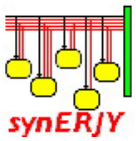


synERJY 5.1

Language Reference Manual

Reinhard Budde
Axel Poigné
Karl-Heinz Sylla

Fraunhofer Institut

Autonome intelligente Systeme

Fraunhofer AiS

February 6, 2007

Preface

synERJY[1] is a programming language and a design environment for embedded reactive systems that combines two paradigms:

- *Object-oriented modelling* for a robust and flexible design.
- *Synchronous execution* for precise modelling of reactive behaviour.

Highlights are that

- *synERJY* provides a deep embedding of the reactive behaviour into the object-oriented data model.
- *synERJY* offers fine-grained integration of synchronous formalisms such as ESTEREL [2], LUSTRE [7], and STATECHARTS [8].[2]

The programming environment supports compilation, configuration, simulation, and testing, as well as verification by model checking. Behavioural descriptions may be edited in graphical or in textual form. Code generators for efficient and compact code in C and several hardware formats are available.

The *synERJY* language and its programming environment are developed at the Fraunhofer Institut Autonome Intelligente Systeme.[3] The programming environment is freely available with a public domain license.

Contact: {reinhard.budde,poigne,sylla}@ais.fraunhofer.de.

[1] *synERJY* may be read as **syn**chronous **e**mbedded **r**eactive **J**ava with a **y** for the sound.

[2] We recommend Halbwachs' book *Synchronous Programming of Reactive Systems* [6] as an excellent introduction.

[3] The development of *synERJY* has been partially supported by the ESPRIT LTR SYRF, "Synchronous formalisms", and the ESPRIT IIM Project CRISYS, "Critical Systems and Instrumentation".

Contents

1 Introduction

synERJY is a strongly typed object-oriented synchronous language for the design of dependable embedded real-time software. It is based on and designed for maximal compatibility with JAVA™. The application area of embedded systems demands for some restrictions, though:

- Neither dynamic loading of packages
- nor threads are not supported,
- arrays are two-dimensional at most, and
- primitive types are machine dependent.

The restrictions should be acceptable in that dynamic loading may impair the predictability of behaviour, in particular with regard to timing constraints. Arrays are one- or two-dimensional for supporting a better memory layout. Two-dimensional arrays are "square" in that the length of all rows and columns are equal.

On the other hand, the language has been extended to accommodate synchronous programming of reactive behaviour. Particularly, *synERJY* features

- signal based broadcast communication combined with
- parallel execution of and within objects,
- a compile-time schedule of activities (such as, e.g., method calls), and
- support for controller design and digital signal processing (by combining the imperative, state machine, and data-flow style of synchronous programming).
- support for vectors and matrices (i.e. arrays of fixed size) for signal processing.
- generation of efficient target code (ANSI-C and hardware formats).

The main concepts of the language are discussed at length in the complimentary *Introduction to synERJY* [3] that is recommended as a first reading.

1.1 The Reactive Execution Model

The synchrony hypothesis. Synchronous programming is a way of specifying discrete-time (control) systems. Many formalisms do the same. However, synchrony as a paradigm avoids ad-hoc solutions and offers a mathematically precise and, as we do believe, also a simple programming model.

Synchronous behaviour is modelled as a machine that, on actuation, reads the input sensors and is then *decoupled from the environment* to compute the subsequent *state* and the value of the output signals, dispatches the output signals to the environment, and waits for the next actuation. Such an execution step is called an *instant*.

Execution, of course, takes time. The delay does not cause harm as long as it is sufficiently small in that an instant always terminates before the next actuation takes place. Then we may use the hypothesis that input is sampled and output is generated "at the same time". The reaction appears to be "instantaneous" with regard to the observable time scale in terms of activations. Berry [2] speaks of the *synchrony hypothesis*. Since time cannot be measured "in between" activations one sometimes speaks of a "zero time model".

Sensors and signals are broadcast in synchronous programming, i.e. at every instant there is a system-wide consistent view of all sensors and signals. In particular

- the value of a signal is never updated at an instant after the value has been read once (*write-before-read* strategy).

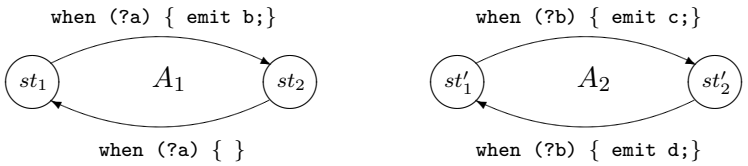
The subsequent state and the output signals must be uniquely determined by the input sensors and the present state.

Synchronous languages nevertheless support concurrency. Possible nondeterminism is resolved by the compiler. If no deterministic scheduling can be found, either due to a *time race* or due to a *causality cycle*, an error message is issued and the program is rejected.

Synchronous formalisms. Roughly, synchronous formalisms may be distinguished according to whether they are based on control flow or data flow.

The formalisms based on control flow (like Esterel [2] or Statecharts [8]) share the idea that signals contribute to the control flow. At an instant, a signal may be emitted to be *present*. Otherwise it is *absent*. One may check for the presence of a signal.

Consider, for example, the following two simple automatons that are supposed to run in parallel



The expression `?a` asks whether the signal `a` is present. Hence, if the automatons are in state st_1 and state st'_1 and if the signal `a` is present the automaton A_1 moves from state st_1 to state st_2 emitting `b`. In parallel the automaton A_2 checks for the presence of `b` provided it is in state st'_1. Since `b` is present the A_2 moves from state st'_1 to state st'_2 emitting `c`. In that the signal `b` is part of the control flow.

Synchronous languages based on data flow (such as LUSTRE[7] and SIGNAL[9]) are inspired by difference equations such as s

$$\begin{aligned} z(n) &= 0.5 * z(n-1) + 0.3 * y(n) + 0.2 * x1(n) \\ y(n) &= 0.5 * x1(n) + 0.5 * x2(n) \end{aligned}$$

Each of the equations is evaluated at time n to compute a new value of the respective variable. $x1$ and $x2$ are assumed to be inputs.

For difference equations, the order of evaluation is determined by the variables only: at an instant, the value of a variable must be computed before it is used in some equation.

It is a minor shift to rewrite the difference equations as *data flow equations*. Consider each of the variables as to refer to an indexed sequence of data, a *data flow*, and we assume that operations like addition and multiplication are defined point-wise.

$$\begin{aligned} y &= 0.5 * x1 + 0.5 * x2 \\ z &= 0.5 * pre(z) + 0.3 * y + 0.2 * x1 \end{aligned}$$

For the second equation, we need what is called a *time shift operator* `pre` such that $\texttt{pre}(z)(n) = z(n-1)$.

1.2 Programming in *synERJY*

synERJY smoothly integrates the control flow and data flow flavours of synchronous programming with object-oriented structuring principles. We superficially discuss two, somewhat artificial, examples to illustrate the integration.[4]

[4] At this stage, one should not expect fully to grasp the meaning of the programs. We recommend [3] as a first reading for a deeper understanding of the language.

Synchrony and object-orientation. The basic structure of *synERJY* compares to that of JAVA™. An application is specified in terms of classes. A class with a method `main` is a *configuration class* from which an application may be generated.

Classes in *synERJY* may be reactive, i.e. exhibit some reactive behaviour. Here is a simple example.

```
class Basic {
    static final time timing = 250msec;
    Sensor button  = new Sensor(new SimInput());
    Signal red_led = new Signal(new SimOutput());

    public Basic () {
        active {
          loop {
              await ?button;
              emit  red_led;
              next;
          };
        };
    };

    public static void main (String[] args) {
       while (instant() == 0) {};
    };
}
```

(cf. `ref_basic.se`) There is sensor `button`, and there is a signal `red_led` that may be emitted. The reactive code is embedded using the `active` statement. It just says that the program waits for the presence of the sensor `button` (The notation `?button` states that one asks for the presence of a sensor or signal), and then immediately emits the signal `red_led`. Then it pauses till the next instant to wait again for the presence of the sensor `button`.

Sensor and signal types behave like reference types. They are created using a constructor. Here the constructors have objects of type `SimInput` and `SimOutput` as arguments. These specify the interaction with the simulator of the *synERJY* tool set. In general, "input" and "output" objects as parameters of a sensor and signal constructor specify the interface of a sensor or signal to the environment. Sensors have parameters of type "input", signals have parameters of type "output".

If several reactive objects are generated in an application, the objects executes in parallel but, since they may share sensors and signals, some scheduling restrictions are imposed.

The method `main` is an infinite loop that calls the system method `instant`. This method executes one step of the reactive machine. It returns an exception code as value. The value 0 states that no exception is thrown.

Integration of synchronous formalism. We demonstrate the integration of two styles – the automatons style and the data flow style – again by a simple example.

```
class UpAndDownCounter {

  Signal<int> count = new Signal<int>(new SimOutput());

  public UpAndDownCounter () {
      active {
          emit count(0);
          automaton {
              init { next state up; };
              state up     // count upwards
                  during {|  count := pre(count) + 1; |}
              when  ($count > 9) { next state down; };
              state down  // count downwards
                  during {| count := pre(count) - 1; |}
              when  ($count < 1) { next state up; };
          };
      };
   };
}
```

(cf. `ref_up-and-down-counter-ref.se`) If started the signal `count` is emitted with value 0, and then the automaton immediately enters state `up`. During being in state `up` the data flow equation `count := pre(count) + 1;` is evaluated. The flow term on the right hand side states that its value at an instant is computed by increasing the value of the signal `count` at the previous instant by 1. When the counter has value 10 state changes from `up` to `down` (the notation `$count` states that the value of a signal is accessed). During being in state `down` the other flow equation is evaluated decreasing the counter.

Note that signals can be emitted as well as constrained by a flow equation. This unification of signals as known of ESTEREL and flow variables as known of LUSTRE is one of the particular highlights of *synERJY*. In general, we say that a signal is *updated* either when it is emitted or constrained by a flow equation.

1.3 Presentation

This reference manual focuses on the synchronous extensions, though it aims for completeness. The style of presentation follows that of the JAVA™ language specification in [5]. which we strongly recommend as complimentary reading. We stress the differences between *synERJY* and JAVA™.

1.4 Compilation and Programming Environment

synERJY is compiled to a simple sub-language of ANSI-C as intermediate code for cross-compiling to different target architectures. The generated code is fairly efficient in that it may run even on small micro controllers. Restricted compilation schemes additionally generate several hardware formats for pure control applications.

synERJY provides a programming environment including a graphic editor for state machine and a simulator for tracing the values of sensors, signals, and fields. These are documented in complimentary user manual [4].

