Project Number 318772

D2.1 Specification of MILS-AADL

Version 1.4
23 September 2014
Final

Public Distribution

RWTH Aachen University

Project Partners: **Fondazione Bruno Kessler**, **fortiss**, **Frequentis**, **LynuxWorks**, **The Open Group**, **RWTH Aachen University**, **TTTech**, **Université Joseph Fourier**, **University of York**

Project Partner Contact Information

Fondazione Bruno Kessler Alessandro Cimatti Via Sommarive 18 38123 Trento, Italy Tel: +39 0461 314320 Fax: +39 0461 314591 E-mail: cimatti@fbk.eu	**fortiss** Harald Ruess Guerickestrasse 25 80805 Munich, Germany Tel: +49 89 36035 22 0 Fax: +49 89 36035 22 50 E-mail: ruess@fortiss.org
Frequentis Wolfgang Kampichler Innovationsstrasse 1 1100 Vienna, Austria Tel: +43 664 60 850 2775 Fax: +43 1 811 50 77 2775 E-mail: wolfgang.kampichler@frequentis.com	**LynuxWorks** Yuri Bakalov Rue Pierre Curie 38 78210 Saint-Cyr-l'Ecole, France Tel: +33 1 30 85 06 00 Fax: +33 1 30 85 06 06 E-mail: ybakalov@lnxw.com
RWTH Aachen University Joost-Pieter Katoen Ahornstrasse 55 D-52074 Aachen, Germany Tel: +49 241 8021200 Fax: +49 241 8022217 E-mail: katoen@cs.rwth-aachen.de	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 894 5845 E-mail: s.hansen@opengroup.org
TTTech Wilfried Steiner Schonbrunner Strasse 7 1040 Vienna, Austria Tel: +43 1 5853434 983 Fax: +43 1 585 65 38 5090 E-mail: wilfried.steiner@tttech.com	**Université Joseph Fourier** Saddek Bensalem Avenue de Vignate 2 38610 Gieres, France Tel: +33 4 56 52 03 71 Fax: +33 4 56 03 44 E-mail: saddek.bensalem@imag.fr
University of York Tim Kelly Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325477 Fax: +44 7976 889 545 E-mail: tim.kelly@cs.york.ac.uk	

Contents

Document Control

Version	Status	Date
0.1	Document outline	14 February 2013
1.0	Preliminary complete version	19 April 2013
1.1	QA review version	30 April 2013
1.2	Final version	22 May 2013
1.3	Post-review version	02 March 2014
1.4	Updated features	23 September 2014

Executive Summary

This document describes the syntax of a MILS dialect of the Architecture Analysis and Design Language (AADL), denoted by MILS-AADL. Essential features of this new language are component definitions in terms of interfaces and implementations, their interaction and dynamic activation, architecture refinement, error modelling, and security-related mechanisms such as encryption and authentication, and the ability to analyse security- and safety-related behavioural constraints.

This goal is achieved by extending a core fragment of AADL by new features to support the representation of security architectures and their effect on quantitative system attributes.

1 Introduction

This document constitutes Deliverable D2.1 of Work Package 2 (Graphical & Declarative Languages) of the EU FP7 project *Distributed MILS for Dependable Information and Communication Infrastructures* (D-MILS; Project Number 318772). It specifies the syntax of a MILS dialect of the Architecture Analysis and Design Language (AADL), denoted by MILS-AADL. The latter is obtained by extending a core fragment of the AADL by new features to support the representation of security architectures and their effect on quantitative system attributes.

The remainder of this document is organised as follows. Section 2 summarises the requirements that guided the design of MILS-AADL. Section 3 gives an overview of the language features which we propose in order to satisfy these requirements. The language design is detailed in the following sections, starting with Section 4 which presents the top-level syntax of system specifications and followed by Sections 5 and 6 which respectively introduce (nominal) component specifications and error models. Section 7 summarises syntactic restrictions on component and error models and their possible interactions. Section 8 sketches the basic properties that should be expressible to support the analysis of different behavioural aspects of a system. The presentation is concluded by an index of grammar symbols and by a list of references.

2 Requirements for MILS-AADL

This section summarises language requirements that are derived from the case studies [5, 7], and that are analysed in greater detail in [6].

- The language shall provide an cohesive and uniform view of the system to be modelled, integrating both hardware (processors, storages, buses, sensors, actuators, ...) and software (processes, threads, data, ...) components and their interactions.
- In order to study the behaviour of discrete systems in a context of continuous physical systems such as power grids, it shall support hybridity in the form of continuous real-valued variables with (linear) time-dependent dynamics. In particular, it shall allow the specification of timed behaviour.
- It shall be possible to model the system's non-nominal behaviour, dealing with component faults and failures, their probabilistic occurrence and their impact on system behaviour and parameters.
- The language shall encompass security-related mechanisms such as encryption and authentication.
- It shall be possible to analyse different aspects of the system such as functional correctness, safety, performance, and security.

3 Overview of MILS-AADL

In order to meet the requirements which were sketched in the previous chapter, we propose to define MILS-AADL as an "extended subset" of the AADL [12, 8] and its Error Model Annex [13, 9]. The resulting features can be summarised as follows:

- The system under consideration is hierarchically organized into *components*, distinguishing between *software*, *hardware* (in AADL terms: *execution platform*), and *composite* components.
- The overall specification can be divided into *packages* to support modularity. Another means for modularisation and encapsulation is the introduction of named *type and value constants*.
- Every component is defined by its type and its implementation. *Component types* specify functional interfaces (that is, externally visible *ports*) as seen by the environment (black-box view). Both (incoming and outgoing) *event* and *data* ports are supported for instantaneous and continuous communication between components, respectively.
- *Component implementations* represent their internal structure (white-box view):
  - the implementation structure of the component as an assembly of *subcomponents*;
  - their (logical) interaction through *event port connections* and *data flows*;
  - their (physical) *binding* (that is, software/hardware deployment) at runtime; and
  - operational *modes* as an abstraction of the concrete component behaviour, possibly representing different system configurations and connection topologies, with mode *transitions* which are spontaneous or triggered by events arriving at ports.

  Also timed and hybrid behaviour can be specified in the implementation. Multiple implementations of a component type can be defined, allowing component variants with the same interface to be modelled.
- The following *software component categories* are supported:
  - *process*: represents a software entity that can be bound to a processor and a memory component, can contain threads;
  - *thread*: a unit of sequential execution which must be the subcomponent of a process; and
  - *data*: data types, involving both discrete (booleans, enumerations, (ranges of) integers, security keys, reals) and analogue (clocks, continuous real-valued variables) value updates using various built-in operators.
- The following *hardware component categories* are supported:
  - *processor*: executes processes and threads;
  - *memory*: stores data and code;
  - *device*: interfaces with or represents the external environment;
  - *bus*: connects hardware and composite components; and
  - *network*: connects composite components.
- The following *composite component categories* are supported:
  - *node*: identifies a network node; and
  - *subject*: identifies a subject in the system's policy architecture; and
  - *system*: represents subsystems or complete systems.
- We distinguish between *atomic* and *non-atomic* component implementations, depending on whether the implementation declares only **`data`** or also other categories of subcomponents,

respectively. (See Section 7 for an overview of which component categories are allowed to have which categories of subcomponents.)

- Component specifications can be equipped with *error models* to support safety and dependability analyses. Again, an error model is defined by its type and its implementation (variant).
- An *error model type* defines an interface in terms of (incoming and outgoing) *error propagations*, which are used to exchange error information between components.
- An *error model implementation* provides the structural details of the error model. It employs *error states* to represent the current configuration of the component with respect to errors. Its actual behaviour is defined by a (probabilistic) machine performing transitions between error states that are triggered by error events and error propagations.
  - *Error events* are internal to the component. They reflect changes of the error state caused by local faults and repair operations and can be annotated with occurrence *rates* to model probabilistic error behaviour.
  - Outgoing *error propagations* report an error state to other components. If their error states are affected, the other components will have an corresponding incoming propagation.

The following sections introduce the concrete syntax of MILS-AADL.

4 Syntax of MILS-AADL

The following sections defines the syntax of our specification language and provides corresponding explanations and examples. The context-free part of this syntax is given in (extended) Backus-Naur Form [11], using the following notations:

- boldface symbols represent keywords (e.g., **`package`**);
- symbols with initial uppercase letters stand for nonterminal symbols (e.g., *SystemSpecification*), they are listed in the index at the end of the document;
- symbols with initial lowercase letters represent terminal symbols (e.g., *identifier*);
- $\alpha \mid \beta$: choice between α and β;
- $\{\alpha\}$: grouping of α;
- $[\alpha]$: zero or one occurrences of α;
- α^*: zero or more occurrences of α; and
- α^+: one or more occurrences of α.

Note the distinction between $[\alpha]$, indicating zero or one occurrences of α, and `[`α`]`, indicating α between literal brackets. *Comments* in specifications are started by a double hyphen ("−−") on any column. They extend to the end of line.

4.1 Comments and Annotation

All characters from a double-minus "−−" symbol onwards until the end of the line, are regarded as a comment. This means these characters will be ignored by the parser. Put differently, the parse trees of a piece of source code with and without comments are identical.

Annotations are a variant on comments, where the annotation is not ignored by the parser, but the annotation does not have any effect on the semantics of the model being described. The annotation *does* appear in the parse tree. The annotation may be placed directly before any of the following nonterminals:

- *PackageSpecification*
- *ComponentDeclaration*
- *ErrorDeclaration*

In the parse tree, the annotation will appear as a child node of the corresponding nonterminal node. The annotation is used for annotating parts of the specification with information relevant to the various verification tools. The annotation text consists of two parts, separated by a colon: a part identifying the tool concerned with this annotation, and free text to be interpreted by this tool.

Annotations
::= *Annotation*$^+$
Annotation
::= {*ToolSymbol* **:** *annotationBody*}

ToolSymbol
::= `smv` | `ocra` | `bip`

The annotation body terminal matches all characters until the closing brace ("}"). The backslash character is an escape character which will be ignored except that the first character following the escape character is matched *verbatim*. This means the backslash character itself can be matched as "\\" and braces can be matched as "\{" and "\}".

4.2 System Specifications

A *system specification* consists of a sequence of package specifications and part declarations. A *package* is a named grouping of declarations that can be used to organize specifications by establishing distinct namespaces. It is divided into public and private segments. Declarations in the *public* segment are visible also outside the package whereas declarations in the *private* segment are visible only within the package. To reference an element in the public segment from outside a package, its name has to be prefixed by the package identifier (separated by double colons "`::`").

A *constant declaration* (cf. Section 4.4) introduces a constant identifier that represents a (discrete, possibly uninterpreted) data type, an uninterpreted function, a constant value or key pair. A *component* or *error (model) declaration* defines the type or an implementation of a component (cf. Section 5) or of an error model (cf. Section 6), respectively. It is possible to declare multiple implementations of a component or of an error model.

