SLIM 3.0 - Syntax and Semantics

Version 1.2

July 27, 2016

Abstract

This document defines the syntax and semantics of the SLIM 3.0 language. SLIM 3.0 consolidates previous versions of the language supported in the COMPASS family of tools. For a more comprehensive overview on using the SLIM language guided by examples, please refer to the COMPASS User Manual [RD6].

Contents

1 Introduction

This document provides the definition of the SLIM syntax and its semantics. It is meant as a reference on using various syntactic features and their meaning. For an introduction to the language, please consult the manual and inspect the examples that are provided with the COMPASS toolset.

The document is structured as follows: First, the concrete syntax is given, which provides a description, grammar and constraints for the various syntactic features. This is followed by the abstract syntax, which describes the model that is created by using the language.

Next, the model extension process is described, based on the description of the model in the abstract syntax. Model extension is the process of integrating the error behavior specified in the model into the nominal specification.

Finally, the semantics of the resulting model are discussed. First, the semantics of individual components specified in the model are presented, followed by the semantics of the complete model in terms of the network of all (connected) components.

2 Concrete Syntax

The following describes the concrete syntax of the SLIM language. SLIM, which is based on AADL, describes a model in terms of components, which can contain subcomponents and connections to other components, effectively describing a hierarchy of components.

This description is structured as follows: First, properties and property sets are introduced. These are syntactic structures that can be applied the the various components and their elements, describing various attributes or parameters of them.

This is followed by a description of SLIM data types and possible expressions over such data. Data is integral to SLIM's behavior, as it describes part of the state of the model. Components of the model can contain data elements, as well as transmit data via their connections.

Finally, the various elements that make up the structure of the model, in terms of components and their elements, are presented (see Section 2.7). Globally, a model consists of a system specification, which is described by packages. Packages contain component definitions, both their types (interfaces) and implementations. Types are introduced first, along with the possible elements they may contain (ports). This is followed by the implementations and their elements (subcomponents, connections and modes).

Every section describing an aspect of the system specification follows the same structure: A description; the possible properties that can be applied to the element; the restrictions that apply to such elements.

2.1 Grammar Notation

The following sections defines the syntax of our specification language. The context-free part of this syntax is given in (extended) Backus-Naur Form, using the following notations:

- boldface symbols represent keywords (e.g., **`package`**);
- symbols with initial uppercase letters stand for nonterminal symbols (e.g., *SystemSpecification*);
- 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.
- $\alpha \mid \ldots \mid \beta$: Any literal character in the range between α and β.

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, and extend to the end of the line ([`\r`] `\n`).

2.2 Property Sets

SLIM uses the AADL "properties" mechanism to extend the AADL syntax with new constructs. Property sets allow the definition of properties (as well as property constants and types) that can be used throughout the SLIM model. Predefined property sets define properties that are used to specify temporal formulas, contracts, and other construct that change the semantics of the SLIM Model. Although the user can specify other property sets, only the properties in these predefined sets are interpreted by COMPASS.

PropertySet

::= **property set** *PropertySetIdentifier* **is**
(*PropertyType* | *PropertyDef* | *PropertyConstant*)$^+$
end *PropertySetIdentifier* **;**

PropertyType

::= *PropertyTypeIdentifier* **: type** *PropertyType***;**

PropertyDef

::= *PropertyIdentifier* **:** *PropertyType* [**=>** *PropertyValue*]
applies to (*ElementType* {**,** *ElementType*}* **) ;**

PropertyConstant

::= *PropertyConstantIdentifier* **: constant** *PropertyType*
=> *PropertyValue***;**

PropertyType

::= **aadlboolean** | **aadlstring** | *PropertyTypeNumeric* |
PropertyTypeEnum | *PropertyTypeRange* | *PropertyTypeUnits* |
PropertyTypeClassifier | *PropertyTypeReference* |
PropertyTypeList | *PropertyTypeRecord* | *PropertyTypeIdentifier*

PropertyTypeNumeric

::= {**aadlinteger** | **aadlreal**} [*PropertyNumeric* **..** *PropertyNumeric*]
[**units** {*UnitsDesignator* | *UnitTypeIdentifier*}]

UnitsDesignator

::= *PropertyTypeIdentifier* | *UnitsList*

PropertyTypeEnum

::= **enum (** *EnumIdentifier* {**,** *EnumIdentifier*}* **)**

PropertyTypeRange

::= **range of** *PropertyTypeNumeric*

PropertyTypeClassifier

::= **classifier (** *ComponentCategory* {**,** *ComponentCategory*}* **)**

PropertyTypeReference

::= **reference (** *ElementType* {**,** *ElementType*}* **)**

PropertyTypeUnits

::= **units** *UnitsList*

UnitsList

::= **(** *UnitIdentifier* {**,** *UnitIdentifier* **=>** *UnitIdentifier* ***** *Number*}* **)**

PropertyTypeList

::= **list of** *PropertyType*

PropertyTypeRecord

::= **record (** (*identifier* **:** *PropertyType* **;**)$^+$ **)**

2.2.1 Property types

Various base property types are defined, listed as follows:

- **`aadlboolean`** : Boolean type, values are either **`true`** or **`false`**.
- **`aadlstring`** : String type, values are quoted strings.
- **`aadlinteger`** : Integer type. Optionally, a unit can be specified, and a range of allowed values.
- **`aadlreal`** : Real (floating point) type. Similar to **`aadlinteger`**, but permits fractions.
- **`range of`** : Allows values that specify a range of the numeric type that follows this type definition.
- **`enumeration`** : Enumeration of identifiers.
- **`reference`** : Permits values that reference named objects in the component hierarchy. It can optionally be followed by a list of named element types that are permitted.
- **`classifier`** : Permits values that reference a classifier. It can optionally be followed by a list of categories of classifier that are permitted.
- **`list of`** : A list of a given type, allows list values.
- **`record`** : A property record, which allows different named values to be specified with heterogeneous types.

2.2.2 Units

Units can be specified in property sets, which can be further associated with numeric values. Units can be specified inline when declaring a numeric type, or can be defined as a separate type and then referenced by the numeric type (however, using a unit type on its own is not valid, and always has to be part of a numeric type).

Units are specified with a base unit, following by derived units expressed as the base unit, or other derived units, multiplied by a constant factor. For instance, a unit to express distance can be specified using millimeter as a base unit, and other units derived from it, such as `centimeter =>millimeter * 10`, `meter =>centimeter * 100`.

In SLIM, units for time are already predefined (`ps`, `Ns`, `Ms`, `Sec`, `Min`, `Hr`), which can always be used without having to be qualified.

2.2.3 Ranges

Ranges can be used to specify a specific domain of numeric values. Ranges can be used as part of a numerical type declaration, limiting the possible values to those included within the range. Alternatively, a type can be declared that accepts ranges as value. For a numerical type that has been declared to use a unit, the range should also be expressed in values using that unit.

2.2.4 Property type declarations

A property set allows a named property type to be specified, which can be further used by properties and property constants. Such types are declared using the **`type`** keyword. A type consists of its identifying name, followed by the type definition (see the previous Section).

2.2.5 Property declarations

A property declaration defines a new property that can be used in the component hierarchy. A property is defined by its name, a type, an optional default value and finally the possible elements the property applies to (which specifies for which elements the property is allowed to be specified).

2.2.6 Property constant declarations

A property constant is a predefined property value. Such constants are declared by the **`constant`** keyword, followed by the type of the constants and its value. Such constants can be used whenever a property value is expected of the same type.

2.2.7 Syntactic Restrictions

N-1 The first property set identifier has to match the end property set identifier.

N-2 The type and default value of a property must match.

N-3 The type and default value of a property constant must match.

N-4 Types, constants, property names and enumeration identifiers must be unique.

N-5 All unit identifiers with a property set must be unique.

N-6 Units may not have cyclic definitions

N-7 Numeric types may only refer to a valid unit type.

N-8 Numeric type ranges must match the specified unit.

N-9 Numeric type ranges may not be empty.

N-10 Units of numeric type ranges must match the same unit type.

N-11 Composite property types may not refer to a unit type directly.

N-12 Record field identifiers must be unique.

N-13 Property value ranges must have matching units.

N-14 Property value lists must have a uniform type.

N-15 Property value record identifiers must be unique.

N-16 Modal property values must have a uniform type.

N-17 All property set names declared within a scope have to be unique.

2.3 SLIM Defined Property Sets

SLIM predefines two property sets for both specifying special purpose attributes of elements in the model, as well as specifying formal properties. One is called `SLIMpropset` and is used for the extensions to the architectural elements such as ports, and one called `CSSP` for the introduction of the Catalogue of System and Software Properties, including generic formal properties specified in different logics, contracts specified in a linear-time temporal logic, patterns of specific properties typically used in system and software design.

The following is an example of an AADL property used to add the "blocking" attribute to (event and event data) ports. SLIM further restricts the syntax to use this attribute only for input ports.

```
Blocking: aadlboolean => true
        applies to (event port, event data port) ;
```

The following is an example of property defined in the CSSP.

```
ReactionTime: record (InputPort: aadlstring;
                      OutputPort: aadlstring;
                      TimeBound: aadlint;)
        applies to (system, process, thread);
```

SLIMpropset The base SLIM property set is defined by `SLIMpropset`. It contains the definitions of the various properties that apply to the elements in the SLIM model. Inside `SLIMpropset`, a few special purpose property types are defined:

- `SlimExpr`: A property type which accepts string values. These string values are parsed as SLIM expressions and are used for for instance invariants and default values. See also Section 2.6.1
- `ClockTimeUnit`: A property type which enumerates the possible time units for clocks, defined as **`enum(`**`Milliseconds, Seconds, Minutes, Hours, Days`**`)`**. See also Section 2.6.1.
- `Formula`: A property type which accepts string values. These string values are parsed as Temporal Formulas. See also Section 2.6.2

For the specification of formal properties and contracts, the following properties have been defined:

- `GenericProperty`: A property that resembles a generic formal property. It is defined as a record containing the following fields:
  - `Name`: A string indicating the name of the property;
  - `Formula`: A `SLIMpropset::Formula` of the formula for the property;
  - `Description`: A string with a plain text description for the property.

- `Contracts`: A list of `Contract` records.
- `Contract`: A record representing a formal contract. It consists of the following fields:
  - `Name`: A string indicating the name of the contract;
  - `Assumption`: A `SLIMpropset::Formula`;
  - `Guarantee`: A `SLIMpropset::Formula`.

  Here, the assumption and guarantee are allowed to refer to the name of a `GenericProperty` or a CSSP property.
- `ContractRefinements`: A list of `ContractRefinement` records. These apply to classifiers only.
- `ContractRefinement`: A record representing a contract refinement. It consists of the following fields:
  - `Contract`: A string indicating the name of the contract being refined;
  - `Subcontracts`: A list of strings referring to the contracts of subcomponents (formatted as SubcomponentName.ContractName);

CSSP The CSSP property set defines various properties and types for the specification of formal properties (in various forms) for components and their elements.

2.4 Properties Association

Properties defined in property sets, as well as those predefined in SLIM, can be associated with various classifiers and named elements defined within classifiers. Properties can be specified in two ways:

1. As part of a package or classifier in the **`properties`** section.
2. As part of a named element between braces (referred to as an in-line property).

In both cases, the property applies either to the element it was defined in, or if the **`applies to`** syntax is used, the owner specified in the **`applies to`** specification. Properties can be overridden by other properties defined on a containing classifier. For instance, a property defined in a component implementation may override one defined for a mode, or the property of a subcomponent may override the same one for a component implementation the subcomponent refers to.

Properties

::= **`properties`** {*PropertyAssoc*$^+$ | **`none`**}

InlineProperties

::= **`{`** *PropertyAssoc*$^+$ **`}`**

PropertyAssoc

::= *PropertyIdentifier* {**`=>`** | **`+=>`**} *PropertyValue*
[**`applies to`** *Reference* [`,` *Reference*]$^+$] **`;`**

Reference

::= *identifier* {`.` *identifier*}*

PropertyValue

::= **`true`** | **`false`** | *PropertyNumeric* | *String* |
EnumIdentifier | *ConstantIdentifier* | *PropertyRange* |
PropertyClassifier | *PropertyReference* |
PropertyList | *PropertyRecord*

PropertyNumeric

::= *Number* [*UnitIdentifier*]

PropertyRange

::= *PropertyNumeric* `..` *PropertyNumeric*

PropertyClassifier

::= **`classifier`** **`(`** *ComponentClassifier* **`)`**

PropertyReference

::= **`reference`** **`(`** *NamedElement* **`)`**

PropertyList

::= **`(`** *PropertyValue* [`,` *PropertyValue*]$^+$ **`)`**

PropertyRecord

::= **`[`** (*identifier* **`=>`** *PropertyValue* **`;`**)$^+$ **`]`**

2.4.1 Syntactic Restrictions

O-1 The type of an assigned property must match its value.

O-2 Assigned properties may only append lists to list valued properties.

O-3 An assigned property may only apply to existing named elements.

O-4 The same component may not assign the same property twice to the same target.

O-5 A property may only be assigned to the category it applies to.

2.5 Data Types

2.5.1 Built-in Data Types

Data types for use in the SLIM model are defined by means of data components (components of the category **`data`**). These data types can be used to specify the data type of data ports as well as data subcomponents. These data can then be used in SLIM expressions (see Section 2.6.1). Some data types are predefined (built-in). The following grammar shows how data types may be specified with SLIM for elements which expect a type.

DataType
  ::= *DataComponentIdentifier* | **`enum(`***EnumIdentifierList***`)`** |
    `[`*ConstantValue***`..`***ConstantValue***`]`** | **`(`***DataType*$\{$**`,`***DataType*$\}^{+}$**`)`**
DataComponentIdentifier
  ::= *identifier*
EnumIdentifierList
  ::= *EnumIdentifier* $\{$**`,`***EnumIdentifier*$\}^{*}$
EnumIdentifier
  ::= *identifier*

The following data types are predefined in SLIM in the `SLIMdatatypes` package, though they do not have to be qualified:

- `bool`: Boolean values, with constants **`true`** and **`false`**;
- `clock`: for data components whose values (in $\mathbb{R}_{\geq 0}$) linearly increase over time, with non-negative `real` constants (see below);
- `continuous`: for data components whose values (in $\mathbb{R}$) change continuously in time;
- **`enum`**: enumeration of abstract values, specified as **`enum(`**id_1**`,`** $\ldots$ **`,`**id_n**`)`** where $n \geq 1$ and each id_i is a distinct identifier;
- `int`: integer values (in $\mathbb{Z} = \{0, 1, -1, \ldots\}$), with constants of the form $[-]\{$`0` $| \ldots |$ `9`$\}^{+}$;
- `real`: floating-point values with discrete value changes, with constants of the form $[-]\{$`0` $| \ldots |$ `9`$\}^{+}[$`.`$\{$`0` $| \ldots |$ `9`$\}^{+}]$.