2 Grammars

We assume the reader to be familiar with context-free grammars which are crucial for syntactic representation.[5] The notation for context-free grammars follows that of the JAVA™ reference manual. Terminal symbols are shown in `fixed width` font, nonterminals in *italic*. A syntactic definition is of the form

Arguments:
    Argument
    Argument *Arguments*

The nonterminal being defined is in the first line followed by a colon, the alternatives listed in the following lines. Long definitions may be extended to a second line using indentation. Optional constructs are indicated by the subscript $\ldots_{opt}$.

In the HTML-version of this document, a nonterminal is linked to its definition. In the TEX-version, superscripts of nonterminals specify the page number of its definition.

The lexical grammar has the non-terminal *Input*[14] as start symbol. It specifies the translation of streams of ASCII input characters into input token.

The syntactic grammar specifies the sequences of input tokens that form a syntactically correct program. Its starting symbol is *CompilationUnit*[40].

[5] A short overview may be found in [5].

3 Lexical structure

3.1 Lexical Translations

The lexical translation transforms a sequence of ASCII characters into a sequence of tokens that are the terminal symbols of the syntactic grammar (*synERJY* does not support Unicode).

3.2 Line Terminators

Lines are terminated by the ASCII characters CR, or LF, or CR LF. The latter is counted as one line terminator.

LineTerminator:
the ASCII LF character, also known as "newline"
the ASCII CR character, also known as "return"
the ASCII CR character followed by the ASCII LF character
InputCharacter:
ASCII Characters but not CR or LF

3.3 Input Elements and Tokens

Input elements that are not whitespace or comments are tokens. Tokens are the terminals of the syntactic grammar.

Input:
InputElements[14]$_{opt}$
InputElements:
InputElement[14]
InputElements[14] *InputElement*[14]
InputElement:
WhiteSpace[15]
Comment[15]
Token[14]
Token:
Identifier[16]
Keyword[16]
Literal[17]
Separator[20]
Operator[20]

3.4 Whitespace

White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators

WhiteSpace:
the ASCII SP character, also known as "space"
the ASCII HT character, also known as "horizontal tab"
the ASCII FF character, also known as "form feed"
LineTerminator[14]

3.5 Comment

There are two kinds of comments: an in-line comment of the form "`/*` *comment* `*/`", and an end of line comment of the form "`//` *comment*". The grammar rules are:

Comment:
TraditionalComment[15]
EndOfLineComment[15]
TraditionalComment:
`/ *` *NotStar*[15] *CommentTail*[15]
EndOfLineComment:
`//` *CharactersInLine*[15]$_{opt}$ *LineTerminator*[14]
CommentTail:
`*` *CommentTailStar*[15]
NotStar[15] *CommentTail*[15]
CommentTailStar:
`/`
`*` *CommentTailStar*[15]
NotStarNotSlash[15] *CommentTail*[15]
NotStar:
InputCharacter[14] but not `*`
LineTerminator[14]
NotStarNotSlash:
InputCharacter[14] but not `*` or `/`
LineTerminator[14]
CharactersInLine:
InputCharacter[14]
CharactersInLine[15] *InputCharacter*[14]

3.6 Identifiers

An identifier is an unlimited-length sequence of ASCII letters and ASCII digits, the first of which must be a letter or _. An identifier cannot be a keyword, a boolean literal, or the null literal. It is not allowed to contain two consecutive _.

Identifier:
    IdentifierChars[16]
    but not a *Keyword*[16] or *BooleanLiteral*[19] or *NullLiteral*[19]
IdentifierChars:
    ASCIILetter[16]
    IdentifierChars[16] *ASCIILetterOrDigit*[16]
ASCIILetter:
    any ASCII character
ASCIILetterOrDigit:
    any ASCII character that is a ASCII letter-or-digit

Two identifiers are equal if they consist of the same sequence of ASCII letters.

3.7 Labels

Labels are used to mark a position within a program. Labels are identifiers followed by two colons (no blanks in between).

Label:
    Identifier[16] : :

3.8 Keywords

The following words are reserved as keywords:

Keyword:
    UsedKeyword[17]
    ReservedKeyword[17]

UsedKeyword: one of

`abstract`	`do`	`implements`	`parameter`	`sustain`
`activate`	`dt`	`import`	`post`	`switch`
`active`	`during`	`import_from_C`	`pre`	`then`
`assert`	`else`	`init`	`precedence`	`this`
`automaton`	`emit`	`instanceof`	`private`	`throw`
`await`	`entry`	`instant`	`ptl`	`transient`
`blacboard`	`exit`	`interface`	`protected`	`until`
`break`	`export_to_C`	`interrupt`	`public`	`void`
`cancel`	`extends`	`invariant`	`reactive`	`volatile`
`case`	`fairness`	`loop`	`return`	`when`
`class`	`final`	`ltl`	`schedule`	`while`
`const`	`for`	`native`	`state`	
`continue`	`formula`	`new`	`static`	
`ctl`	`goto`	`next`	`strictfp`	
`current`	`halt`	`node`	`strongly`	
`default`	`if`	`nothing`	`super`	

ReservedKeyword: one of

`catch`	`finally`	`simple`
`equal`	`not_equal`	`synchronized`

3.9 Literals

A *literal* present a value of primitive type, of `String` type, or of `null` type.

Literal:
- *IntegerLiteral*[17]
- *FloatingPointLiteral*[18]
- *BooleanLiteral*[19]
- *CharacterLiteral*[19]
- *StringLiteral*[19]
- *NullLiteral*[19]
- *TimeLiteral*[20]

Integer literals. An integer literal may be a decimal literal (base 10) or a hexadecimal literal (base 16). Octal literals are not supported.

IntegerLiteral:
- *DecimalNumeral*[17]
- *HexNumeral*[18]

DecimalNumeral:
- `0`

NonZeroDigit[18] *Digits*[18]$_{opt}$

Digits:

Digit[18]

Digits[18] *Digit*[18]

Digit:

`0`

NonZeroDigit[18]

NonZeroDigit: one of

`1 2 3 4 5 6 7 8 9`

HexNumeral:

`0 x` *HexDigits*[18]

`0 X` *HexDigits*[18]

HexDigits:

HexDigit[18]

HexDigit[18] *HexDigits*[18]

HexDigit: one of

`0 1 2 3 4 5 6 7 8 a b c d e f A B C D D F`

An integer literal is of an integral type (cf. Section 4.2). The type is determined automatically by choosing the most appropriate type according to the context. If the context does the type the least possible type is chosen according to the type hierarchy of integral types (cf. Section 4.2). For instance, the literal `15` is of type `byte` (and, hence of all the super types of `byte`). Applying a cast may change the type, e.g. `(int)15` is of type `int`.

Floating-Point literals. The elements of the types `float` and `double` conform with the IEEE 754 32-bit single-precision and 64-bit double-precision binary floating-point formats.

FloatingPointLiteral:

Digits[18] `.` *Digits*[18]$_{opt}$ *ExponentPart*[18]$_{opt}$ *FloatTypeSuffix*[19]

`.`$_{opt}$ *Digits*[18] *ExponentPart*[18]$_{opt}$ *FloatTypeSuffix*[19]

Digits[18] *ExponentPart*[18] *FloatTypeSuffix*[19]

Digits[18] *ExponentPart*[18]$_{opt}$ *FloatTypeSuffix*[19]

ExponentPart:

ExponentIndicator[18] *SignedInteger*[19]

ExponentIndicator: one of

`e E`

SignedInteger:
    Sign[19]$_{opt}$ *Digits*[18]
Sign: one of
    `+ -`
FloatTypeSuffix: one of
    `f F d D`

Floating point literals are of type `double` by default.

Boolean literals. As usual, there are two Boolean values, true and false.

BooleanLiteral: one of
    `true  false`

Character literals. In contrast to JAVA™, character literal are pure ASCII.

CharacterLiteral:
    ' *SingleCharacter*[19] '
SingleCharacter:
    InputCharacter[14] but not ' or

String literals. A string literal consists of zero or more characters enclosed in double quotes.

StringLiteral:
    " *StringCharacters*[19]$_{opt}$ "
StringCharacters:
    StringCharacter[19]
    StringCharacters[19] *StringCharacter*[19]
StringCharacter:
    InputCharacter[14] but not " or

Null literal. There is only one value of null type, the null reference, which always is of null type.

NullLiteral:
    `null`

Time literals. A *time literal* consists of a decimal numeral followed by one time suffixes `msec`, `usec`, `sec`, `min`, or `hour` with the obvious intended meaning. For notational convenience, a time literal may be of the format `12.3456sec`.

TimeLiteral:
    DecimalNumeral[17] *TimeUnitSuffix*[20]
    DecimalNumeral[17] `.` *DecimalNumeral*[17] `sec`
TimeUnitSuffix: one of
    `msec usec sec min hour`

3.10 Separators

The following ASCII characters are the separators:

Separator: one of
    `( ) { } [ ] < > ; , . :`

3.11 Operators

The following ASCII characters are the separators:

Operator: one of

```
=   <   >   !   ~   |   ?   :
==  <=  >=  !=  &&  ||  ++  --
+   -   *   /   &   |   ^   %   <<   >>   >>>
+=  -=  *=  /=  &=  |=  ^=  %=  <<=  >>=  >>>=
```

4 Types, Values, and Variables

4.1 Kinds of Types

Types are either *primitive types*, *reference types*, or signal types. There is a special *null type* with the only value being the null reference `null`.

Type:
    PrimitiveType[21]
    ReferenceType[24]
    SignalType[27]

Every variable and every expression has a type that can be determined at compile time.

4.2 Primitive types

Compared to JAVA™, there are a couple of new primitive types.

- the type `time` (cf. Section 4.2)
- the integral types `uint8`, `uint16`, `uint32`, and `uint64` which enhance the efficiency of compilation, in particular with regard to micro controllers.

PrimitiveType:
    NumericType[21]
    `boolean`
    `bool` *as shorthand*
    `time`
NumericType:
    IntegralType[21]
    FloatingPointType[21]
IntegralType: one of
    `char`
    `byte  unit8`
    `short uint16`
    `int   uint32`
    `long  uint64`
FloatingPointType: one of
    `float  double`

The values of numerical type depend on the target machine that can range up to small target machines. Hence the value scopes specified below are "typical" for a variety of machines.

The value of a variable of primitive type can be changed only by assignment.