SystemSpecification
::= $\{$*PackageSpecification* | *PartDeclaration*$\}^+$
PackageSpecification
::= **`package`** *PackageIdentifier*
`public` *PartDeclaration*$^+$
[**`private`** *PartDeclaration*$^+$]
`end` *PackageIdentifier*`;`
PackageIdentifier
::= *identifier*
PartDeclaration
::= *ConstantDeclarations* | *ComponentDeclaration* | *ErrorDeclaration*
ComponentDeclaration
::= *ComponentType* | *ComponentImplementation*
ErrorDeclaration
::= *ErrorType* | *ErrorImplementation*

Example 1 A package declaring a public CPU component type with two implementations, all being public.

```
package CPUpackage
  public
    processor CPU
      -- type description
    end CPU;
    processor implementation CPU.Intel
      -- implementation description
    end CPU.Intel;
    processor implementation CPU.PowerPC
      -- implementation description
    end CPU.PowerPC;
end CPUpackage;
```

Later we will use the notion of *scopes*, which is defined as follows:

- The list of declarations outside any package forms a scope.
- Each single package forms a scope.

Syntactic Restrictions

A-1 Keywords are not allowed in identifier positions.

A-2 All package identifiers declared in a system specification have to be distinct.

A-3 Cross-references between the public and the private part of the same package are not admitted, that is, a public component implementation may not implement a private component type, and a private component implementation may not implement a public component type. The same applies to error types and implementations.

A-4 The first package identifier has to match the end package identifier.

4.3 Data Types

The following *discrete data types* are predefined:

- **bool**: Boolean values, with constants **true** and **false**;
- **enum**: enumeration of symbolic values, given in the form **enum**(*id*$_1$, ..., *id*$_n$) where $n \geq 1$ and each id_i is a distinct identifier;
- **int**: integer numbers (in $\mathbb{Z} = \{0, 1, -1, \ldots\}$), with numerals of the form [−]{0 | ... | 9}$^+$;
- **real**: floating-point numbers, with numerals of the form [−]{0 | ... | 9}$^+$[.{0 | ... | 9}$^+$];
- [*l*..*u*]: integer values in the range $\{l, \ldots, u\}$ ($l, u \in \mathbb{Z}$, $l \leq u$), with **int** numerals (see above) in the given range;
- **private key**: private keys for dealing with asymmetric encryption and authentication, taken from an asymmetric key pair declared on the global level (cf. Section 4.4);

- **public key**: public keys for dealing with asymmetric encryption and authentication, taken from an asymmetric key pair declared on the global level (cf. Section 4.4);
- **key**: symmetric key for dealing with asymmetric encryption and authentication, taken from a symmetric key pair declared on the global level (cf. Section 4.4);
- **hashed** τ: hashes of values of discrete type τ;
- **keyhashed** τ: keyed hashes of values of discrete type τ;
- **encrypted** τ: values of discrete type τ which have been encrypted;
- **signed** τ: values of discrete type τ which have been signed; and
- $(\tau_0, \ldots, \tau_{n-1})$: tuples of values of discrete types τ_0 through τ_{n-1}, where $n \geq 1$.

Moreover, there exist the following *analogue data types* that allow to model physical behaviour:

- **clock**: for data components whose values (in $\mathbb{R}_{\geq 0}$) linearly increase over time; and
- **continuous**: for data components whose values (in $\mathbb{R}$) continuously change according to differential equations with a constant slope.

Note the distinction between key pairs and keys (parts of a key pair). Key pairs are always declared as constants (cf. Section 4.4). Their parts, the private or public key of a specific key pair, can only be accessed as the default value of a data port, event data port or data subcomponent (cf. "Key selector" in Section 5.1.1). These keys can then be freely communicated as regular data.

DataType
::= *DiscreteDataType* | *AnalogueDataType*

DiscreteDataType
::= **bool** | **enum** (*EnumIdentifierList*) | **int** | **real** |
[*ConstantValue* **..** *ConstantValue*] | *KeyDataType* |
(*DiscreteDataType* { **,** *DiscreteDataType* }$^{+}$) | *TypeConstantClassifier* |
{**encrypted** | **signed** | **hashed** | **keyhashed**} *DiscreteDataType*

EnumIdentifierList
::= *EnumIdentifier* { **,** *EnumIdentifier* }*

EnumIdentifier
::= *identifier*

KeyDataType
::= [**private** | **public**] **key**

AnalogueDataType
::= **clock** [*TimeUnit*] | **continuous**

The syntax of constant values is defined in the next section.

Syntactic Restrictions

B-1 All symbolic identifiers declared in an **`enum`** type have to be distinct.

B-2 In an integer range type of the form `[`v_1`..`v_2`]`, both v_1 and v_2 must be integer numerals or value constant identifiers (see next section) of that type, and $v_1 < v_2$ must hold.

4.4 Constant Declarations

Constant declarations introduce identifiers that represent discrete data types, uninterpreted functions, constant values or key pairs. Here, analogue data types are excluded as they can only be used for modelling hybrid behaviour within component implementations. Keys are taken from key pairs which must be declared as symmetric or asymmetric. The selectors **`pub`** and **`priv`** are only allowed for defining default values, denoting initial knowledge of those keys. Components cannot directly access the keys with selectors during runtime.

Regarding the matching between constant values and discrete data types, the following rules apply:

- **`true`** and **`false`** match **`bool`**;
- each of the symbolic identifiers declared in an **`enum`** type matches that type;
- each $z \in \mathbb{Z}$ matches **`int`**, **`real`**, and every range type `[`l`..`u`]` such that $l \leq z \leq u$;
- each $r \in \mathbb{R}$ matches **`real`**; and
- $(c_0, \ldots, c_{n-1})$ matches $(\tau_0, \ldots, \tau_{n-1})$ whenever c_0 matches τ_0, and c_1 matches τ_1, etc., up to $n-1$, where $n \geq 1$.

ConstantDeclarations
::= **`constants`** *ConstantDeclaration*$^+$
ConstantDeclaration
::= *TypeConstantDeclaration* |
FunctionConstantDeclaration |
ValueConstantDeclaration |
KeyPairDeclaration
TypeConstantDeclaration
::= *TypeConstantIdentifier* **`:`** **`type`** [**`:=`** *DiscreteDataType*]**`;`**
TypeConstantClassifier
::= [*PackageIdentifier* **`::`**] *TypeConstantIdentifier*
TypeConstantIdentifier
::= *identifier*
FunctionConstantDeclaration
::= *FunctionConstantIdentifier* **`:`** **`function`** *DiscreteDataType*{**`,`** *DiscreteDataType*}*

−> *DiscreteDataType*;

FunctionConstantClassifier

::= [*PackageIdentifier*::] *FunctionConstantIdentifier*

FunctionConstantIdentifier

::= *identifier*

ValueConstantDeclaration

::= *ValueConstantIdentifier*: *DiscreteDataType* := *ConstantValue*;

ValueConstantClassifier

::= [*PackageIdentifier*::] *ValueConstantIdentifier*

ValueConstantIdentifier

::= *identifier*

ConstantValue

::= **true** | **false** | *EnumIdentifier* | *integerNumeral* | *realNumeral* |
(*ConstantValue*{, *ConstantValue*}$^{+}$) | *ValueConstantClassifier* |
FunctionConstantClassifier (*ConstantValue*) |
ConstantValue [*ConstantValue*]

KeyPairDeclaration

::= *KeyPairIdentifier*: {**symmetric** | **asymmetric**} **key pair**;

KeyPairIdentifier

::= *identifier*

Example 2 The following listing shows some constant declarations.

```
constants
  Frame: type := (Header, Payload);
  Header: type := [1..max];
  Payload: type;
  pl: function: real -> Payload;
  Level: type := enum(open, secret, top_secret);
  max: int := 255;
  myframe: Frame := (max, pl(0.0));
  mylevel: Level := secret;
  keysUntrusted: asymmetric key pair;
```

Syntactic Restrictions

C-1 All declared type constant, value constant and key pair identifiers have to be distinct.

C-2 All type constant identifiers occurring in the assigned data type of a type constant declaration must be declared as such.

C-3 Declarations of type constant identifiers must not be recursive.

C-4 All value constant and key pair identifiers occurring in the assigned constant value of a value constant declaration must be declared as such.

C-5 Declarations of value constant identifiers must not be recursive.

C-6 In each value constant declaration, the constant value must match the data type declared for the value constant identifier.

4.5 Built-in Operators

The following tables list the operators that can be used to form larger (non-constant) expressions from basic data elements. Here $RNum$ stands for {**clock**, **continuous**, **real**}, $Range$ stands for an arbitrary integer range type and *Key* for {**key**, **symmetric key**, **asymmetric key**}.

Note that it is *not* possible to mix integer, integer range and real values as operands. As integer constants (see above) can represent both a standard integer and an element of a range, their type (**int** or $Range$) depends on the context in which they occur.

Arithmetic Operators Table 1 lists the supported arithmetic operators. Here $\tau_1, \tau_2 \in RNum$, and the maximum type $\max\{\tau_1, \tau_2\}$ is understood with respect to the order **real** $<$ **clock** $<$ **continuous**. Thus, e.g., $+$: **real** $\times$ **real** $\to$ **real**, $+$: **real** $\times$ **clock** $\to$ **clock**, and $+$: **continuous** $\times$ **clock** $\to$ **continuous**.

When the system specification includes time scales, binary operators are not supported that include **clock** s and other types. Such operators are marked with $^{\bar{T}}$ in the following tables.

Relational Operators Table 2 lists the supported relational operators. Here again $\tau_1, \tau_2 \in RNum$, and only values of the the same **enum** type can be compared using = and !=.

Boolean Operators Table 3 lists the supported Boolean operators.

case Operator The **case** operator is a special construct which is used in expressions of the form

$$\textbf{case}\ b_1 : e_1 ; \ldots ; b_n : e_n\ \textbf{otherwise}\ e_0\ \textbf{end}$$

where $n \geq 1$ and every b_i is a **bool**-valued expression. It returns the value of the first expression e_i such that b_i is true. If no such i exists, it returns the value of e_0. Table 4 lists the supported types. Here $n \geq 1$ and $\tau_0, \tau_1, \ldots, \tau_n \in RNum$. In the last row, all **enum** values must be of the same type.