For all of these except clocks, default values can 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 2.10.4.

For the predefined data types, SLIM specifies the domain of possible values that may be used inside SLIM expressions (see Section 2.6.1). These domains are defined as follows:

- **`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 $n \geq 1$.

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 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`. The `TimeUnit` property can be specified to indicate in what time unit the clock value is represented (defaults to `Seconds` for models using time scales).

Properties When specifying a data type, the following properties are applicable to the element that the data type is defined for (i.e. components of category **`data`**, as well as data ports):

- `Default`: A property that accepts values of type `SlimExpr`, which must be a constant value and adhere to the *ConstantExpression* syntax, see Section 2.6.1. This property defines the default value for the element it is defined for. Default values cannot be specified for `clock` s, which are implicitly initialized to 0.
- `TimeUnit`: A property that accepts a value of the `ClockTimeUnit` enumeration. When using time units in the SLIM model, this property must be defined for `clock` data types (it has no meaning for other types). By default it is set to `Seconds`.

Syntactic Restrictions

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

C-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 Section 2.8) of that type, and $v_1 < v_2$ must hold.

2.5.2 Data component types

Further data types can be specified by means of declaring a data component type. Such a data component identifies a new data type (which can be used by data ports and data subcomponents). Using a data component type directly implies the use of an abstract data type. Such a type represents an unknown, arbitrary value domain.

DataComponentType
  ::= **`data`** *DataComponentTypeIdentifier*
      `end` *DataComponentTypeIdentifier* `;`
DataComponentTypeIdentifier
  ::= *identifier*

Syntactic Restrictions

L-1 All data component type names declared within a scope have to be distinct.

L-2 The first data component type identifier has to match the end data component type identifier.

2.5.3 Data component implementations

A data component implementation can be used to specify composite types. A data component implementation may specify data subcomponents which specify the types this data component implementation is composed of (specifying no subcomponents defines the data component implementation to be of an abstract type). Composed data types are treated as tuples of such types.

DataComponentImplementation
  ::= **`data implementation`** *DataComponentImplName*
      [**`subcomponents`** *DataSubcomponent*$^+$]
      `end` *DataComponentImplName* `;`
DataComponentImplName
  ::= *DataComponentTypeIdentifier* `.` *DataComponentImplIdentifier*
DataComponentImplIdentifier
  ::= *identifier*
DataSubcomponent
  ::= *SubcomponentIdentifier* `:` **`data`** *DataType* `;`

Syntactic Restrictions

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

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

M-3 All data subcomponent identifiers must be unique within a data component implementation.

M-4 A data subcomponent may not specify a type that (recursively) contains itself.

M-5 The first data component implementation identifier has to match the end data component implementation identifier.

Syntactic Restrictions

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

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

M-3 All data subcomponent identifiers must be unique within a data component implementation.

M-4 A data subcomponent may not specify a type that (recursively) contains itself.

M-5 The first data component implementation identifier has to match the end data component implementation identifier.

2.6 SLIM Expressions

2.6.1 Basic Expressions

Throughout the SLIM model, expressions may be used to read and manipulate data. The following tables list the operators that can be used to form larger expressions from basic data elements. Note that these operators are only defined for the predefined data types, and types composed of predefined data types.

Here *RNum* stands for {`clock`,`continuous`,`real`}, and *Range* stands for an arbitrary integer range.

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.

Expressions are considered to be *constant*, i.e. a *ConstantExpression*, if it does not contain any variable data. Possible variable data are listed in the expression grammar under the rule *VariableExpression*.

Expression

::= *Expression BinaryOperator Expression* |
UnaryOperator Expression | *Expression TimeScaleOperator* |
`case` *Expression* **`:`** *Expression* {**`;`** *Expression* **`:`** *Expression*}*
`otherwise` *Expression* **`end`**
TupleExpression | *VariableExpression* | *ConstantExpression*
`(` *Expression* **`)`**

TupleExpression

::= **`(`** *Expression* {**`,`** *Expression*}$^+$ **`)`** |
Expression **`[`***ConstantExpression***`]`**

VariableExpression

::= *DataPortIdentifier* | *DataComponentIdentifier* |
`data(` *EventDataPortIdentifier* **`)`** |
FunctionConstantClassifier **`(`** *Expression* {**`,`** *Expression*}$^+$ **`)`** |

ConstantExpression

::= **`true`** | **`false`** | *number* | *EnumIdentifier* | *ValueConstantClassifier*

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

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

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.

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

Table 1: Arithmetic Operators

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 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 $\tau \in \{\texttt{int}, \texttt{real}, \texttt{continuous}\}$.

Tupling Operators SLIM supports the aggregation of data by introducing Cartesian products in the form of tuples, written as **(**x_0**,** $\ldots$ **,**x_n**)** for tuples x_0 through x_n of respective discrete data types τ_0 through τ_n, and corresponding projections. These projections satisfy the following equation:

$$(x_0, \ldots, x_n)[i] = x_i$$

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

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

Syntactic Restrictions

C-3 The operator tables define the allowed typing of expressions.

C-4 All subexpressions involving data variables of type `int`, *Range*, `real`, `clock`

Operator	Type	Meaning
and	$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$	Conjunction
iff	$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$	Biconditional
imp or **implies**	$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$	Implication
not	$\texttt{bool} \to \texttt{bool}$	Negation
or	$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$	Disjunction
xnor	$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$	Negated exclusive disjunction
xor	$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$	Exclusive disjunction

Table 3: Boolean Operators

Operator	Type	Meaning
case	$(\texttt{bool} \times \texttt{bool})^+ \times \texttt{bool} \to \texttt{bool}$	Boolean **case**
	$(\texttt{bool} \times \tau_1) \times \ldots \times (\texttt{bool} \times \tau_n) \times \tau_0 \to \max\{\tau_0, \tau_1, \ldots, \tau_n\}$	Numerical **case** (result type is the largest numerical class of all subexpressions)
	$(\texttt{bool} \times \textbf{enum})^+ \times \textbf{enum} \to \textbf{enum}$	Enumeration **case**

Table 4: **case** Operator

and `continuous` must be linear. Thus, expressions involving the multiplication or division of two variables with these types are not allowed, nor expressions where the denominator of a division is a variable with these types. This restriction allows the reasoner to exploit efficient linear-constraint solvers.

C-5 The **data()** operator may only appear in assignment expressions of state transitions and may only refer to an input event data port if it is the trigger of that transition, see Section 2.10.4.

C-6 The value i of the tuple projection operator must lie within $0 \leq i < n$, where n is the arity of the tuple.

Operator	Type	Meaning
msec	$\tau \rightarrow$ `clock`	Convert value to time in milliseconds
sec	$\tau \rightarrow$ `clock`	Convert value to time in seconds
min	$\tau \rightarrow$ `clock`	Convert value to time in minutes
hour	$\tau \rightarrow$ `clock`	Convert value to time in hours
day	$\tau \rightarrow$ `clock`	Convert value to time in days
msec	`clock` $\rightarrow \tau$	Convert time to a value in milliseconds
sec	`clock` $\rightarrow \tau$	Convert time to a value in seconds
min	`clock` $\rightarrow \tau$	Convert time to a value in minutes
hour	`clock` $\rightarrow \tau$	Convert time to a value in hours
day	`clock` $\rightarrow \tau$	Convert time to a value in days

Table 5: Timescale Operators

Operator	Type	Meaning
(., ... **,.)**	$\tau_0 \times \cdots \times \tau_n \rightarrow \tau_0 \times \ldots \times \tau_n$	Constructing tuple
`.[.]`	$(\tau_0 \times \cdots \times \tau_n) \times$ `int` $\rightarrow \tau_i$	Projection for position *i*

Table 6: Tupling Operators

2.6.2 Temporal Formulas

TemporalFormula
  ::= *AtomicFormula* |
  not *TemporalFormula* |
  TemporalFormula **and** *TemporalFormula* |
  TemporalFormula **or** *TemporalFormula* |
  TemporalFormula {**implies**|**imp**} *TemporalFormula* |
  always *TemporalFormula* |
  never *TemporalFormula* |
  in the future *TemporalFormula* |
  TemporalFormula **until** *TemporalFormula* |
  in the past *TemporalFormula* |
  TemporalFormula **since** *TemporalFormula*;

AtomicFormula
  := *Expression* |
  time_until (*Expression* **)** |
  next (*Expression* **)** |
  last_data (*identifier* **)**;

Syntactic Restrictions

Q-1 The type of expressions used as atomic formulas must be Boolean.

2.7 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 2.8) introduces a constant identifier that represents a (discrete, possibly uninterpreted) data type, an uninterpreted function, or a constant value. A *component* defines the type or an implementation of a component (cf. Section 2.9). It is possible to declare multiple implementations of a component.

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

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.

Properties When specifying a package, the following properties are applicable:

- `Constants`: A property that accepts an **`aadlstring`** formatted according to the grammar given in Section 2.8. If specified, this property defines the constants that are available inside the package.

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.

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

2.8 Constant Declarations

SLIM allows the definition of constant values and (uninterpreted) function for use in data expressions. These constants can be specified by the `Constants` property inside a package. These constants are then made available for this package. The following grammar specifies the syntax that is used inside the string value of the property.

ConstantDeclarations
  ::= **`constants`** *ConstantDeclaration*$^+$

ConstantDeclaration
  ::= *FunctionConstantDeclaration* | *ValueConstantDeclaration*

FunctionConstantDeclaration
  ::= *FunctionConstantIdentifier* **`:`** **`function`**
    DiscreteDataType {**`,`** *DiscreteDataType*}* **`->`** *DiscreteDataType* **`;`**

FunctionConstantClassifier
  ::= [*PackageIdentifier* **`::`**] *FunctionConstantIdentifier*

FunctionConstantIdentifier
  ::= *identifier*

ValueConstantDeclaration
  ::= *ValueConstantIdentifier* **`:`** *DiscreteDataType* **`:=`** *ConstantExpression* **`;`**

ValueConstantClassifier
  ::= [*PackageIdentifier* **`::`**] *ValueConstantIdentifier*

ValueConstantIdentifier
  ::= *identifier*

Syntactic Restrictions

P-1 All declared value constants have to be distinct.

P-2 The definition of a value constant may only refer to other declared value constants.

P-3 Declarations of value constant identifiers must not be recursive.

P-4 In each value constant declaration, the constant value must match the data type declared for the value constant identifier.

2.9 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.

ComponentType
  ::= *ComponentCategory ComponentTypeIdentifier*
    [**`features`** *ComponentFeatures*]
    [*Properties*]
    `end` *ComponentTypeIdentifier* **`;`**

ComponentTypeIdentifier
  ::= *identifier*

ComponentCategory
  ::= *SoftwareCategory* | *HardwareCategory* | *CompositeCategory* | **`abstract`**

SoftwareCategory
  ::= **`data`** | **`process`** | **`thread`**

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

CompositeCategory
  ::= **`node`** | **`subject`** | **`system`**

ComponentFeatures
  ::= *PortDeclaration*$^+$

Properties For component types the following properties are defined:

- `FDIR`: A boolean property indicating the component type is an FDIR component.
- Various properties to specify formal properties and contracts, see Section 2.3

Syntactic Restrictions

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

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

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

B-4 A component type with the `FDIR` property set **`true`** can only be associated to component implementations with the `FDIR` property set **`true`**.

B-5 The `Blocking` property can apply only to input event or event data ports.

B-6 Any identifier in the temporal formulas declared in a formal property of component type (either a generic property or an assumption or a guarantee of a contract) identifies a port of the same component type.

B-7 The assumption and guarantee of a contract must be either the identifier of a formal property of the same component type or a temporal formula.

B-8 The *Identifier* used in the declaration of CSSP::Function must be an output data or event data port.

B-9 The *Term* used in the declaration of a CSSP::Function must have the same type of the output port referred by the property.

2.9.1 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 2.11 gives the corresponding restrictions. To support FDIR analysis, outgoing data ports of type `bool` of FDIR components can be tagged with the `Alarm` property, and outgoing data ports of non-FDIR components can be tagged with the `Observable` property. Incoming event ports that have the property `Blocking` set to **`false`** never block the communication, see also Section 4.2.2.

PortDeclaration

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

EventPortDeclaration

  ::= *EventPortIdentifier* **:** {**`in`** | **`out`**} **`event port`** [*InlineProperties*] **;**

DataPortDeclaration

  ::= *DataPortIdentifier* **:** {**`in`** | **`out`**} **`data port`**
      DiscreteDataType [*InlineProperties*] **;**

EventDataPortDeclaration

  ::= *EventDataPortIdentifier* **:** {**`in`** | **`out`**} **`event data port`**
      DiscreteDataType [*InlineProperties*] **;**

EventPortIdentifier

  ::= *identifier*

DataPortIdentifier

  ::= *identifier*

EventDataPortIdentifier

  ::= *identifier*

Properties For ports the following properties are defined:

- `Alarm`: A Boolean property that indicates that the port is an alarm for FDIR analysis.
- `Observable`: A Boolean property that indicates that the port is an observable parameter.
- `Blocking`: A Boolean property that indicates if the port is blocking. It can only be defined for **`in event data`** and **`in event`** ports. By default the value is **`true`** for components specifying states (see Section 2.10.4), and **`false`** otherwise.
- `Default`: See Section 2.5.
- CSSP properties that are applicable to ports, see also Section 2.3

Syntactic Restrictions

D-1 Section 2.11 specifies which component categories support the declaration of ports.

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

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

D-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.

D-5 The `Observable` property is only supported for outgoing data ports of non-FDIR components.

D-6 The `Alarm` property is only supported for outgoing data ports of type `bool` of FDIR components.

