The Architecture Analysis and Design Language and the Behavior Annex: A Denotational Semantics

Stefan Björnander, Cristina Seceleanu, Kristina Lundqvist, and Paul Pettersson

School of School of Innovation, Design, and Engineering
Mälardalen University, Sweden
{stefan.bjornander, cristina.seceleanu, kristina.lundqvist, paul.pettersson}@mdh.se

January 7, 2011

Abstract

We present a denotational semantics for the Architecture Analysis and Design Language with Behavior Annex and the Computational Tree logic. We also present tool support as an OSATE plug-in as well as the Production Cell case study.

Contents

Chapter 1

Introduction

Time-critical embedded systems play a vital role in, among others, aerospace applications, automotive systems, air traffic control, railway signaling, and medicine. Design and development of such systems is challenging, because the fulfillment of real time requirements and resource constraints has to be proven in the development process. Of high practical interest is the architecture design phase, because the timing behavior and resource consumption of systems depend heavily on the architecture chosen for the system. Furthermore, architectural mistakes that cause a system not to fulfill certain real-time requirements are hard to correct in later development phases. As a result, a development process for embedded systems should include verification techniques in the architecture design phase to provide evidence that a system architecture has the potential to fulfill its real-time requirements.

In this report we present a denotational semantics for the Architecture Analysis and Description Language (AADL) [9] with Behavior Annex [25]. AADL has been chosen, due to the sound specification language and its industrial use for the development of embedded systems in the automotive and avionic area. Denotational semantics has been chosen due to its exact and stringent mathematical notation and its close relationship to functional languages. As a part of the semantic definition, we have implemented the semantics in standard ML and encapsulated it in an OSATE (Open Source AADL Tool Environment[1]) plug-in.

The rest of this report is organized as follows: chapter 2 describes AADL, its Behavior Annex and the Computation Tree Logic (CTL), chapter 3 defines the syntax of the model and some abstract data types, chapter 4 defines the denotational semantics, chapter 6 deals with related work, and chapter 7 finally discusses conclusions and further work.

---

[1] aadl.info

Chapter 2

Background

The denotational semantics of this report is built upon AADL with Behavior Annex and the Computation Tree Logic.

2.1 AADL

AADL[1] is a large and complete language intended for the design of both the hardware and the software of a system. It is an Society of Automotive Engineers (SAE[2]) standard and is based on MetaH and UML [9]. Compared to MARTE [13], AADL is constrained in one respect: a specific phase of the development life cycle is addressed, and other stages cannot be addressed by AADL [8]. The component abstractions of the AADL are separated into three categories. The first category is the application software:

- **Thread.** Can execute concurrently and be organized into thread groups
- **Thread Group.** Component abstraction for logically organizing threads or thread groups components within a process.
- **Process.** Protected address space whose boundaries are enforced at runtime.
- **Data.** Data types and static data.
- **Subprogram.** Model of a subprogram component that represents a callable piece of source code.

The second category is the execution platform (the hardware):

- **Processor.** Schedules and executes threads.
- **Memory.** Stores code and data.
- **Device.** Represents sensors and actuators that interface with the external environment.

---

[1] aadl.info

[2] www.sae.org

- **Bus.** Interconnects processors, memory, and devices.

The third category is the system component. System components are composites that can consist of other systems as well as software or hardware components. The components types are defined using a parameterized set of properties. Furthermore, components communicate with each other through ports. It is possible to define physical port-to-port connections as well as logical flows through chains of ports. Component definitions are divided into component types holding the public (visible to other components) features, and component implementations that define the private parts of the component.

The AADL standard includes runtime semantics for mechanisms of exchange and control of data, including message passing, event passing, synchronized access to shared components, thread scheduling protocols, and timing requirements.

AADL can be used to model and analyze systems already in use as well as to design new systems. AADL can also be used in the analysis of partially defined architectural patterns. Moreover, AADL supports the early prediction and analysis of critical system qualities, such as performance, schedulability, and reliability. Within the core language, property sets can be declared that add new properties for components. Additional models and properties can also be included by utilizing the extension capabilities of the language. The properties and extensions can be used to incorporate analyses at the architectural design level.

AADL components interact through defined interfaces. A component interface consists of directional flow through data ports for state data, event data ports for message data, event ports for asynchronous events, subprogram calls, and explicit access to data components. Application components have properties that specify timing requirements such as period, worst-case execution time, deadlines, space requirements, and arrival rates [11].

There is a number of tools developed for AADL. One of them is OSATE, which is a plug-in for the Eclipse Development Environment[3]. It supports analysis and simulation of AADL models.

2.1.1 The AADL Behavior Annex

In order to increase the expressiveness of AADL, it is possible to add *annexes*. One of them is the Behavior Annex [12] that models an abstract state machine [5]. Each component of the model describes its logic by defining a behavior model, which consists of three parts [10]:

- **States**. The states of the machine, one of them is the initial state.
- **Transitions**. The condition for a transition from one state to another (or between the same state) is determined by a guard: an expression evaluated to a logical value.
- **State Variables**. The state variables are similar to variables in programming languages; they can be inspected and assigned.

---

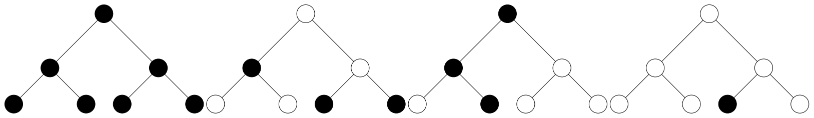
[3] www.eclipse.org

2.1.2 Computation Tree Logic

Computation Tree Logic (CTL) is a branching-time temporal logic; that is, it models time as a tree structure with a non-determined future. There are different paths into the future and any one of them may be the one realized. There are several operators involved in CTL: two child operators (All and Exists), operating on the children of a node, and five path operators (Global, Final, Until, Weak until, and Release), operating on the nodes along one path. Se table 2.1 for a closer description, figure 2.1 for some operator combinations, and section 3.1.4 for an example.

A ϕ	ϕ must be satisfied for every child.
E ϕ	ϕ must be satisfied for at least one child.
G ϕ	ϕ must be satisfied for each node on the path.
F ϕ	ϕ must be satisfied for at least one node on the path.
ϕ U φ	ϕ must be satisfied for each node until (but not necessarily inclusive) φ is satisfied.
ϕ W φ	ϕ must be satisfied for each node until φ is satisfied. The difference against the Until operator is that φ does not have to become satisfied. In that case, ϕ has to be satisfied for each node at the path.
ϕ R φ	φ must be satisfied until (and inclusive) ϕ is satisfied.

Table 2.1: Computational Tree Logic.



(a) All Global P. (b) All Finally P. (c) Exists Global P. (d) Exists Finally P.

Figure 2.1: CTL Operator Combinations.

Chapter 3

Preliminaries

In order to understand the semantics, we need to set up some preliminaries. The semantics is based on the syntax and each semantic rule follows is syntactic counterpart. In order to store the values of the semantic input, we also need some basic abstract data types: list, table, set, and tree.

3.1 The Syntax of Architectural Elements

In this section, we describe the syntax by defining a Backus Normal Form grammar [16] for AADL with Behavior Annex and CTL property specification; it is divided into three parts: the model, the behavior annex, and the property specification. The epsilon (ε) symbol denotes the empty string. It is also possible to add comments rows, beginning with two hyphens and lasting to the end of the row.

Definition ::= *Model PropSpecS*

3.1.1 The AADL Structural Elements

The AADL Model of this report is limited to systems. There are two kinds of systems: the *system* that defines the port interface and the behavior annex and the *system implementation* that defines the subcomponents and the port connections between them.

Model ::= *System SystemImpl*

Components

A system is made up of optional features (input and output ports) and an optional behavior annex (its syntax is given in section 3.1.2). As is evident from the grammar, there has to be at least one system and exactly one system implementation, which occurs at the end of the definition.

System ::= *System System*
| **system** *Identifier SystemBody* **end ;**

SystemBody ::= *OptionalFeatures OptionalAnnex*

The system implementation comprises optional subcomponents and optional connections.

SystemImpl ::= **system implementation** *Identifier* . *Identifier*
SystemImplBody **end ;**

SystemImplBody ::= *OptionalSubcomponents OptionalConnections*

Connections

The features of a system is part of its interface against other systems and is made up by input and output port. They are later connected with each other.

OptionalFeatures ::= **features** *Feature*
| ε

Feature ::= *Feature Feature*
| *Identifier* **: in event port ;**
| *Identifier* **: out event port ;**

Configuration

The configuration is made up of subcomponents and connections. The subcomponents are instances of earlier defined systems (equivalent to classes and objects in object oriented languages) and the connections are drawn between input and output ports in the subcomponents, not the systems. The systems have fact in played out its role when the subcomponents have been defined.

OptionalSubcomponents ::= **subcomponents** *Subcomponent*
| ε

Subcomponent ::= *Subcomponent Subcomponent*
| *Identifier* **: system** *Identifier* **;**

OptionalConnections ::= **connections** *Connection*
| ε

Connection ::= *Connection Connection*
| **: event port** *Identifier* . *Identifier* **->**
Identifier . *Identifier* **;**

3.1.2 The AADL Behavior Annex Structural Elements

As mention in section 2.1.1, the Behavior Annex models a state machine holding states, transitions, and state variables. In this report, however, we extend the annex with *initialization lists* and *action lists*. In both cases, the state variables can be assigned to values of expressions and output port can be triggered. The initializations is a stand alone part of the annex while each action list is connected to a transition.

With these extensions, the behavior annex comprises four parts: state variables, initializations of state variables and output ports, states, and transitions with actions lists.

OptionalAnnex ::= *Annex*
| ε

Annex ::= **annex** *Identifier* **{****
OptionalStateVariables *OptionalInitializations*
OptionalStates *OptionalTransitions* ****}** **;**

Declaration

It is possible to define state variables and states. All state variables have integer types, one of the states is the initial state.

OptionalStateVariables ::= **state variables** *StateVariables*
| ε

StateVariable ::= *StateVariable* *StateVariable*
| *Identifier* **:** **integer** **;**

OptionalStates ::= **states** *State*
| ε

State ::= *State* *State*
| *Identifier* **:** **initial state** **;**
| *Identifier* **:** **state** **;**

Execution

The *OptionalInitializations* grammatical rule simple calls the *Action* grammatical rule because the initialization lists are in fact action lists.

OptionalInitializations ::= **initializations** *Action*
| ε

An transition has a source state, a guard expression, a target state and an optional list of actions.

OptionalTransitions ::= **transitions** *Transition*
| ε

Transition ::= *Transition* *Transition*
| *Identifier* **-[** *ExpressionS* **]->** *Identifier* *OptionalAction*

In the action part, the state variables becomes initialized with values evaluated from expressions and signals are sent to output ports. If the action list is present, it is surrounded by braces; if not, it is replaced by a semicolon.

OptionalAction ::= **{** *Action* **}**
| **;**

Each action is either an assignment of a state variable or the triggering of an outport port. Each individual action is terminated by a semicolon.

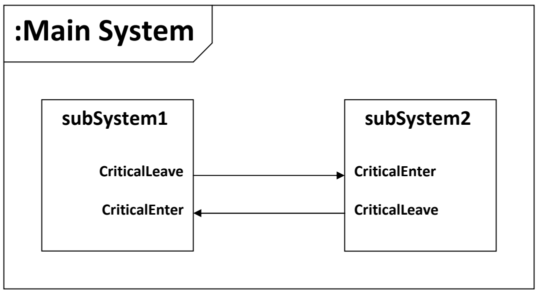


Figure 3.1: Mutual Exclusive Critical Sections.

$$\begin{array}{rcl} \mathit{Action} & ::= & \mathit{Action}\ \mathit{Action} \\ & | & \mathit{Identifier} := \mathit{ExpressionS}\ \mathbf{;} \\ & | & \mathit{Identifier}\ \mathbf{!}\ \mathbf{;} \end{array}$$

An expression can be a value, an identifier representing a state variable or an input port followed by a question mark, or the sum, difference, product, or quotient of two expressions. The *S* in *ExpressionS* stands for *Syntax* in order to distinguish it from the *Expression* type in chapter 4.

$$\begin{array}{rcl} \mathit{ExpressionS} & ::= & \mathit{Value} \\ & | & \mathit{Identifier} \\ & | & \mathit{Identifier}\ ? \\ & | & (\ \mathit{ExpressionS}\ ) \\ & | & \mathit{ExpressionS} + \mathit{ExpressionS} \\ & | & \mathit{ExpressionS} - \mathit{ExpressionS} \\ & | & \mathit{ExpressionS} * \mathit{ExpressionS} \\ & | & \mathit{ExpressionS}\ /\ \mathit{ExpressionS} \end{array}$$

3.1.3 An Example: Mutual Exclusive Critical Sections

Below follows an example of an AADL with Behavior model. It is made up by two subsystems with one critical section each. They communicate with port signals in order to make sure they cannot reach their critical sections at the same time. As seen in the listing below, the subsystems are similar. The only different is that the first subsystem is initialized to trigger a *CriticalLeave* signal, which means that the second subsystem is free to enter its critical section.

Each subsystem starts in the initial state *Waiting* and waits until it receives the *CriticalEnter* signal from the other subsystem. Then it enter its critical section and when it leaves it sends the *CriticalLeave* signal to the other subsystem in order to allow it to enter its critical section.

```
system SubSystem1
  features
    CriticalEnter: in event port;
    CriticalLeave: out event port;
  annex SubSystemAnnex1 {**
   initializations
     CriticalLeave!;
```

```
   states
     Waiting :initial state;
     Critical :state;
   transitions
      Waiting -[CriticalEnter?]-> Critical;
      Critical -[true]-> Waiting {CriticalLeave!;}
   **};
end SubSystem1;

system SubSystem2
  features
    CriticalEnter: in event port;
    CriticalLeave: out event port;
  annex SubSystemAnnex2 {**
   states
     Waiting :initial state;
     Critical :state;
   transitions
      Waiting -[CriticalEnter?]-> Critical;
      Critical-[true]-> Waiting {CriticalLeave!;}
   **};
end SubSystem2;
```

The main system instantiate the two subsystems as subcomponents *subsystem1* and *subsystem2* as well as connecting them to each other with the *CriticalLeave* and *CriticalEnter* ports, see figure 3.1. Also see chapter 5 for a more extensive example.

```
system implementation MainSystem
  subcomponents
    subSystem1: system SubSystem1;
    subSystem2: system SubSystem2;
  connections
    event port subSystem1.CriticalLeave -> subSystem2.CriticalEnter;
    event port subSystem2.CriticalLeave -> subSystem1.CriticalEnter;
end MainSystem.impl;
```

The process of determine whether a source code satisfying a grammar is called *parsing*, see appendix A for an example.

3.1.4 Computation Tree Logic

Syntactically speaking, there are two kinds of properties: the tree and node property. As stated in section 3.3.4, each node of a tree holds a value. The value of a tree property (**true** or **false**) depends on the value of the tree node in question as well as the values of the tree nodes of the subtree. The value of the node property only depends on the value of the tree node.