Time scales The system specification may make use of time scales, which allow time to be expressed using a unit such as seconds or hours. The use of time scales is *optional*. However, if one part of the system specification uses time scales, the entire system must be specified using time scales. When using time scales, it is not possible to directly mix timed (i.e. **clock**) types with

Operator	Type	Meaning
+	$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$	Integer addition
	$\mathit{Range} \times \mathit{Range} \rightarrow \mathit{Range}$	Range addition
	$\tau_1 \times \tau_2 \rightarrow \max\{\tau_1, \tau_2\}$	Real addition$^{\bar{T}}$
−	$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$	Integer subtraction
	$\mathit{Range} \times \mathit{Range} \rightarrow \mathit{Range}$	Range substraction
	$\tau_1 \times \tau_2 \rightarrow \max\{\tau_1, \tau_2\}$	Real subtraction$^{\bar{T}}$
−	$\texttt{int} \rightarrow \texttt{int}$	Unary integer minus
	$\mathit{Range} \rightarrow \mathit{Range}$	Unary range minus
	$\texttt{real} \rightarrow \texttt{real}$	Unary real minus
	$\texttt{clock} \rightarrow \texttt{clock}$	Unary clock minus
	$\texttt{continuous} \rightarrow \texttt{continuous}$	Unary continuous minus
*	$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$	Integer multiplication
	$\mathit{Range} \times \mathit{Range} \rightarrow \mathit{Range}$	Range multiplication
	$\tau_1 \times \tau_2 \rightarrow \max\{\tau_1, \tau_2\}$	Real multiplication
/	$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$	Integer division
	$\mathit{Range} \times \mathit{Range} \rightarrow \mathit{Range}$	Range division
	$\tau_1 \times \tau_2 \rightarrow \max\{\tau_1, \tau_2\}$	Real division
mod	$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$	Integer modulo
	$\mathit{Range} \times \texttt{int} \rightarrow \mathit{Range}$	Range modulo

Table 1: Arithmetic Operators

untimed types. Instead, they can be converted from and to these domain by means of unary time scale operators in expressions. Furthermore, in case time scales are used, clock data types have to be specified using a *TimeUnit*.

Timescale Operators Table 5 lists the supported timescale operators. Here $\tau \in \{\texttt{int}, \texttt{real}, \texttt{continuous}\}$.

Cryptography Operators In MILS-AADL, public-key cryptography is supported via the declaration of private and public keys, taken from symmetric and asymmetric key pairs (cf. Section 4.4), and encryption and decryption algorithms. The latter are used for two purposes:

- Public-key encryption: a message encrypted with a sender's public key cannot be decrypted by anyone except a possessor of the matching private key. This is used to ensure confidentiality.
- Digital signatures: a message signed with a sender's private key can be verified by anyone who has access to the matching public key, thereby proving that the sender had access to the private key and, therefore, is likely to be the person associated with the public key used. This also ensures that the message has not been tampered with.

For a symmetric key pair ks, the public part **pub** (ks) equals the private part **priv** (ks).

Encryption and authentication are each implemented by a pair of functions, **encrypt/decrypt** and **sign/verify**, whose types are given in Table 6 and which satisfy the following equations. For

Operator	Type	Meaning
=	**bool** × **bool** → **bool**	Equivalence
	int × **int** → **bool**	Integer equality
	Range × *Range* → **bool**	Range equality
	$\tau_1 \times \tau_2 \rightarrow$ **bool**	Real equality$^{\bar{T}}$
	enum × **enum** → **bool**	Enumeration equality
!=	**bool** × **bool** → **bool**	Exclusive disjunction
	int × **int** → **bool**	Integer inequality
	Range × *Range* → **bool**	Range inequality
	$\tau_1 \times \tau_2 \rightarrow$ **bool**	Real inequality$^{\bar{T}}$
	enum × **enum** → **bool**	Enumeration inequality
<	**int** × **int** → **bool**	Strictly less on integer
	Range × *Range* → **bool**	Strictly less on ranges
	$\tau_1 \times \tau_2 \rightarrow$ **bool**	Strictly less on real$^{\bar{T}}$
>	**int** × **int** → **bool**	Strictly greater on integer
	Range × *Range* → **bool**	Strictly greater on ranges
	$\tau_1 \times \tau_2 \rightarrow$ **bool**	Strictly greater on real$^{\bar{T}}$
<=	**int** × **int** → **bool**	Less or equal on integer
	Range × *Range* → **bool**	Less or equal on ranges
	$\tau_1 \times \tau_2 \rightarrow$ **bool**	Less or equal on real$^{\bar{T}}$
>=	**int** × **int** → **bool**	Greater or equal on integer
	Range × *Range* → **bool**	Greater or equal on ranges
	$\tau_1 \times \tau_2 \rightarrow$ **bool**	Greater or equal on real$^{\bar{T}}$

Table 2: Relational Operators

Operator	Type	Meaning
and	**bool** × **bool** → **bool**	Conjunction
iff	**bool** × **bool** → **bool**	Equivalence
imp	**bool** × **bool** → **bool**	Implication
not	**bool** → **bool**	Negation
or	**bool** × **bool** → **bool**	Disjunction
xnor	**bool** × **bool** → **bool**	Negated exclusive disjunction
xor	**bool** × **bool** → **bool**	Exclusive disjunction

Table 3: Boolean Operators

any discrete data type τ, ciphertext c of type **encrypted** τ or **signed** τ, (**private**) **key** k_1, and (**public**) **key** k_2:

$$\begin{aligned} \textbf{decrypt}(c, k_1) = m \quad & \text{if } c = \textbf{encrypt}(m, k_2) \text{ and for some key pair } ks, \\ & \quad k_1 = \textbf{priv}(ks) \text{ and } k_2 = \textbf{pub}(ks). \\ \textbf{verify}(c, k_2) = m \quad & \text{if } c = \textbf{sign}(m, k_1) \text{ and for some key pair } ks, \\ & \quad k_1 = \textbf{priv}(ks) \text{ and } k_2 = \textbf{pub}(ks). \end{aligned}$$

Operator	Type	Meaning
case	$(\textbf{bool} \times \textbf{bool})^+ \times \textbf{bool} \rightarrow \textbf{bool}$	Boolean **case**
	$(\textbf{bool} \times \textbf{int})^+ \times \textbf{int} \rightarrow \textbf{int}$	Integer **case**
	$(\textbf{bool} \times \mathit{Range})^+ \times \mathit{Range} \rightarrow \mathit{Range}$	Range **case**
	$(\textbf{bool} \times \tau_1) \times \ldots \times (\textbf{bool} \times \tau_n) \times \tau_0 \rightarrow \max\{\tau_0, \tau_1, \ldots, \tau_n\}$	Real **case**
	$(\textbf{bool} \times \textbf{enum})^+ \times \textbf{enum} \rightarrow \textbf{enum}$	Enumeration **case**

Table 4: **case** Operator

Operator	Type	Meaning
msec	$\tau \rightarrow \textbf{clock}$	Conversion of discrete value to time in milliseconds
sec	$\tau \rightarrow \textbf{clock}$	Conversion of discrete value to time in seconds
min	$\tau \rightarrow \textbf{clock}$	Conversion of discrete value to time in minutes
hour	$\tau \rightarrow \textbf{clock}$	Conversion of discrete value to time in hours
day	$\tau \rightarrow \textbf{clock}$	Conversion of discrete value to time in days
msec	$\textbf{clock} \rightarrow \tau$	Conversion of time to a discrete value in milliseconds
sec	$\textbf{clock} \rightarrow \tau$	Conversion of time to a discrete value in seconds
min	$\textbf{clock} \rightarrow \tau$	Conversion of time to a discrete value in minutes
hour	$\textbf{clock} \rightarrow \tau$	Conversion of time to a discrete value in hours
day	$\textbf{clock} \rightarrow \tau$	Conversion of time to a discrete value in days

Table 5: Timescale Operators

For decryption or verification with the wrong key, the result can be any value of type τ.

Operator	Type	Meaning
encrypt	$\tau \times (\textbf{public key} \cup \textbf{key}) \rightarrow \textbf{encrypted}\ \tau$	Encryption
decrypt	$\textbf{encrypted}\ \tau \times (\textbf{private key} \cup \textbf{key}) \rightarrow \tau$	Decryption
sign	$\tau \times (\textbf{private key} \cup \textbf{key}) \rightarrow \textbf{signed}\ \tau$	Signing
verify	$\textbf{signed}\ \tau \times (\textbf{public key} \cup \textbf{key}) \rightarrow \tau$	Verification

Table 6: Key Operators

Hashing Operator Hashing is supported via the hashing operator.

Tupling Operators MILS-AADL supports the aggregation of data by introducing Cartesian products in the form of tuples, written as $(x_0, \ldots, x_{n-1})$ for tuples x_0 through x_{n-1} of respective discrete data types τ_0 through τ_{n-1}, and corresponding projections. These projections satisfy the following equation:

$$(x_0, \ldots, x_{n-1})[i] = x_i$$

In the projection, the index (between brackets) must be an integer constant i such that $0 \leq i < n$.

Operator	Type	Meaning
hash	$\tau \rightarrow$ **hashed** τ	Hashing
keyhash	$\tau \times$ **key** $\rightarrow$ **keyhashed** τ	Keyed hashing

Table 7: Hashing Operator

Operator	Type	Meaning
(., ..., .)	$\tau_0 \times \cdots \times \tau_{n-1} \rightarrow \tau_0 \times \cdots \times \tau_{n-1}$	Tupling
.[.]	$(\tau_0 \times \cdots \times \tau_{n-1}) \times$ **int** $\rightarrow \tau_i$	Projection for position i

Table 8: Tupling Operators

Operator Precedence The precedence of operators is defined by the following list (from higher to lower precedence). In addition, parentheses (“(”, “)”) can be used to override the standard precedences.

1. **not**, − (unary minus),
2. **msec**, **sec**, **min**, **hour**, day,
3. ⋆, /, **mod**,
4. +, − (subtraction),
5. <, <=, >, >=, =, !=,
6. **and**,
7. **or**, **xor**, **xnor**,
8. **iff**,
9. **imp**,
10. **case**.

Syntactic Restrictions

D-1 The operator tables define the allowed typing of expressions.

D-2 (Sub)Expressions involving **clock** or **continuous** data types must be linear. Thus, expressions involving the multiplication or division of two **clock** or **continuous** sub-expressions are not allowed, nor expressions where the denominator of a division is of type **clock** or **continuous**.

5 Syntax of Component Specifications

5.1 Component Types

A *component type declaration* gives the *category* of the component and establishes its externally visible characteristics, against which other components can operate. Each implementation of the component is required to satisfy this declaration. Interface *features* are ports along which information can be exchanged between components, and (private or public) keys for encryption and authentication purposes.

The following *software component categories* are supported:

- *process*: represents a software entity that can be bound to a processor and a memory component, can contain threads;
- *thread*: a unit of sequential execution which must be the subcomponent of a process; and
- *data*: data types, both for discrete (booleans, enumerations, (ranges of) integers, security keys, reals) and for analogue (clocks, continuous real-valued variables) value updates.

The following *hardware component categories* are supported:

- *processor*: executes processes and threads;
- *memory*: stores data and code;
- *device*: interfaces with or represents the external environment;
- *bus*: connects hardware components; and
- *network*: connects composite components.