2.10 Component Implementations

ComponentImplementation
  ::= *ComponentCategory* **`implementation`** *ComponentImplName*
    [**`subcomponents`** *Subcomponents*]
    [**`connections`** {*EventPortConnections* | *DataFlows*}]
    [**`modes`** *Modes* [**`transitions`** *ModeTransitions*]]
    [*Properties*]
  `end` *ComponentImplName* **`;`**

ComponentImplName
  ::= *ComponentTypeIdentifier* **`.`** *ComponentImplIdentifier*

ComponentImplIdentifier
  ::= *identifier*

Properties For component types the following properties are defined:

- `FDIR`: A boolean property indicating the component implementation is an FDIR component.
- The properties to specify formal properties and contracts as defined in Section 2.3

Syntactic Restrictions

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

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

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

E-4 The type of a component implementation that has the `FDIR` property set to **`true`** must also have the `FDIR` property set to **`true`**.

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

E-6 Any identifier in the temporal formulas declared in a component implementation identifies a port of the corresponding component type or a port of a subcomponent.

E-7 The contract identifier used in a contract refinement must refer to a contract name declared in the component type.

E-8 The identifiers of subcontracts in a contract refinement must refer to a contract name of the component type of the corresponding subcomponent.

2.10.1 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 2.5 are allowed. 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`, default values can be defined by means of the `Default` property.

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, processor component, device or composite component *accesses* a bus;

These bindings are specified by means of the properties `StoredIn`, `RunningOn` and `Accesses`.

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 2.10.4 for details.

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

Properties For subcomponents the following properties are defined:

- `Default`: For data subcomponents, see Section 2.5.
- `Accesses`: A list of bus subcomponent references that this subcomponent accesses
- `StoredIn`: A memory subcomponent reference that this subcomponent is stored in
- `RunningOn`: A processor subcomponent reference that this subcomponent runs on

Syntactic Restrictions

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

F-2 In the second type of subcomponent declaration (*OtherSubcomponent*), *ComponentCategory* must not be **`data`**, and *ComponentClassifier* must refer to the name of a declared component type or implementation.

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

F-4 Section 2.11 specifies the possible subcomponent categories for each component category.

F-5 For all data subcomponents except those of type `clock`, default values have to be defined.

F-6 Clocks must not be assigned default values.

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

F-8 In case of a bus-to-bus binding, the referred component has to be different from the referring one.

F-9 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.

F-10 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 2.10.4).

F-11 The component hierarchy must not be recursive, that is, no component implementation can have itself as an (indirect) subcomponent.

F-12 For each component implementation, the identifiers of all data subcomponents have to be distinct from all data ports declared in the corresponding type specification.

F-13 The value identifiers of all **`enum`** types used in a component type or implementation have to be distinct from all data port, and data subcomponent identifiers of that component.

2.10.2 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 2.9.1). 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 neighbor 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 2.11). 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: for processes, a connection to a processor or memory component is only allowed if it is bound to that component.

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 2.10.4 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). 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(`**$\ldots,m,\ldots$**`);`**

  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(`**$\ldots,m,\ldots$**`);`**

  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(`**$\ldots,m,\ldots$**`);`**

  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*] [*InlineProperties*] **`;`**
EventPortReference
  ::= [*SubcomponentIdentifier* **`.`**] *EventPortIdentifier*
EventDataPortConnection
  ::= **`port`** *EventDataPortReference* **`->`** *EventDataPortReference*
    [*InModes*] [*InlineProperties*] **`;`**
EventDataPortReference
  ::= [*SubcomponentIdentifier* **`.`**] *EventDataPortIdentifier*

Syntactic Restrictions

G-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.

G-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.

G-3 The topological restrictions as specified in the beginning of this section apply (in particular related to process bindings).

G-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 2.10.4).

G-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.

G-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.

G-7 Each output event and event data port of a non-atomic component must be connected by some out-to-out connections (output events must be generated by some subcomponents).

G-8 For each in-to-in connection, the two connected input event (data) ports must have the same value of the blocking property.

2.10.3 Data Flows

We now define data flows to model immediate propagation of data among components. While in an event connection, the data of the receiver is updated after a transition fired be the event, a flow introduces a value update of a port that is 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 generalize 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 (cf. Section 2.11). To reiterate, for processes a connection to a processor or memory component is only allowed if it is bound to that component.

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.)

DataFlows
  ::= *DataFlow*$^+$
DataFlow
  ::= **`port`** *FlowExpression* **`->`** *DataPortReference*
    [*InModes*] [*InlineProperties*] `;`
FlowExpression
  ::= *Expression*
DataPortReference
  ::= [*SubcomponentIdentifier* `.`] *DataPortIdentifier*

Here the symbol *FlowExpression* refers to a SLIM expression of which the allowed variables are limited to *DataPortReference*, see also Section 2.6.1.

Syntactic Restrictions

H-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.

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

H-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.

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

H-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 2.10.4).

H-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.

H-7 In each component implementation, an outgoing data port is allowed to occur either on the left-hand side of mode transition assignments (Section 2.10.4) or as a target port in flows, but not both.

H-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.)

H-9 To exclude undefined values of an incoming data port *p* of a subcomponent *c*, in each mode of every component that uses *c* as a subcomponent, *p* must occur as the target port of a flow.

H-10 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.

H-11 The topological restrictions as specified in the beginning of this section apply (in particular related to process bindings).

2.10.4 Modes, States and Transitions

A component may specify either modes or states to specify different system configurations or behavior respectively. They may not be mixed however, and states are allowed only to be specified for atomic components (that is, those without subcomponents).

Modes In our modeling approach, *modes* can be attached to non-data components. They can be employed to specify different system configurations and connection topologies at runtime, allowing to model *dynamic reconfiguration* within the context of a non-atomic component in response to external events.

Each model specification must declare exactly one **`initial`** or **`activation`** mode. In the first case, 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 system reconfiguration in the next section), execution is resumed in the previous mode, that is, *mode history* is supported. In the second case, the component's execution is started in the given mode after *each* activation. Data subcomponents and outgoing data ports of the component are handled in a similar way: if an initial mode is specified, their values are preserved during deactivation; with an activation mode, they are reset to their default values (if defined) upon each (re-)activation.

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

$$m_1 \texttt{ -[} e \texttt{]-> } m_2$$

where m_1 and m_2 are modes and e is a trigger event for the transition. To define a transition that applies to every mode of the component, the wildcard symbol $\star$ can be used in place of the source mode.

Mode transitions can be *triggered* by incoming events arriving at ports, either externally from an input port or from a subcomponent's output port. A nondeterministic choice between different input triggers can be specified using the keyword **`or`**.

In addition it is possible to associate a guard with a mode transition using the `Guard` property. A *guard* is a logical expression over data ports of the respective component which additionally enables or disables a transition.

A special type of *reactivation* transition can be specified by using the special trigger **`@activation`**. Such a transition is taken upon reactivation of a component after it had been deactivated before. Nondeterminism is allowed. In the case an **`activation`** mode is specified, these transitions take precedence. That is, if the current mode is the source mode of an reactivation transition, then this transition is taken, regardless whether an **`initial`** or **`activation`** mode is specified. After activation, the mode will be set to the target mode of this transition.

Modes

::= *Mode*$^+$

Mode

::= *ModeIdentifier* **`:`** [{**`initial`** | **`activation`**}] **`mode`** [*InlineProperties*] **`;`**

ModeIdentifier

::= *identifier*

ModeTransitions

::= *ModeTransition*$^+$

ModeTransition

::= *SourceMode* **`-[`** *Trigger* {**`or`** *Trigger*}* **`]->`** *ModeIdentifier* [*InlineProperties*] **`;`**

SourceMode

::= *ModeIdentifier* | **`*`**

Trigger

::= *EventPortReference* | **`@activation`**

ModeGuard

::= *Expression*

InModes

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

States States provide an abstraction of the concrete *behavior* of a component, constituting an automata-like formalism for modeling finite state spaces of components. This applies only to atomic components, and is not possible for composite components (those with subcomponents). Similar to modes, at least one **initial** or **activation** state has to be declared.

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

Similar restrictions apply to `continuous` data components. Using linear expressions, their values may be tested in state invariants and state transition guards, and additionally they may be reset to an arbitrary constant. The specification of their dynamic behavior, 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 $a \in \mathbb{R}$) which is again attached as an invariant to a state. Such limited version of differential equations allows to reason with linear constraints. Syntactically, $\dot{x}$ is represented by an apostrophe: `x'`. In case time scales are used, the equation is specified using a time unit in the form $\dot{x} = a$ **per** *TimeUnit*.

References to other data elements, that is, data ports and discrete-type data subcomponents of a component are disallowed in state invariants. If the invariant is absent, then it is assumed to be **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.

A shorthand is available for the specification of invariants. A state may specify the **urgent in** parameter, which specifies the amount of time the system may reside in that state. This implicitly adds a clock to the model specification. Upon entering the state, this clock is reset to zero, and the urgent parameter induces an upper bound on this clock.

Formally, the state behavior of a component is specified by a *hybrid automaton* which operates on the **data** subcomponents 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*.

State Transitions *Transitions* between states are of the general form

$$m_1 \;\texttt{-[}\, e \;\textbf{when}\; g \;\textbf{then}\; f \,\texttt{]->}\; m_2$$

where m_1 and m_2 are modes, e is a trigger event for the transition, g is a guard, and f is an effect. Each of the trigger, the guard, and the effect can be omitted. To define a transition that applies to every state of the component, the wildcard symbol $\star$ can be used in place of the source state.

State transitions may, in addition to being triggered like mode transitions, 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.

Similar to mode transitions also state transitions allow the specification of guards. As an extension, a shorthand for clock guards is available using the **`within to`** notation, which specifies the time interval in which the transition is enabled upon entering its source state. This behavior specifies an implicit clock (see also the **`urgent in`** parameter) which is bound between the specified lower and upper bounds.

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 2.6.1 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 $(z-l)$ **mod** $(u-l+1)+l$ for d.

The resulting combinations of types are listed in Table 7. 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 7: 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.

Reactivation events are also allowed for state transitions. In addition to those specified for modes, effects may be specified (but no guards). Effects apply using the regular semantics. Thus, similar to regular transitions, data subcomponents for which no effect is specified will keep their original value, unless it was disabled in the source mode, but becomes active in the target mode.

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 modeled 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.

States
  ::= *State*$^+$

State
  ::= *identifier* : [*StateType*] `state` [`urgent in` *number*]
    [`while` *Invariant*][*InlineProperties*]`;`

StateType
  ::= `initial` | `activation`

Invariant
  ::= *TrajectoryEquation* | *Expression* | *Invariant* `and` *Invariant*

TrajectoryEquation
  ::= *SubcomponentIdentifier*`'` `=` *ConstantValue* [`per` *TimeUnit*]

ComparisonOperator
  ::= `<` | `>` | `<=` | `>=`

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

StateTransitions
  ::= *StateTransition*$^+$

StateTransition
  ::= *SourceState* `-[` [*StateTrigger*[`(` *Expression* `)`]] [`when` *StateGuard*]
    [`within` *number* `to` *number*] [`then` *StateEffect*] `]->`
    identifier[*InlineProperties*]`;`

SourceState
  ::= *identifier* | `*`

StateTrigger
  ::= *EventPortReference* [`(`*Expression*`)`] | `@activation`

StateGuard
  ::= *Expression*

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

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

Here the terminal symbol *operator* refers to the built-in operators as introduced in Section 2.6.1. The optional expression argument for the *Trigger* must only be given for outgoing event data ports.

Properties For modes, states and transitions the following properties are defined:

- CSSP properties that are applicable to modes and states, see also Section 2.3
- `Guard`: Applies to mode transitions only. This is a property of type `SlimExpr` which specifies the guard of the transition it is associated with.

Syntactic Restrictions

I-1 Section 2.11 specifies which subcomponent categories support modes.

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

I-3 Exactly one mode or state (the starting mode or state) has to be distinguished as **`initial`** or **`activation`** mode.

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

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

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

I-7 Trajectory equations and continuous invariants can refer to `continuous` subcomponents only.

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

I-9 For each `continuous` data component, at most one trajectory equation may be given in each state.

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

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

I-12 Each mode trigger must be

- an incoming 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 outgoing event port of that subcomponent.

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

I-13 Each state 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 or state of the respective transition.

I-14 The mode or state 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 or state of that transition.

I-15 The left-hand side of each assignment in a mode or state transition effect is a distinct data subcomponent that is active in the target mode or state of that transition or an outgoing data port of the current component.

I-16 The right-hand side of each assignment in a mode or state transition effect is a well-typed expression over data ports or data subcomponents of the current component that are active in the source mode or state of that transition.

I-17 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 7.

I-18 (see H-7)

I-19 States may only be defined for atomic components (those components that do not contain subcomponents of category other than data).

2.11 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 8 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 2.10.2, 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 implementations (having threads with event ports as subcomponents), but not for threads (only **`data`** subcomponents, which do not have ports) or for processors (no subcomponents).

Category	**Type elements**	**Implementation elements**	**Bindings**
`abstract`	any	any	any
Software categories			
`data`	—	`subcomponents`: `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 8: Overview of Component Restrictions

2.12 Error modeling

Error models can be used to annotate (non-data) component types and implementations to support safety and dependability analysis. The corresponding extension of the AADL language is defined in the Error Modeling Annex (EMA) [RD7], from which we draw some key ideas for developing our error modeling formalism.

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.