Integral types and values. The values of integral types have the following format and ranges:

- type `byte`: signed 8 bit
  (*synonym*: `int8`)
- type `char`: unsigned 8 bit
  (*synonyms*: `uint8`, `unsigned int8`, `unsigned byte`)
- type `short`: signed 16 bit
  (*synonym*: `int16`)
- type `uint16`: unsigned 16 bit
  (*synonyms*: `uint16`, `unsigned int16`, `unsigned int`)
- type `int`: signed 32 bit
  (*synonym*: `int32`)
- type `uint32`: unsigned 32
  (*synonyms*: `uint32`, `unsigned int32`, `unsigned int`)
- type `long`: signed 64 bit
  (*synonym*: `int64`)
- type `uint64`: unsigned 64 bit
  (*synonyms*: `uint64`, `unsigned int64`, `unsigned long`)

An integer literal is "polymorphic" in that it may be of any type such that its value is a value of that type, e.g. `1` is of any of the integral types, `-1` is of any of the signed types. Each literal has a most specific type that is the type that comprises its value, and that is the smallest such type. Most specific types are, e.g.

- `1` is of type `char`,
- `-1` is of type `byte`,
- `256` is of type `uint16`,

- `-128` is of type `short`, etc.

The type of an integral literal will resolved by the context it is used in (cf. Section 5).
The operations related to integral types are:

- relational operators `==` and `!=` (*equal* and *not equal*).
- comparison operators `<`, `<=`, `>`, and `>=` (*less than*, *less than or equal*, *greater than*, and *greater than or equal*).
- unary numerical operators `+` and `-` (*plus* and *minus*)
- binary numerical operators `+`, `-`, `*`, and `/` (*plus*, *minus*, *multiplication*, and *division*).
- unary numerical operators `++` and `--` (*increment* and *decrement* both prefix and postfix)
- unary logical operator $\sim$ (*complement*)
- binary logical operators `&`, `|`, `&&`, and `||` (*and*, *or*, *conditional and*, and *conditional or*)
- shift operators `<<`, `>>`, and `>>>` (*left shift*, *right shift*, and *arithmetic right shift*)

Floating-points type and values. The values of inegral types have the following format and ranges:

- type `float`: 32 bit floating point
- type `double`: 64 bit floating point

The operations related to floating-point types are:

- relational operators `==` and `!=` (*equal* and *not equal*).
- comparison operators `<`, `<=`, `>`, and `>=` (*less than*, *less than or equal*, *greater than*, and *greater than or equal*).
- unary numerical operators `+` and `-` (*plus* and *minus*)
- binary numerical operators `+`, `-`, `*`, and `/` (*plus*, *minus*, *multiplication*, and *division*).
- unary numerical operators `++` and `--` (*increment* and *decrement* both prefix and postfix)

The Boolean type and values. The Boolean type has the usual two truth values:

- type `boolean`: values `true`, `false`

The operations related to type `boolean` are:

- the relational operators `==` and `!=` (*equality* and *inequality*),
- the prefix operator `!` (*not*),
- the logical operators `&`, `|`, `&&`, `||`, `^` (*and*, *or*, *conditional and*, *conditional or*, and *exclusive or*).

The time type and values. The values of type `time` are time measurements:

- type `time`: target specific length and resolution, e.g. in Linux 64 bit, unit is micro second.

The operations related to type `time` are:

- the relational operators `==` (*equality*) and `!=` (*inequality*),
- comparison operators `<`, `<=`, `>`, and `>=` (*less than*, *less than or equal*, *greater than*, and *greater than or equal*).
- unary numerical operators `+` and `-` (*minus* being symmetric because computation is on time intervals, e.g. `2sec - 3sec` equals `1sec`.)

4.3 Reference Types

Reference types are class or interface types, and array types.

ReferenceType:
ClassOrInterfaceType[25]
ArrayType[25]

Class and interface types may be parameterized by a list of types, the format being

$$\texttt{TypeName<Type}_1\texttt{,...,Type}_n\texttt{>}.$$

ClassOrInterfaceType:
ClassType[25]
InterfaceType[25]
ClassType:
TypeName[36] *TypeParameterDeclaration*[25]$_{opt}$
InterfaceType:
TypeName[36] *TypeParameterDeclaration*[25]$_{opt}$
TypeParameterDeclaration:
`<` *TypeParameters*[25] `>`
TypeParameters:
TypeParameter[25]
TypeParameter[25] `,` *TypeParameters*[25]
TypeParameter:
ClassOrInterfaceType[25]
ClassOrInterfaceType[25] `implements` *TypeName*[36]

Arrays are one-dimensional or two-dimensional with columns and rows of the same length. Vector types are subtypes of one-dimensional array types, and matrix types are subtypes of two-dimensional array types. The difference is that vectors and matrices have a specified size (e.g. `int[5]`). Vector and matrix types support specific vector and matrix operations such as, for instance, scalar products.

ArrayType:
OneDimensionalArrayType[25]
VectorType[25]
TwoDimensionalArrayType[25]
MatrixType[25]
OneDimensionalArrayType:
PrimitiveType[21] `[ ]`
ClassOrInterfaceType[25] `[ ]`
TwoDimensionalArrayType:
PrimitiveType[21] `[,]`
ClassOrInterfaceType[25] `[,]` *VectorType:*
PrimitiveType[21] `[`*ConstantExpression*[122]`]`
ClassOrInterfaceType[25] `[`*ConstantExpression*[122]`]`
MatrixType:
PrimitiveType[21] `[`*ConstantExpression*[122]`,`*ConstantExpression*[122]`]`
ClassOrInterfaceType[25] `[`*ConstantExpression*[122]`,`*ConstantExpression*[122]`]`

Values of reference type. The values of reference types are pointers to an object. An object may be a *class instance*, an *array*, or the special *null* reference.

Objects are created by a class instance creation expression, an array creation expression, by invoking the method `newInstance` of class `Class`, or – in case of strings, and arrays – may be implicitly created by operators. The operations related to reference types are:

- the relational operators `==` and `!=` (*reference equality* and *reference inequality*)
- *field access* using a qualified name or a field access expression
- *method invocation*
- the *cast* operator
- the string concatenation operator `+`
- the operators related to signals (cf. Section 4.4)
- the operators related to arrays, vectors and matrices (cf. Section 11)

The class *Object*. The class `Object` is superclass of all other classes. A variable of type `Object` can reference any object.
The *methods* of class `Object` are:

- The method `getClass` returns the class object related to an object.
- The method `equals` returns `true` if objects are equal, and `false` otherwise.
- The void method `instant` is a compiler defined method that triggers an execution step. At present, calls of `instant` are restricted to methods `main` (of "configuration classes").

The class *String*. Instances are strings of characters. String literals reference instances of class string.

The string concatenation operator `+` implicitly creates a new instance of class `String`.

4.4 Signal Types

Signal types in many ways behave like reference types, but reference is static, being determined at compile time. We distinguish pure signal types, valued signal types, and flow types.

SignalType:
    PureSignalType[27]
    ValuedSignalType[27]
    FlowType[29]

Pure and valued signal types. There are three kinds of signals: signals of kind `Sensor`, `Signal`, or `DelayedSignal` . In the first case we speak of sensors, in the latter two of signals and delayed signals. The gross difference is that sensors are read-only within a program, while the (delayed) signals may be emitted or be constrained by a flow equation.

PureSignalType:
    SignalKind[27]
ValuedSignalType:
    SignalKind[27] *Clock*[27]$_{opt}$ *SignalTypeParameter*[28]
SignalKind: one of
    `Sensor` `Signal` `DelayedSignal` `Delayed`
Clock:
    { *Expression*[122] }

`Delayed` is a shorthand for `DelayedSignal`. The clock is optional for notational convenience. A signal type without clock is equivalent to one with the clock `true`, hence, e.g., the types `Signal{true}<`*T*`>` and `Signal<`*T*`>` are equivalent.

Pure signals may be present or absent only, while valued signals additionally have a value.

Operations related to pure and valued signal types are

- the *presence* operator `?`,
- the *(a)wait for signal* operator `@`, and
- the *value* operator `$`.

The latter is only defined for valued signals. The present operator checks whether a signal is *present* at an instant, or *absent*. The (a)wait for signal

operator yields the amount of time that has passed since the signal has not been present. The value operator accesses the value of a signal after it has been present for last time.

Signals can be *updated* to be present at an instant (with a new value if valued), but only if its clock evaluates to true. Signals of kind `Signal` or `DelayedSignal` are updated using the `emit` statement (*EmitStatement*[86]) or the constraint by a flow equation (*FlowEquation*[92]).

Sensors are updated to be present by callbacks of an input interface (cf. Section 10.6.

The distinction between signals of kind `Signal` and `DelayedSignal` is thus:

- A signal of kind `Signal` is updated to be present at the *same* instant, with a new value if valued.
- A signal of kind `DelayedSignal` is updated to be present at the *next* instant, with a new value if valued.

Values of signal types are references to signals (cf. Section 8). A signal is created by a signal declaration (*SignalDeclaration*[54]).

Signal type parameters. The type parameter of a sensor or signal type may either be a primitive type or a restricted class type.

SignalTypeParameter:
`<` *PrimitiveType*[21] `>`
`<` *ClassType*[25] `>`

The following restrictions apply for class type parameters of sensor or signal types:

- all its fields are of *PrimitiveType*[21]
- the class must be declared as `final`.
- constructors have no parameters.

Flow types and values. Flow types are only used for typing expressions. Flow types are of the form

$$T\{C\}$$

with T being a primitive or a class type, and C being a Boolean flow expression, we refer to as a *clock*. A *flow expression* is an expression of flow type.

Formally flow types are specified by

FlowType:
    StandardType[29] { *Expression*[122] }
StandardType:
    PrimitiveType[21]
    ClassType[25]

The values of flow type are references to flows (cf. Section 8). Flows are only implicitly generated by flow expressions.

Flow types have private constructors only, hence no variables of flow type can be defined.

Flow contexts. Expressions of flow type may only occur in a flow context of the form

{| ...|}

or in a clock expression. Vice versa, all literals and variables within a flow context or clock expression are automatically converted to a corresponding flow type that is on base clock. For instance,

- the literal `3` is of type `int` outside of a flow context, and of type `int{}` within a flow context.
- the literal ``` ``this is a string'' ``` is of type `String` outside of a flow context, and of type `String{}` within a flow context.
- A variable of some primitive or class type T is – of course – of type T outside of a flow context, but of type $T\{\}$ within a flow context.

Flow context are used for the sustain statement (cf. Section 14.14), the during clause of automata states (cf. Section 14.16), and the definition of nodes (cf. Section 9.9).