The following *composite component categories* are supported:

- *node*: identifies a network node;
- *subject*: identifies a subject in the system's policy architecture; and
- *system*: represents subsystems or complete systems.

Component types of category **`data`** cannot be declared. Rather, abbreviating names for data types have to be introduced as constants (cf. Section 4.4).

ComponentType
::= *ComponentCategory ComponentTypeIdentifier*
[**`features`** *ComponentFeatures*]
`end` *ComponentTypeIdentifier*`;`
ComponentTypeIdentifier
::= *identifier*
ComponentCategory
::= *SoftwareCategory* | *HardwareCategory* | *CompositeCategory*
SoftwareCategory

::= **data** | **process** | **thread**

HardwareCategory

::= **bus** | **device** | **memory** | **network** | **processor**

CompositeCategory

::= **node** | **subject** | **system**

ComponentFeatures

::= {*PortDeclaration*}$^+$

Syntactic Restrictions

E-1 All component type identifiers declared within a scope have to be distinct.

E-2 Component types of category **data** cannot be declared.

E-3 The first component type identifier has to match the end component type identifier.

5.1.1 Port Declarations

A *port* represents a communication interface for the directional exchange of *event* and/or *data* information between components. Ports are classified as

- **event port**: interfaces for the *instantaneous* (i.e., message-like) communication of events raised by threads, processes, devices, or composite components; or
- **data port**: interfaces for the *continuous* transmission of typed state data among components; or
- **event data port**: interfaces for *instantaneous* communication of typed state data among components.

Ports are directional: an **in** (**out**) port represents a component's input (output). Data ports of an analogue type (that is, **clock** or **continuous**) are not supported. Not all component categories feature ports; Section 7 gives the corresponding restrictions.

Note that the classification of ports according to their communication type and their input/output behaviour allows, to a certain extent, to derive the characteristics of the respective component:

- *data-oriented* components only offer data ports;
- *control-oriented* components only offer event ports;
- *data/control-oriented* components offer both data and event ports;
- *passive* components only offer input ports;
- *active* components only offer output ports; and
- *reactive* components offer both input and output ports (although from the type declaration it is not deducible whether the output really depends on the input).

PortDeclaration
::= *EventPortDeclaration* | *DataPortDeclaration* | *EventDataPortDeclaration*

EventPortDeclaration
::= *EventPortIdentifier* **:** *Direction* **event port;**

DataPortDeclaration
::= *DataPortIdentifier* **:** *Direction* **data port** *DiscreteDataType* [*DefaultValue*] **;**

EventDataPortDeclaration
::= *EventPortIdentifier* : *Direction* **event data port** *DiscreteDataType* **;**

EventPortIdentifier
::= *identifier*

DataPortIdentifier
::= *identifier*

Direction
::= **in** | **out**

DefaultValue
::= **default** *DefaultConstantValue*

DefaultConstantValue
::= **true** | **false** | *EnumIdentifier* | *integerNumeral* | *realNumeral* |
(*DefaultConstantValue* { , *DefaultConstantValue* }$^{+}$) | *ValueConstantClassifier* |
FunctionConstantClassifier (*DefaultConstantValue*) |
DefaultConstantValue [*DefaultConstantValue*] |
KeySelector

KeySelector
::= { **pub** (*KeyPairIdentifier*) | **priv** (*KeyPairIdentifier*) }

Example 3 The following listing declares a thread component type for the input part of a simple cruise control with two incoming event ports (`engage`, `brake`), an incoming real-valued data port (`speed`), and an outgoing integer-valued data port (`control`).

```
thread Input
  features
    engage: in event port;
    brake: in event port;
    speed: in data port real;
    control: out data port int default 0;
end Input;
```

Syntactic Restrictions

F-1 Section 7 specifies which component categories support the declaration of ports.

F-2 All (event or data) port identifiers declared within a component type have to be distinct.

F-3 The type of each default value, if given, must match the data type of the respective data port.

F-4 The symbolic identifiers of all enum types used in a component type declaration have to be distinct from the data port identifiers declared in that component type.

5.2 Component Implementations

A *component implementation* gives the internal structure of a component in terms of subcomponents and their interaction via (event and data) connections. Moreover it is possible for a non-data component to define its (abstract) nominal behaviour by defining flows between incoming and outgoing data ports, and by assigning modes and mode transitions. Implementations (and types) of components of category **`data`** cannot be specified.

ComponentImplementation
::= *ComponentCategory* **`implementation`** *ComponentImplementationName*
[**`subcomponents`** *Subcomponents*]
[**`connections`** *EventPortConnections*]
[**`flows`** *DataFlows*]
[**`modes`** *Modes* [**`transitions`** *ModeTransitions*]]
`end` *ComponentImplementationName*`;`

ComponentImplementationName
::= *ComponentTypeIdentifier*`.`*ComponentImplementationIdentifier*

ComponentImplementationIdentifier
::= *identifier*

Syntactic Restrictions

G-1 All component implementation names declared within a scope have to be distinct.

G-2 The component type identifier of each declared component implementation name must refer to a component type that is declared within the same scope.

G-3 The component category given in the component implementation must match the component category as specified in the corresponding component type.

G-4 Component implementations of category **`data`** cannot be specified.

G-5 The first component implementation name has to match the end component implementation name.

5.2.1 Subcomponents and Their Physical Bindings

Components may be hierarchically decomposed into collections of interacting *subcomponents*. This is specified in the **subcomponents** part of the component implementation where the implementation structure of the component is defined as an assembly of subcomponent implementations. If a component c introduces a subcomponent c', then c is called the *supercomponent* of c'. The subcomponents of a common supercomponent are called *neighbour components*. If a subcomponent has only one associated implementation, then it suffices to give its type in the subcomponent declaration.

For **data** subcomponents, the data types described in Section 4.3 are admitted. If a component contains only **data** (or no) subcomponents, it is called *atomic*, otherwise *non-atomic*.

For all data subcomponents except those of type **clock** and (**private** or **public**) **key**, default values have to be defined. Clocks are different from usual data elements as their access is limited: they are always initialized with a zero value, may only be compared to a constant, and reset to zero. Similar restrictions apply to continuous data components. Details are explained in Section 5.2.4. Data subcomponents of type (**private** or **public**) **key** are used for exchanging keys and can receive their values by assignments of incoming data ports of the same type. Default values for these keys are allowed but not required.

Moreover physical *bindings* between the subcomponents can be established. The following kinds of bindings are supported:

- a process is **stored in** memory and **running on** a processor;
- a bus, memory, or processor component **accesses** a bus;
- a network component **accesses** a network; and
- a device or composite component **accesses** a bus or a network.

The activation of a subcomponent can depend on the mode of the component, thus supporting the specification of different system configurations at runtime. See Section 5.2.4 for details.

Subcomponents
::= *Subcomponent*$^+$
Subcomponent
::= *DataSubcomponent* | *OtherSubcomponent*
DataSubcomponent
::= *SubcomponentIdentifier* **:** **data** *DataType* [*DefaultValue*] [*InModes*] [*TimeUnit*] **;**
SubcomponentIdentifier
::= *identifier*
OtherSubcomponent
::= *SubcomponentIdentifier* **:** *ComponentCategory ComponentClassifier*
[*Bindings*] [*InModes*] **;**
ComponentClassifier
::= [*PackageIdentifier* **::**] *ComponentTypeIdentifier*[**.** *ComponentImplementationIdentifier*]

Bindings
::= *Binding*$^+$
Binding
::= {**stored in** | **running on** | **accesses**} *SubcomponentIdentifier*

Example 4 1. A simple model of a computer system with a CPU running a process with a contained thread, a memory component storing the process, and a bus connecting CPU and memory. (For simplicity, no connections are established, and no internal behaviour is defined.)

```
system Computer
end Computer;
system implementation Computer.Impl
  subcomponents
    CPU: processor Processor.Impl accesses BUS;
    MEM: memory Memory.Impl accesses BUS;
    BUS: bus Bus.Impl;
    PRC: process Process.Impl stored in MEM running on CPU;
end Computer.Impl;

processor Processor
end Processor;
processor implementation Processor.Impl
end Processor.Impl;

memory Memory
end Memory;
memory implementation Memory.Impl
end Memory.Impl;

bus Bus
end Bus;
bus implementation Bus.Impl
end Bus.Impl;

process Process
end Process;
process implementation Process.Impl
  subcomponents
    THR: thread Thread.Impl;
end Process.Impl;

thread Thread
end Thread;
thread implementation Thread.Impl
```

```
end Thread.Impl;
```

2. The next specification demonstrates that **system** components can also be employed to model abstract activities, without determining their later realisation in hard- or software. It defines a composite system with two concurrent activities.

```
system Abstract
end Abstract;
system implementation Abstract.Impl
  subcomponents
    act1: system Activity.Impl1;
    act2: system Activity.Impl2;
end Abstract.Impl;

system Activity
  -- type description
end Activity;
system implementation Activity.Impl1
  -- implementation description
end Activity.Impl1;
system implementation Activity.Impl2
  -- implementation description
end Activity.Impl2;
```

3. The following specification demonstrates the use of encryption keys. It additionally employs the features of subcomponent declaration and data flows, which will be introduced later in Section 5.2.3.

   The system is essentially taken from [1]. It implements a simple cryptographic controller where information flows from a red (secure) network to a black (insecure) one, and where the payload is encrypted before using the public key `mykey`.

```
--
-- Crypto controller example
--
-- Properties to be verified:
-- outgoing header is identical to incoming header:
--   fst(root.outframe) = fst(root.inframe)
-- outgoing payload is encryption of incoming payload:
--   snd(root.outframe) = encrypt(snd(root.inframe), mykey)
--
constants
  Frame: type := (Header, Payload);
  Header: type := int;
  Payload: type := int;
  MyKeys: asymmetric key pair;
```

```
--
-- Overall system with four components connected by channels
--
system CryptoController
  features
    inframe: in data port Frame;
    outframe: out data port Frame;
end CryptoController;
system implementation CryptoController.Imp
  subcomponents
    red: node Splitter.Imp accesses channels;
    bypass: node Bypass.Imp accesses channels;
    crypto: node Crypto.Imp accesses channels;
    black: node Merger.Imp accesses channels;
    channels: network CryptoNet.Imp;
  flows
    port inframe -> red.frame;
    port red.header -> bypass.inheader;
    port red.payload -> crypto.inpayload;
    port bypass.outheader -> black.header;
    port crypto.outpayload -> black.payload;
    port black.frame -> outframe;
end CryptoController.Imp;

--
-- Splitter component for decomposing frames into header and payload
--
node Splitter
  features
    frame: in data port Frame;
    header: out data port Header;
    payload: out data port Payload;
end Splitter;
node implementation Splitter.Imp
  flows
    port fst(frame) -> header;
    port snd(frame) -> payload;
end Splitter.Imp;

--
-- Bypass component for forwarding header
--
node Bypass
```

```
  features
    inheader: in data port Header;
    outheader: out data port Header;
end Bypass;
node implementation Bypass.Imp
  flows
    port inheader -> outheader;
end Bypass.Imp;

--
-- Crypto component for encrypting payload
--
node Crypto
  features
    inpayload: in data port Payload;
    outpayload: out data port Payload;
end Crypto;
node implementation Crypto.Imp
  constants
    mykey: public key default pub(MyKeys);
  flows
    port encrypt(inpayload, mykey) -> outpayload;
end Crypto.Imp;

--
-- Merger component for re-assembling frames from header and payload
--
node Merger
  features
    header: in data port Header;
    payload: in data port Payload;
    frame: out data port Frame;
end Merger;
node implementation Merger.Imp
  flows
    port (header, payload) -> frame;
end Merger.Imp;

network CryptoNet
end CryptoNet;
network implementation CryptoNet.Imp
end CryptoNet.Imp;
```

Syntactic Restrictions

H-1 All declared subcomponent identifiers have to be distinct.