2.13 Error Model Types

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 2.10.2 for similar restrictions regarding port connections):

- from a hardware or composite component to a hardware or composite component if both access a common bus,
- from a bus 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 identifier
         [features ErrorFeatures]
       end identifier;
ErrorFeatures
  ::=  ErrorFeature+
ErrorFeature
  ::=  ErrorPropagation
ErrorPropagation
  ::=  InPropagation | OutPropagation
InPropagation
  ::=  identifier:  in error propagation;
OutPropagation
  ::=  identifier:  out error propagation;
```

Syntactic Restrictions

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

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

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

2.14 Error Model Implementations

An *error model implementation* provides the structural details of the error model. The actual behavior of an error model implementation is given by a (probabilistic) state machine whose *states* are the error states declared in the error model implementation.

Error events are internal to the component; they reflect changes of the error state caused by local faults and repair operations. Moreover an *occurrence distribution* can be attached to an error event, to model the probabilistic nature of faults.

Clocks explicitly define clock components that allow timed behavior to be introduced in the error specification. The behavior of these clocks is the same as for nominal models.

Error states are employed to represent the current configuration of the component with respect to errors. The specification distinguishes between the *initial* or *activation* state (which needs to be unique), and actual *error* states. The meaning of initial and activation states is similar to that of initial and activation modes (cf. Section 2.10.4). If an initial state is given, the error model is put in that state only in the beginning of system execution, supporting *error history* during deactivation phases. With an activation state, the error model starts over again in that state after each (re-)activation of the respective component. This distinction is useful, e.g., for modeling the different error behavior of hardware and software components: while reactivating a hardware component will generally not remove the cause of an error, this is usually the case for software components. The states of an error model implementation can further specify invariants or urgency in the same fashion as for states in the nominal models.

Reset events can be sent from the nominal model (see Section 2.10.4) to the error model of the same component, trying to repair a fault which has occurred. It is up to the error model to specify whether the repair operation is successful.

Reactivation triggers operate in the same fashion as for mode models. Upon reactivation of the error model, this transition is taken, possibly nondeterministically.

Transitions between states can be triggered by error events, **`reset`** events, **`@activation`** triggers, or error propagations. To define a transition that applies to every state of the error model, the wildcard symbol $\star$ can be used in place of the source state. Note that error transitions may refer to clocks of the error implementation in their guards and effects. The same linearity conditions apply as for mode component implementations.

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 Name
         [events ErrorEvents]
         [clocks ErrorClocks]
         [states ErrorStates]
         [transitions ErrorTransitions]
       end Name;
ErrorEvents
  ::=  ErrorEvent+
ErrorEvent
  ::=  identifier : error event [Occurrence];
Occurrence
  ::=  occurrence Distribution
Distribution
  ::=  poisson number [per TimeUnit]
ErrorClocks
  ::=  ErrorClock+
ErrorClock
  ::=  identifier : data clock [TimeUnit];
ErrorStates
  ::=  ErrorState+
ErrorState
  ::=  identifier : StateType state [urgent in number] [while Invariant];
ErrorTransitions
  ::=  ErrorTransition+
ErrorTransition
  ::=  SourceState -[ErrorTrigger [when ModeGuard]
         [within number to number] [then ModeEffect]]-> identifier;
SourceState
  ::=  identifier | *
ErrorTrigger
  ::=  identifier | reset | @activation
```

Syntactic Restrictions

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

K-2 The first identifier of each error implementation name must refer to an error model type that is defined within the same scope.

K-3 All error events declared within an error model implementation have to be distinct.

K-4 Both the source and the target state of an error transition must refer to (not necessarily distinct) states of the current error model.

K-5 The error transition trigger must be

- an error event or
- an (incoming or outgoing) error propagation or
- the keyword **`reset`**.

K-6 The transitions leaving a specific error state must either all be probabilistic with a Poisson distribution or non-probabilistic.

K-7 Non-determinism is not allowed in error model implementations, that is, the triggers of all transitions leaving a specific error state must be distinct.

K-8 Each state in the specification must be reachable by a sequence of transitions from the starting state.

K-9 The source, trigger and destination combination of an error transition must be unique, i.e., an unique error transition can only be declared once.

K-10 The first error implementation identifier has to match the end error implementation identifier.

K-11 The identifiers of error events and error propagations within the scope of the error implementation have to be distinct.

K-12 The rates of events with a **`poisson`** distribution must be positive.

K-13 The identifiers of error states within an error implementation have to be distinct.

K-14 Error states may not be defined in both the error model type and implementation.

K-15 Exactly one error state in the implementation (the *starting state*) has to be distinguished as either **`initial`** or **`activation`**.

K-16 The identifiers of clocks within an error implementation have to be distinct.

K-17 State invariants must refer to an existing clock within the error implementation.

K-18 State invariants must be convex.

K-19 State invariants must be expressions of type `bool`.

K-20 State urgency values may not be negative.

K-21 Transition guards may only refer to clocks declared in the same error implementation.

K-22 Transition guards must be expressions of type `bool`.

K-23 Transition time delay bounds may not be negative, and the lower bound has to be smaller than the upper bound.

K-24 The left-hand side of each assignment in the error transition effect must be a distinct clock.

K-25 Transitions effects may only refer to clocks declared in the same error implementation.

K-26 Probabilistic error states may not have invariants.

K-27 Probabilistic error states may not declare an urgency parameter.

K-28 Transitions leaving from probabilistic error states may not have guards.

K-29 Transitions leaving from probabilistic error states may not specify time delays.

K-30 Reactivation transitions may not have guards.

K-31 Reactivation transitions may not use the **`within`**-notation.

K-32 If the system specification makes use of time scales, error events must specify a time unit.

K-33 If the system specification makes use of time scales, error clocks must specify a time unit.

K-34 If the system specification makes use of time scales, mode urgency has to be expressed using a time unit.

K-35 If the system specification makes use of time scales, transition time delays have to be specified using time units

2.15 Fault Injections

To specify the association between an error model and a nominal model, so called *fault injections* are used. A fault injection specifies at minimum the error model implementation that is associated with a nominal component. Fault injections can further specify the effect on a data component or port when the error model is in a certain state, as well as a set of modes that the nominal model can be in during certain error states and possible blocking of event ports in certain error states.

Fault injections are specified by means of the following properties:

- `ErrorModel`: A classifier value referencing an error model implementation. This specifies for a nominal component the associated error model.

- `FaultEffects`: A list of `FaultEffect` values, which are records with a `state` field that accepts a string referencing an error state of the associated `ErrorModel`, a `target` field which is a reference to the element to which the fault effect applies, and an `effect` field of type `SlimExpr` specifying an expression with the fault effect that must apply in the specified error state.

- `ForcedModes`: A list of `ForcedMode` values, which are records with a `state` field that accepts a string referencing an error state of the associated `ErrorModel`, and a `modes` field which is a list of references to modes or states in the component which the component is allowed to be in during the specified error state.

- `InhibitList`: A list of `Inhibit` values, which are records with a `state` field that accepts a string referencing an error state of the associated `ErrorModel`, and a `ports` field which is a list of references to event (data) ports of the component which will be disabled in during the specified error state.

It is possible to specify fault injections on component types, implementations as well as non-data subcomponents. Such fault injections apply in the order *Subcomponent* > *Implementation* > *Type*.

If the error model reaches a state for which a `FaultEffect` property is specified, this effect is applied, overriding any possible transition effect or data flow for as long as the error model is in this state. When no fault effect applies any longer, the data flows return to their original expressions, but data component that were modified due to a transition effect are not reset to their original value. It is an error in the model if a `FaultEffect` invalidates a nominal state invariant of the current active state (it will block the associated error event or propagation from occurring).

With the `ForcedModes` property, it is possible which modes (or states) in the nominal model are valid when the error model is in a particular error state. When the error model is in such a state, any transition from a mode in this list to a mode not in this list becomes disabled. When the error model moves from one state to another due to any event or propagation, but not **`reset`**, if the active mode (or state) of the nominal component is not in the `ForcedMode` list of the target state, a forced transition is taken to the first mode listed in the `ForcedMode` property. A transition in the nominal model with the **`reset`** transition can only be taken if the synchronizing error transition targets a state for which either the `ForcedMode` property is not specified, or contains the target mode of the nominal transition. It is an error in the model if a forced mode transition is inhibited by an invariant that does not hold in the current state (it will block the associated error event or propagation from occurring).

If the error model reaches a state for which a `Inhibit` property is specified, any transition that is triggered by the referenced event (data) ports becomes disabled.

3 Abstract Syntax

3.1 Data Types

The following sets are used to define types of data in a SLIM model. The set *AbsPrimDTyp* contains the set of Boolean values $\mathbb{B}$, the set of integers $\mathbb{Z}$, the set of real numbers $\mathbb{R}$, sets of enumeratives, and the intervals of integers. *AbsDTyp* is the union of *AbsPrimDTyp* and tuples of *AbsPrimDTyp*. *AbsATyp* is the set of real functions $\mathbb{R} \to \mathbb{R}$. Finally, $\mathit{AbsTyp} = \mathit{AbsDTyp} \cup \mathit{AbsATyp}$.

3.2 SLIM Model

A SLIM model S is a tuple $\langle \Sigma, \mathit{CTyp}, \mathit{CImpl} \rangle$ consisting of a set of global symbols Σ, a set *CTyp* of component types, and a set *CImpl* of component implementations. These are defined in the following sections.

3.2.1 Global Symbols

The set Σ of global symbols consists of a set of constant symbols V_Σ and a set of function symbols U_Σ. Each constant $v \in V_\Sigma$ has a type $\mathit{typ}(v) \in \mathit{AbsDTyp}$. Each function symbol $f \in U_\Sigma$ has a type $\mathit{typ}(f) \in \mathit{AbsDTyp}^+$ (i.e. tuples of *AbsDTyp*).

3.2.2 Component Types

Each component type $c \in \mathit{CTyp}$ consists of

- a set of input data ports $\mathit{IDPrt}(c)$,
- a set of output data ports $\mathit{ODPrt}(c)$,
- a set of input event ports $\mathit{IEPrt}(c)$,
- a set of output event ports $\mathit{OEPrt}(c)$.

(event data ports are represented by a pair of data and event ports).

Let us denote $\mathit{Prt}(c)$ the set of ports of component type c, i.e. $\mathit{Prt}(c) := \mathit{IDPrt}(c) \cup \mathit{ODPrt}(c) \cup \mathit{IEPrt}(c) \cup \mathit{OEPrt}(c)$, with $\mathit{IPrt}(c)$ the set of input ports, i.e., $\mathit{IPrt}(c) := \mathit{IDPrt}(c) \cup \mathit{IEPrt}(c)$, with $\mathit{OPrt}(c)$ the set of output ports, i.e., $\mathit{OPrt}(c) := \mathit{ODPrt}(c) \cup \mathit{OEPrt}(c)$, with $\mathit{DPrt}(c)$ the set of data ports, i.e., $\mathit{DPrt}(c) := \mathit{IDPrt}(c) \cup \mathit{ODPrt}(c)$, and with $\mathit{EPrt}(c)$ the set of event ports, i.e., $\mathit{EPrt}(c) := \mathit{IEPrt}(c) \cup \mathit{OEPrt}(c)$, .

Each port $p \in \mathit{Prt}(c)$ is associated with a set of values $\mathit{typ}(p) \in \mathit{AbsDTyp}$, which represents the value that the data associated with the port can have in a specific time.

Each output data port $p \in \mathit{ODPrt}(c)$ has a default value $\mathit{dfl}(c, p)$.

Each input data port $p \in \mathit{IEPrt}(c)$ has a Boolean attribute $\mathit{blk}(c, p)$, denoting if it is blocking or not.

3.2.3 Component Implementations

Each component implementation $im \in CImpl$ is associated with a component type $typ(im) \in CTyp$. For simplicity, we write $IEPrt(im)$, $OEPrt(im)$, $IDPrt(im)$, $ODPrt(im)$, $dfl(im,p)$, ... instead of $IEPrt(typ(im))$, $OEPrt(typ(im))$, $IDPrt(typ(im))$, $OEPrt(typ(im))$, $dfl(typ(im),p)$, ...

Each composite component implementation $im \in CImpl$ consists of

- a finite non-empty set $Mod(im)$ of modes,
- its *initial mode*, $ini(im) \in Mod(im)$,
- the *invariant* of each mode, $inv(im,m)$,
- a set $Sub(im)$ of subcomponents,
- a set $Act(im,m) \subseteq Sub(im)$ of active subcomponents for each mode $m \in Mod(im)$, such that $Sub(im) = \bigcup_{m \in Mod(im)} Act(im,m)$,
- a set $Con(im,m)$ of event connections for each mode $m \in Mod(im)$, such that

$$\begin{array}{rrl} Con(im,m) \subseteq & IEPrt(im) \times & SIE(im,m) \\ \cup & SOE(im,m) \times & OEPrt(im) \\ \cup & SOE(im,m) \times & SIE(im,m) \end{array}$$

  where $SIE(im,m) := \{sc.p \mid sc \in Act(im,m), p \in IEPrt(sc)\}$ and $SOE(im,m) := \{sc.p \mid sc \in Act(im,m), p \in OEPrt(sc)\}$.

- a set $Flw(im,m)$ of data flows for each mode $m \in Mod(im)$, such that $Flw(im)$ is set of pairs $\langle p,a \rangle$ where p is a port in $ODPrt(im) \cup SID(im)$, a is an expression over $IDPrt(im) \cup SOD(im)$, where $SOD(im) := \{sc.p \mid sc \in Sub(im), p \in ODPrt(sc)\}$, and $SID(im) := \{sc.p \mid sc \in Sub(im), p \in IDPrt(sc)\}$,
- a set $MTr(im)$ of mode transitions, where each mode transition is a tuple $\langle m,t,g,f,m' \rangle$, where $m,m' \in Mod(im)$, t is a trigger, g denotes a guard, and f is an effect,
- and a set $RTr(im)$ of reactivation transitions, where each reactivation transition is a tuple $\langle m,f,m' \rangle$, where $m,m' \in Mod(im)$, and f is an effect.

Every subcomponent $s \in Sub(im)$ is associated with a type $typ(s) \in CTyp \cup CImpl \cup AbsTyp$.

As abbreviations we use $CSub(im) := \bigcup_{m \in Mod(im)} \{sc \in Act(im,m) \mid typ(sc) \in CTyp \cup CImpl\}$ for the set of (direct) control subcomponents of im, $DSub(im) := \bigcup_{m \in Mod(im)} \{sc \in Act(im,m) \mid typ(sc) \in AbsTyp\}$ for the set of data subcomponents of im, $DAct(im,m) := \{sc \in Act(im,m) \mid typ(sc) \in AbsTyp\}$ for the set of data subcomponents of im that are active in mode m.

Each data subcomponent $d \in DSub(im)$ has a default value $dfl(im,d)$.

Together, the data subcomponents and the data ports constitute the *data elements* of a component implementation

$$Dat(im,m) := DAct(im,m) \cup DPrt(im)$$

Again we set $Dat(im) := \bigcup_{m \in Mod(c)} Dat(im,m)$.

If $CSub(im) = \emptyset$, then im is called *atomic*.