The *PropSpecS* (the *S* stands for *Syntax*) grammatical rule supports the *all* and *exists* child operators as well as the *global*, *final*, *until*, *weak until*, and *release* path operators. As mentioned in section 2.1.2, the outmost operators must be a child and path operator pair. In this syntax, the operators must be given in that order. The syntax is also case sensitive, the operators shall be given in lower-case letters.

PropSpecS ::= **a g** *PropSpecS*
| **a f** *PropSpecS*
| **a** *PropSpecS* **u** *PropSpecS*
| **a** *PropSpecS* **w** *PropSpecS*
| **a** *PropSpecS* **r** *PropSpecS*
| **e g** *PropSpecS*
| **e f** *PropSpecS*
| **e** *PropSpecS* **u** *PropSpecS*
| **e** *PropSpecS* **w** *PropSpecS*
| **e** *PropSpecS* **r** *PropSpecS*
| *NodePropSpecS*

Node Property Specifications

The *NodePropSpecS* (the *S* stands for *Syntax*) grammatical rule is a logical, relational, or arithmetic expression. It can also be an identifer followed by a dot and another identifer identifying the name of a subcomponent and the name of a state or state variable.

NodePropSpecS ::= **(** *PropSpecS* **)**
| **not** *PropSpecS*
| *PropSpecS* **and** *PropSpecS*
| *PropSpecS* **or** *PropSpecS*
| *PropSpecS* = *PropSpecS*
| *PropSpecS* ! = *PropSpecS*
| *PropSpecS* < *PropSpecS*
| *PropSpecS* <= *PropSpecS*
| *PropSpecS* > *PropSpecS*
| *PropSpecS* >= *PropSpecS*
| *PropSpecS* + *PropSpecS*
| *PropSpecS* − *PropSpecS*
| *PropSpecS* ∗ *PropSpecS*
| *PropSpecS* / *PropSpecS*
| *Identifier* **.** *Identifier*

An Example of a Property Specification

In section 3.1.3, a main system holding two subsystem with one critical section each was presented. In order to make sure that the two subsystems never reach their critical section at the same time, the property specification below can be formulated. The *all* and *global* operator combination decides whether the expression is always evaluated to **true**, which in turn means that the two subsystems never reach their critical section at the same time.

```
a g not (subSystem1.Critical and subSystem2.Critical)
```

3.2 Values

An identifier is a string of character that begins with a letter or an underscore and is followed by a number (possible zero) of letters, digits, or underscores. An integer can hold any integer value. A boolean value is either **false** or **true**. A connection holds the index of the sending subcomponent in the global subcomponent list (see section 4.1.2) together with the output port name as well as the index of the receiving subcomponent and the input port name.

An expression is a value, the name of a state variable or an input port, or an relational or arithmetic arithmetic expression. An action is either the sending of a signal through an output port or the assignment of a evaluated expression value to a state variable. A transition is the source state integer value, the guard expression, the target state integer value, and a (possible empty) list of actions. A system is the current state integer value (initialized to zero, representing the initial state), the symbol table holding the input and output ports, the state variable, and states as well as a (possible empty) list of initializations (equivalent to actions), and a (possible empty) list of transitions. A Value is a state, boolean, or integer value, or an action, transition, or system.

Identifier = [a-zA-Z_][a-zA-Z0-9_]*
Integer = {..., -3, -2, -1, 0, 1, 2, 3, ...}
Boolean = {**false**, **true**}
Connection = Integer × Identifier × Integer × Identifier
Expression = *value* Value + *identifier* Identifier +
eq (Expression × Expression) +
ne (Expression × Expression) +
lt (Expression × Expression) +
le (Expression × Expression) +
gt (Expression × Expression) +
ge (Expression × Expression) +
add (Expression × Expression) +
sub (Expression × Expression) +
mul (Expression × Expression) +
div (Expression × Expression)
Action = *send* Identifier + *assign* (Identifier × Expression)
Transition = Identifier × Expression × Identifier × List
System = Integer × Table × List × List
Value = *state* Integer + *boolean* Boolean + *integer* Integer +
action Action + *transition* Transition +
system System

3.3 Abstract Data Types

In an AADL model, identifers are bound to values that needs to be stored for further use. Therefore, we need the abstract data types List, Table, Set, and Tree to holds values.

3.3.1 List

List is a recursively defined abstract data type with the operations *list_empty* that returns an empty list, *list_insert* that adds a value at the beginning of the list, *list_add* that adds a value at the end of the list, *list_get* that returns the value at the given index in the list, *list_set* that updates the value at the given index, *list_index_of* that returns the index of a given value, and *list_split* that returns the first value and the rest of the list as a pair, and *list_merge* that appends the second list to the first one.

List = *list_null* + *list_enter* List

list_empty : List
list_empty =
    list_null

list_insert : Value × List → List
list_insert value list =
    list_enter (*value*, *list*)

$\mathit{list\_add} : \text{Value} \times \text{List} \to \text{List}$
$\mathit{list\_add}\ \mathit{value}_1\ (\mathit{list\_enter}\ (\mathit{value}_2,\ \mathit{tail})) =$
$\quad \mathit{list\_enter}\ (\mathit{value}_2,\ \mathit{list\_add}\ \mathit{value}_1\ \mathit{tail})$
$\mathit{list\_add}\ \mathit{value}\ \mathit{list\_null} =$
$\quad \mathit{list\_enter}\ (\mathit{value},\ \mathit{list\_null})$
$\mathit{list\_get} : \text{Integer} \times \text{List} \to \text{Value}$
$\mathit{list\_get}\ \mathbf{0}\ (\mathit{list\_enter}\ (\mathit{value},\ \mathit{tail})) =$
$\quad \mathit{value}$
$\mathit{list\_get}\ \mathit{index}\ (\mathit{list\_enter}\ (\mathit{value},\ \mathit{tail})) =$
$\quad \mathit{list\_get}\ (\mathit{index}\ \mathbf{-}\ \mathbf{1})\ \mathit{tail}$
$\mathit{list\_set} : \text{Integer} \times \text{Value} \times \text{List} \to \text{List}$
$\mathit{list\_set}\ \mathbf{0}\ \mathit{value}_1\ (\mathit{list\_enter}\ (\mathit{value}_2,\ \mathit{tail})) =$
$\quad \mathit{list\_enter}\ (\mathit{value}_1,\ \mathit{tail})$
$\mathit{list\_set}\ \mathit{index}\ \mathit{value}_1\ (\mathit{list\_enter}\ (\mathit{value}_2,\ \mathit{tail})) =$
$\quad \mathit{list\_enter}\ (\mathit{value}_2,\ \mathit{list\_set}\ \mathit{value}_1\ (\mathit{index}\ \mathbf{-}\ \mathbf{1})\ \mathit{tail})$
$\mathit{list\_index\_of} : \text{Value} \times \text{List} \to \text{Integer}$
$\mathit{list\_index\_of}\ \mathit{value}_1\ (\mathit{list\_enter}\ (\mathit{value}_2,\ \mathit{tail})) =$
$\quad \mathbf{if}\ \mathit{value}_1 = \mathit{value}_2\ \mathbf{then}\ \mathbf{0}$
$\quad \mathbf{else}\ (\mathit{list\_index\_of}\ \mathit{value}_1\ \mathit{tail})\ \mathbf{+}\ \mathbf{1}$
$\mathit{list\_split} : \text{List} \to (\text{Value} \times \text{List})$
$\mathit{list\_split}\ (\mathit{list\_enter}\ (\mathit{value},\ \mathit{tail})) =$
$\quad (\mathit{value},\ \mathit{tail})$
$\mathit{list\_merge} : \text{List} \times \text{List} \to \text{List}$
$\mathit{list\_merge}\ \mathit{list}\ (\mathit{list\_enter}\ (\mathit{head},\ \mathit{tail})) =$
$\quad \mathit{list\_enter}\ (\mathit{head},\ \mathit{list\_merge}\ \mathit{list}\ \mathit{tail})$
$\mathit{list\_merge}\ \mathit{list}\ \mathit{list\_null} =$
$\quad \mathit{list}$

3.3.2 Table

Table is a recursively abstract data that associates identifers (keys) with values. It holds the operations *table_empty* that returns an empty table, *table_set* that associates an identifier with a value (if the identifier already is associated with a value, that value is dismissed), *table_get* that look up the value associated with the given identifier, *table_merge* that merges two tables into one (if the same identifier is associated with a value in both tables, the value of the second table is associated with the identifier in the resulting table), and *table_to_list* that returns a list holding the values (not the keys) of the table.

$\text{Table} = \mathit{table\_null} + \mathit{table\_enter}\ ((\text{Identifier} \times \text{Value}) \times \text{Table})$

$\mathit{table\_empty} : \text{Table}$
$\mathit{table\_empty} =$
$\quad \mathit{table\_null}$
$\mathit{table\_set} : \text{Identifier} \times \text{Value} \times \text{Table} \to \text{Table}$
$\mathit{table\_set}\ \mathit{ident}_1\ \mathit{value}_1\ (\mathit{table\_enter}\ ((\mathit{ident}_2,\ \mathit{value}_2),\ \mathit{rest})) =$
$\quad \mathbf{if}\ \mathit{ident}_1 = \mathit{ident}_1\ \mathbf{then}\ \mathit{table\_enter}\ ((\mathit{ident}_1,\ \mathit{value}_2),\ \mathit{rest})$
$\quad \mathbf{else}\ \mathit{table\_enter}\ ((\mathit{ident}_2,\ \mathit{value}_2),\ \mathit{table\_set}\ \mathit{ident}_1\ \mathit{value}_1\ \mathit{rest})$
$\mathit{table\_set}\ \mathit{ident}\ \mathit{value}\ \mathit{table\_null} =$
$\quad \mathit{table\_enter}\ ((\mathit{ident},\ \mathit{value}),\ \mathit{table\_null})$

```
table_get : Identifier × Table → Value
table_get $ident_1$ (table_enter (($ident_2$, value), rest)) =
    if $ident_1 = ident_2$ then value
    else table_get $ident_1$ rest
table_merge : Table × Table → Table
table_merge (table_enter ((ident, value), rest)) =
    table_enter ((ident, value), table_to_table rest)
table_to_table table_null table =
    table
table_to_list : Table → List
table_to_list (table_enter ((ident, value), rest)) =
    list_insert value (table_to_list rest)
table_to_list value list_null =
    list_empty
```

3.3.3 Set

Set is a recursively defined abstract data type holding the operations *set_empty* that returns an empty table, *set_add* that adds a value to the set (unless it is already present), and *set_exists* that decides whether a value is present in the set.

```
Set = set_null + set_enter Set

set_empty : Set
set_empty =
    set_null
set_add : Value × Set → Set
set_add $value_1$ (set_enter ($value_2$, tail)) =
    if $value_1 = value_2$ then set_enter ($value_2$, tail)
    else set_add $value_1$ tail
set_add value set_null =
    set_enter (value, set_null)
set_exists : Value × Set → Boolean
set_exists $value_1$ (set_enter ($value_2$, tail)) =
    if $value_1 = value_2$ then true
    else set_exists $value_1$ tail
set_exists value set_null =
    false
```

3.3.4 Tree

Tree is a recursively defined abstract data type holding the operations *tree_create* that returns a tree with one node holding the given value, *tree_add_child* that adds a child node, holding the given value, to the root node of the tree, *tree_set_value* that sets the value of the root node of the tree, and *tree_get_children* that returns a list holding the subtrees of the root node of the tree.

```
Tree = tree_null + tree_enter (Value × List)

tree_create : Value → Tree
tree_create value =
    tree_enter (value, tree_null)
```

$$\mathit{tree\_add\_child} : \text{Value} \times \text{Tree} \rightarrow \text{Tree}$$
$$\mathit{tree\_add\_child}\ \mathit{child}\ (\mathit{tree\_enter}\ (\mathit{value},\ \mathit{child\_list})) = \quad \mathit{tree\_enter}\ (\mathit{value},\ \mathit{list\_add}\ \mathit{child}\ \mathit{child\_list})$$

$$\mathit{tree\_get\_value} : \text{Tree} \rightarrow \text{Value}$$
$$\mathit{tree\_get\_value}\ (\mathit{tree\_enter}\ (\mathit{value},\ \mathit{child\_list})) = \quad \mathit{value}$$

$$\mathit{tree\_get\_children} : \text{Tree} \rightarrow \text{List}$$
$$\mathit{tree\_get\_children}\ (\mathit{tree\_enter}\ (\mathit{value},\ \mathit{child\_list})) = \quad \mathit{child\_list}$$

Chapter 4

Semantics

In AADL, the semantics is made up by systems. Formally, a system is a tuple $(S, s_0, A_{init}, V, P_{in}, P_{out}, T)$ where S is a non-empty finite set of states and $s_0 \in S$ is the compulsory initial state. V is a possible empty finite set of state variables. $A_{init} \subseteq V \times E + P_{out}$ is a set of initializations, which can be either state variables assigned to expressions or signals sent to outports. P_{in} and P_{out} is the possible empty finite sets of inports and outports, respectively. $T \subseteq S \times E \times S \times A$ is a possible empty finite set of transitions. The possible empty finite action set $A \subseteq V \times E + P_{out}$ is a set of assignments and output signal triggings similar to the A_{init} set above. E is a set of expression recursively defined as $E = \mathit{variable}\ V + \mathit{value}\ C + \mathit{inport}\ P_{in} + \mathit{add}\ (E \times E) + \mathit{sub}\ (E \times E) + \mathit{mul}(E \times E) + \mathit{div}\ (E \times E)$, where C is a constant integer value. The input port expression has boolean type.

The semantics of this chapter is divided into several steps:

- **The AADL Model.** For each system, its input and output ports are saved in a symbol table together with the behavior annex's states and state variables. For each system implementation, the subcomponent are stored in a global subcomponent list (see section 4.1.2) and the connections between the subcomponents are stored in a global connection list.
- **The AADL Behavior Annex.** For each behavior annex, its states and state variables are stored in a symbol table. The initializations and transitions are stored in lists.
- **The CTL Property Specification.** Made up by a combination of CTL logic operators and regular expressions and is stored in an intern format that are used in the Property Evaluation section below.
- **Initialization.** The initialization list of each subcomponent is executed and the result is stored in the global subcomponent list.
- **Expression Evaluation.** When a transition is to be taken, we need to evaluate the guard expression into a boolean value in order to decide whether to take the transition. When a state variable is to be assigned to an expression in a initialization list or a transition action list, we also need to evaluate the expression into a value, in this case an integer value.
- **Connection.** Before the transition list of each subcomponent is traversed, we first need to execute any connections. That is, if an output port is set to **true** in one subcomponent, we set it to **false** and set its corresponding input port (in the same or another subcomponent) to **true**.
- **Generation.** We create a state tree where each node has an subcomponent list as value. Determinism (exactly one transition can be taken) generates a

path from the root node to a leaf node. In case of non-determinism (several transitions can be taken) each possible transition generates a new sub tree as a child.

- **Property Specification Evaluation.** When the state tree is generated, it needs to be evaluates against the CTL property in order to decide whether the property yields **true**.