H-2 In the second type of subcomponent declaration (*OtherSubcomponent*), *ComponentCategory* must be different from **data**, and *ComponentClassifier* must refer to the name of a component implementation. If the corresponding component type has only one implementation, then it suffices to give the type identifier.

H-3 The component category given in a subcomponent declaration must match the component category as specified in the corresponding component implementation.

H-4 Section 7 specifies the possible subcomponent categories for each component category.

H-5 For all data subcomponents except those of type **clock** and (**private** or **public**) **key**, default values have to be defined.

H-6 Clocks must not be assigned default values.

H-7 Every default value must match the data type of the respective subcomponent.

H-8 The binding **stored in** can only be specified for a **process** subcomponent and must refer to a declared **memory** subcomponent,

H-9 The binding **running on** can only be specified for a **process** subcomponent and must refer to a declared **processor** subcomponent,

H-10 The binding **accesses** is subject to the restrictions as specified in Section 7. In case of a bus-to-bus or network-to-network binding, the referred component has to be different from the referring one.

H-11 Whenever a subcomponent c_1 is bound to a subcomponent c_2, c_2 has to be active in each mode where c_1 is active.

H-12 If given, all identifiers in an **in modes** list must be distinct, and must refer to modes as declared in the **modes** part of the component implementation (see Section 5.2.4).

H-13 The component hierarchy must not be recursive, that is, no component implementation can have itself as an (indirect) subcomponent.

H-14 For each component implementation, the identifiers of all data subcomponents have to be distinct from all data port and key identifiers declared in the corresponding type specification.

H-15 The value identifiers of all **enum** types used in a component type or implementation have to be distinct from all data port, key, and data subcomponent identifiers of that component.

5.2.2 Event Port Connections

Representations of the interactions among components are restricted to defined relations established between interface elements, i.e., event ports, data ports and event data ports (see Section 5.1.1). With regard to the first two kinds of ports, *event port connections* and *event data port connections* establish directed interactions between the event ports of components. We distinguish the following types of connections:

- *in-to-in*: from an incoming event port of a component to an incoming event port of one of its subcomponents,
- *out-to-out*: from an outgoing event port of a component to an outgoing event port of its supercomponent, and
- *out-to-in*: from an outgoing event port of a component to an incoming event port of one its neighbour components.

Note that this excludes in-to-out connections and connections from a component to itself.

Moreover, not all component categories support ports (cf. Section 7). Additionally, introducing out-to-in connections between hardware and composite components requires a physical coupling between those components, which further restricts the possible topology of such connections: they can only be established between neighbouring devices or composite components which are accessing a common bus or network.

The presence of a connection can depend on the mode of the component, thus supporting the specification of different connection topologies at runtime. If the **`in modes`** clause is not present, the connection is implicitly declared to be active in all modes of the respective component. See Section 5.2.4 for details.

Event ports and event data ports support *fan-in* and *fan-out*, that is, the same event port can respectively be the target and source of several connections (even in the case where these are simultaneously active). Fan-out and fan-in is not supported by all verification tools and will therefore yield a warning. In the case of fan-out, it is not allowed to have pairs of connections that go from the same event port of one component to (different) event ports of another component, and that are active in the same mode. More concretely, this excludes the following cases:

- Declarations of in-to-in connections of the form

  `port` p `->` $c.p_1$ **`in modes`** $(\ldots, m, \ldots)$ `;` $\ldots$ `;`
  `port` p `->` $c.p_2$ **`in modes`** $(\ldots, m, \ldots)$ `;`

  where p is an incoming event port of the current component, and c is a subcomponent with incoming event ports p_1 and p_2;
- declarations of out-to-in connections of the form

  `port` $c.p$ `->` $c'.p_1$ **`in modes`** $(\ldots, m, \ldots)$ `;` $\ldots$ `;`
  `port` $c.p$ `->` $c'.p_2$ **`in modes`** $(\ldots, m, \ldots)$ `;`

  where c and c' are (different) subcomponents of the current component with outgoing event port p and incoming event ports p_1 and p_2, respectively; and

- declarations of out-to-out connections of the form

  port $c.p$ -> p_1 **in modes** ($\ldots, m, \ldots$) ; $\ldots$;
  port $c.p$ -> p_2 **in modes** ($\ldots, m, \ldots$) ;

  where c is a subcomponent with outgoing event port p, and p_1 and p_2 are outgoing event ports of the current component.

EventPortConnections
::= (*EventPortConnection* | *EventDataPortConnection*)$^+$
EventPortConnection
::= **port** *EventPortReference* -> *EventPortReference* [*InModes*] ;
EventPortReference
::= [*SubcomponentIdentifier* .] *EventPortIdentifier*
EventPortConnection
::= **port** *EventDataPortReference* -> *EventDataPortReference* [*InModes*] ;
EventDataPortReference
::= [*SubcomponentIdentifier* .] *EventDataPortIdentifier*

Example 5 The following specification extends Example 3. It reuses the type specification of the `Input` thread, and adds another thread `Output` which converts (integer) `control` data into (real) `throttle` data. Both are embedded in a common process `Cruise` which forwards input (`engage`, `brake`, `speed`) and output (`throttle`) information to and from the two threads, respectively.

```
process Cruise
  features
    engage: in event port;
    brake: in event port;
    speed: in data port real default 0.0;
    throttle: out data port real default 0.0;
end Cruise;
process implementation Cruise.Impl
  subcomponents
    input: thread Input.Impl;
    output: thread Output.Impl;
  connections
    port engage -> input.engage;
    port brake -> input.brake;
  flows
    port speed -> input.speed;
```

```
    port input.control -> output.control;
    port output.throttle -> throttle;
end Cruise.Impl;

thread Input
  features
    engage: in event port;
    brake: in event port;
    speed: in data port real;
    control: out data port int default 0;
end Input;
thread implementation Input.Impl
end Input.Impl;

thread Output
  features
    control: in data port int;
    throttle: out data port real default 0.0;
end Output;
thread implementation Output.Impl
end Output.Impl;
```

Syntactic Restrictions

I-1 For each event port connection, the subcomponent identifiers referred in that connection must exist in the component implementation in which the port connection is defined.

I-2 For each event port connection, both the source and the target port must be declared in the respective component type as event ports, and the source and the target component must be different.

I-3 The topological restrictions regarding out-to-in connections as specified in the beginning of this section apply.

I-4 If given, all identifiers in the **in modes** list must be distinct, and must refer to modes as declared in the **modes** part of the component implementation (see Section 5.2.4).

I-5 For each mode in which a connection is active, the subcomponents that are referenced in the source and the target port must also be active in that mode.

I-6 Fan-out of event ports to the same component is disallowed, meaning pairs of jointly active connections that go from the same outgoing event port of one component to (different) event ports of another component.

5.2.3 Data Flows

So far, the values of data ports can only be defined by introducing defaults (see Section 5.1.1). Later in Section 5.2.4 we will see how to describe more complex computations using assignments in mode transitions. These, however, may cause problems that can arise from the delayed execution of transitions due to interleaving of concurrent activities etc.

To avoid the necessity to introduce additional control features in order to guarantee the value consistency between ports, our language supports the declaration of *flows*. A flow introduces a value update of a port as an immediate reaction to an update of (one or more) other ports. More concretely, flows are specified similarly to event port connections where the source part is now an expression over

- incoming data ports of the current component and
- outgoing data ports of its subcomponents

and the target part is

- an outgoing data port of the current component or
- an incoming data port of one of its subcomponents

such that the type of the source expression equals that of the target port.

Thus data flows generalise event port connections in several ways as they

- support in-to-out dependencies (from an incoming to an outgoing data port of the current component),
- allow a target data port to depend on more than one source data port, and
- can modify forwarded values rather than just copy them.

On the other hand, similar restrictions regarding the physical coupling between hardware and composite components apply. Flows from outgoing to incoming data ports can only be established between neighbouring devices or composite components which are accessing a common bus or network.

Just like an event port connection, the presence of a flow can be made dependent on the mode of the component using an **`in modes`** clause. If it is absent, the flow is implicitly declared to be active in all modes of the component. Another similarity is the support for fan-out: the same data port is allowed to occur in the source expression of several flows. On the other hand, every (incoming or outgoing) data port is restricted to a single incoming flow in each mode. (It cannot have a "fan-in" from different sources as its value would not be well-defined in this case.) As for event and event-data ports, not all verification tools support fan-out, so using fan-out triggers a warning.

DataFlows
::= *DataFlow*$^{+}$
DataFlow
::= **`port`** *FlowExpression* -> *DataPortReference* [*InModes*]**`;`**

FlowExpression
::= *DataPortReference* | *ConstantValue* | *operator*(*FlowExpression*, . . . , *FlowExpression*)
DataPortReference
::= [*SubcomponentIdentifier* **.**] *DataPortIdentifier*

Here the terminal symbol *operator* refers to the built-in operators as introduced in Section 4.5.

Example 6 In the following simple adder component, a value update of one of the input ports immediately causes an update of the associated output port. This way, the consistency between input and output ports is always guaranteed.

```
device Adder
  features
    input1: in data port bool default true;
    input2: in data port bool default true;
    output: out data port (bool, bool);
end Adder;
device implementation Adder.Impl
  flows
    port (input1 and input2, input1 xor input2) -> output;
end Adder.Impl;
```

Syntactic Restrictions

J-1 For each data flow, the subcomponent identifiers referred in that flow must exist in the component implementation in which the port connection is defined.

J-2 The source part of each flow must a well-typed expression over incoming data ports of the respective component and outgoing data ports of its subcomponents.

J-3 The target port of each flow must be an outgoing data port of the respective component or an incoming data port of one of its subcomponents.

J-4 For each flow, the type of the source expression must be equal to that of the target port.

