation for RTEdge^{TM}.

Textual ASCII notation	Mathematical notation

`atomic K1 {`
`  implements F1;`
`  attribute a1 : bool`
`  attribute a2 : int32;`
`  activities m1, m2;`
`  behaviour M1;`
`}`
	$K_1 = (F_1, \Sigma_1, A_1, M_1)$ where:
	• F_1 is an interface such as the one shown in Table 1 on page 19
	• $\Sigma_1 = (V_1, type_1)$ where
	– $V_1 = \{a_1, a_2\}$
	– $type_1 = \{a_1 \mapsto \texttt{bool}, a_2 \mapsto \texttt{int32}\}$
	• $A_1 = \{m_1, m_2\}$
	• M_1 is a state machine such as the one shown in Table 1 on page 19

`composite K3 {`
`  implements F1;`
`  part r1 : K1;`
`  part r2 : K2;`
`  part r3 : K1;`
`  connector l1 : r1.p1 - r2.p3;`
`  connector l2 : r1.p2 - p1;`
`  connector l3 : p2 - r3.p2;`
`}`
	$K_3 = (F_3, H_3, R_3, role_3, C_3)$ where:
	• F_3 is an interface such as the one shown in Table 1 on page 19
	• $H_1 = \{K_1, K_2\}$
	• $R_1 = \{r_1, r_2, r_3\}$
	• $role_3 = \{r_1 \mapsto K_1, r_2 \mapsto K_2, r_3 \mapsto K_1, \mathsf{self} \mapsto K_3\}$
	• $C_3 = \{l_1, l_2, l_3\}$
	– $l_1 = (r_1.p_1, r_2.p_3)$
	– $l_2 = (r_1.p_2, \mathsf{self}.p_1)$
	– $l_3 = (\mathsf{self}.p_2, r_3.p_2)$

Table 2: Correspondence between mathematical notation and ASCII notation for RTEdge^{TM}.