The *definition* semantics rule defines the overall process of the semantics of this chapter. First, the *model* semantic rule extract the subcomponent table and the list of connections between the subcomponents, then the *prop_spec* semantic rule returns the CTL property specification for further use. Moreover, the global subcomponent list (converted from the subcomponent table with the *table_to_list* operation) is then initialized by the *traverse_init_list* semantic rule. The resulting subcomponent list is then stored as the value of the root node of the state tree. When the state tree has been generated by the *generate_tree* semantic rule, it is evaluated against the CTL property, which finally yields the boolean result.

The information for each system is stored in the tuple (*state*, *symbol_table*, *init_list*, *trans_list*), where *state* is the current state of the annex, *symbol_table* is holding the input and output ports of the system as well as states and state variables of the annex, *init_list* holds the list of initializations, and *trans_list* holds the list of transitions. For each system, one such tuple is associated with the name of the system (the name of the behavior annex is ignored). The table is then used in the subcomponent section, where tuples are copied and placed in the subcomponent table (it needs to be a table instead of list since components are needed to become looked up in the CTL property specifications), which is global since only one system implementation is allowed. The connections are placed in the global connection list. The subcomponent table is then transformed into a list and initialized, it is repletely traversed during the construction of the state tree and finally the property specification is evaluated against the state tree.

```
definition : Model × PropSpecS → Boolean
definition ⟦M PS⟧ =
    let (system_table, conn_list) = model M in
    let prop_spec = prop_spec PS system_table in
    let inst_list = initalize_subcomponent_list (table_to_list subcomp_table) in
    let init_tree = tree_create inst_list in
    let state_tree = generate_tree inst_list conn_list set_empty init_tree in
        evaluate_prop_spec prop_spec state_tree
```

4.1 The AADL Model

The purpose of the semantic rules of this section is to collect and store information about the model. There are three global tables and lists: the information about the systems is stored in the system table (*system_table*), the information about the subcomponents is stored in the subcomponent table (*subcomp_table*), and the information about the connections between the subcomponent is stored the connection list (*conn_list*). As there is only one system implementation, the subcomponent table and connection list are global. The system implementation name is ignored.

```
model : Model → Table
model ⟦S SI⟧ =
    let system_table = system S in
        system_impl SI system_table
```

4.1.1 System

A system is made up of optional features (input and output ports) and an optional behavior annex (the semantics of the behavior annex is given in section 4.2). The semantic rule *system* returns a system table holding information (current state, symbol table, initialization list, and transition list) of each system.

```
system : System → Table
system ⟦$S_1$ $S_2$⟧ =
    let system_table$_1$ = system $S_1$ in
    let system_table$_2$ = system $S_2$ in
        table_merge system_table$_1$ system_table$_2$
system ⟦system $I$ $SB$ end ;⟧ =
    table_set $I$ (system_body $SB$) table_empty
```

The system body contains an optional lists of features (input and output ports) and an optional behavior annex (see section 4.2 for its semantics). The system is made up of its current state, symbol table (holding input and output ports, states, and state variables), initialization list, and transitions list. The current state of the system is initialized to zero, representing the initial state.

```
system_body : SystemBody → Value
system_body ⟦$OF$ $OA$⟧ =
    let in_out_port_table = optional_features $OF$
    let (var_state_table, init_list, trans_list) = optional_annex $OA$ in
    let symbol_table = table_merge in_out_port_table var_state_table in
        system (0, symbol_table, init_list, trans_list)
```

The input and output ports are boolean values, initialized to **false**. The *feature* semantic rule needs neither the system table nor the subcomponent table, its task is to simple collect the ports and store them as boolean values in the symbol table.

```
optional_features : OptionalFeatures → Table
optional_features ⟦features $F$⟧ =
    feature $F$
optional_features ⟦⟧ =
    table_empty
```

The *feature* semantic rule associates the name of each input and output port to the boolean value **false**.

```
feature : Feature → Table
feature ⟦$F_1$ $F_2$⟧ =
    let port_table$_1$ = feature $F_1$ in
    let port_table$_2$ = feature $F_2$ in
        table_merge port_table$_1$ port_table$_2$
feature ⟦$I$ : in event port⟧ =
    table_set $I$ (boolean false) table_empty
feature ⟦$I$ : out event port⟧ =
    table_set $I$ (boolean false) table_empty
```

4.1.2 System Implementation

The system implementation is constituted by optional subcomponents and connections between the subcomponents (not between the systems). The *system_impl_body* semantic rule needs the system table in order to look up systems and instantiate subcomponent of them. It generates and returns a subcomponent table holding the subcomponents and a list of connections between them.

```
system_impl : SystemImpl × Table → (Table × List)
system_impl ⟦system implementation $I_1.I_2$ SIB end ;⟧ system_table =
    system_impl_body SIB system_table

system_impl_body : SystemImplBody × Table → (Table × List)
system_impl_body ⟦OS OC⟧ system_table =
    let subcomp_table = optional_subcomponents OS system_table in
    let conn_list = optional_connections OC subcomp_table in
        (subcomp_table, conn_list)
```

The *subcomponent* semantic rule contributes to the global subcomponent table. For each subcomponent, it stores a copy of the system by looking it up in the system table. If there is no subcomponents, an empty table is returned.

```
optional_subcomponents : OptionalSubcomponents × Table → Table
optional_subcomponents ⟦subcomponents S⟧ system_table =
    subcomponent S system_table
optional_subcomponents ⟦⟧ system_table =
    table_empty

subcomponent : Subcomponent × Table → Table
subcomponent ⟦$SC_1$ $SC_2$⟧ system_table =
    let $subcomp\_table_1$ = subcomponent $SC_1$ system_table in
    let $subcomp\_table_2$ = subcomponent $SC_2$ system_table in
        table_merge $subcomp\_table_1$ $subcomp\_table_2$
subcomponent ⟦$I_1$ : system $I_2$ ;⟧ system_table =
    let subcomp = table_get $I_2$ system_table in
        table_set $I_1$ subcomp table_empty
```

For each connection, the *connection* semantic rule gathers its information: the index of source and target subcomponent in the subcomponent list (transformed from table to list by *table_to_list*) and the name of the input and output ports. This information is then stored in the global connection list.

```
optional_connections : OptionalConnections × Table → List
optional_connections ⟦connections S⟧ subcomp_table =
    connection S subcomp_table
optional_connections ⟦⟧ subcomp_table =
    list_empty

connection : Connection × Table → List
connection ⟦$C_1$ $C_2$⟧ subcomp_table =
    let $conn\_list_1$ = connection $C_1$ subcomp_table in
    let $conn\_list_2$ = connection $C_2$ subcomp_table in
        table_merge $conn\_list_1$ $conn\_list_2$
connection ⟦ event port $I_1$. $I_2$ -> $I_3$. $I_4$ ;⟧ subcomp_table =
    let inst_list = table_to_list subcomp_table in
    let outsystem_record = table_get $I_1$ subcomp_table in
    let outsystem_index = list_index_of outsystem_record inst_list in
    let insystem_record = table_get $I_3$ subcomp_table in
    let insystem_index = list_index_of insystem_record inst_list in
        list_add (connection (outsystem_index, $I_2$, insystem_index, $I_4$))
```

4.2 The AADL Behavior Annex

The behavior annex is the part of the AADL model that defines the behavior of the model. It is based on the Abstract State Machine [5]. It holds states (among which one is the initial state) and state variables, which can be initialized. It also holds input and output signals connected to the ports of the surrounding system, the output signals can be initialized. Finally, it holds a set of transitions between the states [10]. Each transition can be equipped with a guard; that is, a boolean expression that has to evaluated to **true** for the transition to be granted (technically, each transition has a guard; however, it can be limited to the value **true**). A transition can also be equipped with a list of actions; that is, assignments of state variables or sending of signals to output ports. The annex semantic rules does not need the component and subcomponent tables. They just return a tuple holding the symbol table with the states and state variables, the initialization list, and the transitions of the annex. If there is no annex, an empty table and empty lists are returned.

$$
\begin{array}{l}
\mathit{optional\_annex} : \text{OptionalAnnex} \to (\text{Table} \times \text{List} \times \text{List}) \\
\mathit{optional\_annex}\ [\![A]\!] = \\
\quad \mathit{annex}\ A \\
\mathit{optional\_annex}\ [\![ ]\!] = \\
\quad (\mathit{table\_empty}, \mathit{list\_empty}, \mathit{list\_empty})
\end{array}
$$

The *annex* semantic rule collects information about the annex parts. Even thought the annex is named, we discard the name. As the annex is surrounded by a system, the name of the system will be sufficient. The state variable table and the state table is merged into one, we assume that each state variable and state is given a unique name and that the state variable and state name sets are disjunct.

$$
\begin{array}{l}
\mathit{annex} : \text{Annex} \to (\text{Table} \times \text{List} \times \text{List}) \\
\mathit{annex}\ [\![\textbf{annex}\ I\ \textbf{\{**}\ \mathit{OSV}\ \mathit{OI}\ \mathit{OS}\ \mathit{OT}\ \textbf{**\}}\ \textbf{;}]\!] = \\
\quad \textbf{let}\ \mathit{var\_table} = \mathit{optional\_state\_variables}\ \mathit{OSV}\ \textbf{in} \\
\quad \textbf{let}\ \mathit{init\_list} = \mathit{optional\_initalizations}\ \mathit{OI}\ \textbf{in} \\
\quad \textbf{let}\ \mathit{state\_table} = \mathit{optional\_states}\ \mathit{OS}\ \textbf{in} \\
\quad \textbf{let}\ \mathit{trans\_list} = \mathit{optional\_transitions}\ \mathit{OT}\ \mathit{state\_table}\ \textbf{in} \\
\quad\quad (\mathit{table\_merge}\ \mathit{var\_table}\ \mathit{state\_table}, \mathit{init\_list}, \mathit{trans\_list})
\end{array}
$$

All state variables hold integer type and are associated with the zero value. However, they may be initialized to other integer values by the *initialization* semantic rule below.

$$
\begin{array}{l}
\mathit{optional\_state\_variables} : \text{OptionalStateVariables} \to \text{Table} \\
\mathit{optional\_state\_variables}\ [\![\textbf{state variables}\ \mathit{SV}]\!] = \\
\quad \mathit{state\_variable}\ \mathit{SV} \\
\mathit{optional\_state\_variables}\ [\![ ]\!] = \\
\quad \mathit{table\_empty}
\end{array}
$$

$$
\begin{array}{l}
\mathit{state\_variable} : \text{StateVariable} \to \text{Table} \\
\mathit{state\_variable}\ [\![\mathit{SV}_1\ \mathit{SV}_2]\!] = \\
\quad \textbf{let}\ \mathit{state\_table}_1 = \mathit{state\_variable}\ \mathit{SV}_1\ \textbf{in} \\
\quad \textbf{let}\ \mathit{state\_table}_2 = \mathit{state\_variable}\ \mathit{SV}_2\ \textbf{in} \\
\quad\quad \mathit{table\_merge}\ \mathit{state\_table}_1\ \mathit{state\_table}_2 \\
\mathit{state\_variable}\ [\![I\ \textbf{:}\ \textbf{integer}\ \textbf{;}]\!] = \\
\quad \mathit{table\_add}\ I\ (\mathit{integer}\ \mathbf{0})\ \mathit{table\_empty}
\end{array}
$$

The *state* semantic rule is called with the integer value one as the parameter *number*. In this way, the initial state will be associated with the integer value zero and the other states will associated with unique positive integer values. As the semantic rule is called with the integer value one, and it is increased by one each time a state is associated with a value. The non-initial states will be given consecutive positive integer values (starting from one).

$$\begin{array}{l}
\mathit{optional\_states} : \text{OptionalStates} \rightarrow \text{Table}\\
\mathit{optional\_states}\ \llbracket \textbf{states}\ S \rrbracket =\\
\quad \textbf{let}\ (\mathit{state\_table}, \mathit{state\_number}) = \mathit{states}\ SV\ \mathbf{1}\ \textbf{in}\\
\qquad \mathit{state\_table}\\
\mathit{optional\_states}\ \llbracket\,\rrbracket =\\
\quad \mathit{table\_empty}
\end{array}$$

$$\begin{array}{l}
\mathit{state} : \text{State} \times \text{Integer} \rightarrow (\text{Table} \times \text{Integer})\\
\mathit{state}\ \llbracket S_1\ S_2 \rrbracket\ \mathit{state\_number} =\\
\quad \textbf{let}\ (\mathit{state\_table}_1, \mathit{state\_number}_1) = \mathit{state}\ S_1\ \mathit{state\_number}\ \textbf{in}\\
\quad \textbf{let}\ (\mathit{state\_table}_2, \mathit{state\_number}_2) = \mathit{state}\ S_2\ \mathit{state\_number}_1\ \textbf{in}\\
\qquad (\mathit{table\_merge}\ \mathit{state\_table}_1\ \mathit{state\_table}_2, \mathit{state\_number}_2)\\
\mathit{state}\ \llbracket I\ \textbf{: initial state} \rrbracket\ \mathit{state\_number} =\\
\quad (\mathit{table\_set}\ I\ (\mathit{state}\ \mathbf{0})\ \mathit{table\_empty}, \mathit{state\_number})\\
\mathit{state}\ \llbracket I\ \textbf{: state} \rrbracket\ \mathit{state\_number} =\\
\quad (\mathit{table\_set}\ I\ (\mathit{state}\ \mathit{state\_number})\ \mathit{table\_empty}, \mathit{state\_number} + \mathbf{1})
\end{array}$$

The *optional_initialization* semantic rule just calls the *action* semantic rule, since they perform the same task: gather the action associated with a transition or initialization, respectively, in a list. An action is either the assignment of a value to a state variable or the sending of a signal to an outport.

$$\begin{array}{l}
\mathit{optional\_initialization} : \text{OptionalInitializations} \rightarrow \text{List}\\
\mathit{optional\_initialization}\ \llbracket \textbf{initializations}\ I \rrbracket =\\
\quad \mathit{action}\ I\\
\mathit{optional\_initialization}\ \llbracket\,\rrbracket =\\
\quad \mathit{list\_empty}
\end{array}$$

For each transition, we look up the integer value of the source and target state (we assume they are stored in the symbol table) as well as collect the guard expression and the possible empty action list.