J-5 If given, all identifiers in the **in modes** list must be distinct, and must refer to modes as declared in the **modes** part of the component implementation (see Section 5.2.4).

J-6 For each mode in which a flow is active, the subcomponents that are referenced in the source and the target port must also be active in that mode.

J-7 In each component implementation, an outgoing data port is allowed to occur either on the left-hand side of mode transition assignments (Section 5.2.4) or as a target port in flows, but not both.

J-8 For each mode of a component, every data port is allowed to occur at most once as a target port in all flows that are active in that mode (no fan-in for data ports.)

J-9 To exclude undefined values of an incoming data port p of a component c, in each mode of every component that uses c as a subcomponent, p must occur as the target port of a flow, or it must have a default value. In particular, every incoming data port of the root component must have a default value.

J-10 To exclude undefined values of an outgoing data port p that occurs on the left-hand side of mode transition assignments or as a target of flows in some but not all modes of that component, p must have a default value.

J-11 The union of the dependency relations between all data ports of the overall system which is imposed by data flows over all modes of each component has to be acyclic.

5.2.4 Modes and Mode Transitions

Modes In our modelling approach, *modes* can be attached to non-data components. They serve two purposes:

- They provide an abstraction of the concrete *behaviour* of a component, constituting an automata-like formalism for modelling the finite control part of components. This applies, in particular, to atomic components.
- They can be employed to specify different *system configurations* and *connection topologies* at runtime, allowing to model the mode-dependent activation of subcomponents and their connections and flows within the context of a non-atomic component in response to external events.

Each modal specification must give exactly one **`initial`** mode. The component starts its execution in that mode only in the beginning of system execution, that is, when the component is activated for the first time. After its de- and re-activation (see the explanations on mode-dependent activation in the next section), execution is resumed in the previous mode, that is, *mode history* is supported. Data subcomponents and outgoing data ports of the component are handled in a similar way: their values are preserved during deactivation. Specifically, the mode and data values are not reset to their initial values after re-activation. This resetting behavior should be modeled explicitly.

Formally, the mode behavior of a component is specified by a *hybrid automaton* which operates on the **`data`** elements that are declared in the **`subcomponents`** part of the component's implementation, and on the outgoing data ports that are declared in the **`features`** part of the component's type. Here **`clock`** data components are different from the usual ones as their access is limited: they may only be reset to zero. After reset, they start increasing their value implicitly as time progresses. All clocks in the system proceed at the same linear rate. The value of a clock component therefore denotes the amount of time that has elapsed since its last reset. Thus, clocks can be considered as *timers*.

Linear conditions on the values of the clocks (including time units) can be attached to both modes and mode transitions. In the first case, they act as *mode invariants* which constrain the amount of time that may be spent in a location. In the second case, they represent *mode transition guards* that enable or disable a transition (see below).

Similar restrictions apply to **`continuous`** data components. Using linear expressions, their values may be tested in invariants and guards, and they may be reset to a constant. The specification of their dynamic behaviour, however, allows more freedom: it is given by a *trajectory equation*, that is, a differential equation of the form $\dot{x} = a$ (where $\dot{x}$ denotes the derivation with respect to time and $a \in \mathbb{R}$ is a real-valued constant) which is again attached as an invariant to a mode. (Syntactically, $\dot{x}$ is represented by an apostrophe: `x'`.)

References to other data elements, that is, data ports and discrete-type data subcomponents of a component are disallowed in invariants. If the invariant is absent, then it is assumed to be constantly **`true`**. Note that clocks can be considered as special continuous components which obey the trajectory equation $\dot{x} = 1$, and which can only be reset to zero.

Mode Transitions *Transitions* between modes are of the general form

$$m_1 \text{ -[} e(v) \textbf{ when } g \textbf{ then } f \text{]-> } m_2$$

where m_1 and m_2 are modes, e is a trigger event for the transition, v is the event data, g is a guard, and f is an effect. Each of the trigger, event data, the guard, and the effect can be omitted. The event data only applies to events over **`event data`** ports. For incoming trigger events with data, v is a variable; for outgoing ones, v is an arbitrary expression.

Transitions can

- be externally *triggered* by incoming events arriving at ports, or
- occur *spontaneously*, either with the generation of an output event that triggers input events in other components or as an *invisible* (i.e., event-free) transition.

A *guard* is a logical expression over data subcomponents and data ports of the respective component which restricts the enabledness of a transition. In particular, clock and continuous components can be referred to in guards. However, the latter are only occurred to allow in linear expressions. The same applies to real- and integer-valued data elements.

An *effect* defines the impact of the transition by specifying update operations for the values of data subcomponents and outgoing data ports. In particular, *clock resets* are possible updates, that is, assignments of the form $c := 0[.0]$ where c denotes a clock component. Another possible form of assignment is $d := a$ where d denotes a continuous data component and $a \in \mathbb{R}$. Other assignments to clock or continuous components are not allowed. Additionally, for tuples, it is allowed to assign a new value at a particular index of the form $t[i] := v$, where t is a tuple of type $(\tau_1, \ldots, \tau_n)$, $1 \leq i \leq n$ and v is of type τ_i. Please see Section 4.5 for an overview of supported operators. Moreover it must be guaranteed that every data element occurring on the right-hand side of an assignment has a defined value, and that the result type of the right-hand side matches the type of the left-hand side, with the following exceptions:

- expressions of type **`clock`** or **`continuous`** can also be assigned to data elements of type **`real`** and
- expressions of an integer range type can be assigned to data elements of another integer range type, involving a modulo operation of the value of the former exceeds the range of the latter. More exactly, the assignment of a value $z \in \mathbb{Z}$ to a data element d of range type `[`l`..`u`]` yields the new value $(z - l) \bmod (u - l + 1) + l$ for d.

The resulting combinations of types are listed in Table 9. Here τ denotes any discrete data type other than **real**, and $RNum$ abbreviates $\{$**clock**, **continuous**, **real**$\}$.

Operator	Type	Meaning
:=	$\tau \to \tau$	Discrete assignment
	$\{0, 0.0\} \to$ **clock**	Clock reset
	$\mathbb{R} \to$ **continuous**	Continuous constant assignment
	$RNum \to$ **real**	Real assignment
	$(\tau_1, \ldots, \tau_n) \times$ **int** $\to \tau_i$	Tuple index assignment at index i

Table 9: Assignment Operator

If the **when** or the **then** clause are absent, then the guard is assumed to be of the form **true**, and the effect is assumed to be the empty list of assignments, respectively.

For semantic reasons it is not possible to give an output event as an effect. If an incoming event triggers an outgoing event, then this has to be modelled as two separate transitions: the first is triggered by the incoming event and leads to an intermediate mode in which (only) the outgoing event can be emitted.

Note that guards only *restrict* the possible transitions but do not *force* them to be taken. The only way to specify this kind of behaviour is by employing mode invariants.

Mode-dependent activation of subsystems is supported by making the presence of subcomponents (Section 5.2.1), event port connections (Section 5.2.2) and data flows (Section 5.2.3) dependent on the current mode (see nonterminal symbol *InModes*). Whenever a mode transition adds or removes a subcomponent, connection or flow, we say that the former *activates* or *deactivates* the latter. The second specification in Example 7 presents an application of this kind.

Moreover this concept allows to define the fine-grained behaviour of a component *in a certain mode* by using nested modes, as illustrated by the third specification in Example 7.

Modes
::= *Mode*$^+$
Mode
::= *ModeIdentifier*`:` [**initial**] **mode** [**while** *Invariant*]`;`
ModeIdentifier
::= *identifier*
Invariant
::= *TrajectoryEquation* | *Expression* | *Invariant* **and** *Invariant*
TrajectoryEquation
::= *SubcomponentIdentifier*`'` `=` *ConstantValue* [**per** *TimeUnit*]
ComparisonOperator
::= < | > | <= | >=

TimeUnit
::= `day` | **`hour`** | **`min`** | **`sec`** | **`msec`**

ModeTransitions
::= *ModeTransition*$^{+}$

ModeTransition
::= *ModeIdentifier* `-[` [*Trigger*] [**`when`** *Guard*] [**`then`** *Effect*] `]->` *ModeIdentifier*`;`

Trigger
::= *EventPortReference* [`(` *Expression* `)`]

Guard
::= *Expression*

Effect
::= *Assignment* {`;` *Assignment*}*

Assignment
::= (*DataPortReference* | *SubcomponentIdentifier*) `:=` *Expression*

Expression
::= *DataPortReference* | *SubcomponentIdentifier* | *ConstantValue* |
operator(*Expression*, . . . , *Expression*)

InModes
::= **`in modes`** (*ModeIdentifier* {`;` *ModeIdentifier*}*)

Here the terminal symbol *operator* refers to the built-in operators as introduced in Section 4.5. The optional argument for the *Trigger* must only be given for event data ports, and composite expressions are only allowed for outgoing event data ports. For an incoming event data port, *Expression* in the definition of *Trigger* must be the identifier of a data subcomponent or outgoing data port of the same type as the event data.

Example 7 1. The following specification extends the `Input` thread given in Example 5 by introducing two modes, `idle` and `enabled`. The thread is initially in the first mode. It switches to `enabled` when the cruise control is engaged, and then compares the current speed from the incoming data port `speed` to the nominal speed which is kept in local data component `set`. Depending on the outcome of this comparison, the value of outgoing data port `control` is defined. The thread returns to `idle` mode whenever the brake is used, or more than `timeout` time units have passed.

```
constants
  timeout: int := 100;

thread Input
  features
    engage: in event port;
    brake: in event port;
```

```
    speed: in data port real default 0.0;
    control: out data port [-1..1] default 0;
end Input;

thread implementation Input.Impl
  subcomponents
    timer: data clock;
    set: data real default 50.0;
  modes
    idle: initial mode;
    enabled: mode while timer <= timeout min;
  transitions
    idle -[engage then timer := 0]-> enabled;
    enabled -[when speed < set then control := 1]-> enabled;
    enabled -[when speed > set then control := -1]-> enabled;
    enabled -[brake]-> idle;
    enabled -[when timer >= timeout min]-> idle;
end Input.Impl;
```

2. The next example shows a redundant, self-configuring system with two CPUs. A monitor process supervises their work and initiates a switch to the alternative configuration if necessary. Initially, the system is started in `primary` mode. Later, the second specification in Example 8 will show a possible implementation of the monitor process.

```
system Redundant
end Redundant;
system implementation Redundant.Impl
  subcomponents
    cpu1: processor CPU.Impl in modes (primary);
    cpu2: processor CPU.Impl in modes (backup);
    mon: process Monitor.Impl;
  modes
    primary: initial mode;
    backup: mode;
  transitions
    primary -[mon.switch]-> backup;
    backup -[mon.switch]-> primary;
end Redundant.Impl;

process Monitor
  features
    switch: out event port;
end Monitor;
process implementation Monitor.Impl
  -- ...
```

```
end Monitor.Impl;

processor CPU
end CPU;
processor implementation CPU.Impl
end CPU.Impl;
```