Operators on flows. All operators and methods on primitive types or class types lift to operators and methods of the corresponding flow type on base clock within a flow context, for instance:

- Given the flow expressions `true` and `false` of type `boolean{`C`}`, flow expression `true && false` is of type `boolean{`C`}`.
- Given a data method with signature `int add(int x, int y)` of some fictitious class `Int`, and an object x of type `Int`, then the expression `x.add(3,5)` is of type `int{}` within a flow context. Note that `x` is of type `Int{}`.

Particular operators related to flow types are

- The operator `pre` that accesses the value a flow had the previous time it was present.
- The operator `->` (*arrow*) that distinguishes between the first instant at which a flow context has started to be evaluated, and later instants.
- The down-sampling operator `when`, and
- the *up-sampling operator* `current`.

Inheritance. Flow types are super types to valued sensor types. Sensor types are super types to signal and delayed signal types, in particular

- `Signal` is a subtype of `Sensor`.
- `DelayedSignal` is a subtype of `Sensor`.
- `Sensor`$\{C\}$`<`T`>` is a subtype of $T\{C\}$.
- `Signal`$\{C\}$`<`T`>` is a subtype of `Sensor`$\{C\}$`<`T`>`.
- `DelayedSignal`$\{C\}$`<`T`>` is a subtype of `Sensor`$\{C\}$`<`T`>`.

About clocks. A signal can only be updated to be present at an instant if its clock expression evaluates to true.

Within a flow context (cf. Section 14.18), a signal can only updated by a flow equation. The constraint applies only if the clock of the signal evaluates to true, *and* if the flow context is active.

Outside of any flow context, a signal be updated only by using the `emit` statement (cf. *EmitStatement*[86]). If a signal is updated using an emit statement it must have the clock `true` (i.e. have a type `Signal<`T`>` resp. `Signal{true}<`T`>`).

4.5 Variables

Kinds. A variable stores a value of its associated type. The value is changed by assignment for instance. A variable of primitive types holds a value of the respective type, a variable of reference type a reference, either a `null` reference, or a reference to an object of appropriate type.

Kinds of variables are:

- *Class variables*: `static` field in a class declaration initialised to a default value, or a field, implicitly public static, and final, in an interface declaration.
- *Instance variables*: field in a class definition that is not `static`. Initialised to a default value.
- *Array components*: unnamed variable for a newly created array. Initialised to a default value.
- *Type parameters*: created for each type call, initialised by the type parameter value.
- *Method parameters*: created for each method call, initialised by the parameter value.
- *Constructor parameters*: created for each method call, initialised by the parameter value.
- *Local variables*: declared and initialised to a default when entering the block where the variable is declared.
- *Sensor or signal variables*: are (blank) final. They are initialised either by a sensor or signal creation expression or the assignment of a constructor parameter of a reactive method.

There are no flow variables.

A variable can be declared `final`. A final variable may only be assigned to once, either in its declaration or in the constructor. A blank final is a final variable whose declaration lacks an initialiser.

A final variable always contains the same value once it has been assigned.

Defaults Values. Every variable in a program must have a value before its value is used. Each variable is initialised with a default value when it is created. Default values are.

- – All integral types `0`.
  – All float types `0.0`.
  – `boolean`: `false`.
  – *reference types*: `null`.
  – *sensor and signal types*: no default value. Sensors and signals must be initialised explicitly.

- *method parameter*: arguments when invoked.
- *constructor parameter*: arguments when invoked.
- *exception-handler parameter*: arguments when invoked.
- *local variable*: by initialisation or assignment before it is used.

5 Conversions and Promotion

There are no implicit type conversions. The programmer has to indicate a type conversion explicitly using a cast.

The only promotion rules apply to integral literals. A priori, the type of an integral literal is its most specific type. In context of a binary arithmetic operator the literal may be promoted to a type of the other operand if this is a possible integral type for that literal. For instance, if `x` is of type `int` then `1` will be promoted to be of type `int` in `x + 1`.

The rule for promotion is that both operands of a binary arithmetic operator may be promoted to the smallest type that is a possible type for each of them. Promotion will cause a typing error if a unique such type cannot be determined, e.g.:

- Let `x` be of type `int`. Then `x + 1` is of type `int`.
- A type cannot be determined for `1 + 2` but
- using a cast `(int)1 + 2` constrains the type of the expression to `int`.

6 Names

6.1 Declarations

Names refer to entities in a program. These may be class or interface types, fields, methods, parameters, or local variables.

A declaration binds some program entities to an identifier that may be used within to refer to the respective entity. Declared entities are:

- a class, declared in a class type declaration (cf. Section 9),
- an interface, declared in a interface type declaration (cf. Section 10),
- a field, declared in a class type declaration (cf. Section 9), or a constant field, declared in an interface type declaration (cf. Section 10)
- a method, declared in a class or interface type declaration (cf. Sections 9,10),
- a parameter of a method or constructor of a class (cf. Section 9), or of an abstract method of an interface (cf. Section 10),
- a local variable, declared in a block (cf. Section 13.3), in a for statement (cf. Section 13.11), in a reactive block (cf. Section 14.3), or in a flow context (cf. Section 14.14, 14.16, 9.9).

Constructors use the name of the class they are declared in.

Names may be *simple* being an identifier or *qualified* consisting of a qualified name followed by an "." and an identifier.

synERJY restricts naming according to the conventions of JAVA™ proposed in [5]:

- The name of a class or Interface starts with a capital letter followed by a sequence of letters or digits with at least one non-capital letter, or a single capital letter. Otherwise,
- for all other names, either all letters are capital, or it starts with a non-capital letter.

Identifiers may occur in other contexts than that of names as are: declarations, class instance creation (cf. Section 16.2, field access (cf. Section 16.8, method invocations (cf. Section 16.5, and labels (cf. Section 3.7.

6.2 Scope of a Declaration

The scope of a declaration is that part of a program within which the declared entity can be referred to using a simple name. If a point in a program is included in the scope of a declaration we say that the declaration *is in scope* at that point.

Scoping rules are:

- The scope of a class or interface type is all the class or interface declarations of an application.
- The scope of the declaration of a a member – that is a field, or method – declared in or inherited by a class is the body of the class.
- The scope of a parameter of the method or constructor is the body of the respective method or constructor.
- The scope of a local variable declaration is the rest of the block that immediately encloses it. The scope of a local variable declared in the *ForInit*[80] part of a for statement includes: (i) its own initializer, (ii) any declarator to the right of the *ForInit*[80] part, (iii) The expression and the *ForUpdate*[80] part, and the statement.

There is *no shadowing* in *synERJY*: within the scope of a declaration no declaration with the same name is allowed.

6.3 Members

Reference types have members that are

- Members of a class type may be fields and methods. They may either be declared in the class, inherited from its direct superclass (except for `Object`), or inherited from its direct superinterfaces.
- Members of an interface type may be static fields and abstract methods. They may either be declared in the class, or inherited from its direct superinterfaces.
- Members of an one-dimensional array are the parameterless method `length`, and the members inherited from `Object`.
- Members of an two-dimensional array are the parameterless methods `rows` and `columns`, and the members inherited from `Object`.

6.4 Meaning

The meaning of a name depends on the context in which it is used. Names are generated by the grammar rules

TypeName:
Identifier[16]
ExpressionName:
Identifier[16]
AmbiguousName[36] . *Identifier*[16]
FieldName:
Identifier[16]
AmbiguousName[36] . *Identifier*[16]
MethodName:
Identifier[16]
AmbiguousName[36] . *Identifier*[16] *AmbiguousName:*
Identifier[16]
AmbiguousName[36] . *Identifier*[16]

The meaning of a name can be determined as follows:

- A *TypeName*[36] may occur
  - in an extend clause of a class declaration (*Super*[50]),
  - as the type of a formal parameter (*FormalParameter*[55]),
  - as the type of a local variable (*LocalVariableDeclarationStatement*[75]),
  - as the class type to be instantiated in a class instance creation expression (*ClassInstanceCreationExpression*[99]),
  - as a direct superclass or super-interface of an anonymous class (*ClassInstanceCreationExpression*[99]),
  - as the element type of a array type (*ArrayType*[25],
  - as the type used in an cast operator (*CastExpression*[110]), or
  - as the value type of signal (*SignalType*[27]).
- an *ExpressionName*[36] may occur
  - as the array reference expression in a array access expression (*ArrayAccess*[108]) or vector or matrix access expression (*VectorOrMatrixAccess*[109]),
  - as a postfix expression (*PrefixPostfixExpression*[112]), or

  - as the left-hand operand of an assignment operator (*Assignment*[76]).

- a *MethodName*[36] may occur

  - before the opening bracket in a method invocation expression (*MethodInvocation*[101],*ReactiveMethodInvocation*[105], *NodeInvocation*[106]).

- An *AmbiguousName*[36] may occur to the left of a "." in a qualified *ExpressionName*[36], *MethodName*[36], or *AmbiguousName*[36]

Ambiguous names can be reclassified.

- If an ambiguous name is a simple name being a simple identifier then it is reclassified as an *ExpressionName*[36] if it is in the scope of the declaration of a field, a local variable, or a parameter.

- If an ambiguous name is a qualified name then it is reclassified as an *ExpressionName*[36] if the name on the left of the "." is a type name, and if it is a field, or method of the class denoted by the type

Since all classes are declared top-level, type names are simple names. Hence name classified as type name denotes the respective class type.

The meaning of a name classified as an expression name is determined as follows:

- If the expression name is a simple name, i.e. an identifier. then

  - if the identifier appears within a scope of the declaration of a local variable, or a parameter, then the name denotes the respective entity. The type of the identifier is that of the declared entity.
  - if the identifier appears within a class declaration and therein within the scope of a field declaration, then
    - if the field is declared to be final, the name refers to the value of this field, or
    - otherwise, the identifier denotes the variable declared by the field declaration.

    The type of the expression name is that of the field.

- If the expression name is a qualified name of the form $X.y$ with y being an identifier.

  - If X is a type name that names a class type, then there must be exactly one member field of the respective class type with name y.

* If that field is final, then $X.y$ denotes the value of that class variable.
* Otherwise, $X.y$ denotes the class variable.

In both cases, the type of the expression name is that of the class variable.

– If X is a type name that names an interface type, then there must be exactly one member field of the respective class type with name y.

The name $X.y$ then denotes the value of that field, and the type of the expression name is that of the class variable.