$$\begin{array}{l}
\mathit{optional\_transition} : \text{OptionalTransision} \rightarrow \text{List}\\
\mathit{optional\_transition}\ \llbracket \textbf{transitions}\ I \rrbracket =\\
\quad \mathit{transition}\ I\\
\mathit{optional\_transition}\ \llbracket\,\rrbracket =\\
\quad \mathit{list\_empty}
\end{array}$$

$$\begin{array}{l}
\mathit{transition} : \text{Transition} \times \text{Table} \rightarrow \text{List}\\
\mathit{transition}\ \llbracket T_1\ T_2 \rrbracket\ \mathit{symbol\_table} =\\
\quad \textbf{let}\ \mathit{trans\_list}_1 = \mathit{transition}\ S_1\ \mathit{symbol\_table}\ \textbf{in}\\
\quad \textbf{let}\ \mathit{trans\_list}_2 = \mathit{transition}\ S_2\ \mathit{symbol\_table}\ \textbf{in}\\
\qquad \mathit{list\_merge}\ \mathit{trans\_list}_1\ \mathit{trans\_list}_2\\
\mathit{transition}\ \llbracket I_1\ \textbf{-[}\ E\ \textbf{]->}\ I_2\ \text{OA} \rrbracket\ \mathit{symbol\_table} =\\
\quad \textbf{let}\ \mathit{state}\ \mathit{source\_state} = \mathit{table\_get}\ I_1\ \mathit{symbol\_table}\ \textbf{in}\\
\quad \textbf{let}\ \mathit{state}\ \mathit{target\_state} = \mathit{table\_get}\ I_2\ \mathit{symbol\_table}\ \textbf{in}\\
\qquad\quad \mathit{list\_add}\ (\mathit{transition}\ (\mathit{source\_state}, \mathit{expression}\ E, \mathit{target\_state},\\
\qquad\qquad \mathit{optional\_action}\ OA))\ \mathit{list\_empty}
\end{array}$$

The *action* semantic rule collects the action associated with a transition. An action may be the assignment of a state variable or the sending of a signal through a port connection. We store the name of the state variable together with the expression in the action list or the name of the output port.

```
optional_action : OptionalAction → List
optional_action ⟦{ A }⟧ =
    action A
optional_action ⟦;⟧ =
    list_empty

action : Action → List
action ⟦$A_1$ $A_2$⟧ =
    let $action\_list_1$ = action $A_1$ in
    let $action\_list_2$ = action $A_2$ in
        list_merge $action\_list_1$ $action\_list_2$
action ⟦I := E⟧ =
    list_add (action (assign (I, expression E))) list_empty
action ⟦I ! ;⟧ =
    list_add (action (init I)) list_empty
```

An expression may be an identifier (representing a state variable), value or input port as well as the addition, subtraction, multiplication, or division of two expressions.

```
expression : ExpressionS → Expression
expression ⟦V⟧ =
    value V
expression ⟦I⟧ =
    identifier I
expression ⟦I ?⟧ =
    receive I
expression ⟦( E )⟧ = E
expression ⟦$E_1$ + $E_2$⟧ =
    add ($E_1$, $E_2$)
expression ⟦$E_1$ - $E_2$⟧ =
    sub ($E_1$, $E_2$)
expression ⟦$E_1$ * $E_2$⟧ =
    mul ($E_1$, $E_2$)
expression ⟦$E_1$ / $E_2$⟧ =
    div ($E_1$, $E_2$)
```

4.3 CTL Property Specification

There are two kinds of property specifications: tree and node property. The tree specification affects the subtree of a node as well of the value of the node itself. The tags *single* and *double* are necessary since we need to catch the path operators with one operand (*global* and *final*) as well as those with two operands (*until*, *weak*, and *release*).

WidthOp = *all* + *exists*
DepthOp = *global* + *final* + *until* + *weak* + *release*
TreePropSpec = *single* PropSpec + *double* (PropSpec × PropSpec)
PropSpec = *tree_prop_spec* (WidthOp × DepthOp × TreePropSpec) + *node_prop_spec* NodePropSpec

The node specification only affects the value of the node. On the whole, it is similar to regular expressions. However, it can also be made up by a constant value, state variable, or state. As the former two cases are syntactically equivalent, the *path_prop_spec* semantic rule uses the symbol table to distinguish them.

$$\begin{aligned}
&\mathrm{NodePropSpec} = \mathit{not}\ \mathrm{PropSpec}\ +\\
&\quad \mathit{and}\ (\mathrm{PropSpec} \times \mathrm{PropSpec}) + \mathit{or}\ (\mathrm{PropSpecn} \times \mathrm{PropSpec})\ +\\
&\quad \mathit{eq}\ (\mathrm{PropSpec} \times \mathrm{PropSpec}) + \mathit{ne}\ (\mathrm{PropSpec} \times \mathrm{PropSpec})\ +\\
&\quad \mathit{lt}\ (\mathrm{PropSpec} \times \mathrm{PropSpec}) + \mathit{le}\ (\mathrm{PropSpec} \times \mathrm{PropSpec})\ +\\
&\quad \mathit{gt}\ (\mathrm{PropSpec} \times \mathrm{PropSpec}) + \mathit{ge}\ (\mathrm{PropSpec} \times \mathrm{PropSpec})\ +\\
&\quad \mathit{add}\ (\mathrm{PropSpec} \times \mathrm{PropSpec}) + \mathit{sub}\ (\mathrm{PropSpec} \times \mathrm{PropSpec})\ +\\
&\quad \mathit{mul}\ (\mathrm{PropSpec} \times \mathrm{PropSpec}) + \mathit{div}\ (\mathrm{PropSpec} \times \mathrm{PropSpec})\ +\\
&\quad \mathit{value}\ \mathrm{Value} + \mathit{state}\ (\mathrm{Integer} \times \mathrm{Integer})\ +\\
&\quad \mathit{variable}\ (\mathrm{Integer} \times \mathrm{Identifier})
\end{aligned}$$

Each tree property specification must begin with a width operator (*all* or *exists*) followed by a depth operator (*until*, *weak*, or *release*). It can also be a node property specification.

$$\begin{aligned}
&\mathit{prop_spec} : \mathrm{PropSpecS} \times \mathrm{Table} \to \mathrm{PropSpec}\\
&\mathit{prop_spec}\ [\![\mathbf{a}\ \mathbf{g}\ \mathit{PS}]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec} = \mathit{prop_spec}\ \mathit{PS}\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{all}, \mathit{global}, \mathit{single}\ \mathit{prop_spec})\\
&\mathit{prop_spec}\ [\![\mathbf{a}\ \mathbf{f}\ \mathit{PS}]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec} = \mathit{prop_spec}\ \mathit{PS}\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{all}, \mathit{final}, \mathit{single}\ \mathit{prop_spec})\\
&\mathit{prop_spec}\ [\![\mathbf{a}\ \mathit{PS}_1\ \mathbf{u}\ \mathit{PS}_2]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec}_1 = \mathit{prop_spec}\ \mathit{PS}_1\ \mathit{subcomp_table}\ \mathbf{in}\\
&\quad \mathbf{let}\ \mathit{prop_spec}_2 = \mathit{prop_spec}\ \mathit{PS}_2\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{all}, \mathit{until}, \mathit{double}\ (\mathit{prop_spec}_1, \mathit{prop_spec}_2))\\
&\mathit{prop_spec}\ [\![\mathbf{a}\ \mathit{PS}_1\ \mathbf{w}\ \mathit{PS}_2]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec}_1 = \mathit{prop_spec}\ \mathit{PS}_1\ \mathit{subcomp_table}\ \mathbf{in}\\
&\quad \mathbf{let}\ \mathit{prop_spec}_2 = \mathit{prop_spec}\ \mathit{PS}_2\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{all}, \mathit{weak}, \mathit{double}\ (\mathit{prop_spec}_1, \mathit{prop_spec}_2))\\
&\mathit{prop_spec}\ [\![\mathbf{a}\ \mathit{PS}_1\ \mathbf{r}\ \mathit{PS}_2]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec}_1 = \mathit{prop_spec}\ \mathit{PS}_1\ \mathit{subcomp_table}\ \mathbf{in}\\
&\quad \mathbf{let}\ \mathit{prop_spec}_2 = \mathit{prop_spec}\ \mathit{PS}_2\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{all}, \mathit{release}, \mathit{double}\ (\mathit{prop_spec}_1, \mathit{prop_spec}_2))\\
&\mathit{prop_spec}\ [\![\mathbf{e}\ \mathbf{g}\ \mathit{PS}]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec} = \mathit{prop_spec}\ \mathit{PS}\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{exists}, \mathit{global}, \mathit{single}\ \mathit{prop_spec})\\
&\mathit{prop_spec}\ [\![\mathbf{e}\ \mathbf{f}\ \mathit{PS}]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec} = \mathit{prop_spec}\ \mathit{PS}\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{exists}, \mathit{final}, \mathit{single}\ \mathit{prop_spec})\\
&\mathit{prop_spec}\ [\![\mathbf{e}\ \mathit{PS}_1\ \mathbf{u}\ \mathit{PS}_2]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec}_1 = \mathit{prop_spec}\ \mathit{PS}_1\ \mathit{subcomp_table}\ \mathbf{in}\\
&\quad \mathbf{let}\ \mathit{prop_spec}_2 = \mathit{prop_spec}\ \mathit{PS}_2\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{exists}, \mathit{until}, \mathit{double}\ (\mathit{prop_spec}_1, \mathit{prop_spec}_2))\\
&\mathit{prop_spec}\ [\![\mathbf{e}\ \mathit{PS}_1\ \mathbf{w}\ \mathit{PS}_2]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec}_1 = \mathit{prop_spec}\ \mathit{PS}_1\ \mathit{subcomp_table}\ \mathbf{in}\\
&\quad \mathbf{let}\ \mathit{prop_spec}_2 = \mathit{prop_spec}\ \mathit{PS}_2\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{exists}, \mathit{weak}, \mathit{double}\ (\mathit{prop_spec}_1, \mathit{prop_spec}_2))\\
&\mathit{prop_spec}\ [\![\mathbf{e}\ \mathit{PS}_1\ \mathbf{r}\ \mathit{PS}_2]\!]\ \mathit{subcomp_table} =\\
&\quad \mathbf{let}\ \mathit{prop_spec}_1 = \mathit{prop_spec}\ \mathit{PS}_1\ \mathit{subcomp_table}\ \mathbf{in}\\
&\quad \mathbf{let}\ \mathit{prop_spec}_2 = \mathit{prop_spec}\ \mathit{PS}_2\ \mathit{subcomp_table}\ \mathbf{in}\\
&\qquad \mathit{tree_prop_spec}\ (\mathit{exists}, \mathit{release}, \mathit{double}\ (\mathit{prop_spec}_1, \mathit{prop_spec}_2))
\end{aligned}$$

```
$\mathit{prop\_spec}\ [\![\mathit{NPS}]\!]\ \mathit{subcomp\_table} =$
    $\mathit{node\_prop\_spec}\ \mathit{NPS}\ \mathit{subcomp\_table}$
```

The node specification only affects the value of the node. On the whole, it is similar to the regular expression. However, it can also be made up by a constant value, state variable, or state. As the former two cases are syntactically equal, the *prop_spec_value* semantic rule uses the symbol table to distinguish them.

```
$\mathit{node\_prop\_spec} : \mathrm{NodePropSpecS} \times \mathrm{Table} \rightarrow \mathrm{PropSpec}$
$\mathit{node\_prop\_spec}\ [\![\texttt{(}\ \mathit{PS}\ \texttt{)}]\!]\ \mathit{subcomp\_table} =$
    $\mathit{prop\_spec}\ \mathit{PS}\ \mathit{subcomp\_table}$
$\mathit{node\_prop\_spec}\ [\![\mathbf{not}\ \mathit{PS}]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec} = \mathit{prop\_spec}\ \mathit{PS}\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_node\_prop\_spec}\ (\mathit{not}\ \mathit{prop\_spec})$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \mathbf{and}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{and}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \mathbf{or}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{or}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \texttt{=}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{eq}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \texttt{!=}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{ne}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \texttt{<}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{lt}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \texttt{<=}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{le}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \texttt{>}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{gt}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \texttt{>=}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{ge}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \texttt{+}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{add}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
$\mathit{node\_prop\_spec}\ [\![\mathit{PS}_1\ \texttt{-}\ \mathit{PS}_2]\!]\ \mathit{subcomp\_table} =$
    $\mathbf{let}\ \mathit{prop\_spec}_1 = \mathit{prop\_spec}\ \mathit{PS}_1\ \mathit{subcomp\_table}\ \mathbf{in}$
    $\mathbf{let}\ \mathit{prop\_spec}_2 = \mathit{prop\_spec}\ \mathit{PS}_2\ \mathit{subcomp\_table}\ \mathbf{in}$
        $\mathit{node\_prop\_spec}\ (\mathit{sub}\ (\mathit{prop\_spec}_1,\ \mathit{prop\_spec}_2))$
```

```
node_prop_spec $[\![PS_1 * PS_2]\!]$ subcomp_table =
    let prop_spec$_1$ = prop_spec $PS_1$ subcomp_table in
    let prop_spec$_2$ = prop_spec $PS_2$ subcomp_table in
        node_prop_spec (mul (prop_spec$_1$, prop_spec$_2$))
node_prop_spec $[\![PS_1 / PS_2]\!]$ subcomp_table =
    let prop_spec$_1$ = prop_spec $PS_1$ subcomp_table in
    let prop_spec$_2$ = prop_spec $PS_2$ subcomp_table in
        node_prop_spec (div (prop_spec$_1$, prop_spec$_2$)) node_prop_spec $[\![I_1 . I_2]\!]$ subcomp_table =
    let inst_record = table_get $I_1$ subcomp_table in
    let inst_index = list_index_of inst_record (table_to_list subcomp_table) in
    let system (state, symbol_table, init_list, trans_list) = inst_record in
    let value = table_get $I_2$ subcomp_table in
        node_prop_spec (prop_spec_value inst_index value)
```

```
prop_spec_value : Integer × Value → NodeProp
prop_spec_value inst_index (state state_value) =
    state (inst_index, state_value)
prop_spec_value inst_index (integer int_value) =
    variable (inst_index, int_value)
```

4.4 Initialization

Before the process of generating the state tree begins, the subcomponent table must become initialized; that is, the assignments state variables and the triggering of outport signals in the initialization part of the annexes must be performed. The *initalize_subcomponent_list* semantic rule traverses the global subcomponent list and, for each subcomponent, calls *initalize_subcomponent* that initializes the subcomponent by traversing the initialization list. The parameter of the semantic rule is the global subcomponent list and the return value is the same list with the symbol table of each subcomponent updated in accordance to its initialization list. The *list_insert* list function adds the new value at the beginning of the list (in contrast to the *list_add* list function that adds the value at the end of the list).

```
initalize_subcomponent_list : List → List
initalize_subcomponent_list inst_list =
    if not (list_is_empty inst_list) then
        let (subcomponent$_1$, tail$_1$) = list_split inst_list in
        let subcomponent$_2$ = initalize_subcomponent subcomponent in
        let tail$_2$ = initalize_subcomponent_list tail$_1$ in
            list_insert subcomponent$_2$ tail$_2$
    else list_empty
```

```
initialize_subcomponent : Value → Value
initialize_subcomponent record subcomp_table =
    let system (state, symbol_table$_1$, init_list, trans_list) = record in
    let symbol_table$_2$ = traverse_action_list init_list symbol_table$_1$ in
        system (state, symbol_table$_2$, init_list, trans_list)
```

The *traverse_action_list* semantic rule traverses the action list and, for each action in the list, calls the *execute_action* semantic rule that executes the action and updates the symbol table. When the complete list is traversed, the resulting symbol table is returned.

```
traverse_action_list : List × Table → Table
traverse_action_list action_list symbol_table$_1$ =
    if not (list_is_empty action_list) then
        let (action, tail) = list_split action_list in
        let symbol_table$_2$ = execute_action action symbol_table$_1$ in
            traverse_action_list tail symbol_table$_2$
    else symbol_table$_1$
```