3.2.4 SLIM Model Instance

A SLIM Model Instance is given by a SLIM Model and a component type or implementation chosen as root instance. Formally, a SLIM Model Instance is a tuple $SI = \langle \Sigma, CTyp, CImpl, root \rangle$ where $S = \langle \Sigma, CTyp, CImpl \rangle$ is a SLIM Model and $root$ is in $CTyp \cup CImpl$. If the SLIM Model S contains only one component implementation $root(S)$ which is not instantiated as subcomponent of another component, we sometimes refer to the SLIM Model as it were a SLIM Model Instance assuming that $root(S)$ is the *root*.

3.3 Abstract model corresponding to the concrete specification

3.3.1 Basic Sets

The following basic sets are derived by the language grammar:

Ide: the set of *identifiers*;

Nam: the set of *names*, given by $Nam := Ide.Ide$;

Cat: the set of *component categories*, given by

$$\begin{aligned} Cat := \{&\mathbf{bus}, \mathbf{data}, \mathbf{device}, \mathbf{memory}, \mathbf{network}, \\ &\mathbf{node}, \mathbf{process}, \mathbf{processor}, \mathbf{system}, \mathbf{thread}\}; \end{aligned}$$

DTyp: the set of *discrete data types*, given by

$$\begin{aligned} DTyp := &\{\texttt{bool}, \texttt{int}, \texttt{real}, \} \cup \\ &\{\mathbf{enum}\texttt{(}e_1\texttt{,} \ldots \texttt{,} e_n\texttt{)} \mid n \geq 1, e_i \in Ide\} \cup \\ &\{\texttt{[}l\texttt{..}u\texttt{]} \mid l, u \in \mathbb{Z}, l \leq u\} \cup \\ &\{\texttt{(}tp_1\texttt{,}tp_2\texttt{)} \mid tp_1, tp_2 \in DTyp\} \end{aligned}$$

where $n \geq 1$, and $e_i \in Ide$ for each $i \in [n]$; and

ATyp: the set of *analogue data types*, given by

$$ATyp := \{\texttt{clock}, \texttt{continuous}\}.$$

3.3.2 Data Types

For each type $tp \in DTyp \cup ATyp$, we let

$$D(tp) := \begin{cases} \mathbb{B} & \text{if } tp = \texttt{bool} \\ \{e_1, \ldots, e_n\} & \text{if } tp = \texttt{\textbf{enum(}}e_1\texttt{,} \ldots \texttt{,} e_n\texttt{\textbf{)}} \\ \mathbb{Z} & \text{if } tp = \texttt{int} \\ \mathbb{R} & \text{if } tp = \texttt{real} \\ \{l, \ldots, u\} & \text{if } tp = \texttt{[}l\texttt{..}u\texttt{]} \\ D_{tp_1} \times D_{tp_2} & \text{if } tp = \texttt{\textbf{(}}tp_1\texttt{,} tp_2\texttt{\textbf{)}} \\ \mathbb{R} \to \mathbb{R} & \text{if } tp = \texttt{clock} \\ \mathbb{R} \to \mathbb{R} & \text{if } tp = \texttt{continuous} \end{cases}$$

3.3.3 Component Types

CTyp is the set of *component types* defined in the concrete specification *S* as follows. Whenever *S* contains

$$cc\ name\ \ldots\ \texttt{\textbf{end}}\ name$$

where $cc \in Cat$, $name \in Nam$, then $name \in CTyp$.

Ports Let $c \in CTyp$. The *ports* of *c* are defined as follows. Whenever the specification of *c* contains **`features`** $\ldots$ p**`:`** pc**`;`** $\ldots$ with

- $pc =$ **`in event port`**, then $p \in IEPrt(c)$ and $typ(c, p) := \emptyset$,
- $pc =$ **`out event port`**, then $p \in OEPrt(c)$ and $typ(c, p) := \emptyset$,
- $pc =$ **`in data port`** tp, then $p \in IDPrt(c)$ and $typ(c, p) := D(tp)$,
- $pc =$ **`out data port`** $tp\ldots$, then $p \in ODPrt(c)$ and $typ(c, p) := D(tp)$,
- $pc =$ **`in event data port`** tp, then $p^e \in IEPrt(c)$, $p^d \in IDPrt(c)$ and $typ(c, p) := D(tp)$, and
- $pc =$ **`out event data port`** tp, then $p^e \in OEPrt(c)$, $p^d \in ODPrt(c)$ and $typ(c, p) := D(tp)$.

As abbreviations we use:

$$\begin{aligned} EPrt(c) &:= IEPrt(c) \cup OEPrt(c), \\ DPrt(c) &:= IDPrt(c) \cup ODPrt(c). \end{aligned}$$

3.3.4 Component Implementations

CImpl is the set of *component implementations* defined in the concrete specification *S* as follows. Whenever *S* contains

$$cc\ \texttt{\textbf{implementation}}\ type.name\ \ldots\ \texttt{\textbf{end}}\ type.name$$

where $cc \in Cat$, $type.name \in Nam$, then $type.name \in CTyp$ and $typ(type.name) = type$.

Modes The set $Mod(i)$ of modes of a component implementation $i \in CImpl$ is defined as follows. Whenever the concrete specification of i in S contains

$$\texttt{modes} \ldots m\texttt{:} \; [\texttt{initial}] \, \texttt{mode while} \; iv\texttt{;} \ldots,$$

then

- $m \in Mod(i)$,
- $ini(i) := m$ if the **`initial`** attribute is present, and
- $inv(i,m) := iv$;

If the specification of a component implementation $im \in CImpl$ does not contain any mode, then we assume that $Mod(i) = \{ini(i)\}$ for some **`initial`** mode $ini(i) = m_0$ with $inv(i,m_0) = \top$.

Subcomponents The sets $Sub(im)$ and $Act(im,m)$ of subcomponents and active subcomponents of a component implementation $im \in CImpl$ are defined as follows. Whenever the concrete specification of im in S contains

$$\texttt{subcomponents} \ldots sc\texttt{:} \; cc \; type \; bn \; sc' \ldots \texttt{in modes(}m_1\texttt{,} \ldots \texttt{,}m_n\texttt{);} \ldots$$

where $bn \in \{\texttt{accesses}, \texttt{running on}, \texttt{stored in}\}$, $sc, sc' \in Ide$, $cc \in Cat$, $type \in CTyp \cup CImpl \cup DTyp \cup ATyp$, and $n \geq 1$, then $sc \in Sub(im)$, $typ(sc) = type$ and for every $i \in [n]$ $sc \in Act(im,m_i)$.

Event Port Connections Let $im \in CImpl$ and $m \in Mod(im)$. The set $Con(im,m)$ of *event port connections* of im active in m are defined as follows. Whenever the specification of im contains

$$\texttt{connections} \ldots \texttt{port} \; sc_1.p_1 \; \texttt{->} \; sc_2.p_2 \; \texttt{in modes(}m_1\texttt{,} \ldots \texttt{,}m_n\texttt{);} \ldots$$

(where, for each $i \in [2]$ and $j \in [n]$, $sc_i \in Act(im,m_j) \cup \{\varepsilon\}$, $p_1 \in IEPrt(sc_1) \cup OEPrt(sc_1)$, and $p_2 \in IEPrt(sc_2) \cup OEPrt(sc_2)$), then

$$(sc_1.p_1, sc_2.p_2) \in Con(im)$$

and, for each $j \in [n]$, $(sc_1.p_1, sc_2.p_2) \in Act(im,m_j)$. Thus, according to the syntactic restrictions imposed in Section 2,

$$\begin{aligned} Con(im,m) \subseteq \; & IEPrt(im) \times SIE(im,m) \\ \cup \; & SOE(im,m) \times OEPrt(im) \\ \cup \; & SOE(im,m) \times SIE(im,m) \end{aligned}$$

as defined in Section 3.2.3.

Data Flows Let $im \in CImpl$ and $m \in Mod(im)$. The set $Flw(im,m)$ of *data flows* of *im* are defined as follows. Whenever the specification of *im* contains

$$\texttt{flows} \ldots \texttt{port}\ a\ \texttt{->}\ p\ \texttt{in modes(}m_1\texttt{,} \ldots \texttt{,} m_n\texttt{);} \ldots$$

where, for each $i \in [n]$, $p \in ODPrt(im) \cup SID(im,m_i)$, a is an expression over $IDPrt(im) \cup SOD(im,m_i)$, $SOD(im,m) := \{sc.p \mid sc \in Act(im,m), p \in ODPrt(c.sc)\}$, and $SID(im,m) := \{sc.p \mid sc \in Act(im,m), p \in IDPrt(c.sc)\}$, then

$$(a,p) \in Flw(im,m_i)$$

and, for each $j \in [n]$, $((a,p) \in Act(im,m_j)$.

Event Data Connections Let $im \in CImpl$ and $m \in Mod(im)$. Whenever the specification of *im* contains

$$\texttt{connections} \ldots \texttt{port}\ sc_1.p_1\ \texttt{->}\ sc_2.p_2\ \texttt{in modes(}m_1\texttt{,} \ldots \texttt{,} m_n\texttt{);} \ldots$$

(where, for each $i \in [2]$ and $j \in [n]$, $sc_i \in Act(im,m_j) \cup \{\varepsilon\}$, $p_1^e \in IEPrt(sc_1) \cup OEPrt(sc_1)$, and $p_2^e \in IEPrt(sc_2) \cup OEPrt(sc_2)$), then

$$(sc_1.p_1^e, sc_2.p_2^e) \in Con(im)$$

and, for each $j \in [n]$, $(sc_1.p_1^e, sc_2.p_2^e) \in Act(im,m_j)$.

Additionally,

$$(sc_1.p_1^d, sc_2.p_2^d) \in Flw(im,m_i)$$

and, for each $j \in [n]$, $((sc_1.p_1^d, sc_2.p_2^d) \in Act(im,m_j)$

Thus, for each event data port connection, both an event and data connection is created. to the mapped event ports p^e and data ports p^d.

Default values Using the `default` attribute, an implementation specification assigns default values to data subcomponents. In our semantics, this assignment is represented by the mapping $dfl(im,d)$. We assume that $dfl(im,d) = 0$ for every clock $d \in Clk(im)$.

Mode Transitions The set of *mode transitions* of a component implementation $im \in CImpl$, $MTr(im)$, is defined as follows: whenever the specification of *im* contains an entry of the form

$$\texttt{transitions} \ldots m\ \texttt{-[}t\ \texttt{when}\ g\ \texttt{then}\ f\,\texttt{]->}\ m'\texttt{;} \ldots$$

where $m,m' \in Mod(im)$, t is a trigger, g denotes a guard, and f is an effect (as defined in Section 2, then

$$(m,t,g,f,m') \in MTr(c).$$

Event data port references in transitions are split into their separated event and data counterparts. To this end, for each transition where $(p,a) = t$ and $p^e \in OEPrt(im)$, t is replaced by p^e, and $f \uplus \{p^d := a\}$. Furthermore, for each transition where $t = p^e \in IEPrt(im)$, for each `data(`t`)` expression in mode transition effects, the expression is replaced by p^d.

Reactivation Transitions The set of *reactivation transitions* of a component implementation $im \in CImpl$, $RTr(im)$, is defined as follows: whenever the specification of im contains an entry of the form

$$\texttt{transitions}\ldots m\,\texttt{-[@activation then}\,f\,\texttt{]->}\,m'\texttt{;}\ldots$$

where $m, m' \in Mod(im)$ and f is an effect, then

$$(m, f, m') \in RTr(im).$$

3.4 Temporal Formulas

The abstract syntax of properties is defined by the following table, where each row maps the constructs in the concrete syntax to the corresponding version in the abstract notation.

constraint	:=	*atom* |	ϕ	:=	$a \mid$
		not *constraint* |			$\neg\phi \mid$
		constraint **and** *constraint* |			$\phi \wedge \phi \mid$
		constraint **or** *constraint* |			$\phi \vee \phi \mid$
		constraint **implies** *constraint* |			$\phi \rightarrow \phi \mid$
		always *constraint* |			$G\phi \mid$
		never *constraint* |			$G\neg\phi \mid$
		in the future *constraint* |			$F\phi \mid$
		constraint **until** *constraint* |			$\phi\, U \phi$
		in the past *constraint* |			$O\phi \mid$
		constraint **since** *constraint*;			$\phi\, S\phi;$
atom	:=	**true** |	a	:=	$\top \mid$
		false |			$\bot \mid$
		term relation term |			$t \bowtie t \mid$
		time_until(*term*) *relation term* |			$\rhd_{\bowtie t} t \mid$
		change(*port* **)** |			$v' \neq v \mid$
		term ;			$t;$
term	:=	*port* |	t	:=	$v \mid$
		constant |			$c \mid$
		uninterpreted_function **(** *term* **)** |			$f(t) \mid$
		term function term |			$t \star t \mid$
		next(*port* **)** |			$v';$
		last_data(*event_port* **)** ;			$ld(t);$

3.5 Model Extension

The integration of the nominal and the error model is the so-called *(fault) model extension*. It modifies each nominal control component model for which a (non-trivial) error model is defined by enriching it by the error model specification, thus producing the extended model which represents both the nominal and the failure behavior. Informally, the extended model is obtained as follows:

- For each control component in the system, (an instantiation of) the associated error model is attached as a new **`system`** subcomponent to the nominal specification.
- The starting mode of the error subcomponent is defined to be the starting state of the error model. It is an **`initial`**/**`activation`** mode if the starting state is of type **`initial`**/**`activation`**, respectively.
- Each state transition of the error model gives rise to a mode transition of the error subcomponent.
- The set of event ports of the nominal model is extended by adding all error propagations (*propagation ports*) of the respective error model, in order to simulate the forwarding of error information via propagations by event communication.
- Correspondingly, the set of event port connections has to be extended by *propagation port connections*.

Error component translation For a given error model type E_t and implementation E_i, the nominal specification for both the corresponding component type and component implementation is defined as follows:

- A unique *name* is generated for the component type.
- For each outgoing error propagation p in E_t, a corresponding port is added to $\mathit{OEPrt}(c)$ with $\mathit{typ}(c,p) := \emptyset$;
- For each incoming error propagation p in E_t, a corresponding port is added to $\mathit{IEPrt}(c)$ with $\mathit{typ}(c,p) := \emptyset$. Furthermore, $\mathit{blk}(c,p) = \mathsf{false}$;
- For each error event e in E_i, a corresponding port is added to $\mathit{OEPrt}(c)$ with $\mathit{typ}(e,p) := \emptyset$. The event is furthermore annotated with the occurrence probability, if any;
- A reset port is added to $\mathit{IEPrt}(c)$ for the **`reset`** event.
- A special data port dp, named `errorState`, is added to $\mathit{ODPrt}(c)$, for which holds that $\mathit{typ}(c,dp) := \{s_1,\ldots,s_n\}$, for all states $s \in E_i$. This port contains an enumeration of values that correspond to the states specified in E_i.