– If X is an expression name its type T must be a reference type, there must be exactly one member field of T with name y. Then if the field is either

* a field of an interface type,
* a final field of a class type, or
* the method `length` of an one-dimensional type,
* the methods `columns` and `rows` of a two-dimensional type,

the expression name $X.y$ denotes the value of the respective fields, and has the same type.

Otherwise $X.y$ denotes a variable that may either be a class or instance variable that determines the type of $X.y$ as well.

6.5 Access Control

Accessibility is a static property to be determined at compile time only depends on types and declaration modifiers. Reference types are accessed by qualified names (cf. Section 6), or by field access (*FieldAccess*[107]) or method invocation expressions (*MethodInvocation*[101]).

A member (class, interface, field, or method) of a reference (class, interface, or array) type or a constructor of a class type is accessible only if the member or constructor permits access:

- If the member or constructor is declared `public` then access is permitted.
- If the member or constructor is declared `protected` then access is permitted only according to the following conditions:

  - Let C be the class in which a protected member m is declared. Access is permitted only within the body of a subclass C' of C. Further, if $Id : main.tex, v1.132006/12/2015 : 29 : 51apExp$ denotes an instance field or instance method, then, if the access is by a field access expression (*FieldAccess*[107]) $E.Id$, where E is a primary expression (*Primary*[98]), or by a method invocation expression (*MethodInvocation*[101]) $E.Id(\ldots)$, where E is a Primary expression, then the access is permitted if and only if the type of E is C' or a subclass of C'.

- If the member or constructor is declared private, then access is permitted if and only if it occurs within the body of the class that encloses the declaration of the member.

- Let C be the class in which a protected constructor is declared and let S be the innermost class in whose declaration the use of the protected constructor occurs. Then:

  - If the access is by a superclass constructor invocation `super`$(\ldots)$ or by a qualified superclass constructor invocation of the form E.`super`$(\ldots)$, where E is a primary expression, then the access is permitted.
  - If the access is by an anonymous class instance creation expression of the form `new` $C(\ldots)\ldots$ or by a qualified class instance creation expression of the form E.`new` $C(\ldots)\ldots$, where E is a primary expression, then the access is permitted.
  - Otherwise, if the access is by a simple class instance creation expression of the form `new` $C(\ldots)$ or by a qualified class instance creation expression of the form E.`new` $C(\ldots)$, where E is a primary expression, then the access is not permitted.

One should note that

- Class, interface types, array types are always declared public, hence may be accessed by any code.

- All members of a class are implicitly `private`.

- All members of an interface, array type are implicitly `public`.

7 Compilation Unit

A compilation unit is a list of type declarations. Packages are not supported.

CompilationUnit:
    TypeDeclarations[40]$_{opt}$
TypeDeclarations:
    TypeDeclaration[40]
    TypeDeclarations[40] *TypeDeclaration*[40]
TypeDeclaration:
    ClassDeclaration[49]
    InterfaceDeclaration[63]

8 Flows, Sensors, and Signals

Values of sensors, signals, and flow expressions[6] are references to flows. A formal definition of flows is quite involved, hence only an informal presentation is offered to illustrate the basic ideas.

8.1 Traces and Flows

Traces. Traces are a prerequisite for defining flows. Let a *trace* be a sequence of data (of the same type). At an instant, a trace may be *accessible* or *inaccessible*. If it is accessible it has a value. A trace may, for instance, be inaccessible since it is the trace of a local signal enclosed by a block that is not active (i.e. not being evaluated) at an instant.

For visualisation, we use a diagram of the form

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	...
d		d_0	d_1			d_2		d_3	...

The trace d is accessible at an instant if there is an entry d_n. An empty slot marks inaccessibility. The index i indicates the instants.

Standard operators on primitive or reference types or methods of reference types lift to the corresponding traces by defining the operations element-wise at every instant, e.g.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
d		d_0	d_1			d_2		...
d'		d'_0	d'_1			d'_2		...
$d + d'$		$d_0 + d'_0$	$d_1 + d'_1$			$d_2 + d'_2$		...

The definition requires that d, d', and $d + d'$ are accessible at the same instant.

Literals and variables of primitive or reference type determine traces. For instance, the literal `3` determines a trace that has the value 3 at every instant

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
d	3	3	3	3	3	3	3	...

[6] i.e. expressions of flow type.

Down-sampling and upsampling. The operator *when* *down-samples* the trace d at a *frequency* specified by the Boolean "clock" flow c as in, e.g.,

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		d_0	d_1			d_2		d_3	d_4		...
c		t	f			f		t	f		...
d `when` c		d_0						d_3			...

The flow d *when* b is accessible at an instant if and only if the flow b has the value *true* at that instant.

We expect *up-sampling* as a counter part to down-sampling. The operator *current* latches the value of a trace till the next sampling.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		d_0	d_1			d_2		d_3	d_4		...
c		t	f			f		t	f		...
$e = d$ `when` c		d_0						d_3			...
`current(`e`)`		d_0	d_0			d_0		d_3	d_3		...

Up-sampling should generate a trace that is "as fast" as the trace originally down-sampled meaning that `current(`d'`)` is accessible if and only if d (resp. c) is accessible.

The up-sampled trace `current(`e`)` must be accessible if and only if d (resp. c) is accessible. Hence the definition of `current(`e`)` depends on the trace e and the frequency at which d is accessible. If we indicate accessibility of d by adding a *sampling frequency* in the diagram

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
e		d_0						d_3			...
ν		.	.			.		.	.		...

this clearly provides sufficient information for constructing the up-sampled trace

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
`current(`e`)`		d_0	d_0			d_0		d_3	d_3		...

Iterating up-sampling. We generalise the idea in that we specify several sampling frequencies as in

i	0	1	2	3	4	5	6	7	8	9	...
e		d_0						d_3			...
ν_2		.	.			.		.	.		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

Note that the ν_0 is "faster" than ν_1, etc.

Using these informations we can construct the up-samplings

i	0	1	2	3	4	5	6	7	8	9	...
e' = current(e)		d_0	d_0			d_0		d_3	d_3		...
e'' = current(e')	δ	d_0	d_0		d_0	d_0	d_0	d_3	d_3		...
e''' = current(e'')	δ	d_0	d_0	d_0	d_0	d_0	d_0	d_3	d_3	d_3	...

The trace ν_0 cannot be up-sampled any more since it is on base clock.

Flows. Obviously, traces are not sufficient to support up-sampling. A more sophisticated structure is needed that consists of the data trace plus all the sampling frequencies as illustrated by the diagram

i	0	1	2	3	4	5	6	7	8	9	...
d		d_0						d_1			...
ν_2		.	.			.		.	.		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

We refer to such a structure as a *flow*.

Operations on flows. The operations on traces naturally lift to flows. To be well defined, it is required that binary operations can be applied only if the two argument flows have the same sampling frequencies, e.g.

i	0	1	2	3	4	5	6	7	8	9	...
d		d_0						d_1			...
d'		d'_0						d'_1			...
d'		$d_0 + d'_0$						$d_1 + d'_1$			...
ν_2		.	.			.		.	.		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

Here the frequencies are the same for all the three flows hence we use only one diagram for convenience. For another example let

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		d_0	d_1			d_2		d_3	d_4		...
c		t	f			f		t	f		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

then down-sampling `e = d when c` changes the hierarchy of frequencies

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
e		d_0						d_3			...
ν_2		.	.			.		.	.		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

8.2 Flow Expressions

Flows for semantics. Expressions with a type of the form $T\{C\}$ are referred to as *flow expressions*. Values of flow expressions are references to flows that satisfy certain conditions: if

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		d_0	d_1			d_2		d_3	d_4		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

is a flow referenced by a flow expression of type $T\{C\}$ then

- the data d_0,d_1,d_2,d_3,d_4, ... must be of type T, and
- the clock (flow) expression must reference a Boolean flow of the form

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
c	f	t	t		f	t		t	t		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

The data trace d is accessible if and only if the data trace of c has value *true*. The sample frequencies of c (here only ν_0) are extended by the sampling frequency defining the accessibility of c (here ν_1).

Operations. Operations on flow expressions implicitly create new flows. E.g given integer expressions E_1 and E_2 with values being references to flows d_1 and d_2, the expression $E_1 + E_2$ generates a new flow $d_1 + d_2$ as defined above that is referenced by $E_1 + E_2$.

8.3 Sensors

The type `Sensor{`C`}<`T`>` is a subtype of $T\{C\}$. Thus values of sensors of type `Sensor{`C`}<`T`>` are references to flows satisfying the conditions stated above. Note that `Sensor<`T`>` is a shorthand for `Sensor{true}<`T`>`.

Pure sensor of type `Sensor{`C`}` should be considered as having the null type as data type T. Since the null type has one value values of such sensors are references to flows of the form

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		*null*	*null*			*null*		*null*	*null*		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

The data trace only states that d is accessible or inaccessible. Hence we rather use

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		.	.			.		.	.		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

for presentation.

Sensors additionally provide three methods to access flows:

- The *presence* operator $?s$ yields the value true at an instant if the sensor has been updated.
- The *value* operator $\$s$ yields at instant t the value of the sensor when it was last updated. This operator is defined only for valued sensors, i.e. sensors of type `Sensor{`C`}<`T`>`.
- The *(a)wait for signal* operator $@s$ yields at instant t the amount of (real) time that has passed since when the sensor s was not present.

To get an understanding of the latter operator we augment the data trace with a fictitious time trace τ. The τ_is are meant to correspond to the "real time" measured at the beginning of an instant.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	...
$@d$	0	$\tau_1 - \tau_0$	0	0	$\tau_4 - \tau_3$	$\tau_5 - \tau_3$	0	0	$\tau_8 - \tau_7$	...
d			d_0				d_1			...
ν_1	.	.	.		.	.	.	.	.	...
ν_0	.	.	.	.	.	.	.	.	.	...
τ	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	...

Note that

- the idea is consistent with the synchrony paradigm since the time difference is determined at the beginning of an instant very much like an input sensor, and that
- the granularity of "real time" is determined by the time differences between the instants.
- The sigal d is present during all of an instant. Hence @d equals 0 in the subsequent instant; no time has been spent for waiting with the signal d being not present.

There are a predefined input sensor `dt` of type `Sensor<time>`. The signal `dt` yields the amount of time (in terms of system time) that has passed between two instants (i.e. $\tau_{i+1} - \tau_i$ in terms of the diagram above). Note that `dt` is not defined in the very first instant. In case it is used an exception will be thrown.