The *evaluate_expression* semantic rule does not only return the value of the expression, it does also return a new symbol table as the expression may include the reception of an input port signal. In that case the boolean value of the signal is set to **false** and the resulting symbol table is returned. Note that state variables as well as input and output port signals are stored in the same symbol table.

```
execute_action : Value × Table → Table
execute_action (action (send I)) symbol_table =
    table_set I (boolean true) symbol_table
execute_action (action (assign (I, E))) symbol_table$_1$ =
    let (value, symbol_table$_2$) = evaluate_expression E symbol_table$_1$ in
        table_set I value symbol_table$_2$
```

4.5 Expression Evaluation

The *evaluate_expression* semantic rule evaluates an expression. Besides the expression, it also takes a symbol table as parameter. It returns the value together with the result symbol table. The resulting symbol table is different from the original table if the expression includes the reception of an input port. In that case, its boolean value become changed from **true** to **false** in the new symbol table, since the input signal should be read only once.

```
evaluate_expression : Expression × Table → (Value × Table)
evaluate_expression (value V) symbol_table =
    (V, symbol_table)
evaluate_expression (identifier I) symbol_table =
    (table_get I symbol_table, symbol_table)
evaluate_expression (receive I) symbol_table =
    (table_get I symbol_table, table_set I (boolean false) symbol_table)
evaluate_expression (not E) symbol_table =
    let (boolean bool_value, symbol_table$_1$) =
        evaluate_expression E$_1$ symbol_table in
        (boolean (not bool_value), symbol_table$_1$)
evaluate_expression (and (E$_1$, E$_2$))) symbol_table =
    let (boolean bool_value$_1$, symbol_table$_1$) =
        evaluate_expression E$_1$ symbol_table in
    let (boolean bool_value$_2$, symbol_table$_2$) =
        evaluate_expression E$_2$ symbol_table$_1$ in
        (boolean (bool_value$_1$ and bool_value$_2$), symbol_table$_2$)
evaluate_expression (or (E$_1$, E$_2$))) symbol_table =
    let (boolean bool_value$_1$, symbol_table$_1$) =
        evaluate_expression E$_1$ symbol_table in
    let (boolean bool_value$_2$, symbol_table$_2$) =
        evaluate_expression E$_2$ symbol_table$_1$ in
        (boolean (bool_value$_1$ or bool_value$_2$), symbol_table$_2$)
```

```
evaluate_expression (eq ($E_1$, $E_2$))) symbol_table =
    let (value$_1$, symbol_table$_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (value$_2$, symbol_table$_2$) =
        evaluate_expression $E_2$ symbol_table$_1$ in
        (evaluate_equal value$_1$ value$_2$, symbol_table$_2$)
evaluate_expression (ne ($E_1$, $E_2$))) symbol_table =
    let (value$_1$, symbol_table$_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (value$_2$, symbol_table$_2$) =
        evaluate_expression $E_2$ symbol_table$_1$ in
        (evaluate_not_equal value$_1$ value$_2$, symbol_table$_2$)
evaluate_expression (lt ($E_1$, $E_2$))) symbol_table =
    let (integer int_value$_1$, symbol_table$_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (integer int_value$_2$, symbol_table$_2$) =
        evaluate_expression $E_2$ symbol_table$_1$ in
        (integer (int_value$_1$ < int_value$_2$), symbol_table$_2$)
evaluate_expression (le ($E_1$, $E_2$))) symbol_table =
    let (integer int_value$_1$, symbol_table$_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (integer int_value$_2$, symbol_table$_2$) =
        evaluate_expression $E_2$ symbol_table$_1$ in
        (integer (int_value$_1$ <= int_value$_2$), symbol_table$_2$)
evaluate_expression (gt ($E_1$, $E_2$))) symbol_table =
    let (integer int_value$_1$, symbol_table$_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (integer int_value$_2$, symbol_table$_2$) =
        evaluate_expression $E_2$ symbol_table$_1$ in
        (integer (int_value$_1$ > int_value$_2$), symbol_table$_2$)
evaluate_expression (ge ($E_1$, $E_2$))) symbol_table =
    let (integer int_value$_1$, symbol_table$_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (integer int_value$_2$, symbol_table$_2$) =
        evaluate_expression $E_2$ symbol_table$_1$ in
        (integer (int_value$_1$ >= int_value$_2$), symbol_table$_2$)
evaluate_expression (add ($E_1$, $E_2$))) symbol_table =
    let (integer int_value$_1$, symbol_table$_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (integer int_value$_2$, symbol_table$_2$) =
        evaluate_expression $E_2$ symbol_table$_1$ in
        (integer (int_value$_1$ + int_value$_2$), symbol_table$_2$)
evaluate_expression (sub ($E_1$, $E_2$))) symbol_table =
    let (integer int_value$_1$, symbol_table$_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (integer int_value$_2$, symbol_table$_2$) =
        evaluate_expression $E_2$ symbol_table$_1$ in
        (integer (int_value$_1$ − int_value$_2$), symbol_table$_2$)
```

```
evaluate_expression (mul ($E_1$, $E_2$))) symbol_table =
    let (integer $int\_value_1$, $symbol\_table_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (integer $int\_value_2$, $symbol\_table_2$) =
        evaluate_expression $E_2$ $symbol\_table_1$ in
        (integer ($int\_value_1 * int\_value_2$), $symbol\_table_2$)
evaluate_expression (div ($E_1$, $E_2$))) symbol_table =
    let (integer $int\_value_1$, $symbol\_table_1$) =
        evaluate_expression $E_1$ symbol_table in
    let (integer $int\_value_2$, $symbol\_table_2$) =
        evaluate_expression $E_2$ $symbol\_table_1$ in
        (integer ($int\_value_1$ / $int\_value_2$), $symbol\_table_2$)
```

The *eq* and *ne* operators above accept that both operands are either integer and boolean, why we need to call the *evaluate_equal* and *evaluate_not_equal*, respectively. The rest of the relational operates only accept integer values, and so do the arithmetic operators.

```
evaluate_equal : Value × Value → Value
evaluate_equal (integer $int\_value_1$) (integer $int\_value_2$) =
    boolean ($int\_value_1 = int\_value_2$)
evaluate_equal (boolean $bool\_value_1$) (boolean $bool\_value_2$) =
    boolean ($bool\_value_1 = bool\_value_2$)
evaluate_not_equal : Value × Value → Value
evaluate_not_equal (integer $int\_value_1$) (integer $int\_value_2$) =
    boolean ($int\_value_1 \; != int\_value_2$)
evaluate_not_equal (boolean $bool\_value_1$) (boolean $bool\_value_2$) =
    boolean ($bool\_value_1 \; != bool\_value_2$)
```

4.6 Connection

The *traverse_connection_list* semantic rule traverses the connection list and, for each connection, calls *execute_connection* that execute the connection.

```
traverse_connection_list : List × List → List
traverse_connection_list conn_list subcomp_list =
    if not (list_is_empty conn_list) then
        let ($connection_1$, $tail_1$) = list_split conn_list in
        let $connection_2$ = execute_connection $connection_1$ subcomp_list
        let $tail_2$ = traverse_connection_list $tail_1$ $connection_2$
            traverse_connection_list $tail_2$ $connection_2$
    else subcomp_list
```

In the *execute_connection* semantic rule, we first look up the subcomponent of the output port and the output port boolean value. If it is **true**, we set its value to **false** and look up the subcomponent of the input port and set the its value to **true**. Since the value of the ports has been altered in both the symbol tables of the output and input port subcomponents, we need to set the new subcomponent values in the subcomponent list (*subcomp_list*).

```
execute_connection : Value × List → List
execute_connection (conn (out_index, out_ident, in_index, in_ident))
    subcomp_list$_1$ =
    let out_subcomponent$_1$ = list_get out_index subcomp_list$_1$ in
    let system (out_state, out_table$_1$, out_init_list, out_trans_list) =
        out_subcomponent$_1$ in
    if is_true (table_get out_ident out_table$_1$) then
        let out_table$_2$ = table_set out_ident (boolean false) out_table$_1$ in
        let out_subcomponent$_2$ =
            system (out_state, out_table$_2$, out_init_list, out_trans_list) in
        let subcomp_list$_2$ =
            list_set out_index out_subcomponent$_2$ subcomp_list$_1$ in
        let in_subcomponent$_1$ = list_get in_index subcomp_list$_1$ in
        let system (in_state, in_table$_1$, in_init_list, in_trans_list) =
            in_subcomponent$_1$ in
        let in_table$_2$ = table_set in_ident (boolean true) in_table$_1$ in
        let in_subcomponent$_2$ =
            (in_state, in_table$_2$, in_init_list, in_trans_list) in
            list_set in_index in_subcomponent$_2$ subcomp_list$_2$
    else subcomp_list
```

4.7 Generation

In this section, we generate the state tree. What makes it somewhat complicated is that we construct the tree recursively: *generate_tree* calls *traverse_subcomponent_list*, which calls *traverse_trans_list*, which calls *execute_transition*, which finally calls *generate_tree* recursively. Each possible transition in each subcomponent generates a new sub tree. If several transitions can be taken (in the same subcomponent or in a different one), one sub tree for each transition will be constructed. The idea is that from the start, the tree is made up of one single node that holds the subcomponent list in its initial state. Then *traverse_subcomponent_list* goes through the subcomponents and for each subcomponent goes *traverse_trans_list* through each transition. For each transition that can be taken, *execute_transition* updates the subcomponent list so that the transition is taken and create a new sub tree with that list as root value, and attach that sub tree as a child to the parameter main tree. Then it finally calls *generate_tree* which recursively continues to create sub trees until no more transitions can be taken or the subcomponent list already has been attached to a ancestor node.

The *generate_tree* semantic rule first traverses the connection list by calling *traverse_connection_list* in order to establish any current connections, which results in an updated subcomponent list. Then it traverses the subcomponent list which generates a sub tree that is attached to the parameter main tree. However, one of its parameter is a set stored with subcomponent lists that keeps track of the subcomponents. If the same list reoccur (that is, one of the ancestor nodes holds the same subcomponent list), the generation is aborted.

```
generate_tree : List × List × Set × Tree → Tree
generate_tree subcomp_list conn_list $set_1$ main_tree =
    if not (set_exists subcomp_list $set_1$) then
        let $set_2$ = set_add subcomp_list $set_1$ in
        let $sub\_tree_1$ = tree_create subcomp_list in
        let $subcomp\_list_2$ = traverse_connection_list conn_list subcomp_list
        in let $sub\_tree_2$ = traverse_subcomponent_list $subcomp\_list_2$
                                   conn_list $set_2$ $sub\_tree_1$ in
            tree_add_child $sub\_tree_2$ main_tree
    else main_tree
```

The *traverse_subcomponent_list* semantic rule traverses the subcomponent list and, for each subcomponent, traverse the transition list. As the traversing of the transitions list may affect the state and the symbol table of the subcomponent, it need to be stored in the subcomponent list. In this case, the complete subcomponent list is kept since we need it in the recursive call to *execute_transition* below. Instead of splitting the list as in similar cases above, we increase the index of the list. Each call to *traverse_trans_list* generates a new tree that becomes the parameter to the next call to *traverse_subcomponent_list.* When the list is is completely traversed, the resulting tree of the last call to *traverse_subcomponent_list* is returned.

```
traverse_subcomponent_list : Integer × List × List × Set × Tree → Tree
traverse_subcomponent_list inst_index subcomp_list conn_list set $tree_1$ =
    if inst_index < (list_size subcomp_list) then
        let system (state, symbol_table, init_list, trans_list) =
            list_get inst_index subcomp_list in
        let $tree_2$ = traverse_trans_list trans_list inst_index subcomp_list
            conn_list set $tree_1$ in
            traverse_subcomponent_list (inst_index + 1) subcomp_list
            conn_list set $tree_2$
    else $tree_1$
```

The *traverse_trans_list* semantic rule traverses the transition list, and for each transition calls *execute_transition* that returns a new tree, which in turn becomes the parameter tree in the next call to *traverse_trans_list.* When the list is completely traversed, it returns the resulting tree from the last call to *execute_transition.*

```
traverse_trans_list : List × Integer × List × List × Set × Tree → Tree
traverse_trans_list trans_list inst_index subcomp_list conn_list set $tree_1$ =
    if (list_size trans_list) > 0 then
        let (head, tail) = list_split trans_list in
        let $tree_2$ = execute_transition head inst_index subcomp_list
            conn_list set $tree_1$ in
            traverse_trans_list tail inst_index subcomp_list conn_list set $tree_2$
    else $tree_1$
```

The *execute_transition* semantic rule executes a transition. If the current state of the subcomponent is equal to the source state of the transition and the guard expression evaluates to **true**, the transition is taken and the subcomponent list is updated. As each taken transition generates a new sub tree, *generate_tree* is called with the updated subcomponent list.

```
execute_transition : Value × Integer × List × List × Set × Tree → Tree
execute_transition trans_value inst_index subcomp_list1 conn_list set
    main_tree =
    let transition (source_state, guard_expr, target_state, action_list) =
        trans_value in
    let record1 = table_get inst_index subcomp_list in
    let system (state, symbol_table1, init_list, trans_list) = record1 in
    let (boolean is_guard, symbol_table2) =
        evaluate guard_expr symbol_table1 in
        if (state = sourceState) and is_guard then
            let symbol_table3 = traverse_action_list init_list symbol_table2 in
            let record2 =
                system (target_state, symbol_table3, init_list, trans_list)
            let subcomp_list2 = list_set inst_index record2 subcomp_list1 in
                generate_tree subcomp_list2 conn_list set main_tree
        else main_tree
```

4.8 Property Specification Evaluation

When the state tree of section 4.7 has become generated, we are finally ready to evaluate it against the property specification created in section 4.3. First, we define the two auxiliary semantic rules *is_true* and *is_false* that decide whether a boolean value is **true** or **false**, respectively.

```
is_true : Value → Boolean
is_true (boolean B) = B

is_false : Value → Boolean
is_false (boolean B) = not B
```

The *evaluate_prop_spec* semantic rule evaluate a tree or node property.

```
evaluate_prop_spec : PropSpec × Tree → Value
evaluate_prop_spec (tree_prop_spec (width_op, depth_op, T)) tree =
    boolean_value (evaluate_tree T tree width_op depth_op)
evaluate_prop_spec (node_prop_spec N) tree =
    evaluate_node N tree
```

The *evaluate_children* and *evaluate_tree* semantic functions call each other alternately. Initially, *evaluate_tree* is called for the root node, it calls *evaluate_children* for its children, which in turn calls *evaluate_tree* for each of the children. These alternately calls continue until the property specification has been satisfied or a leaf in the tree has been reached.