The component type is then added to the specification such that $\mathit{name} \in \mathit{CTyp}$.

The component implementation i is generated as follows:

- A unique name is generated for the component implementation, matching the generated component type name.
- For each *clk* defined in the list of clocks, a data subcomponent *dclk* is added to $Sub(i)$.
- For each *s* in the list of error states of E_i, a mode *m* is added to $Mod(i)$ where
  - $ini(i) := m$ iff *s* is an **`initial`** state, or
  - $inv(i,m) := iv$ iff the invariant *iv* is defined for *s*, and
  - $StateMap(E_i,s) = m$;
- For each *et* in the set of *state transitions* of E_i, a transition *t*is added to $MTr(i)$ where if *et* is of the form

  $$s\,\texttt{-[}t\ \textbf{when}\ g\ \textbf{then}\ f\,\texttt{]->}\ s'\texttt{;}$$

  then
  $$t = (StateMap(E_i,s), t, g, f', StateMap(E_i,s'))$$
  , with $f' = f \uplus \{\texttt{errorState} := s'\}$
- For each *ert* in the set of *reactivation transitions* of E_i, a reactivation transition *rt*is added to $RTr(i)$ where if *ert* is of the form

  $$s\,\texttt{-[@activation}\ \textbf{then}\ f\,\texttt{]->}\ s'\texttt{;}$$

  then
  $$rt = (StateMap(E_i,s), f', StateMap(E_i,s'))$$
  , with $f' = f \uplus \{\texttt{errorState} := s'\}$
- For every *s* in the list of error states of E_i without an outgoing **`reset`** transition, the dummy transition

  $$(StateMap(E_i,s), \textbf{reset}, \mathsf{true}, \varepsilon, StateMap(E_i,s')) \in MTr'(i)$$

- References in invariant, guard or transition effect expressions are updated to point to the corresponding element in $Sub(i)$.

Error component integration In order to insert the error components into the nominal specification, the nominal component type *ct* and implementation *ci* are extended as follows (where *c* and *i* are the component type and implementation corresponding to the associated error model respectively):

- For each outgoing event port $p \in OEPrt(c)$ matching an outgoing error propagation, a corresponding port is added to $OEPrt(ct)$;
- For each incoming event port $p \in IEPrt(c)$ matching an incoming error propagation, a corresponding port is added to $IEPrt(ct)$;

- An outgoing data port `errorState` is added to $\mathit{ODPrt}(ct)$, matching the `errorState` port of c;
- The error implementation i is added to the subcomponents of ci: $sc_e \in \mathit{Sub}(ci)$ where $\mathit{typ}(sc_e) = i$;
- For each $m \in \mathit{Modes}(ci)$, and each $p \in \mathit{OEPrt}(i)$, $(m, (sc_e.p), \mathsf{true}, \varepsilon, m)$ is added to $\mathit{MTr}(ci)$;
- For each outgoing event port $p \in \mathit{OEPrt}(c)$ matching an outgoing error propagation, and for all modes $m \in \mathit{Mod}(ci)$, a connection is added $(sc_e.p, p) \in \mathit{Con}(ci, m)$;
- For each incoming event port $p \in \mathit{IEPrt}(c)$ matching an incoming error propagation, and for all modes $m \in \mathit{Mod}(ci)$, a connection is added $(p, sc_e.p) \in \mathit{Con}(ci, m)$;
- The `errorState` data port connection is added to c: for each $m \in \mathit{Mod}(ci)$,

$$\mathit{DCon}'(ci, m) := \mathit{DCon}(ci, m) \uplus \{(i.\texttt{errorState}, \texttt{errorState})\};$$

Port connections are made between instances of ci and its supercomponent, sibling components and subcomponents to support error propagations between components. Note that it is not necessary for a component to be associated with an error model for it to add error propagations, it suffices if one of its subcomponents has such an association. Connections can roughly be divided into three types:

- Propagations towards the environment.
- Propagations towards subcomponents;
- Propagations between sibling components;

The first type is already treated in the construction of the extended implementation. The other two types are specified in the following. Here, for any component implementation ci, we let $\mathit{ESub}(ci) \subseteq \mathit{Sub}(ci)$ be the set of subcomponents of ci which have an associated error model.

For any component implementation ci with an associated error model subcomponent sc_e, for all modes $m \in \mathit{Mod}(ci)$ and for all $sc \in \mathit{ESub}(ci)$ where $sc \in \mathit{Act}(ci, m)$:

- For each outgoing event port $p \in \mathit{OEPrt}(\mathit{typ}(sc))$ matching an outgoing error propagation, a connection is added $(sc.p, sc_e.p) \in \mathit{Con}(ci', m)$ iff $p \in \mathit{IEPrt}(\mathit{typ}(sc_e))$;
- For each incoming event port $p \in \mathit{IEPrt}(\mathit{typ}(sc))$ matching an incoming error propagation, a connection is added $(sc_e.p, sc.p) \in \mathit{Con}(ci', m)$ iff $p \in \mathit{OEPrt}(\mathit{typ}(sc_e))$.

Connections between subcomponents are added as well. For all modes $m \in \mathit{Mod}(ci)$ and for all pairs $(sc_2, sc_1) \in \mathit{Acc}(c, m) \cup \mathit{Acc}(c, m)^{-1} \cup \mathit{Run}(c, m) \cup \mathit{Sto}(c, m)$ where $(sc_2, sc_1) \in \mathit{ESub}(ci) \times \mathit{ESub}(ci)$:

- For each outgoing event port $p \in OEPrt(typ(sc_1))$ matching an outgoing error propagation, a connection is added $(sc_1.p, sc_2.p) \in Con(ci, m)$, iff $p \in IEPrt(typ(sc_2))$;

After integrating the error model, the following modifications are applied, in order:

- Application of force mode transitions;
- Insertion of fault effects;
- Event inhibition.

Applying forced modes Forced modes are specified by the tuple (s_i, fm), where

- s_i is the error state for which the forced modes are specified;
- $fm = \{m_1, \ldots, m_n\}$ specify the set of allowed modes in the specified error state;

If upon entering the error state $StateMap(E_i, s_i)$ the active mode of the component ci is not in fm, a transition is forced. To this end, the following is performed for each forced modes specification (s_i, fm):

Let for each $m_e \in Mod(i)$ the function

$$FM(s_i) := \begin{cases} fm & \text{if there is a forced mode } (s_i, fm) \\ Mod(ci) & \text{otherwise} \end{cases}$$

specify the allowed modes for each possible error state s_i. Furthermore, let $FM^{-1}(m) = \{s_i \mid m \in FM(s_i)\}$.

For each mode $m_n \in Mod(ci)$, and each $p \in IEPrt(i) \cup OEPrt(i)$:

- $inv(ci, m_n)$ is replaced by **(** $inv(ci, m_n)$ **and** $\bigvee_{s \in FM^{-1}(m_n)} (\texttt{errorState} = s)$**)**.
- For each mode $m \in Mod(ci)$, and for each $p \in IEPrt(i) \cup OEPrt(i)$, if there is a transition $(m_e, p, g, f, m'_e) \in MTr(i)$ where $m \notin FM(s)$, with $s = StateMap^{-1}(E_i, m'_e)$, add the transition $(m, p, \texttt{errorState} = s, \varepsilon, m_1)$, where $FM(s) = (m_1, \ldots, m_n)$.

The invariant prevents entering a mode that is not in the forced mode list. The added transitions provide the synchronizing transitions to force entering the first specified forced mode.

Inserting failure effects Failure effects are specified as a tuple (s_i, p, e_f), where

- s_i is the error state in which the failure effect applies;
- p is the targeted port or data component;
- e_f is the failure effect expression;

Failure effects are applied using either of two methods, depending on whether or not the target variable if the target of a data flow connection or not. For each individual failure effect (s_i, p, e_f), the following is performed:

- For each $(p,a) \in \mathit{Flw}(i,.)$ this connection is replaced by the flow $(p,$ **case** `errorState =` s_i **then** e_f **otherwise** a **end;**);
- For each (m,t,g,f,m') in $\mathit{MTr}(i)$ where $t \neq$ **reset** and t is not an error event:
  - If p `:=` $a \in f$, this assignment is replaced by $p :=$ **case** `errorState =` s_i **then** e_f **otherwise** a **end;** [1];
  - Otherwise $f = f \uplus \{p := e_f\}$.
- For each $(m,f,m') \in \mathit{RTr}(i)$:
  - If p `:=` $a \in f$, this assignment is replaced by $p :=$ **case** `errorState =` s_i **then** e_f **otherwise** a **end;** ;
  - Otherwise $f = f \uplus \{p := e_f\}$.
- For each $(m,t,g,f,m') \in \mathit{MTr}(i)$ where $t =$ **reset** or an error event, and $(s,t,g',f',s_i) \in \mathit{MTr}(\mathit{typ}(sc_e))$:
  - If p `:=` $a \in f$, this assignment is replaced by $p :=$ **case** `errorState =` s_i **then** e_f **otherwise** a **end;** ;
  - Otherwise if not $(p,a) \in \mathit{Flw}(i,m')$ then $f = f \uplus \{p$ `:=`$e_f\}$.

Event inhibition Event inhibitions are specified as tuples (s_i, ie), where $ie = \{e_1, \ldots, e_n\}$ is the set of inhibited event (data) ports. For each event inhibition:

- For each (m,t,g,f,m') in $\mathit{MTr}(ci)$, if $t \in ie$, then the guard g is replaced by **(** g **and** `errorState !=` s_i **)**.

[1]As an optimization, nested case expressions can be flattened into a single statement

4 Semantics

4.1 Overview

We define the semantics of a SLIM Model Instance *SI* both in terms of traces and automata: if the SLIM Model is instantiated in a component implementation (i.e. a component implementation is chosen as root), we define the corresponding automaton and the traces of *SI* coincide with the traces of the automaton; if, instead, the SLIM Model is instantiated in a component type, there is no corresponding automaton, and we define the traces of the component type in terms of its interface.

In the following, we define the semantics of component implementations in terms of a new automata model, called event-data automata, which is employed to formalize the local behavior of a component. We then define the network of event-data automata taking into account the interaction of subcomponents.

Finally, we define the traces of a component type and of a component implementation.

4.2 Formalizing the Local Behavior of Components

4.2.1 Event-Data Automata

An *event-data automaton (EDA)* is a tuple of the form

$$\mathfrak{A} = (M, m_0, X, v_0, \chi, \varphi, E, \longrightarrow, \rightrightarrows, blk)$$

where

- M is a finite set of *modes*,
- $m_0 \in M$ denotes the *starting mode*,
- X is a finite set of *variables*, partitioned into
  - *input variables*, IX,
  - *output variables*, OX, and
  - *local variables*, LX,
- $v_0 \in V_X$ is the *initial valuation* where V_X denotes the set of all *valuations*, that is, partial functions that assign values to the elements of X,
- $\chi : M \to (V_{LX} \to \mathbb{B})$ specifies the *mode constraints* (where we assume that $\chi(m_0)(v_0|_{LX}) =$ `true`),
- $\varphi : M \to (LX \to \mathbb{R})$ specifies the *trajectory equations* by associating with each local variable its derivative in the current mode[2],
- E is a finite set of *events*, partitioned into

[2] If $\varphi(m)(x) = 0$, then x is a discrete variable, if $\varphi(m)(x) = 1$, then it is a clock, and otherwise it is a hybrid variable.

  - *input events*, IE, and
  - *output events*, OE, and

- $\longrightarrow \subseteq M \times E_\tau \times (V_X \to \mathbb{B}) \times (V_X \to V_X) \times M$ is a finite *(mode) transition relation* where $E_\tau := E \cup \{\tau\}$. Transitions are represented in the form $m \xrightarrow{e,g,f} m'$, and e, g, and f are called the *trigger*, the *guard*, and the *effect*, respectively. Here f is allowed to modify only output and local variables, that is, $f(v)(x) = v(x)$ for each $v \in V_X$ and $x \in IX$.

- $\rightrightarrows \subseteq M \times (V_X \to V_X) \times M$ is a finite *reactivation transition relation*. Transitions are represented in the form $m \overset{f}{\rightrightarrows} m'$, and f is called the *effect*. Here f is allowed to modify only output and local variables, that is, $f(v)(x) = v(x)$ for each $v \in V_X$ and $x \in IX$.

- $blk \subseteq E$ is a set of blocking events.

4.2.2 Semantics of Event-Data Automata

The operational semantics of an EDA is given as a labeled transition system whose states, called *configurations*, are pairs of modes and valuations. Transitions either model the passing of time, involving an update of the non-discrete variables, or are internally triggered by events, including the invisible event τ. The second case requires the guard of the respective transition to be enabled, and then modifies the valuation of the variables according to the transition effect.