8.4 Signals

Signal types are subtypes of sensor types. The difference between sensors and signals is that signals, but not sensors, can be updated by an `emit` statement (*EmitStatement*[86]) or by a flow constraint statement (*FlowEquation*[92]). Only input sensors are updated by callback of an input interface (cf. Section 10.6). The following conditions apply

- A sensor or signal s can be updated at instant t only if it accessible, i.e. if its clock is true.
- A sensor or signal can change value only if it is updated.
- If a signal s is updated using the `emit` statement, it must have the clock `true`.
- If a sensor is updated using an input interface (cf. Section 10.6) then it must have clock `true`.

To indicate presence in a flow diagram bold face will be used as in

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		$\mathbf{d_0}$	d_0			d_0		$\mathbf{d_1}$	d_1		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

In case of pure signals an asterisk is used

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
d		$*$	.			.		$*$	.		...
ν_1	.	.	.		.	.	.	.	.		...
ν_0	.	.	.	.	.	.	.	.	.	.	...

9 Classes

Class declarations define new reference types and their implementation. *Reactive classes* exhibit reactive behaviour. A class is reactive if

- its constructor has a tail of the form

  `active { ... }`,

  Such an constructor is called an *reactive constructor*.

- it comprises a sensor or signal declaration, or

- a reactive method (cf. Section 9.9).

- the modifier `reactive` is used.

Reactive classes are only allowed to have one reactive constructor.

The section discusses the common semantics of top level and anonymous classes.*Nested classes are not supported.*

Classes may declared to be abstract. A reactive class is abstract if some method is abstract and it does not have a reactive constructor. An abstract class cannot be instantiated, but can be extended by subclasses. If a class is declared final it cannot have subclasses. Classes are by default public.

Each class except the class `Object` is a subclass of a single existing class, and may implement interfaces (cf. Section 10.

The body of a class declares fields (9.7) and methods (9.9), instance and static initializers (9.9), and constructors (9.12). The members of a class include both declared and inherited members. The scope of a member is the entire declaration of the class in which the member is declared.

Instance initializers are blocks of executable code that may be used to help initialise an instance when it is created.

Constructors are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded.

9.1 Class Declarations

synERJY supports the specification of parameterised classes. A parameterised class may have one or more type parameters. The specification of a parameterised list may be taken as a prototypical example of a parameterised class.

```
class List<T> {

   public List() {
       tail = null;
       length = 0;
   };

   private List(T elem, List<T> list) {
       head = elem;
       tail = list;
       length++;
   };

   List<T> tail;
   T head;
   int length;

   public T       head() { return head; };
   public List<T> tail() { return tail; };
   public void append(T elem) {
       tail = new List<T>(head,tail);
      head = elem;
   };
}
```

(cf. `ref_param-types.se`) The class name `List` has a type parameter `T` indicated by the square brackets. The type parameter may be used like any other type within the class body. For instance, there is a field `head` the type of which is specified by the type parameter. Similarly, the field `tail` is of type `List<T>`. Type parameters may as well be used in methods or in constructors for specification of the types of parameters or the result type.

The general scheme of class declarations is as follows:

ClassDeclaration:
    ClassHeader[49] *ClassBody*[51]
ClassHeader:
    ClassModifiers[50]$_{opt}$ `class` *ClassDeclarator*[49] *Super*[50]$_{opt}$ *Interfaces*[51]$_{opt}$
ClassDeclarator:
    Identifier[16] *FormalTypeParameterDeclaration*[50]$_{opt}$

The identifier in a class declaration specifies the name of the class.

9.2 Class Modifiers

All classes are public by default. They may redundantly declared to be `public` *(there are no access modifier* `private` *and* `protected`*).* Classes may be `abstract` or `final` *(but not* `static` *or* `strictfp`).

ClassModifiers:
    `public`$_{opt}$ *ClassModifier*[50]
ClassModifier: one of
    `abstract` `final` `reactive`

9.3 Type Parameter Declarations

A class may be parameterised by one or more type parameters.

FormalTypeParameterDeclaration:
    `<` *FormalTypeParameters*[50] `>`
FormalTypeParameters:
    FormalTypeParameter[50]
    FormalTypeParameter[50] `,` *FormalTypeParameters*[50]
FormalTypeParameter:
    TypeName[36]
    TypeName[36] `implements` *TypeName*[36]

9.4 Superclasses and Subclasses

The optional extends clause specifies the direct *superclass* of the current class that is said to be a direct *subclass* of the respective superclass.

Super:
    `extends` *ClassOrInterfaceType*[25]

If a class does not have a specified direct superclass the class `Object` implicitly is a direct superclass. The class `Object` is the only class that does not have a direct superclass.

9.5 Superinterfaces

The implements clause lists the names of interfaces that are direct superinterfaces of the class being declared.

Interfaces:
`implements` *InterfaceTypes*[51]
InterfaceTypes:
InterfaceType[25]
InterfaceType[25] **,** *InterfaceTypes*[51]

The class is said to implement all its direct superinterfaces. Either the class is declared to be abstract, or all the methods of each direct superinterface must be implemented by a method declaration in that class or by some method declaration inherited from its direct superclass.

9.6 Class Body and Member Declaration

A class body contains the declarations of members of the class, i.e. fields, classes, interfaces, and methods, as well as instance initializers, static initializers, and declarations of constructors.

ClassBody:
{ *ClassBodyEntities*[51]$_{opt}$ }
ClassBodyEntities:
ClassBodyEntity[51]
ClassBodyEntity[51] *ClassBodyEntities*[51]
ClassBodyEntity:
ClassBodyDeclaration[51]
ClassBodySpecification[61]
ClassBodyDeclaration:
ClassMemberDeclaration[51]
ConstructorDeclaration[59]
ClassMemberDeclaration:
FieldDeclaration[52]
SignalDeclaration[54]
MethodDeclaration[55]
ReactiveMethodDeclaration[58]
NodeDeclaration[59]

The scope of a declaration of a member declared in or inherited by a class type is the entire body of that class type.

Members of a class type are

- either inherited from a direct superclass,

- or from a direct superinterface, or
- are declared in the body of the class.

Members of a class are inherited by a subclass only if they are declared `public` or `protected`, `private` members are not inherited.

Constructors are not members and therefore not inherited.

Class body specification will be explained in Section 9.13.

9.7 Field Declaration

The variables of a class type are introduced by *field declarations*.

FieldDeclaration:
FieldModifiers[52]$_{opt}$ *PrimitiveType*[21] *VariableDeclarators*[52] **;**
FieldModifiers[52]$_{opt}$ *ReferenceType*[24] *VariableDeclarators*[52] **;**
VariableDeclarators:
VariableDeclarator[52]
VariableDeclarator[52] **,** *VariableDeclarators*[52]
VariableDeclarator:
Identifier[16]
Identifier[16] **=** *VariableInitializer*[52]
VariableInitializer:
Expression[122]
ArrayInitializer[68]

A field declaration may declare several fields of the same type by using more than one field declarator. The identifier may be used to refer to the declared field. The scope of a field is the declaration of the class in which it is declared. The type determines the type of values a field may take.

All declared fields of a class must have different names. Methods and fields may have the same name since being used in different contexts.

Field modifiers. The list of field modifiers may comprise only one of the access modifiers. *The field modifier* `transient` *is not supported.* There are, however, a number of modifiers not defined in JAVA™.

FieldModifiers:
FieldModifier[53]
FieldModifier[53] *FieldModifiers*[52] *FieldTargetModifier*[53]$_{opt}$

FieldModifier: one of

`protected` `private`

`public` `public` { *ClassTypes*[53] }

`final` `static`

FieldTargetModifier:

`native` `native`{ *StringLiteral*[19] }

`import_from_C` `import_from_C`{ *StringLiteral*[19] }

`export_to_C` `export_to_C`{ *StringLiteral*[19] }

ClassTypes:

ClassType[25]

ClassType[25] `,` *ClassTypes*[53]

Fields are private by default in synERJY.

There is only one incarnation of a `static` field. A static field, also called a *class variable*, is incarnated when the class is initialised.

If a field is not declared static (a *non-static* field) it is called an *instance variable*. Whenever a new instance of a class is created, a new variable associated with that instance is created for every instance variable of that class or any of its superclasses.

A blank final class variable must be definitely assigned by a static initializer of the class in which it is declared.

A blank final instance variable must be definitely assigned at the end of every constructor of the class in which it is declared.

The name of a final and static field may be *imported* from or *exported* to the target language. If it is imported, it must be declared in the target language with an appropriate type. The field is not allowed to be initialised in its declaration. If it is exported, its name and type is exported to the target language. For convenience, a field may be renamed using the modifiers `import_from_C`{ *StringLiteral*[19] } or `export_to_C`{ *StringLiteral*[19]. A field with a *FieldTargetModifier*[53] is always static and final by definition, but the modifiers `static` and `final` may be used optionally. The keywords `native` and `import_from_C` are synonyms.

Initialisation of fields. A field declaration may include a variable initializer if no target modifier is used. The semantics is that of an assignment to a declared variable (cf. *Assignment*[76]).

The initializer of a class variable will be evaluated only once when the class is initialised, that of an instance variable each time an instance of the class is generated. If used in a local variable declaration statement (cf.

LocalVariableDeclarationStatement[75]) it will be executed each time the local variable statement is executed.

Hiding. No hiding of fields is allowed, i.e. a field f_1 declared within the scope of another field f_2 must always have a different name.

9.8 Signal Declarations

Sensor and signal declarations declare variables of sensor or signal type. Variables of sensor or signal type are always blank final and non-static. The format is restricted.

SignalDeclaration:
    SignalType[27] *SignalDeclarator*[54] `;`
SignalDeclarator:
    Identifier[16]
    Identifier[16] `=` `new` *SignalType*[27] `(` *Expression*[122]$_{opt}$ `)`

Clock expressions do not occur in the type constructors following the `new` operator (We omit the clocks here for convenience of notation.)

If the signal is of kind `Sensor`, its constructor must have an argument of interface types `Input` (cf. Section 10.6). We then refer to the signal as an *input sensor*.

For signals of kind `Signal` and `DelayedSignal` the argument is optional, but it must be of type `Output` (cf. Section 10.7). If an argument is given, we speak of an *output signal*.

The format implies that sensors and signals are assigned statically and unconditionally at compile time (see also Section 9.12) .

9.9 Method Declarations

A method declares executable code that can be invoked, passing a fixed number of values as arguments (cf. Section 16.5). We often speak of a "data method" in comparison to reactive methods that hold reactive code (cf. Section 9.10).

A *class method*, i.e. a method declared to be `static`, is invoked relative to the class type. An instance method is invoked with respect to some particular object that is an instance of the class type. A method whose declaration does not indicate how it is implemented must be declared abstract.