The *evaluate_children* traverses the children of the root node of a tree. If there is no children, we have reached a leaf of the tree. Different values are returned depending of the *depth* operator. In case of the *global* operator, the property has to hold for each node on the path from the root node to the leaf. Therefore, the **and** operator is applied to the node property values, and **true** is returned at the end of the path. In case of *final* operator, it is enough that one property holds for the path from the root node to the leaf node. Therefore, the **or** operator is applied to the node property values and **false** is returned at the end of the path. In case of the *until* operator, the property has to hold for at least one node on the path, which finally gives **false** as return value. On the other hand, in case of the *weak* operator, it does not need to be the case; therefore, **true** is returned. The same goes for the *release* operator, **true** is returned in that case as well.

If the root node of the tree has one child, we simple evaluate it by calling *evaluate_tree*. If it has more than one children, we need to look into the *width* operators. In case of the *all* operator, the property has to hold for all child nodes, why we apply the **and** operator between the property value of the first child node and the evaluation of the rest of the children. In case of the *exists* operator, the property has to hold for only one of the children, why we instead apply the **or** operator.

```
evaluate_children : TreeProp × List × WidthOp × DepthOp → Boolean
evaluate_children TP child_list width_op depth_op =
    case (list_size child_list) of
        0 ⇒ case depth_op of
                  global ⇒ true
                | final ⇒ false
                | until ⇒ false
                | weak ⇒ true
                | release ⇒ true
      | 1 ⇒ evaluate_tree TP (list_get 0 child_list) width_op depth_op
      | default ⇒ let (head, tail) = list_split child_list in
          case width_op of
                all ⇒ (evaluate_tree TP head width_op depth_op) and
                      (evaluate_children TP tail width_op depth_op)
              | exists ⇒ (evaluate_tree TP head width_op depth_op) or
                      (evaluate_children TP tail width_op depth_op)
```

The *evaluate_tree* semantic rule evaluates the property of the root node of the tree and compare it against the children. In case of the *global* operator, the property has to hold for the root node and all the nodes to the leaf nodes. In case of the *final* operator, it is enough if the property holds for one of them. In case of the *until*, *weak*, and *release* operator, there are two properties to consider. In both the *until* and *weak* cases, either the second property has to hold for the root node; if it does not, the first property has to hold for the root node and the operators has to hold for the rest of the children. However, they differ at the end of the path, see the *evaluate_children* semantic rule above. Finally, in case of the *release* operator, the second operator has to hold for the root node; if it does not, either the first property has to hold for the root node or the operator has to hold for the rest of the children.

```
evaluate_tree : TreeProp × Tree × WidthOp × DepthOp → Boolean
evaluate_tree PS tree width_op depth_op =
    case depth_op of
        global ⇒ let (single subProp) = PS in
            (is_true (evaluate_prop_spec subProp tree)) and
            (evaluate_children PS (tree_get_children tree) width_op depth_op)
      | final ⇒ let (single subProp) = PS in
            (is_true (evaluate_prop_spec subProp tree)) or
            (evaluate_children PS (tree_get_children tree) width_op depth_op)
      | until ⇒ let (double ($subProp_1$, $subProp_2$)) = PS in
            (is_true (evaluate_prop_spec $subProp_2$ tree)) or
            ((is_true (evaluate_prop_spec $subProp_1$ tree)) and
            (evaluate_children PS (tree_get_children tree) width_op depth_op))
      | weak ⇒ let (double ($subProp_1$, $subProp_2$)) = PS in
            (is_true (evaluate_prop_spec $subProp_2$ tree)) or
            ((is_true (evaluate_prop_spec $subProp_1$ tree)) and
            (evaluate_children PS (tree_get_children tree) width_op depth_op))
```

```
        | release ⇒ let (double ($subProp_1$, $subProp_2$)) = PS in
            (is_true (evaluate_prop_spec $subProp_2$ tree)) and
            ((is_true (evaluate_prop_spec $subProp_1$ tree)) or
            (evaluate_children PS (tree_get_children tree) width_op depth_op))
```

The *evaluate_node* semantic rule is relative simple. It is similar to the *evaluate_expression* semantic rule in section 4.5 and evaluates the node property.

```
evaluate_node : NodeProp × Tree → Value
evaluate_node (not_prop_spec PS) tree =
    let boolean bool_value = evaluate_prop_spec PS tree in
        boolean (not bool_value)
evaluate_node (and_prop_spec ($PS_1$, $PS_2$)) tree =
    let boolean $bool\_value_1$ = evaluate_prop_spec $PS_1$ tree in
    let boolean $bool\_value_2$ = evaluate_prop_spec $PS_2$ tree in
        boolean ($bool\_value_1$ and $bool\_value_2$)
evaluate_node (or_prop_spec ($PS_1$, $PS_2$)) tree =
    let boolean $bool\_value_1$ = evaluate_prop_spec $PS_1$ tree in
    let boolean $bool\_value_2$ = evaluate_prop_spec $PS_2$ tree in
        boolean ($bool\_value_1$ or $bool\_value_2$)
evaluate_node (eq_prop_spec ($PS_1$, $PS_2$)) tree =
    let $value_1$ = evaluate_prop_spec $PS_1$ tree in
    let $value_2$ = evaluate_prop_spec $PS_2$ tree in
        evaluate_equal $value_1$ $value_2$
evaluate_node (ne_prop_spec ($PS_1$, $PS_2$)) tree =
    let $value_1$ = evaluate_prop_spec $PS_1$ tree in
    let $value_2$ = evaluate_prop_spec $PS_2$ tree in
        evaluate_not_equal $value_1$ $value_2$
evaluate_node (lt_prop_spec ($PS_1$, $PS_2$)) tree =
    let integer $int\_value_1$ = evaluate_prop_spec $PS_1$ tree in
    let integer $int\_value_2$ = evaluate_prop_spec $PS_2$ tree in
        integer ($int\_value_1$ < $int\_value_2$)
evaluate_node (le_prop_spec ($PS_1$, $PS_2$)) tree =
    let integer $int\_value_1$ = evaluate_prop_spec $PS_1$ tree in
    let integer $int\_value_2$ = evaluate_prop_spec $PS_2$ tree in
        integer ($int\_value_1$ <= $int\_value_2$)
evaluate_node (gt_prop_spec ($PS_1$, $PS_2$)) tree =
    let integer $int\_value_1$ = evaluate_prop_spec $PS_1$ tree in
    let integer $int\_value_2$ = evaluate_prop_spec $PS_2$ tree in
        integer ($int\_value_1$ > $int\_value_2$)
evaluate_node (ge_prop_spec ($PS_1$, $PS_2$)) tree =
    let integer $int\_value_1$ = evaluate_prop_spec $PS_1$ tree in
    let integer $int\_value_2$ = evaluate_prop_spec $PS_2$ tree in
        integer ($int\_value_1$ >= $int\_value_2$)
evaluate_node (add_prop_spec ($PS_1$, $PS_2$)) tree =
    let integer $int\_value_1$ = evaluate_prop_spec $PS_1$ tree in
    let integer $int\_value_2$ = evaluate_prop_spec $PS_2$ tree in
        integer ($int\_value_1$ + $int\_value_2$)
evaluate_node (sub_prop_spec ($PS_1$, $PS_2$)) tree =
    let integer $int\_value_1$ = evaluate_prop_spec $PS_1$ tree in
    let integer $int\_value_2$ = evaluate_prop_spec $PS_2$ tree in
        integer ($int\_value_1$ − $int\_value_2$)
```

```
evaluate_node (mul_prop_spec ($PS_1$, $PS_2$)) tree =
    let integer int_value$_1$ = evaluate_prop_spec $PS_1$ tree in
    let integer int_value$_2$ = evaluate_prop_spec $PS_2$ tree in
        integer (int_value$_1$ * int_value$_2$)
evaluate_node (div_prop_spec ($PS_1$, $PS_2$)) tree =
    let integer int_value$_1$ = evaluate_prop_spec $PS_1$ tree in
    let integer int_value$_2$ = evaluate_prop_spec $PS_2$ tree in
        integer (int_value$_1$ / int_value$_2$)
evaluate_node (state_prop S) tree =
    evaluate_state S tree
evaluate_node (variable_prop V) tree =
    evaluate_variable V tree
evaluate_node (value_prop V) tree =
    V
```

The *evaluate_state* semantic rule looks up the subcomponent and the state's integer value and compares it to the parameter stat value.

```
evaluate_state : (Integer × Integer) × Tree → Value
evaluate_state (inst_index, state_value) tree =
    let subcomp_list = tree_get_value tree in
    let record = list_get inst_index subcomp_list in
    let system (state, symbol_table, init_list, trans_list) = record in
        boolean_value (state = state_value)
```

The *evaluate_variable* semantic rule looks up the subcomponent and the state variable's integer value.

```
evaluate_variable : (Integer × Identifier) × Tree → Value
evaluate_variable (inst_index, var_ident) tree =
    let subcomp_list = tree_get_value tree in
    let record = list_get inst_index subcomp_list in
    let system (state, symbol_table, init_list, trans_list) = record in
        table_get var_ident symbol_table
```

Chapter 5

Tool Support and Case Study

In order to proof the correctness of the semantics of chapter 4, we have developed a tool that is an Eclipse[1] plug-in on top of the OSATE Framework[2], which in itself is an Eclipse plug-in, see figure 5.1. It evaluates a property specification on an AADL model with behavior annexes.

The model and property specification are translated into standard ML format, the translator is written in Java[3], CUP[4], and JLex[5]. We have tested our tool on the Production Cell case study in chapter 5.2.

The semantics itself is implemented in Standard ML[6] and is made up of seven source code files:

- **Utilities.ml**. Definition of basic functions, trace printing and abstract data types, such as Map, List, Tree, and Set.
- **Storage.ml**. Definitions of values and the Storage abstract data type.
- **Parser.ml**. Parses and semantically checks the AADL with Behavior Annex input model. The model needs to be translated into ML source code, see figure 5.3 and section 5.1.
- **Evaluator.ml**. The evaluation of expressions.
- **Initializor.ml**. The initialization of each subcomponent.
- **PropSpec.ml**. The parsing and evaluation of a property specification against the state tree.
- **Generator.ml**. The generation of the state tree.

The first part of the tool is a parser written in CUP and JLex that translates the AADL with Behavior Annex model into a format readable to Standard ML, see section 5.1.

The tool needs three directory paths: the *Standard ML Path*, *Semantics Path*, and the *Temporary Path*, see figure 5.2. In the temporary path directory, four files are generated:

---

[1] www.eclipse.org

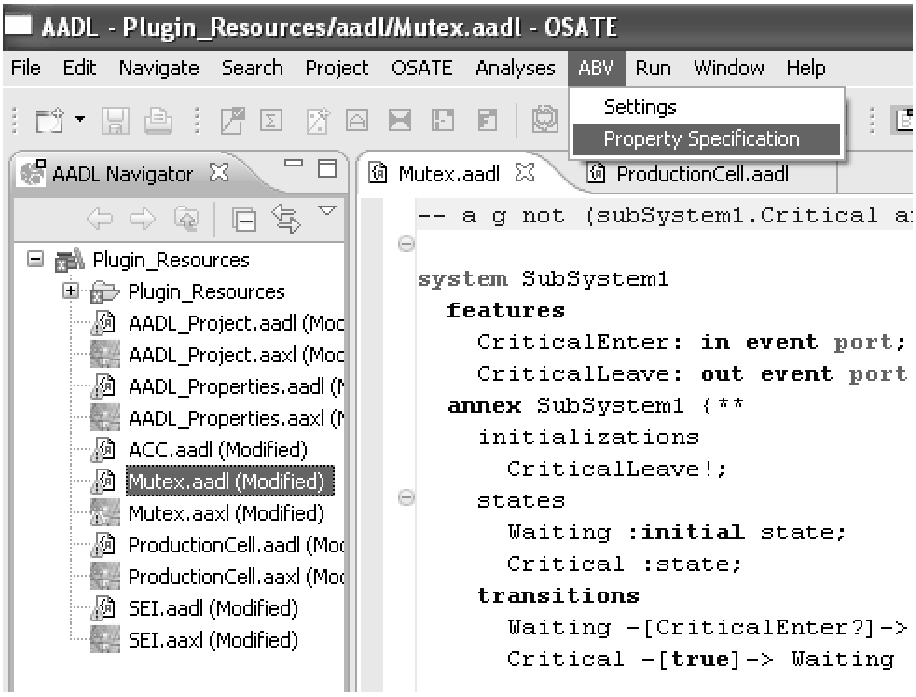
[2] www.aadl.info/aadl/currentsite/tool/osate.html

[3] www.oracle.com/technetwork/java/index.html

[4] http://www2.cs.tum.edu/projects/cup/

[5] www.cs.princeton.edu/ appel/modern/java/JLex

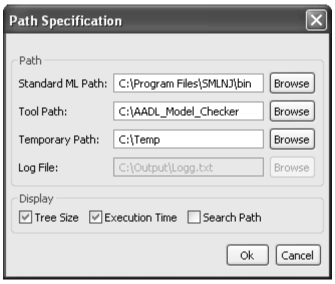
[6] www.smlnj.org



```
-- a g not (subSystem1.Critical a

system SubSystem1
  features
    CriticalEnter: in event port;
    CriticalLeave: out event port
  annex SubSystem1 {**
    initializations
      CriticalLeave!;
    states
      Waiting :initial state;
      Critical :state;
    transitions
      Waiting -[CriticalEnter?]->
      Critical -[true]-> Waiting
```

Figure 5.1: The Semantics Tool.



Standard ML Path: C:\Program Files\SMLNJ\bin
Tool Path: C:\AADL_Model_Checker
Temporary Path: C:\Temp
Log File: C:\Output\Logg.txt

☑ Tree Size ☑ Execution Time ☐ Search Path

Figure 5.2: Path Specification.

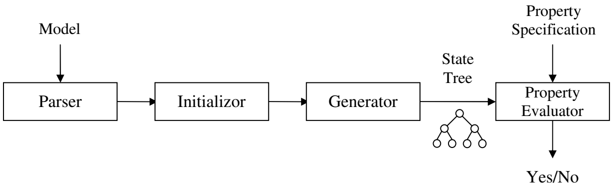


Figure 5.3: The Semantics Tool Modules.

- **Input.ml**. The AADL with Behavior Annex model as well as the property specification translated into ML by the parser mentioned above.
- **Semantics.bat**. The tool starts a subprocess executing this batch file. With the settings of figure 5.2, it looks like the following:

```
"C:\Program Files\SMLNJ\bin\sml" < "C:\Temp\Semantics.ml"
                                 > "C:\Temp\Semantics.log"
```

- **Semantics.ml**. The ML source file that do the actually work. It includes the ML files above together with the ML versions of the AADL with Behavior Annex model and the property specification. Finally, it starts the semantic evaluation by calling the main function. With the settings of figure 5.2, it will looks like the following:

```
use "C:\\Semantics\\Utilities.ml";
use "C:\\Semantics\\Storage.ml";
use "C:\\Semantics\\Parser.ml";
use "C:\\Semantics\\Evaluator.ml";
use "C:\\Semantics\\Initializor.ml";
use "C:\\Semantics\\PropSpec.ml";
use "C:\\Semantics\\Generator.ml";
use "C:\\Temp\\Input.ml";
ParseSpecification Model PropSpec;
```

- **Semantics.log**. The output of the execution of the ML files are logged, read and interperted by the tool.