The definition of the semantics employs the following notation. Given a valuation $v \in V_X$, a time delay $t \in \mathbb{R}_{>0}$, and a mapping $\varphi : LX \to \mathbb{R}$ of trajectory equations, the notation $v + t \cdot \varphi$ denotes the corresponding temporal modification of the local variables, that is, for each $x \in X$,

$$(v + t \cdot \varphi)(x) := \begin{cases} v(x) + t \cdot \varphi(x) & \text{if } x \in LX \\ v(x) & \text{otherwise} \end{cases}$$

Formally, the *semantics* of an EDA is given by the labeled transition system

$$(Cnf, \kappa_0, L, \longrightarrow)$$

with

- the set of *(local) configurations* $Cnf := M \times V_X$,
- the *initial configuration* $\kappa_0 := (m_0, v_0) \in Cnf$,
- the set of *transition labels* $L := \mathbb{R}_{>0} \cup E_\tau$, and
- the *(local) transition relation* $\longrightarrow \subseteq Cnf \times L \times Cnf$, given by
  - *time transition*: $(m, v) \xrightarrow{t} (m, v + t \cdot \varphi(m))$ if
    - $t \in \mathbb{R}_{>0}$ and

    * the invariant stays valid for t time units[3]: $\chi(m)(v|_{LX} + t \cdot \varphi(m)) = \mathsf{true}$.
  - *internal or event transition*: $(m, v) \xrightarrow{e} (m', f(v))$ if
    * $e \in E_\tau$ and
    * in the current mode m, an e-transition is enabled where the invariant of the target mode is valid after applying the transition effect: there exists $m \xrightarrow{e,g,f} m'$ in $\mathfrak{A}$ such that $g(v) = \mathsf{true}$ and $\chi(m')(f(v)|_{LX}) = \mathsf{true}$.
  - *non-blocking transition*: $(m, v) \xrightarrow{e} (m, v)$ if
    * $e \notin blk$,
    * in the current mode m, no e-transition is enabled, i.e. there is no $m \xrightarrow{e,g,f} m'$ in $\mathfrak{A}$ such that $g(v) = \mathsf{true}$ and $\chi(m')(f(v)|_{LX}) = \mathsf{true}$.
  - *reactivation transition*: $(m, v) \rightrightarrows (m', f(v))$ if
    * in the current mode m, an reactivation-transition is enabled where the invariant of the target mode is valid after applying the transition effect: there exists $m \overset{f}{\rightrightarrows} m'$ in $\mathfrak{A}$ such that $\chi(m')(f(v)|_{LX}) = \mathsf{true}$.

4.2.3 Representing the Local Behavior of Component Implementations as Event-Data Automata

Given a component implementation, we can now define an EDA that represents the local behavior of the component. Here we denote the value of a given Boolean expression b (such as a mode invariant or a transition guard) with respect to a valuation by $[\![b]\!] : V_X \to \mathbb{B}$. Likewise, $[\![a]\!](v)$ denotes the value of an expression a with respect to the valuation $v \in V_X$.

The definition is based on the following associations:

- The meaning of modes in the SLIM component and in the EDA is identical.
- Incoming and outgoing data ports are interpreted as input and output variables, respectively, and data subcomponents are interpreted as local variables.
- Events in the EDA are either SLIM event ports, or are used to represent the event communication between a supercomponent and one of its (active) subcomponents. In the second case, they are of the form $sc.p$ where sc is the identifier of the subcomponent, and p its event port. Here an incoming event port in the subcomponent gives rise to an output event in the EDA of the supercomponent, and vice versa.
- Initial valuations, mode constraints and trajectory equations are directly taken from the SLIM Model.
- (Reactivation) Transition effects are determined as follows:
  - incoming data ports are not modified,

[3]Owing to the linearity of constraints, it suffices to check them in the target valuation only.

- outgoing data ports are updated according to the SLIM transition effect, if given, and not modified otherwise, and
- data subcomponents which are active in the target mode of the transition are updated according to the SLIM transition effect, if given, or else reset to their default value if they were inactive in the source mode, and not modified otherwise.

Formally, the association is defined as follows. For each component implementation $im \in CImpl$, $\mathfrak{A}_{im} = (M, m_0, X, v_0, \chi, \varphi, E, \longrightarrow, \rightrightarrows, blk)$ is given by letting

- $M := Mod(im)$,
- $m_0 := stm(im)$,
- $X := IX \cup OX \cup LX$ where
  - $IX := IDPrt(im)$,
  - $OX := ODPrt(im)$, and
  - $LX := \bigcup_{m \in Mod(im)} DAct(im, m)$,
- $v_0 := dfl(im)$,
- for every $m \in Mod(im)$, $\chi(m)$ is determined by the constraints occurring in $inv(im, m)$,
- for every $m \in Mod(im)$, $\varphi(m)$ is determined by the trajectory equations occurring in $inv(c, m)$,
- $E := IE \cup OE$ where[4]
  - $IE := IEPrt(im) \cup \{sc.p \mid sc \in CSub(im), p \in OEPrt(c.sc)\}$ and
  - $OE := OEPrt(im) \cup \{sc.p \mid sc \in CSub(im), p \in IEPrt(c.sc)\}$,
- $\longrightarrow := \{(m, p, [\![g]\!], [\![f]\!], m') \mid (m, p, g, f, m') \in MTr(im)\}$ where $[\![f]\!] : V_X \to V_X$ is defined as follows: $[\![f]\!](v) := v'$ with
  - for each $d \in IDPrt(im)$, $v'(d) := v(d)$,
  - for each $d \in ODPrt(im)$,

$$v'(d) := \begin{cases} [\![a]\!](v) & \text{if } f \text{ contains assignment } d := a \\ v(d) & \text{otherwise} \end{cases}$$

  - for each $d \in DAct(im, m')$,

$$v'(d) := \begin{cases} [\![a]\!](v) & \text{if } f \text{ contains assignment } d := a \\ dfl(c, d) & \text{else if } d \notin DAct(im, m) \\ v(d) & \text{otherwise} \end{cases}$$

[4]Here the construction can be optimized by only considering those events that occur as transition labels in c-transitions.

- $\rightrightarrows := \{(m, [\![f]\!], m') \mid (m, f, m') \in RTr(im)$ or $f = id$ and m' is the activation mode$\}$ where $[\![f]\!] : V_X \to V_X$ is defined the same as for $\longrightarrow$
- $blk = \{e \in E \mid blk(im, e)\}$ if im is atomic, while $blk = \emptyset$ if im is not atomic.

4.3 Formalizing the Global Behavior of Systems

Now we have to specify how the EDAs that represent single components interact with each other. This interaction is highly dynamic; local transitions can cause subcomponents to become (in-)active, and can change the topology of event and data port connections and flows. On the level of the formal model this means that both the activation of the component EDAs and their interconnection depend on the modes of the EDAs.

4.3.1 Networks of Event-Data Automata

A *network of event-data automata (NEDA)* is a tuple of the form

$$\mathfrak{N} = ((\mathfrak{A}_i)_{i\in[n]}, \alpha, EC, DD)$$

where

- each $\mathfrak{A}_i$ is an EDA of the form $\mathfrak{A}_i = (M_i, m_0^i, X_i, v_0^i, \chi_i, \varphi_i, E_i, \longrightarrow_i, \rightrightarrows_i, blk_i)$ $(i \in [n])$,
- $\alpha : M \to 2^{[n]}$ is the *activation mapping* (where $M := \prod_{i=1}^n M_i$ denotes the set of *global modes*),
- $EC : M \to \{i.e \mid i \in [n], e \in E_i\}^2$ is the *event connection mapping*, and
- $DD : M \to (\{i.x \mid i \in [n], x \in IX_i \cup OX_i\} \dashrightarrow \{j.a \mid j \in [n], a \in Exp(IX_j) \cup OX_j\})$ is the *data dependence mapping* where $Exp(IX_j)$ denotes the set of all expressions over IX_j.

4.3.2 Semantics of Networks of Event-Data Automata

The *semantics* of a NEDA is given by the labeled transition system

$$(Cnf, \kappa_0, L, \Longrightarrow)$$

which is defined in terms of the local transition systems $(Cnf_i, \kappa_0^i, L_i, \longrightarrow_i)$ $(i \in [n])$ of the constituent EDAs as follows (please refer to the next list for the definitions of the auxiliary functions):

- the set of *(global) configurations* is given by $Cnf := \prod_{i=1}^n Cnf_i$,
- the *initial configuration* is $\kappa_0 := (\kappa_0^1, \ldots, \kappa_0^n)$,
- the set of *transition labels* is $L := \mathbb{R}_{>0} \cup \{\tau\} \cup E_1$, and

- the *(global) transition relation*, $\Longrightarrow \subseteq Cnf \times L \times Cnf$, is given by
  - *time transition*: models the passing of time involving all active EDAs.

    $$\kappa = (\kappa_1, \ldots, \kappa_n) \overset{t}{\Longrightarrow} (\kappa'_1, \ldots, \kappa'_n)$$

    if there exists $t \in \mathbb{R}_{>0}$ such that for each $i \in [n]$, $\kappa_i \xrightarrow{t} \kappa'_i$ if $i \in \alpha(mod(\kappa))$, and $\kappa'_i = \kappa_i$ otherwise.
  - *internal transition*: represents an invisible step of an active EDA.

    $$\kappa = (\kappa_1, \ldots, \kappa_n) \overset{\tau}{\Longrightarrow} cns_\kappa(\kappa')$$

    if there exist $i \in \alpha(mod(\kappa))$, $\kappa_i \xrightarrow{\tau}_i \kappa'_i$, and, for all $j \in [n] \setminus \{i\}$, $\kappa_j = \kappa'_j$.
  - *multiway communication transition*: an output event is sent from an active EDA to all active EDAs connected to that event.

    $$\kappa = (\kappa_1, \ldots, \kappa_n) \overset{e}{\Longrightarrow} cns_\kappa(\kappa')$$

    if there exist $i \in \alpha(mod(\kappa))$, with $i \neq 1$, $oe \in OE_i$, $\kappa_i \xrightarrow{oe}_i \kappa'_i$, and either there exists $(i.oe, 1.e) \in EC(mod(\kappa))$ or $e = \tau$, and, for all $j \in [n] \setminus \{i\}$, either of the following holds:
    * $j \in \alpha(mod(\kappa))$ and there exists $ie \in IE_j$ s.t. $(i.oe, j.ie) \in EC(mod(\kappa))$ and $\kappa_j \xrightarrow{ie}_j \kappa'_j$,
    * $j \in \alpha(mod(\kappa))$ and there exists no $ie \in IE_j$ s.t. $(i.oe, j.ie) \in EC(mod(\kappa))$ and $\kappa_j = \kappa'_j$,
    * $j \in \alpha(mod(\kappa')) \setminus \alpha(mod(\kappa))$ and $\kappa_j \rightrightarrows_j \kappa'_j$,
    * $j \notin \alpha(mod(\kappa))$ and $j \notin \alpha(mod(\kappa'))$ and $\kappa_j = \kappa'_j$.
  - *input transition*: the first EDA provides an input event that is sent to all active EDAs connected to that event.

    $$\kappa = (\kappa_1, \ldots, \kappa_n) \overset{e}{\Longrightarrow} cns_\kappa(\kappa')$$

    if $e \in IE_1$ and, for all $j \in [n]$, either of the following holds:
    * $j \in \alpha(mod(\kappa))$ and there exists $ie \in IE_j$ s.t. $(1.e, j.ie) \in EC(mod(\kappa))$ and $\kappa_j \xrightarrow{ie}_j \kappa'_j$,
    * $j \in \alpha(mod(\kappa))$ and there exists no $ie \in IE_j$ s.t. $(1.e, j.ie) \in EC(mod(\kappa))$ and $\kappa_j = \kappa'_j$,
    * $j \in \alpha(mod(\kappa')) \setminus \alpha(mod(\kappa))$ and $\kappa_j \rightrightarrows_j \kappa'_j$,
    * $j \notin \alpha(mod(\kappa))$ and $j \notin \alpha(mod(\kappa'))$ and $\kappa_j = \kappa'_j$.

The definition employs the following auxiliary functions:

- $mod : Cnf \to M$, extracting the mode information from a given global configuration:

  $$mod(\kappa_1, \ldots, \kappa_n) := (mod(\kappa_1), \ldots, mod(\kappa_n)) \text{ with } mod(m, v) := m.$$

- $cns_\kappa : Cnf \to Cnf$, making a global configuration consistent by taking the (unique) solution of the equation system that is implied by the data dependence mapping. In addition, input or output variables that have been disconnected in the transition (that is, the variable occurs as a target in the data dependence relation of the old mode but in no data dependence of the new mode) are reset to their default values:
$$cns_\kappa((m_1,v_1),\ldots,(m_n,v_n)) := ((m_1,v'_1),\ldots,(m_n,v'_n))$$
if, for each $i \in [n]$ and $x \in IX_i \cup OX_i$,
$$v'_i(x) = \begin{cases} [\![a]\!](v'_j) & \text{if } DD(m_1,\ldots,m_n)(i.x) = j.a \\ v^i_0(x) & \text{if } DD(m_1,\ldots,m_n)(i.x) \text{ undefined,} \\ & \quad \text{and } DD(mod(\kappa))(i.x) \text{ defined} \\ v_i(x) & \text{if both } DD(m_1,\ldots,m_n)(i.x) \text{ and} \\ & \quad DD(mod(\kappa))(i.x) \text{ undefined} \end{cases}$$

4.3.3 Representing the Global Behavior of Component Implementations as Networks of Event-Data Automata

We now define the network associated to a component implementation. Apart from the activation mapping which can directly be taken from the SLIM Model, the essential idea is to analyze the connection and flow structure of event and data ports in the SLIM component implementation for generating the corresponding NEDA. For the event part this means that, for a given global mode of the system, all end-to-end (that is, multistep) connections between event ports are determined, and are taken into account in the event connection (*EC*) mapping. Here the following cases need to be considered:

- multistep out-to-in connections, involving zero or more direct out-to-out, one direct out-to-in, and zero or more direct in-to-in connections,
- multistep in-to-in connections, originating in the main component and involving zero or more direct in-to-in connections, and
- multistep out-to-out connections, ending in the main component and involving zero or more direct out-to-out connections.

The data dependence (*DD*) mapping can directly be determined from the data port connections and flows as defined in the SLIM specification.