A method may be implemented by platform-dependent native code. Further, The implementation of a method may be exported to the target language.

MethodDeclaration:
    MethodHeader[55] *MethodBody*[57]
MethodHeader:
    MethodModifiers[56]$_{opt}$ *ResultType*[55] *MethodDeclarator*[55]
ResultType:
    Type[21]
    `void`
MethodDeclarator:
    Identifier[16] `(` *FormalParameters*[55]$_{opt}$ `)`

The result type specifies the type of value that the method returns or uses the keyword void to indicate that the method does not return a value.

A class can declare a method with the same name as the class or a field of the class. *Two methods with the same name and the same number of parameters are not allowed to be members of the same class.*[7]

A method must not have a method body if it specified to be `abstract` or `native`.

Formal parameters. Each formal parameter consists of a type, and an identifier that specifies the name of the parameter:

FormalParameters:
    FormalParameter[55]
    FormalParameters[55] `,` *FormalParameter*[55]
FormalParameter:
    Type[21] *Identifier*[16]

A method or constructor without parameters has only an empty pair of parentheses. All parameters must have different names.

When the method or constructor is invoked, parameter variables of the declared parameter type are created and initialized by the values of the actual argument expressions initialize newly created parameter variables, before execution of the body of the method or constructor.

[7] In contrast to JAVA™ where the methods may be distinguished by the parameter types as well.

Method signature. The signature of a method consists of the name, and the list of types of its formal parameters.

Method modifiers. The list of method modifiers may comprise only one of the access modifiers. *The method modifiers* `synchronised` *and* `strictfp` *are not supported.* There are, however, a number of modifiers not known in JAVA™.

MethodModifiers:
MethodModifier[56]
MethodModifiers[56] *MethodModifier*[56] *MethodTargetModifier*[56]$_{opt}$
MethodModifier: one of

`protected`	`private`	
`public`	`public` { *ClassTypes*[53] }	
`abstract`	`static`	`final`

MethodTargetModifier: one of

`import_from_C`	`import_from_C` { *StringLiteral*[19] }
`export_to_C`	`export_to_C` { *StringLiteral*[19] }
`interrupt` (*StringLiteral*[19])	

Default modifier of a method is `private`*.*

If a method is declared to be *abstract* it does not provide an implementation but only specifies its signature. An implementation must be provided by a subclass if it is not abstract. A compile-time error occurs if a method declaration that contains the keyword abstract also contains any one of the keywords `private`, `static`, `final`, `native`, `import_from_C`, `export_to_C`, or `interrupt`.

If a method is declared to be *static* it is called a *class method*. Class methods must be invoked without reference to a specific object.

If a method is declared to be `final` no overriding by subclasses can take place. A `private` method and all methods declared in a `final` class are `final` by default.

If a method is declared to be `native` its implementation is imported from the target language. Then the body of a method is omitted, indicating that the implementation is omitted. The name of a method (and its implementation) may be exported to the target language using the modifier `export_to_C`. A method may be renamed while imported or exported. Only data methods may be imported or exported, i.e. any of the target modifiers should not be used with the modifier `reactive`. The keywords `native` and `import_from_C` are synonyms.

A method with the target modifier `interrupt` is used to interact with the interrupt system of the target architecture (e.g. a micro processor). If an intrrupt, the name of which is given by the string literal, is raised by target system, then the method is executed. As usually, interrupts should be handled with care.

Method body. A method body is a block guarded by a pre- and a post-condition.

MethodBody:
PreCondition$^{57}{}_{opt}$ *Block*$^{74}{}_{opt}$ *PostCondition*$^{57}{}_{opt}$;

If a method is declared `void`, then its body must not contain any return statement (cf. *ReturnStatement*^{81}) that has an *Expression*^{122}.

If a method is declared to have a return type, then every return statement in its body must have an *Expression*^{122} the type of which matches to the return type of the method.

Pre- and postconditions are guards that may throw an exception (cf. Section 12.

PreCondition:
`pre assert` *Assertions*^{73}
PostCondition:
`post assert` *Assertions*^{73}

At first the precondition is evaluated, then the block defining the behaviour of a data method is executed. A postcondition is executed directly after the execution of a return statement.

Inheritance, and Overriding A class inherits all the non-private methods from its direct superclass and direct superinterfaces that are accessible (cs. Section 6.5) and not overridden by a declaration in the class. A class may inherit more than one method with the same signature, but then the methods must be abstract.

An instance method m_1 overrides another method m_2 with the same signature that is declared in a superclass or superinterface either if m_2 is non-private and accessible. A compile time error occurs if m_2 is a static method. Further, if m_2 is abstract, we say that m_1 implements m_2. Overriding is transitive, i.e. if a method m_1 overrides a method m_2, and if m_2 overrides m_2, then m_1 overrides m_3.

It is required that the types of parameters and the return type must be compatible, if a method overrides another. The access modifier of an overriding method must provide at least as much access as the overridden method.

An overridden method can be accessed only by using the keyword `super` in a method invocation expression but not with a qualified name or a cast to a superclass type.

Note that hiding is not allowed.

Overloading. If two methods being member of a class have the same name but a different number of parameters they are said to be *overloaded*.[8]

Initializers An instance initializer is executed when an instance of a class is created (cf. *ClassInstanceCreationExpression*[99]). A static initializer is executed when the class is initialized.

9.10 Reactive Method Declarations

A reactive method declaration declares a method with a reactive body. A reactive method may be called from within a reactive block (*ReactiveBlock*[82]). A reactive method is statically linked at compile time (cf. section 16.6).

ReactiveMethodDeclaration:
    ReactiveMethodHeader[58] *ReactiveBlock*[82]
ReactiveMethodHeader:
    ReactiveMethodModifiers[58]$_{opt}$ `void` *MethodDeclarator*[55]
ReactiveMethodModifiers:
    `private`$_{opt}$ `reactive`$_{opt}$

The modifier `reactive` declares a method to be reactive. Reactive methods are private by default. No other modifiers are allowed for a reactive method. Reactive methods must have result type `void`.

The modifier `reactive` is optional if the body can be determined to be a reactive body, i.e. if a reactive statement occurs in a body. If the compiler cannot infer that a method is reactive, it is assumed to be a data method by default. In that case it is mandatory to use the modifier `reactive` if a method is required to be reactive.

[8] In contrast, of JAVA™ where methods may be overloaded if the signatures are different.

9.11 Node Declarations

A node declaration declares a node with a body being a flow context. A node corresponds to a block in typical data data flow diagrams, for instance in SIMULINK (cf. section 8). A node may be only be called from within a flow context (*FlowContext*[92]). A node is statically linked at compile time (*nodeinvocation*[106]).

NodeDeclaration:
    NodeHeader[59] *FlowContext*[92]
NodeHeader:
    `private`$_{opt}$ `node` *MethodDeclarator*[55]

The clocks of a node are "relative" with regard to a node invocation in that the formal parameters inherit the clocks of the arguments (cf. Section 16.7. However, the clocks of a node are checked to be consistent relative to the clocks of the formal parameters. This presumes that *a node does not access any signal declared as a class member*. These have "absolute" clocks that cannot be compared to the relative clocks of a node.

The clock calculus is explained at length in the *synERJY* Language Introduction [3].

9.12 Constructor Declarations

A constructor is used to create an instance of a class. *synERJY requires that* at least one constructor declaration exists *in a class declaration.*

ConstructorDeclaration:
    ConstructorModifier[60]$_{opt}$ *ConstructorDeclarator*[59]
        ConstructorBody[60]
ConstructorDeclarator:
    TypeName[36] `(` *FormalParameters*[55]$_{opt}$ `)`

The type name must be the name of the class that contains the constructor declaration. Otherwise, the constructor declaration looks just like a method declaration without a result type. Constructors are invoked by the class instance creation expressions (cf. *ClassInstanceCreationExpression*[99]).

Formal parameters correspond to that of method declarations. The *signature* of a constructor is determined by the list of types of the formal parameters.

Constructors may be overloaded if they differ in the number of parameters.[9] The overloading is resolved at compile time by each class instance creation expression.

Constructor modifiers. Constructors may be public, protected, or private.

ConstructorModifier: one of
`public` `protected` `private`

Constructor body. The first statement of a constructor body may be an explicit invocation of another constructor of the same class or of the direct superclass.

The last statement – the *constructor tail* – may be an *active statement* that specifies the reactive behaviour of a class. Then the respective class is a reactive class.

Otherwise the body of a constructor is like the body of a data method (except that pre and postconditions cannot be used).

ConstructorBody:
`{` *ConstructorInvocation*$^{60}{}_{opt}$ *ConstructorBodyEnd*^{60} `}`
ConstructorInvocation:
`this` `(` *Arguments*$^{99}{}_{opt}$ `)`
`super` `(` *Arguments*$^{99}{}_{opt}$ `)`
ConstructorBodyEnd:
BlockStatements$^{74}{}_{opt}$ *ConstructorTail*^{60}
ConstructorTail:
SensorOrSignalInitializations$^{60}{}_{opt}$ *ActiveStatement*$^{60}{}_{opt}$
SensorOrSignalInitializations:
SensorOrSignalInitialization^{60}
SensorOrSignalInitialization^{60} *SensorOrSignalInitializations*^{60}
SensorOrSignalInitialization:
Identifier^{16} `=` *Identifier*^{16} `;`
ActiveStatement:
`active` *ReactiveBlock*^{82}

A `return` without an expression is allowed to occur in a constructor if the class is not reactive.

[9] In contrast to JAVA™ where the constructors must differ in terms of the signatures.

There are two cases of an explicit constructor invocation. A superclass constructor invocation begins with the keyword `super`, and an alternate constructor invocation begins with the keyword `this`.

A constructor is *reactive* if it comprises a sensor or signal initialization and/or an active statement. A reactive class is restricted to have at most one reactive constructor.

Default constructor Unlike to JAVA™, every class is required to have at least one constructor.

Sensor or signal initialisation. Sensor or signal initialization is restricted to assign a formal parameter to a sensor or signal variable only. Hence a sensor or signal variable that is not initialized when declared (cf. section 9.8) refers to a sensor or signal declared in some superclass. The reference is static, and part of what is called a *signal bus* in [3].

Active statement. The `active` statement embeds reactive code. It is the starting point of generation of reactive code. The reactive code of all reactive classes is executed concurrently communicating only via the sensor or signals of the signal bus, i.e. those sensor or signals that are actual parameters of a constructor call.