Similar to the semantics of chapter 4, the tool works in two steps: first the state tree is generated holding each possible model state, then the tree is evaluated against the CTL property specification. See figure 5.3.

5.1 Input Language

Since the semantics is implemented in ML, its input needs to be translated into ML format. Therefore, a parser has been developed. It accepts an AADL with Behavior Annex model or a CTL property specification. In both cases, a corresponding ML syntax tree is generated. For instance, the property specification in section 2.1.2 is translated into the following ML value. The value holding the property specification is always named *PropSpec*. In case of an AADL with Behavior Annex model, the name is always *Model*.

```
val PropSpec = (tree_parse (all, global, (single_parse (node_parse
(not_parse (node_parse (and_parse ((node_parse (ident_parse
("subsystem1", "Critical"))), (node_parse (ident_parse
("subsystem2", "Critical"))))))))));
```

However, note that the parser is just a parser, it only translates the code from one format to another. All type checking is done by the ML semantics implementation described in the previous section.

5.2 Case Study

In order to validate the approach of this report, a case study of a Production Cell system has been constructed in AADL, its source code is given in appendix B. It is based on an automated manufacturing system which is modeled on an industrial plant in Karlsruhe in Germany. It was first described by Lewerentz in [18]. Ouimet defined it in TASM in [21] as depicted in Figure 5.4.

The system is not controlled by a central unit. Instead, the components communicates with each other through port connections. The components work concurrently, when a component is ready to accept a new block it notifies the preceding component, which in turn acknowledges that is has loaded the block. There is also a signal acknowledging that the loading location of the component is free.

The system is composed of the robot arms *Loader*, *BeltToPress*, *PressToPress*, the conveyer belts *FeedBelt* and *DepositBelt* as well as the *Press*. The system input is a set of blocks arriving in a crate and the output is the same blocks with bolts attached to them delivered by the deposit belt for further processing. Once a block has been loaded, [illegible] iption.

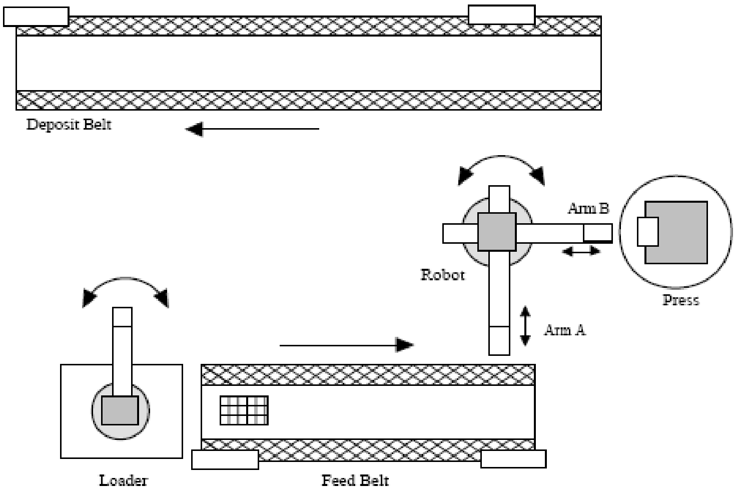


Figure 5.4: The Production Cell System as presented in [21].

While waiting, the loader is parked at the crate with the magnet turned off. When it receives a signal from the feed belt that it is ready to receive a new block, the loader turns the magnet on, moves the arm onto the beginning of the feed belt, turns the magnet off, and signals the feed belt that it has loaded a block.

When the feed belt receives the loading signal, it starts the belt so the block moves towards the end point. When the block has reached the end point, the belt is turned off and a signal is sent backwards to the loader robot arm that the block has been picked up. Then the robot arm moves back to the crate and waits for the next ready signal from the feed belt. The feed belt also sends a signal forwards to the belt-to-press robot arm that the block is ready to be picked up. Then it waits until the loader places a new block on the belt.

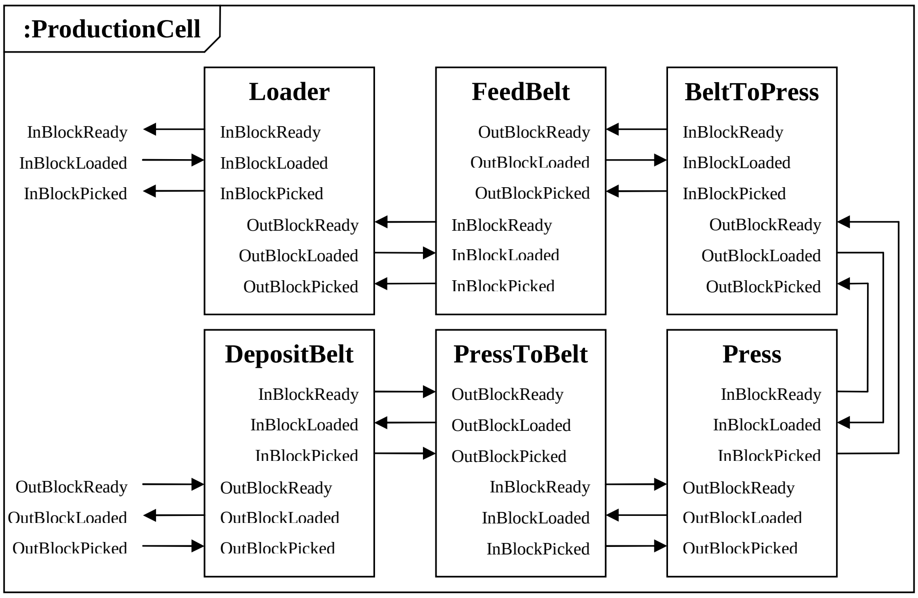


Figure 5.5: Schematic Description of the Production Cell System.

When the belt-to-press robot arm receives the loading signal from the feed belt, it waits for the ready signal from the press. When it receives it, the robot arm turns on the magnet and picks up the block, sends the picked-up signal backwards to the feed belt, moves the block and drops it on the input plate of the press. When it receives the picked-up signal from the press, it moves the arm back to the feed belt and waits for the next loaded signal.

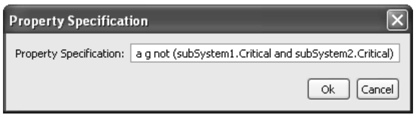
The press moves the block into the press position, sends the picked-up signal backwards to the belt-to-press robot arm and starts pressing a bolt into the block. When the bolt pressing process is finished, the block is moved to the departure position, the ready signal is sent backwards to the belt-to-press robot arm and the loaded signal is sent forwards to the press-to-belt robot arm.

When the press-to-belt robot arm receives the loaded signal, it waits for the ready signal from the deposit belt. When it receives it, it picks up the block by turning on the magnet, moves the block onto the deposit belt, turn off the magnet and sends the loaded signal to the deposit belt. When it receives the picked-up signal from the deposit belt, it moves backwards to the press and sends the ready signal to the press.

Finally, when the deposit belt receives the loaded signal from the press-to-belt robot arm, it starts the belt, sends the picked-up signal to the press-to-belt robot arm and moves the block to further processing. When the block has reached the end of the feed-belt, it stops the belt and sends the ready signal backwards to the press-to-belt robot arm.

The source code for the Production Cell system is given in appendix B. It has the systems Loader, FeedBelt, BeltToPress, Press, PressToBelt, and DepositBelt. These systems are instantiated into the subcomponents feedBelt, beltToPress, press, pressToBelt, depositBelt, and storer.

In order to assure that a block is always moved through the Production Cell system and not become (permanently) stuck somewhere on the way, we examen whether the state variable *storedBlocks* in the *storer* subcomponent will be assigned the value one, meaning that one block has gone thought the whole system. The test can be formally



Property Specification: a g not (subSystem1.Critical and subSystem2.Critical)

Figure 5.6: Property Specification.

Property Specification: "a f storer.StoredBlocks = 1".
Result: True.

Figure 5.7: Property Solution.

stated in CTL as follows:

```
a f storer.StoredBlocks = 1
```

As evident from figure 5.6 and 5.7, all paths finally lead to a state where *storedBlocks* is equals to one, meaning that the block will always be dragged through the Production Cell system.

Chapter 6

Related Work

There has been some attempts to define semantics for AADL and its annexes. Al-Nayeem et al. [2] present an architecture pattern for ensuring synchronous computation semantics using the Physically-Asynchronous Logically-Synchronous (PALS) protocol [23]. They have also developed a modeling framework in AADL to automatically transform a synchronous design of a real-time distributed system into an asynchronous design satisfying the PALS protocol.

Abdoul et al. [1] presents an AADL model transformation that covers three aspects: structure, behavior description and execution semantics. They complete the AADL meta model in order to improve system behavior, they also implement these rules using the Kermeta meta modeling platform.

Varona-Gomez and Villar [26] presents the AADL simulation tool AADS, which supports the performance analysis of the AADL specification throughout the refinement process from the initial system architecture until the complete, detailed application and execution platform are developed. In this way, AADS enables the verification of the initial timing constraints during the complete design process.

Rugina et al. [22] presents an iterative dependency-driven approach for dependability modeling using AADL, which is part of a complete framework that allows the generation of dependability analysis and evaluation models from AADL models to support the analysis of software and system architectures, in critical application domains.

Relying on the MARTE Time Model [13] and the operational semantics of the Clock Constraint Specification Language (CCSL) [19], Mellat et al. [20] equip UML activities to the execution semantics of an AADL specification as a part of a broader effort to build a generic simulator for UML models with the semantics explicitly defined within the model.

Gui et al. [14] regards AADL as an Model-Driven Architecture method. They use the linear hybrid automata to abstract the semantics of the software components explicitly and use the TIMES tool [3] to simulate the semantics of linear hybrid automata and the scheduling execution trace of AADL software components, respectively.

Berthomieu et al. [4] give a high-level view of the tools involved and describe the successive transformations performed by their verification process. They also report on an experiment carried out in order to evaluate our framework and give the first experimental results obtained on real-size models.

Sokolsky et al. [24] discuss the use of formal methods for the analysis of architectural models expressed in AADL. They describe the system as a collection of interacting components where the AADL standard prescribes semantics for the thread components and rules of interaction between threads and other components in the system. They also present a semantics-preserving translation of AADL models into the real-time process algebra ACSR [17], which allows schedulability analysis of AADL models.

França et al. [12] present an evaluation of the AADL Behavioral Annex currently in evaluation phase. They relate their experiment with respect to a development concerning the re-engineering of flight software. This experiments has led them to introduce hierarchical aspects and study the link especially with AADL modes. They discuss the definition of a semantics for the AADL execution model and propose some enhancements.

Yang et al. [27] propose a formal semantics for the AADL behavior annex using Timed Abstract State Machine (TASM). They give the semantics of AADL default execution model, then they formally define some aspects semantics of behavior annex. A prototype of real-time behavior modeling and verification is proposed, and a case study is given to validate its feasibility.

De Niz et al. [6] present an approach to model replication patterns in AADL and analyze potentially unintended behaviors. That approach takes advantage of the strong semantics of AADL to model replication patterns at the architecture level. They develop two AADL models, where the first one defines the intended behavior in synchronous call sequences, and the second model describes the replication architecture. These two models are then compared using a differential model in Alloy [15] where the requirements of the first model and the concurrency and potential failure of the second are combined.

Chapter 7

Conclusions and Further Work

We have developed a denotational semantics with which help it is possible to prove CTL property specifications of a model defined in AADL with Behavior Annex. We have also developed a tool that implements the semantics, a tool with a graphical user interface as an OSATE plug-in that has been tested on the Production Cell system.

There are several ways to continue the work of this report. One obvious approach is to optimize the algorithms behind the semantics when it comes to state tree generation and property specification evaluation. In our approach, we use a set to determine whether a state is repeated along a path in the tree. However, is should be possible to use a global set that is capable of catching whether a state is repeated along another path. It should also be possible to evaluate the tree "on the fly"; that is, the evaluation takes place during the tree generation. In that way, no more of the tree than necessary in order to determine the value of the property specification will be generated.

Another interesting extension of the semantics is to add time annotation to the transitions in order to determine the minimal an maximal time frame for a property specification to be satisfied. One other possible way forward it to look into other architecture design languages, such as MARTE [19] or EAST-ADL [7] as input language for the denotational semantics.

Appendix A

An Example of Parsing

The process of determine whether a source code satisfying a grammar is called *parsing*. The parsing methods can be divided into *top-down parsing*, starting from the grammar start symbol and ending with the source code, or *bottom-up parsing*, starting with the source code and ending with the grammar start symbol. Tables A.1, A.2, A.3, and A.4 illustrates a top-down parsing of the *SubSystem1* system below.

```
system SubSystem1
  features
    CriticalEnter: in event port;
    CriticalLeave: out event port;
  annex SubSystemAnnex1 {**
   initializations
     CriticalLeave!;
   states
     Waiting :initial state;
     Critical :state;
   transitions
      Waiting -[CriticalEnter?]-> Critical;
      Critical -[true]-> Waiting {CriticalLeave!;}
   **};
end SubSystem1;
```

System
system *Identifier SystemBody* **end ;**
system SubSystem1 *SystemBody* **end ;**
system SubSystem1 *OptionalFeatures OptionalAnnex* **end ;**
system SubSystem1 features *Feature OptionalAnnex* **end ;**
system SubSystem1 features *Feature Feature OptionalAnnex* **end ;**
system SubSystem1 features *Identifier* **: in event port ;** *Feature OptionalAnnex* **end ;**
system SubSystem1 features CriticalEnter : in event port ; *Feature OptionalAnnex* **end ;**

Table A.1: The top-down parsing of the *SubSystem1* system, part 1.

system SubSystem1 features CriticalEnter : in event port ; *Identifier* **: out event port ;** *OptionalAnnex* **end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; *OptionalAnnex* **end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; *Annex* **end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex *Identifier* {** *OptionalStateVariables OptionalInitializations OptionalStates OptionalTransitions* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** *OptionalInitializations OptionalStates OptionalTransitions* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations** *Identifier* **!;** *OptionalStates OptionalTransitions* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ;** *OptionalStates OptionalTransitions* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states** *State OptionalTransitions* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states** *State State OptionalTransitions* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states** *Identifer* **: initial state ;** *State OptionalTransitions* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ;** *State OptionalTransitions* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ;** *Identifier* **: state ;** *OptionalTransitions* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ;** *OptionalTransitions* **} **; end ;**

Table A.2: The top-down parsing of the *SubSystem1* system, part 2.

system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions** *Transition* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions** *Transition Transition* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions** *Identifier* -[*ExpressionS*]-> *Identifier OptionalAction Transition* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting** -[*ExpressionS*]-> *Identifier OptionalAction Transition* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting** -[*Identifier* ?]-> *Identifier OptionalAction Transition* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting** -[**CriticalEnter** ?]-> *Identifier OptionalAction Transition* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting** -[**CriticalEnter** ?]-> **Critical** *OptionalAction Transition* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting** -[**CriticalEnter** ?]-> **Critical ;** *Transition* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting** -[**CriticalEnter** ?]-> **Critical ;** *Identifier* -[*ExpressionS*]-> *Identifier OptionalAction* **} **; end ;**
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** **initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting** -[**CriticalEnter** ?]-> **Critical ; Critical** -[*ExpressionS*]-> *Identifier OptionalAction* **} **; end ;**

Table A.3: The top-down parsing of the *SubSystem1* system, part 3.

system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting -[CriticalEnter ?]-> Critical ; Critical -[true]-> *Identifier OptionalAction* **} ; end ;
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting -[CriticalEnter ?]-> Critical ; Critical -[true]-> Waiting *OptionalAction* **} ; end ;
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting -[CriticalEnter ?]-> Critical ; Critical -[true]-> Waiting *Identifier* ! ; **} ; end ;
system SubSystem1 features CriticalEnter : in event port ; CriticalLeave : out event port ; annex SubSystemAnnex1 {** initializations CriticalLeave ! ; states Waiting : initial state ; Critical : state ; transitions Waiting -[CriticalEnter ?]-> Critical ; Critical -[true]-> Waiting CriticalLeave! ; **} ; end ;

Table A.4: The top-down parsing of the *SubSystem1* system, part 4.