3. It is possible to model the behavior of a supercomponent in a certain mode by defining nested modes. The following specification gives a system with two modes `m` and `n`, having nested submodes `m_0`, ..., `m_k` and `n_0`, ..., `n_l`, respectively.

```
system super
end super;

system implementation super.impl
  subcomponents
    s: system sub.impl1 in modes (m);
    t: system sub.impl2 in modes (n);
  modes
    m: initial mode;
    n: mode;
  -- transitions...
end super.impl;

system sub
end sub;

system implementation sub.impl1
  modes
    m_0: initial mode;
    m_1: mode;
    -- more modes...
    m_k: mode;
  -- transitions...
end sub.impl1;

system implementation sub.impl2
  modes
    n_0: initial mode;
    n_1: mode;
    -- more modes...
    n_l: mode;
  -- transitions...
end sub.impl2;
```

4. The next specification, borrowed from [10], models a thermostat. The continuous variable `temp` represents the temperature. In mode `Off` (`On`), the heater is switched off (on), and the temperature falls (rises) according to the trajectory equation $\dot{t} = -0.1$ ($\dot{t} = 2$), where t is measured in minutes. Initially, the heater is off and the temperature is 20 (degrees). The invariant of mode `Off` and the guard of the first transition express that the heater may go on as soon as the temperature falls below 19 (degrees), and at the latest it will go on when the temperature falls to 18 (degrees).

```
device Thermostat
end Thermostat;

device implementation Thermostat.Impl
  subcomponents
    temp: data continuous default 20.0;
  modes
    Off: initial mode while temp' = -0.1 per min and temp >= 18.0;
    On: mode while temp' = 2.0 per min and temp <= 22.0;
  transitions
    Off -[when temp < 19.0]-> On;
    On -[when temp > 21.0]-> Off;
end Thermostat.Impl;
```

Syntactic Restrictions

K-1 Section 7 specifies which subcomponent categories support modes.

K-2 All declared mode identifiers have to be distinct.

K-3 Exactly one mode (the starting mode) has to be distinguished as **`initial`** mode.

K-4 Each mode in the specification must be (syntactically) reachable by a sequence of transitions from the starting mode.

K-5 The data subcomponents referred to by an invariant must be active in the mode for which that invariant is defined.

K-6 Clock invariants must only refer to **`clock`** subcomponents.

K-7 Trajectory equations and continuous invariants must only refer to **`continuous`** subcomponents.

K-8 Time units must and may only be used in comparisons involving **`clock`** data subcomponents.

K-9 For each **`continuous`** data component, at most one trajectory equation may be given in each mode.

K-10 Default values for data subcomponents, if given, must satisfy the invariant of the starting mode.

K-11 The source and target mode of a transition must both refer to (not necessarily distinct) modes of the current component.

K-12 Each mode transition trigger, if given, must be

- an (incoming or outgoing) event port of the current component or
- a name of the form $c.p$ where c refers to a subcomponent that is active in m and p is an event port of that subcomponent.

where m refers to the source mode of the respective transition.

K-13 The mode transition guard, if given, must be a well-typed **`bool`**-valued expression which only refers to data ports or data subcomponents that are active in the source mode of that transition. All subexpressions of type **`int`**, $Range$, **`real`**, **`clock`** and **`continuous`** must be linear.

K-14 The left-hand side of each assignment in a mode transition effect is a distinct data subcomponent that is active in the target mode of that transition or an outgoing data port of the current component.

K-15 The right-hand side of each assignment in a mode transition effect is a well-typed expression over data ports or data subcomponents of the current component that are active in the source mode of that transition.

K-16 The right-hand side of each assignment must be consistent with respect to the left-hand side, i.e., it must respect the typing rules given in Table 9.

K-17 (see J-7)

6 Syntax of Error Model Specifications

Error models can be used to annotate (non-data) component types and implementations to support safety and dependability analyses. The corresponding extension of the AADL language is defined in the Error Model Annex [13], on which also our error modelling formalism is based. For detailed explanations we refer to [9].

The error behavior of a complete system emerges from the combination of the individual component error models. Failing components can affect other components because the components interact or one component is a hardware resource that another component is bound to. Thus the system error model is a composition of the error models of its components where the composition is derived from the system hierarchy.

6.1 Error Model Types

Again, an error model is defined by its type and its implementation (variant). An *error model type* defines an interface in terms of (incoming and outgoing) error propagations. *Error propagations* are used to exchange error information between components. Errors can only be propagated between components which are physically connected, in the following sense (cf. Section 5.2.2 for similar restrictions regarding event port connections):

- from a hardware or composite component to a hardware or composite component if both access a common bus or network,
- from a bus or network to a hardware or composite component if the latter accesses the former,
- from a processor to a process if the process is running on that processor,
- from a memory component to a process if the process is stored in that component,
- from a non-atomic component to each of its direct (non-data) subcomponents, and
- from a subcomponent to its direct supercomponent.

ErrorType
::= **error model** *ErrorTypeIdentifier*
[**features** *ErrorFeatures*]
end *ErrorTypeIdentifier*;

ErrorTypeIdentifier
::= *identifier*

ErrorFeatures
::= *ErrorPropagation*$^{+}$

ErrorPropagation
::= *PropagationIdentifier*: *Direction* **propagation;**

PropagationIdentifier
::= *identifier*

Syntactic Restrictions

L-1 All error model type identifiers declared within a scope have to be distinct.

L-2 All error propagation identifiers declared within an error model type have to be distinct.

L-3 The first error type identifier has to match the end error type identifier.

6.2 Error Model Implementations

In the implementation(s) of an error model, *error states* are employed to represent the current configuration of the component with respect to errors. There can be a number of *error* states. Exactly one state is marked **initial**. The meaning of the initial state is similar to that of the initial mode(cf. Section 5.2.4). The error model is put in that state only in the beginning of system execution, supporting *error history* during deactivation phases.

The actual behavior of an error model implementation is given by a (probabilistic) machine operating on error states. *Transitions* between states can be triggered by error events or error propagations.

Error events are internal to the component; they reflect changes of the error state caused by local faults and repair operations. Moreover an *occurrence rates* can be attached to an error event, to model the probabilistic nature of faults. For semantic reasons, the transitions leaving a specific error state must either all be probabilistic, i.e., equipped with a rate, or none of them. Moreover, nondeterminism is not allowed in error model implementations, that is, the triggers of all transitions leaving a specific error state must be distinct.

Outgoing *error propagations* report an error state to other components. If their error states are affected, the other components will have an corresponding incoming propagation.

ErrorImplementation
::= **error model implementation** *ErrorImplementationName*
[**states** *ErrorStates*]
[**events** *ErrorEvents*]
[**transitions** *ErrorTransitions*]
end *ErrorImplementationName*;

ErrorImplementationName
::= *ErrorTypeIdentifier*.*ErrorImplementationIdentifier*

ErrorImplementationIdentifier
::= *identifier*

ErrorStates
::= *ErrorState*$^{+}$

ErrorState
::= *StateIdentifier*: [**initial**] **state**;

StateIdentifier
::= *identifier*
ErrorEvents
::= *ErrorEvent*$^{+}$
ErrorEvent
::= *EventIdentifier*`:` **`event`** [*ErrorRate*]`;`
EventIdentifier
::= *identifier*
ErrorRate
::= **`rate`** *ConstantValue* [**`per`** *TimeUnit*]
ErrorTransitions
::= *ErrorTransition*$^{+}$
ErrorTransition
::= *StateIdentifier* `-[` *ErrorTrigger* `]->` *StateIdentifier*`;`
ErrorTrigger
::= *EventIdentifier* | *PropagationIdentifier*

Example 8 1. The following specification presents an error model with transient and permanent faults (cf. [9, p. 11]). Initially, the component is in the `ErrorFree` state. A fault is either transient or permanent, both cases occurring with a certain rate (0.2 and 0.1 events per hour, respectively). In the first case, the error is expected to disappear within 30 seconds.

```
error model TransientPermanent
end TransientPermanent;

error model implementation TransientPermanent.General
  states
    ErrorFree: initial state;
    TransientError: state;
    PermanentError: state;
    Resetting: state;
  events
    TransientFault: event rate 0.2 per hour;
    PermanentFault: event rate 0.1 per hour;
    Works: event rate 2.0 per min;
  transitions
    ErrorFree -[TransientFault]-> TransientError;
    ErrorFree -[PermanentFault]-> PermanentError;
    TransientError -[Works]-> ErrorFree;
end TransientPermanent.General;
```

2. Now we extend the second specification from Example 7 by error models. Again we represent a redundant system with two CPUs, supervised by a monitor system. The failure of a CPU is reported by the `CPUfails` propagation, which leads to the `Failed` error state (see **`error model implementation`** `CPUError.Impl`). The propagation is catched by `MonError.Impl` and causes a transition to state `Failure`, upon which the **`system implementation`** `Monitor.Impl` reacts (using a fault injection which is not shown here) by going to mode `Change` in which event `Switch` is emitted. The latter initiates the system-level switch to the alternative configuration.

```
system Redundant
end Redundant;
system implementation Redundant.Impl
  subcomponents
    CPU1: processor CPU.Impl in modes (Primary);
    CPU2: processor CPU.Impl in modes (Backup);
    Mon: system Monitor.Impl;
  modes
    Primary: initial mode;
    Backup: mode;
  transitions
    Primary -[Mon.Switch]-> Backup;
    Backup -[Mon.Switch]-> Primary;
end Redundant.Impl;

system Monitor
  features
    Switch: out event port;
end Monitor;
system implementation Monitor.Impl
  subcomponents
    Failure: data bool default false;
  modes
    Stay: initial mode;
    Change: mode;
  transitions
    Stay -[when Failure]-> Change;
    Change -[Switch]-> Stay;
end Monitor.Impl;

processor CPU
end CPU;
processor implementation CPU.Impl
end CPU.Impl;

error model MonError
```

```
  features
    CPUfails: in propagation;
end MonError;
error model implementation MonError.Impl
  states
    OK: initial state;
    Failure: state;
  transitions
    OK -[CPUfails]-> Failure;
end MonError.Impl;

error model CPUError
  features
    CPUfails: out propagation;
end CPUError;
error model implementation CPUError.Impl
  states
    OK: initial state;
    Failed: state;
  transitions
    OK -[CPUfails]-> Failed;
end CPUError.Impl;
```