Formally, given the collection of components in the SLIM specification S, the association of a corresponding NEDA,
$$\mathfrak{N}_S = ((\mathfrak{A}_i)_{i\in[n]}, \alpha, EC, DD),$$
can be defined as follows:

- each $\mathfrak{A}_i := \mathfrak{A}_{c_i}$ ($i \in [n]$) is constructed as described in Section 4.2.3,
- the activation mapping $\alpha : M \to 2^{[n]}$ is derived from Act as follows: for each $(m_1,\ldots,m_n) \in M$,

  - $1 \in \alpha(m_1, \ldots, m_n)$ and
  - whenever $i \in \alpha(m_1, \ldots, m_n)$, then $Act(i, m_i) \subseteq \alpha(m_1, \ldots, m_n)$,

- for each $(m_1, \ldots, m_n) \in M$,

$$\begin{aligned}
&EC(m_1, \ldots, m_n) := \\
&\qquad \{(i.op, j.ip) \mid i, j \in [n], op \in OE_i, ip \in IE_j, (c_i.op, c_j.ip) \in Con^+\} \\
&\quad \cup \quad \{(i.(sc.ip), j.ip) \mid i, j \in [n], sc \in Act(c_i, m_i), c_i.sc = c_j, sc.ip \in OE_i\} \\
&\quad \cup \quad \{(j.op, i.(sc.op)) \mid i, j \in [n], sc \in Act(c_i, m_i), c_i.sc = c_j, sc.op \in IE_i\} \\
&\quad \cup \quad \{(1.ip, i.ip') \mid i \in [n], ip \in IE_1, ip' \in IE_i, (c_1.ip, c_i.ip') \in Con^*\} \\
&\quad \cup \quad \{(i.op', 1.op) \mid i \in [n], op \in OE_1, op' \in OE_i, (c_i.op', c_1.op) \in Con^*\}
\end{aligned}$$

  and

- for each $(m_1, \ldots, m_n) \in M$, $i \in [n]$, and $x \in IX_i \cup OX_i$,

$$DD(m_1, \ldots, m_n)(i.x) := \begin{cases} j.y & \text{if } (y, sc.x) \in DCon(c_j, m_j) \text{ and} \\ & \quad c_j.sc = c_i \\ & \text{or } (sc_1.y, sc_2.x) \in DCon(c_k, m_k) \\ & \quad \text{and} c_k.sc_1 = c_j, c_k.sc_2 = c_i \\ & \text{or } (sc.y, x) \in DCon(c_i, m_i) \text{ and} \\ & \quad c_i.sc = c_j \\ i.a & \text{if } (a, x) \in Flw(c_i, m_i) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Here the notation $(c_i.op, c_j.ip) \in Con^+$ means that there is a multistep connection from the output port *op* of component c_i to the input port *ip* of component c_j, in the global mode $(m_1, \ldots, m_n)$, in the order

1. (zero or more) out-to-out connections,
2. (exactly one) out-to-in connection, and
3. (zero ore more) in-to-in connections.

Similarly, the notations $(c_1.ip, c_i.ip') \in Con^*$ and $(c_i.op', c_1.op) \in Con^*$ refer to a (possibly) empty sequence of in-to-in and out-to-out connections, respectively.

4.4 Traces

Given a set of data variables V (with $typ(v) \in AbsTyp$ for every $v \in V$) and a set of events E, a trace over V and E is a sequence $\sigma = s_0, e_0, s_1, e_1, \ldots$ such that:

- for all $i \geq 0$, s_i is an assignment to $V \cup \{T\}$,
- for all $i \geq 1$, e_i is an event in $E \cup \{\delta\}$,
- the sequence $s_0(T), s_1(T), \ldots$ is weakly monotonic and diverging,

- for all $i \geq 1$ if $e_i = \delta$ then $s_i(V) = s_{i-1}(V)$,
- for all $i \geq 1$ if $e_i \neq \delta$ then $s_i(T) = s_{i-1}(T)$.

The trace defines a continuous behavior, i.e. a state $\sigma(i,t)$ for every $i \geq 0$, $s_i(T) \geq t \geq s_{i+1}(T)$, defined as: for all $v \in V$, if $typ(v) \in AbsDTyp$, then $\sigma(i,t)(v) = s_i(v)$, while if $typ(v) \in AbsATyp$ and $s_i(v) = f$, then $\sigma(i,t)(v) = f(t)$.

4.4.1 Traces of an EDA

Given an EDA $\mathfrak{A} = \langle M, m_0, X, v_0, \chi, \varphi, E, \longrightarrow, \rightrightarrows, blk \rangle$, let $V_{\mathfrak{A}} := \{v_M\} \cup X$ and $E_{\mathfrak{A}} := E \cup \{\tau, @\}$, where v_M is a variable whose domain is M. A trace of $\mathfrak{A}$ is a trace $s_0, e_0, s_1, e_1, \ldots$ over $V_{\mathfrak{A}}$ and $E_{\mathfrak{A}}$ such that:

- $s_0(v_M) = m_0$ and $s_0(X) = v_0$,
- for all $i \geq 0, t \geq 0$, $\sigma(i,t) \models \chi(s_i(v_M))$,
- for all $i \geq 0, t \geq 0$, $\frac{d\sigma(i,t)}{dt} = \varphi(s_i(v_M))$,
- for all $i \geq 0$, if $e_i \neq \delta$, then either of the following holds:
  - *internal or event transition*:
    - $e_i \in E_\tau$ and
    - there exists $s_i(v_M) \xrightarrow{e,g,f} s_{i+1}(v_M)$ in $\mathfrak{A}$ such that $s_i(X) \models g$, $s_{i+1}(X) = f(s_i(X))$, and $s_{i+1}(v_M) \models \chi(s_{i+1}(v_M))$.
  - *non-blocking transition*:
    - $e_i \notin blk$ and
    - there exists no $s_i(v_M) \xrightarrow{e,g,f} s_{i+1}(v_M)$ in $\mathfrak{A}$ such that $s_i(X) \models g$, $s_{i+1}(X) = f(s_i(X))$, and $s_{i+1}(v_M) \models \chi(s_{i+1}(v_M))$.
  - *reactivation transition*:
    - $e_i = @$,
    - there exists $s_i(v_M) \overset{f}{\rightrightarrows} s_{i+1}(v_M)$ in $\mathfrak{A}$ such that $s_{i+1}(X) = f(s_i(X))$, and $s_{i+1}(v_M) \models \chi(s_{i+1}(v_M))$.

4.4.2 Traces of a Model Instance

If the root instance is a component type $c \in CTyp$, then the model traces are the traces of the set of variables $V(c) := DPrt(c)$ and the events $E(c) = EPrt(c) \cup \{\tau\}$.

If the root instance is an atomic component implementation $im \in CImpl$, then the model traces are the traces of the set of variables $V(im) := Dat(im) \cup \{mode\}$ and the events $E(im) = EPrt(im) \cup \{\tau\}$.

If the root instance is a component implementation $im \in CImpl$, then the model traces are the traces of the set of variables and events defined recursively as follows:

- $V(im) = Dat(im) \cup \{mode\} \cup \{sc.v \mid sc \in CSub(im), v \in V(typ(sc))\}$,
- $E(im) = EPrt(im) \cup \{\tau\} \cup \{sc.e \mid sc \in CSub(im), e \in E(typ(sc))\}$.

4.5 Temporal Formulas

Global symbols in Σ are interpreted, as in standard first-order logic, by an interpretation mapping $\mathscr{I}$ that maps each symbol in a constant or functions in the structure corresponding symbol's type. For example, if f is a function symbol with type $typ(f) = \mathbb{R}, \mathbb{B}$, then $\mathscr{I}$ is a function from reals to Boolean values.

Given an interpretation mapping $\mathscr{I}$ and a trace $\sigma = s_0, e_0, s_1, e_1, \ldots$, let $\pi := \langle \mathscr{I}, \sigma \rangle$. $\pi \models \phi$ iff $\pi_0 \models \phi$, which is defined as follows:

$\sigma_i(v) = s_i(v)$
$\sigma_i(c) = c$
$\sigma_i(f(t)) = \mathscr{I}(f)(\sigma_i(t))$
$\sigma_i(t_1 \star t_2) = \sigma_i(t_1) \star \sigma_i(t_2)$
$\sigma_i(t') = \sigma_{i+1}(t)$
$\sigma_i(\rhd\phi) = \sigma_j(T) - \sigma_i(T)$ where j is the smallest index $\geq i$ such that $\sigma_j \models \phi$
$\sigma_i(ld(e)) = \sigma_j(data(e))$ where j is the biggest index $\leq i$ such that $\sigma_j(e)$ is true, or $\sigma_i(ld(e)) = \mathscr{I}(ld(e))$ if such j does not exist
$\sigma_i \models \top$
$\sigma_i \not\models \bot$
$\sigma_i \models e$ iff $e = e_i$
$\sigma_i \models t_1 \bowtie t_2$ iff $\sigma_i(t_1) \bowtie \sigma_2(t_2)$
$\sigma_i \models \neg\phi$ iff $\sigma_i \not\models \phi$
$\sigma_i \models \phi_1 \wedge \phi_2$ iff $\sigma_i \models \phi_1$ and $\sigma_i \models \phi_2$
$\sigma_i \models \phi_1 \vee \phi_2$ iff $\sigma_i \models \phi_1$ or $\sigma_i \models \phi_2$
$\sigma_i \models \phi_1 \rightarrow \phi_2$ iff $\sigma_i \not\models \phi_1$ or $\sigma_i \models \phi_2$
$\sigma_i \models G\phi$ iff for all $j \geq i$ $\sigma_j \models \phi$
$\sigma_i \models F\phi$ iff there exists $j \geq i$ $\sigma_j \models \phi$
$\sigma_i \models \phi_1\ U\ \phi_2$ iff for some $j \geq i$, $\sigma^j \models \phi_2$ and for all $0 \leq k < j$, $\sigma^k \models \phi_1$
$\sigma_i \models O\phi$ iff there exists j, $0 \leq j \leq i$, $\sigma_j \models \phi$
$\sigma_i \models \phi_1\ S\ \phi_2$ iff for some j, $0 \leq j \leq i$, $\sigma^j \models \phi_2$ and for all $j < k \leq i$, $\sigma^k \models \phi_1$

4.6 Probabilistic semantics

The probabilistic semantics of SLIM are defined in terms of a Markov Chain, more specifically an Interactive Markov Chain (IMC). In effect, the reachable state space of the NEDA is transformed into a transition system, which is extended with probabilistic transitions with exponential distribution based on the rates attached to error events. Probabilistic models are only supported for NEDA which do not contain any timed transitions. When probabilistic analysis is performed, such transitions are ignored.

An interactive Markov chain is defined as the tuple $\mathscr{I} = (S, Act, \rightarrow, \Rightarrow, s_0)$, where

- S is the set of states of the Markov chain;
- Act the set of possible actions including the internal action τ;
- $\rightarrow \subseteq S \times Act \times S$ the action transition relation;

- $\Rrightarrow \subseteq S \times \mathbb{R}_{>0} \times S$ the Markovian (or probabilistic) transition relation;
- s_0 the initial state.

Let $\mathfrak{N} = ((\mathfrak{A}_i)_{i\in[n]}, \alpha, EC, DD)$ be a NEDA. Taking the labeled transition system $(Cnf, \kappa_0, L, \Longrightarrow)$ imposed by $\mathfrak{N}$, we define $\mathscr{I}$ as follows:

- $S = Cnf$;
- $Act = L \setminus \mathbb{R}$;
- $\rightarrow = \{(\kappa \times l \times \kappa) \mid (\kappa, l, \kappa') \in \Longrightarrow\}$, where l not associated with an error rate and $l \notin \mathbb{R}$;
- $\Rightarrow = \{(\kappa \times \lambda \times \kappa) \mid (\kappa, l, \kappa') \in \Longrightarrow\}$, where $l \in E$ is associated with an error rate $\lambda \in \mathbb{R}_{>0}$;
- $s_0 = \kappa_0$.

The states of the IMC are simply the possible configurations of the NEDA. The transitions of the NEDA are separated into action transitions, and Markovian transitions. Transitions which are labeled by an error event which has been associated with a rate λ are transformed into Markovian transitions and timed transitions are ignored. Otherwise an action transition is added.

4.6.1 Maximal Progress

When there is a state $s \in S$ such that $(s, l, s') \in \rightarrow \wedge (s, \lambda, s'') \in \Rightarrow$ (for some $s', s'' \in S, l \in L$, and $\lambda \in \mathbb{R}$), the probabilistic transition (s, λ, s'') will be ignored. This is due to the *maximal progress assumption*, where it is assumed that an action $l \in Act$ takes zero time to execute. As it holds that the probability of the time for transition (s, λ, s'') to be zero is exactly zero (i.e. $1 - e^{\lambda t}$ with t being 0), it is assumed that transition (s, l, s') will always be taken.

5 Comparison with AADL

Though SLIM strives to be compliant with AADL, there are some differences between the two languages. This section gives an overview of the differences between the two languages.

The primary difference lies in the fact that SLIM only supports a subset of AADL. To name a few, flows, feature groups and subprograms do not occur in SLIM. In Table 9 an indication is given of supported elements. More constructs may be available in AADL that are not listed here. For a full overview, refer to [RD6].

Other differences in syntax can primarily be summarized as follows:

- Expressions as data port connection sources;
- States and state transitions;
- Port connection semantics;
- Error models;
- Predefined properties;
- Case sensitivity, with lowercase keywords;
- Some extra reserved keywords that otherwise are valid identifiers in AADL.

As SLIM uses its own behavioral semantics, there are some deviations from AADL to accommodate for this. This relates to the first three items mentioned in the list above. Port connections are always synchronizing, therefore there is no queuing of data or events. To avoid problems in the semantics of reading and writing of data, the source of a data port connection is allowed to be some expression, see Section 2.10.3. SLIM further defined states and state transitions to reason over data values specified for a component. They allow modifying data explicitly (with transition effects), guard transitions and place invariants on states.

A syntax for error models is directly integrated into the language. For details on these error models, see Section 2.12

SLIM does not interpret the properties that are defined by default for AADL, and defines a default set of properties on its own (see Section 2.3). Default AADL properties may still be used, but are not taken into account for analysis.

SLIM is case sensitive. This holds for keywords and identifiers, where keywords have to be written lowercase. In particular, identifier that differ in casing only are considered distinguishable. Finally, as SLIM integrates a few extra language concepts, such as expressions and error models, extra keywords are introduced, which in pure AADL would be considered valid identifiers.

Table 9: Supported AADL constructs

Feature	Supported
Annexes	N
Property sets	Y
Basic properties	Y
List and record properties	Y
Reference and classifier properties	Y
Public/Private Packages	Y
Component types and implementations	Y
Extending types and implementations	N
Thread	Y
Thread group	Y
Process	Y
Data	Y
Subprogram	N
Subprogram group	N
Processor	Y
Virtual processor	N
Memory	Y
Bus	Y
Virtual bus	N
Device	Y
System	Y
Abstract	Y
Prototypes	N
Features	Only in *or* out ports
Flows	N
Modes in types	N
Subcomponents	Y
Subcomponent refinement	N
Subcomponent arrays	N
Calls	N
Connections	Partially for port connections, see Section 2.10.2
Flows	N
Modes in implementations	Y
Inherited modes	N
Mode-specific properties	N