Appendix B

The Production Cell Source Code

```
system Loader
  features
    -- Forwards.
    InFeedBeltReady: in event port;
    OutBlockReady: out event port;
  annex Loader {**
   state variables
     LoadedBlocks :integer;
   initializations
     LoadedBlocks := 0;
   states
     Waiting :initial state;
     Loading :state;
   transitions
      Waiting -[(LoadedBlocks < 1) and
                (InFeedBeltReady?)]-> Loading;
      Loading -[true]-> Waiting {OutBlockReady!;
                        LoadedBlocks := LoadedBlocks + 1;}
   **};
end Loader;

system FeedBelt
  features
    -- Backwards.
    InBlockReady: in event port;
    InFeedBeltReady: out event port;
    -- Forwards.
    InArmReady: in event port;
    OutBlockReady: out event port;
  annex FeedBelt {**
   initializations
     InFeedBeltReady!;
   states
     NoBlock_MotorOff :initial state;
     BlockAtBeginning_MotorOn, BlockAtEnd_MotorOff :state;
```

```
    transitions
      NoBlock_MotorOff -[InBlockReady?]-> BlockAtBeginning_MotorOn;
      BlockAtBeginning_MotorOn -[true]-> BlockAtEnd_MotorOff {OutBlockReady!;}
      BlockAtEnd_MotorOff -[InArmReady?]-> NoBlock_MotorOff {InFeedBeltReady!;}
  **};
end FeedBelt;

system BeltToPress
  features
    -- Backwards.
    InBlockReady: in event port;
    InArmReady: out event port;
    -- Forwards.
    PressReady: in event port;
    OutBlockReady: out event port;
  annex BeltToPress {**
   initializations
     InArmReady!;
   states
     MagnetOff_AtBelt_Retracted :initial state;
     MagnetOn_AtBelt_Retracted :state;
     MagnetOn_AtBelt_Extracted :state;
     MagnetOn_AtPress_Extracted :state;
     MagnetOn_AtPress_Retracted :state;
     MagnetOff_AtPress_Retracted :state;
     MagnetOff_AtPress_Extracted :state;
     MagnetOff_AtBelt_Extracted :state;
    transitions
      MagnetOff_AtBelt_Retracted -[InBlockReady?]-> MagnetOn_AtBelt_Retracted;
      MagnetOn_AtBelt_Retracted -[true]-> MagnetOn_AtBelt_Extracted;
      MagnetOn_AtBelt_Extracted -[true]-> MagnetOn_AtPress_Extracted;
      MagnetOn_AtPress_Extracted -[true]-> MagnetOff_AtPress_Retracted;
      MagnetOff_AtPress_Retracted -[PressReady?]-> MagnetOff_AtPress_Retracted;
      MagnetOff_AtPress_Retracted -[true]-> MagnetOff_AtPress_Extracted;
      MagnetOff_AtPress_Extracted -[true]-> MagnetOff_AtBelt_Extracted;
      MagnetOff_AtBelt_Extracted -[true]-> MagnetOff_AtBelt_Retracted
          {InArmReady!; OutBlockReady!;}
  **};
end BeltToPress;

system Press
  features
    -- Backwards.
    InBlockReady: in event port;
    PressReady: out event port;
    -- Forwards.
    OutArmReady: in event port;
    OutBlockReady: out event port;
  annex ArmA {**
   initializations
     PressReady!;
   states
     Waiting :initial state;
     Pressing :state;
```

```
    transitions
      Waiting -[InBlockReady?]-> Pressing;
      Pressing -[true]-> Waiting {OutBlockReady!; PressReady!;}
  **};
end Press;

system PressToBelt
  features
    -- Backwards.
    InBlockReady: in event port;
    OutArmReady: out event port;
    -- Forwards.
    OutFeedBeltReady: in event port;
    OutBlockReady: out event port;
  annex PressToBelt {**
   initializations
     OutArmReady!;
   states
     MagnetOff_AtPress_Retracted :initial state;
     MagnetOn_AtPress_Retracted :state;
     MagnetOn_AtPress_Extracted :state;
     MagnetOn_AtBelt_Extracted :state;
     MagnetOn_AtBelt_Retracted :state;
     MagnetOff_AtBelt_Retracted :state;
     MagnetOff_AtBelt_Extracted :state;
     MagnetOff_AtPress_Extracted :state;
    transitions
      MagnetOff_AtPress_Retracted -[InBlockReady?]-> MagnetOn_AtPress_Retracted;
      MagnetOn_AtPress_Retracted -[true]-> MagnetOn_AtPress_Extracted;
      MagnetOn_AtPress_Extracted -[true]-> MagnetOn_AtBelt_Extracted;
      MagnetOn_AtBelt_Extracted -[true]-> MagnetOff_AtBelt_Retracted;
      MagnetOff_AtBelt_Retracted -[true]-> MagnetOn_AtBelt_Retracted;
      MagnetOn_AtBelt_Retracted -[true]-> MagnetOff_AtBelt_Extracted;
      MagnetOff_AtBelt_Extracted -[true]-> MagnetOff_AtPress_Extracted;
      MagnetOff_AtPress_Extracted -[OutFeedBeltReady?]->
          MagnetOff_AtPress_Retracted {OutBlockReady!;}
  **};
end PressToBelt;

system DepositBelt
  features
    -- Backwards.
    InBlockReady: in event port;
    OutFeedBeltReady: out event port;
    -- Forwards.
    StorerReady: in event port;
    OutBlockReady: out event port;
  annex FeedBelt {**
   initializations
     OutFeedBeltReady!;
   states
     NoBlock_MotorOff :initial state;
     BlockAtBeginning_MotorOn, BlockAtEnd_MotorOff :state;
    transitions
```

```
      NoBlock_MotorOff -[InBlockReady?]-> BlockAtBeginning_MotorOn;
      BlockAtBeginning_MotorOn -[StorerReady?]-> BlockAtEnd_MotorOff
          {OutBlockReady!;}
      BlockAtEnd_MotorOff -[true]-> NoBlock_MotorOff {OutFeedBeltReady!;}
  **};
end DepositBelt;

system Storer
  features
    -- Backwards.
    StorerReady: out event port;
    InStorerBlockReady: in event port;
  annex Storer {**
   state variables
     StoredBlocks :integer;
   initializations
     StorerReady!;
     StoredBlocks := 0;
   states
     Waiting :initial state;
     Storing :state;
    transitions
      Waiting -[InStorerBlockReady?]-> Storing;
      Storing -[true]-> Waiting {StoredBlocks := StoredBlocks + 1;
                                 StorerReady!;}
    **};
end Storer;

system implementation ProductionCell.impl
  subcomponents
    loader: system Loader;
    feedBelt: system FeedBelt;
    beltToPress: system BeltToPress;
    press: system Press;
    pressToBelt: system PressToBelt;
    depositBelt: system DepositBelt;
    storer: system Storer;
  connections
    -- Loader -> FeedBelt
    event port feedBelt.InFeedBeltReady -> loader.InFeedBeltReady;
    event port loader.OutBlockReady -> feedBelt.InBlockReady;
    -- FeedBelt -> BeltToPress
    event port beltToPress.InArmReady -> feedBelt.InArmReady;
    event port feedBelt.OutBlockReady -> beltToPress.InBlockReady;
    -- BeltToPress -> Press
    event port press.PressReady -> beltToPress.PressReady;
    event port beltToPress.OutBlockReady -> press.InBlockReady;
    -- Press -> PressToBelt
    event port pressToBelt.OutArmReady -> press.OutArmReady;
    event port press.OutBlockReady -> pressToBelt.InBlockReady;
    -- PressToBelt -> DepositBelt
    event port depositBelt.OutFeedBeltReady -> pressToBelt.OutFeedBeltReady;
    event port pressToBelt.OutBlockReady -> depositBelt.InBlockReady;
    -- DepositBelt -> Storer
```

```
    event port storer.StorerReady -> depositBelt.StorerReady;
    event port depositBelt.OutBlockReady -> storer.InStorerBlockReady;
end ProductionCell.impl;
```

```
evaluate_children : TreeProp × List × WidthOp × DepthOp → Boolean
evaluate_children TP child_list width_op depth_op =
    case (list_size child_list) of
        0 ⇒ case depth_op of
                global ⇒ true
              | final ⇒ false
      | 1 ⇒ evaluate_tree TP (list_get 0 child_list) width_op depth_op
      | default ⇒ let (head, tail) = list_split child_list in
          case width_op of
              all ⇒ (evaluate_tree TP head width_op depth_op) and
                    (evaluate_children TP tail width_op depth_op)
            | exists ⇒ (evaluate_tree TP head width_op depth_op) or
                       (evaluate_children TP tail width_op depth_op)
evaluate_tree : TreeProp × Tree × WidthOp × DepthOp → Boolean
evaluate_tree PS tree width_op depth_op =
    case depth_op of
        global ⇒ let (single subProp) = PS in
          (is_true (evaluate_prop_spec subProp tree)) and
          (evaluate_children PS (tree_get_children tree) width_op depth_op)
      | final ⇒ let (single subProp) = PS in
          (is_true (evaluate_prop_spec subProp tree)) or
          (evaluate_children PS (tree_get_children tree) width_op depth_op)
```

Bibliography

[1] Thomas Abdoul, Joël Champeau, Philippe Dhaussy, Pierre Yves Pillain, and Jean-Charles Roger. AADL execution semantics transformation for formal verification. In *ICECCS*, pages 263–268. IEEE Computer Society, 2008.

[2] Abdullah Al-Nayeem, Mu Sun, Xiaokang Qiu, Lui Sha, Steven P. Miller, and Darren D. Cofer. A formal architecture pattern for real-time distributed systems. In Theodore P. Baker, editor, *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 161–170. IEEE Computer Society, 2009.

[3] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS '03)*, 2003.

[4] B. Berthomieu, Jean-Paul Bodeveix, Silvano Dal Zilio, Pierre Dissaux, Mamoun Filali, Pierre Gaufillet, Sebastien Heim, and F. Vernadat. Formal verification of AADL models with fiacre and tina (regular paper). In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 19/05/2010-21/05/2010*, page (electronic medium). SIA/3AF/SEE, 2010.

[5] Egon Borger and Robert Stark. *Abstract State Machines - A Method for High-level System Design and Analysis.* Springer-Verlag Berlin And Heidelberg Gmbh and Co. Kg, 2003.

[6] Dionisio de Niz and Peter H. Feiler. Verification of replication architectures in AADL. In *ICECCS*, pages 365–370. IEEE Computer Society, 2009.

[7] Vincent Debruyne, Franoise Simonot-Lion, and Yvon Trinquet. EAST-ADL an architecture description language validation and verification aspects. *Architecture Description Languages*, 176:181–195, 2005.

[8] M. Faugere. MARTE: Also an UML Profile for Modelling AADL Applications Engineering Complex Computer Systems. In *12th IEEE International Conference*, pages 359 – 364, 2007.

[9] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis and Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Society of Automotive Engineers, 2006.

[10] Peter Feiler and Bruce Lewis. SAE Architecture Analysis and Design Language (AADL) Annex Volume 1. Technical Report AS5506/1, Society of Automobile Engineers, 2006.

[11] Peter Feiler and Ana Rugina. Dependability modeling with the architecture analysis and design language (AADL). Technical Report CMU/SEI-2007-TN-043, Carnegie Mellon University, 2007.

[12] Ricardo Bedin França, Jean-Paul Bodeveix, Mamoun Filali, Jean-François Rolland, David Chemouil, and Dave Thomas. The AADL behaviour annex - experiments and roadmap. In *ICECCS*, pages 377–382. IEEE Computer Society, 2007.

[13] Object Management Group. UML Profile for MARTE, beta 2. Technical Report ptc/08-06-08, The ProMARTE Consortium, http://hdl.handle.net/2142/11897, 2008.

[14] Shenglin Gui, Lei Luo, Yun Li, and Lijie Wang. Formal schedulability analysis and simulation for aadl. In *The 2008 International Conference on Embedded Software and Systems (ICESS2008)*, 2008.

[15] D. Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, MIT, 2000.

[16] Donald E. Knuth. *Communications of the ACM*, 7:735–736, 1964.

[17] I. Lee, P. Bremond-Gregoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. In *Proceedings of the IEEE*, pages 158–171, 1994.

[18] C. Lewerentz and T. Lindner. Formal development of reactive systems, case study production cell. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems*, Lecture Notes in Computer Science, pages 21–54. Springer-Verlag, 1995.

[19] F. Mallet. Ccsl: specifying clock constraints with UML/Marte. *ISSE*, 4(3):309–314, 2008.

[20] Frédéric Mallet, Charles André, and Julien DeAntoni. Executing AADL models with UML/MARTE. In *ICECCS*, pages 371–376. IEEE Computer Society, 2009.

[21] Martin Ouimet and Kristina Lundqvist. Modeling the Production Cell System in the TASM Language. Technical Report ESL-TIK-00209, Embedded Systems Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA, 2007.

[22] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaaniche. An architecture-based dependability modeling framework using AADL. In *Proc. 10th IASTED International Conference on Software Engineering and Applications (SEA'2006), Dallas (USA), 13-15 November2006 (13/11/2006)*, pages 222–227, April 06 2006.

[23] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. C. Olveczky. MPALS: Physically Asynchronous Logically Synchronous Systems. Technical report, University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/11897, 2009.

[24] Oleg Sokolsky, Insup Lee, and Duncan Clarke. Schedulability analysis of AADL models. In *IPDPS*. IEEE, 2006.

[25] The SAE Technical Standards Board. The annex behavior specification. Technical Report AS5506, SAE International, 2007.

[26] Roberto Varona-Gomez and Eugenio Villar. AADL simulation and performance analysis in systemC. In *ICECCS*, pages 323–328. IEEE Computer Society, 2009.

[27] Zhibin Yang, Kai Hu, Dianfu Ma, and Lei Pi. Towards a formal semantics for the AADL behavior annex. In *DATE*, pages 1166–1171. IEEE, 2009.