3. The following specification re-models the bit adder example from [2]. The `Main` system is composed of five subcomponents: an `Adder` component which takes its two inputs from `Bit` components, which in turn receive bit values from two `Random` components. Faults are introduced by associating error model `BitError` with system implementation `Bit.Impl`. In this error model, the occurrence of a fault causes a change to the `inverted` state, in which the value of the `output` port of `Bit` is inverted (which again has to be specified in the user interface).

```
system Main
  features
    output: out data port bool default true;
end Main;
system implementation Main.Impl
  subcomponents
    adder: system Adder accesses path;
    bit1: system Bit accesses path;
    bit2: system Bit accesses path;
    rnd1: system Random accesses path;
    rnd2: system Random accesses path;
    path: bus Bus.Impl;
  flows
    port rnd1.output -> bit1.input;
```

```
    port rnd2.output -> bit2.input;
    port bit1.output -> adder.input1;
    port bit2.output -> adder.input2;
    port adder.output -> output;
end Main.Impl;

system Adder
  features
    input1: in data port bool;
    input2: in data port bool;
    output: out data port bool default true;
end Adder;
system implementation Adder.Impl
  modes
    init: initial mode;
  transitions
    init -[then output := input1 xor input2]-> init;
end Adder.Impl;

system Bit
  features
    input: in data port bool;
    output: out data port bool default true;
end Bit;
system implementation Bit.Impl
  modes
    init: initial mode;
  transitions
    init -[then output := input]-> init;
end Bit.Impl;

system Random
  features
    output: out data port bool default true;
end Random;
system implementation Random.Impl
  modes
    init: initial mode;
  transitions
    init -[then output := true]-> init;
    init -[then output := false]-> init;
end Random.Impl;

bus Bus
```

```
end Bus;
bus implementation Bus.Impl
end Bus.Impl;

error model BitError
end BitError;
error model implementation BitError.Impl
  states
    normal: initial state;
    inverted: state;
  events
    fault: event;
  transitions
    normal -[fault]-> inverted;
end BitError.Impl;
```

Syntactic Restrictions

M-1 All error model implementation names declared within a scope have to be distinct.

M-2 The first error implementation name has to match the end error implementation name.

M-3 The first identifier of each error implementation name must refer to an error model type that is defined within the same scope.

M-4 All error state identifiers declared within an error model implementation have to be distinct.

M-5 Exactly one state must be marked as **`initial`**.

M-6 All error events declared within an error model implementation have to be distinct.

M-7 Each error occurrence rate, if given, must be a positive real-valued constant.

M-8 Both the source and the target state of an error transition must refer to (not necessarily distinct) states of the current error model. In particular, if the **`states`** clause is missing from the error implementation specification, then also the **`transitions`** clause has to be absent.

M-9 The error transition trigger must be

- an error event or
- an (incoming or outgoing) error propagation.

M-10 The transitions leaving a specific error state must either all be probabilistic or none of them.

M-11 The triggers of all transitions leaving a specific error state must be distinct.

M-12 Each state in the specification must be (syntactically) reachable by a sequence of transitions from the starting state.

Category	Type elements	Implementation elements	Bindings
Software categories			
`data`	—	—	—

`process`	`port`
	`key`
		`subcomponents`: `thread`, `data`
		`connections`
		`flows`
		`modes/transitions`
			`stored in`
			`running on`

`thread`	`port`
	`key`
		`subcomponents`: `data`
		`flows`
		`modes/transitions`
			—

Hardware categories			
`bus`	—	—	`accesses bus`

`device`	`port`
	`key`
		`subcomponents`: `data`
		`flows`
		`modes/transitions`
			`accesses bus/network`

`memory`	—	`subcomponents`: `memory`	`accesses bus`
`network`	—	—	`accesses network`
`processor`	—	—	`accesses bus`
Composite categories			

`node`	`port`
	`key`
		`subcomponents`:
		`data`, `process`, `bus`,
		`device`, `memory`,
		`processor`, `system`
		`connections`
		`flows`
		`modes/transitions`
			`accesses bus/network`

`system`	`port`
	`key`
		`subcomponents`:
		`data`, `process`, `bus`,
		`device`, `memory`,
		`processor`, `system`
		`connections`
		`flows`
		`modes/transitions`
			`accesses bus/network`

Table 10: Overview of Component Restrictions

7 Overview of Component Restrictions

To keep the presentation simple, the context-free grammar given in the previous chapter defines a superset of the specifications which are actually admitted. Table 10 gives an overview of the syntactic restrictions which are additionally imposed. Note that `data` components are not considered as they do not allow user-defined specifications of types and implementations.

Also note that, according to Section 5.2.2, connections can only be established between components which offer event ports, either between the subcomponents of a supercomponent or between the subcomponent and its supercomponent. Therefore, e.g., connections are supported for process im-

plementations (having threads with event ports as subcomponents), but not for threads (only **`data`** subcomponents, which do not have ports) or for processors (no subcomponents).

8 Property Specifications

This section sketches the basic properties that should be expressible to support the analysis of different behavioural aspects of a system in execution, such as functional correctness, safety, performance, and security.

Component Instances As properties will be addressed on the level of components, the first notation to be fixed is that for component instances. The hierarchical organisation of component specifications suggests a tree-structured representation where

- `root` stands for the top-level component of the system and
- whenever c is a component with a subcomponent identified by sc (cf. Section 5.2.1), $c.sc$ refers to that subcomponent.

For example, the `Crypto` component of the crypto-controller system as described in Example 4 is denoted by `root.crypto`.

Nominal Modes and Error States Basic properties can refer to the following configuration information of a component instance c.

- c`.mode` for representing the current nominal mode (see Section 5.2.4) of c and
- c`.error` for representing the current state of the error model (see Section 6.2) that is attached to c.

These denotations can be used in (in-)equations referring to the corresponding mode and error state identifiers, respectively. For example, the equation `root.mode = idle` can be used to test whether the `Input` thread from Example 7(1) is idle. Similarly, `root.CPU1.error = Failed` expresses that the first CPU in Example 8(2) has failed, assuming that error model `CPUerror` is attached to `CPU1`.

Data Elements Basic properties can refer to the following data elements of a component instance c.

- $c.p$ for representing the current value of data port p (see Section 5.1.1) of c and
- $c.d$ for representing the current value of data subcomponent d (see Section 5.2.1) of c.

These denotations can be used in (well-typed) relational expressions referring to constant values. For example, the expression

```
snd(root.outframe) = encrypt(snd(root.inframe), mykey)
```

checks that the crypto controller from Example 4 works as expected. Another example is given by the inequation `temp >= 20.0`, which can be used to test the temperature value for the thermostat system from Example 7(4).

Security properties The following security-related properties are considered basic properties:

- `knowsKey(`c`,`k`)` expresses that component c knows key k.

Information Flow

- `isIndependentOf(`c`,`i`,`o`)` expresses the property that output port o of component c does not depend on the input port i of component c.

Combining Basic Properties The basic properties as introduced above (and Boolean combinations thereof) act as atomic propositions about system states, on top of which the actual analyses, such as model checking of temporal logic formulae, are performed.

It might turn out later that certain analyses need to determine whether the current mode of a component was reached via a specific transition. If this is the case, then it requires the introduction of transition labels as part of a mode transition system (cf. Section 5.2.4).

9 Comparison with COMPASS-AADL

The dialect of AADL that is introduced in the present document, MILS-AADL, is essentially based on another variant of AADL that has been developed in the framework of the COMPASS project [4]. This language, called COMPASS-AADL in the scope of this document, is described in detail in [3]. Although being quite similar, MILS-AADL and COMPASS-AADL are (syntactically) incompatible as the former exhibits both additional and missing language features with respect to the latter. The modifications are mainly motivated, on the one hand, by the requirement to support the modelling and analysis of security aspects in MILS-AADL. On the other hand, COMPASS-AADL features language constructs for dealing with fault detection, isolation and recovery, which were deemed unnecessary for MILS-AADL. Another reason for applying minor syntactic changes was the adaptation of MILS-AADL to AADL V2, which was introduced in 2012, i.e., after the definition of COMPASS-AADL.

The following list gives a brief summary of the differences between MILS-AADL and COMPASS-AADL.

- General language features:
  - Comment syntax has been extended with generic tool annotations for extensible tool support.
  - New data types for security modelling and aggregation have been added, such as encryption keys, encrypted data, and tuples. They are supported by corresponding built-in operators.
  - Constant declarations have been added to facilitate the modular and user-friendly specification of named types, value constants, and encryption keys.
  - The timed specification of system behaviour now supports time units.
- Nominal component specifications:
  - In order to support the specification of security policies, new component categories (**`network`** and **`node`**) have been introduced. On the other hand, **`thread`** `group` is no longer supported.
  - MILS-AADL specification not only support event and data ports, but also the combination of both in the form of event-data ports.
  - The transmission of values along data ports is now uniformly represented by data flows, making data port connections obsolete.
  - Some keywords related to fault detection, isolation and recovery have been omitted: `alarm`, `fdir`, `observable`, and `passive`.
- Error model specifications:
  - Error states are now declared in the error model implementation (rather than the error type). They are thus considered to be internal objects and are not visible to the environment.

References

[1] Carolyn Boettcher, Rance DeLong, John Rushby, and Wilmar Sifre. The MILS Component Integration Approach to Secure Information Sharing. In *27thAIAA/IEEE Digital Avionics Systems Conference*, St. Paul, MN, October 2008.

[2] M. Bozzano and A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *International Journal on Software Tools for Technology Transfer*, 9(1):5–24, 2007.

[3] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, dependability, and performance analysis of extended AADL models. *The Computer Journal*, 54(5):754–775, May 2011. doi: 10.1093/com.

[4] The COMPASS project web site. `http://compass.informatik.rwth-aachen.de/`.

[5] Safety and security requirements for fortiss Smart Microgrid demonstrator. Technical Report D1.1, Version 1.4, D-MILS Project, March 2014. `http://www.d-mils.org/page/results`.

[6] Requirements for distributed MILS technology. Technical Report D1.3, Version 1.2, D-MILS Project, August 2013. `http://www.d-mils.org/page/results`.

[7] Safety and security requirements for Frequentis Voice Service demonstrator. Technical Report D1.2, Version 1.3, D-MILS Project, March 2014. `http://www.d-mils.org/page/results`.

[8] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An introduction. Technical Note CMU/SEI-2006-TN-011, CMU Software Engineering Institute, 2006.

[9] Peter H. Feiler and Ana Rugina. Dependability modeling with the Architecture Analysis & Design Language (AADL). Technical Note CMU/SEI-2007-TN-043, CMU Software Engineering Institute, 2007.

[10] T.A. Henzinger. The theory of hybrid automata. In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 278–292. IEEE CS Press, 1996.

[11] Information technology – syntactic metalanguage – extended BNF. Standard 14977, ISO/IEC, December 1996.

[12] Architecture Analysis & Design Language (AADL) (rev. B). SAE Standard AS5506B, International Society of Automotive Engineers, September 2012.

[13] Architecture Analysis and Design Language Annex (AADL), Volume 1, Annex E: Error Model Annex. SAE Standard AS5506/1, International Society of Automotive Engineers, June 2006.