9.13 Class Specifications

A class body may comprise

- a specification of *reactive precedences* (cf. Section 15),
- specifications of an *invariant*.

ClassBodySpecification:
PrecedenceSpecification[94]
InvariantSpecification[61]
PropositionSpecification[124]
ConstraintSpecification[126]

Specification of invariants. An invariant is a means to throw an exception (cf. Section 12,

InvariantSpecification:
`invariant` { *Assertions*[73]$_{opt}$ }

The assertion is executed whenever a pre- or postcondition of the respective class is evaluated.

10 Interfaces

An interface declaration specifies a new reference type. Members are constants and abstract methods (*but not classes and interfaces*). An interface has no implementation, but its abstract methods may be implemented by otherwise unrelated classes. Interfaces are a means to share method names without sharing an implementation.

An interface can be declared to be direct extension of one or more other interfaces meaning that it inherits all the members of the interfaces it extends. No hiding is allowed. A class may directly implement one or more interfaces, meaning that any instance of the class implements all the abstract methods specified by the interface or interfaces. A class necessarily implements all the interfaces that its direct superclasses and direct superinterfaces do.

A variable of interface type may have a value that is a reference to any instance of a class which is declared to implement the specified interface.

10.1 Interface Declarations

The interface declaration specifies the name of an interface, its members, and eventually the interfaces it extends.

InterfaceDeclaration:
`interface` *Identifier*[16] *ExtendsInterfaces*[63]$_{opt}$ *InterfaceBody*[64]

There are no modifiers. Interfaces are always public and abstract in *synERJY*.

10.2 Superinterfaces

The declared interface extends each of the other named interfaces if an extend clause exists. Then it inherits the members of each of the other named interfaces that are direct superinterfaces of the interface being declared. Implements of the declared interface by a class implies that all the superinterfaces are implemented.

ExtendsInterfaces:
`extends` *InterfaceTypes*[51]

10.3 Interface Body and Member Declaration

The interface body specifies the members of the interface.

InterfaceBody:
InterfaceMemberDeclarations[64]$_{opt}$
InterfaceMemberDeclarations:
InterfaceMemberDeclaration[64]
InterfaceMemberDeclaration[64] *InterfaceMemberDeclarations*[64]
InterfaceMemberDeclaration:
ConstantDeclaration[64]
AbstractMethodDeclaration[64]

Members are those declared in the interface body or those inherited. The scope of a member of the declared interface is the entire body of the interface. Names of members are always public.

If there are several paths by which the same member declaration might be inherited from an interface, the member is considered to be inherited only once, and it may be referred to by its simple name.

10.4 Field (Constant) Declaration

Field declaration in an interface body are `public`, `static`, and `final` by definition, i.e. such a field denotes a constant.

ConstantDeclaration:
Type[21] *FieldName*[36] `=` *VariableInitializer*[52]

Every field must have an initialization expression. The variable initializer is evaluated and the assignment performed exactly once, when the interface is initialized

10.5 Abstract Method Declarations

A method in an interface body is always `public` and `abstract`, hence no method body is provide.

AbstractMethodDeclaration:
ResultType[55] *MethodDeclarator*[55]

Inheritance and overriding. An interface inherits all the non-private methods from its direct superinterfaces that are accessible and not overridden by a declaration in the interface. An interface may inherit more than one method with the same signature, but then the methods must be abstract, and either the return types are compatible, or both the methods are void.

A method declared in an interface overrides all methods with the same signature in the superinterfaces of the interface. Further, it is required that the return type must be the same, or that both methods must be void, if a method overrides another.

Overloading. If two methods being member of an interface have the same name but different number of parameters they are said to be *overloaded*.[10]

10.6 The Interface `Input`

The interface `Input` is a so-called marker interface: it expects that an implementation has certain properties.

Every implementation of the interface `Input` must have as members

- a parameterless method `new_val` with the result type being `boolean`, and
- and parameterless method `get_val`.

The interface is used as parameter type for sensors (signals of kind `Sensor`).

An implementation of the interface `Input` determines the behaviour of an input sensor:

- An *input sensor* is set to be present at an instant if the method `new_val` returns the value true. If the control signal is valued then its value is updated by the return value obtained by calling the method `get_val`.

The method `new_val` is called at every instant, the method `get_val` only if the method `new_val` returns the value true.

Whether an implementation of the interface `Input` provides the methods `new_val` and `get_val`, and whether the result type of the method `get_val` is compatible with the value type of the respective input sensor is checked by the compiler.

10.7 The interface `Output`

The interface `Output` also is a marker interface.

Every implementation of the interface `Output` must have a member being

- a void method `put_val`

[10] In contrast methods are overloaded in JAVA™ if the signatures differ.

The interface is used as parameter type for signals of kind `Signal` and `DelayedSignal`.

An implementation of the interface `Output` determines the behaviour of an output signal:

- If an *output control signal* is updated (by emmittance or a flow equation) the method `put_val` is called. If the signal is valued the method `put_val` has this value as actual parameter.

Whether an implementation of the interface `Output` provides the method `put_val`, and whether the parameter type of the method `get_val` is compatible with the value type of the respective output signal is checked by the compiler.

11 Arrays, Vectors, and Matrices

Array types and variables. Arrays in *synERJY* are restricted in that arrays are one- or two-dimensional only. Note that two-dimensional arrays are homogeneous in that the length of "lines" is always equal as is the length of "columns".

Arrays are objects that may be dynamically generated. They are comprised of a number of variables (its *components*) that are referenced by *array access expressions*. The components of an one-dimensional array of length n are indexed from 0 to $n-1$, that of a two-dimensional are indexed from $(0,0)$ to $(m-1,n-1)$ with m being the number of colums, and n being the number of rows.

Components of an array must have the same type, being either of primitive type, of class or of interface type.

One dimensional arrays have types of the form T`[]`, and two-dimensional arrays have types of the form T`[,]` with T being a primitive type, a class type or an interface type.

Array variables have values being references to arrays.

Arrays are created using a *array creation expression*, by a *array initializer*, or are implicitly created by applying an array operation.

One-dimensional arrays are accessed using an access expression of the form A`[`E`]` where A is an expression referencing an one-dimensional array, and E is an integer expression indexing the array. Two-dimensional arrays are accessed using an access expression of the form A`[`E_1`,` E_2`]` where A is an expression referencing a two-dimensional array, and E_1 and E_2 are integer expressions indexing the array (cf. *ArrayAccess*[108]).

Array accesses are checked at run-time; if an index is used that is smaller than zero or greater or equal than the length of the one-dimensional array, resp. the dimensions of the two-dimensional array, the exception

`array_index_out_of_bounds_exception`

will be thrown.

The only member of an one-dimensional array is the public parameterless method `length` that specifies the number of components. The only members of a two-dimensional array are the public parameterless methods `columns` and `rows` that specify the number of columns and rows.

Array initializers. If the variable has an initializer it is evaluated when a array variable is created. An initializer of a one-dimensional array is of the form {e_1`, ...,` e_n} with the e_i's being expressions. The initializer is of a

two-dimensional array is the form $\{\{e_{0,0}, \ldots, e_{0,n}\}, \ldots, \{e_{m,0}, \ldots, e_{m,n}\}\}$ with the $e_{i,j}$'s being expressions.

ArrayInitializer:
    OneDimensionalArrayInitilizer[68]
    TwoDimensionalArrayInitilizer[68]
OneDimensionalArrayInitilizer:
    { *VariableInitializers*[68]$_{opt}$ }
VariableInitializers:
    VariableInitializer[52]
    VariableInitializer[52] , *VariableInitializers*[68]
TwoDimensionalArrayInitilizer:
    OneDimensionalArrayInitilizer[68] , *OneDimensionalArrayInitilizer*[68]

An array initializer may be specified in the declaration of an array creation expression (cf. *ArrayCreationExpression*[100]) This creates a vector or matrix and provides some initial values. The number of expressions in an array initializer determines the number of components of an array. Evaluation of an matrix initializer is from the left to the right.

Vector and matrix types and variables. Vectors and matrices are subtypes of one rsp. two-dimensional arrays. The particular features are:

- Dimensions are part of the type declaration,
- Standard vector and matrix operators are defined (see below)
- Vectors and arrays cannot be generated dynamically.

Vector and matrix variables have values being references to vectors and matrices.

Vectors and matrices are created using a *vector/matrix creation expression*, by an *vector/matrix initializer*, or are implicitly created by applying an vector/matrix operation.

Vectors are accessed using an access expression of the form V`[`E`]` where V is an expression referencing a vector, and E is an integer expression indexing the array. Matrices are accessed using an access expression of the form M`[`E_1, E_2`]` where M is an expression referencing a matrix, and E_1 and E_2 are integer expressions indexing the array (cf. *VectorOrMatrixAccess*[109]).

Vector/Matrix accesses are checked at *compile-time*; if an index is used that is smaller than zero or greater or equal than the length of the vector/matrix a typing error will be displayed.

Vector and matrix types and variables. Vectors and matrices behave like one- resp. two-dimensional arrays except that they restricted to conform to the size requirements as specified in the type, e.g.

```
int[3]     x = { 1, 2, 3 };
float[3,2] y = { { 1.1f, 1.2f}, { 2.1f, 2.2f}, { 3.1f, 3.2f} };
```

Vector and matrix operators. Since we distinguish between vectors and matrices as types, we gain some notational freedom using overloading. We discuss each the operators separately comparing it with its mathematical equivalent. Values must always be of the same type.

- ***Scalar product.*** The multiplication `a * b` of two vectors `a` and `b` is defined by

  $$\bar{a} * \bar{b} = \sum_{i=0}^{N-1} a_i * b_i$$

  where the vectors must both be of length N.

- ***Matrix multiplication.*** The multiplication `a * b` of two matrices `a` and `b` is defined by

  $$\bar{\bar{a}} * \bar{\bar{b}} = \begin{pmatrix} \sum_{i=0}^{N-1} a_{0,k} * b_{k,0} & & \cdots \\ & \cdots & \\ \cdots & & \sum_{i=0}^{N-1} a_{M-1,i} * b_{i,L-1} \end{pmatrix}$$

  where the matrices are of type $T[M, N]$ and $T[N, L]$. The result has type $T[M, L]$.

- ***Multiplication with a scalar.*** The multiplication `a * b` of a vector `a` and a scalar `b` is defined by

  $$\bar{a} * b = (a_0 * b, \ldots, a_{N-1} * b)$$

  and similarly for matrices

  $$\bar{\bar{a}} * b = \begin{pmatrix} a_{0,0} * b & & \cdots \\ & \cdots & \\ \cdots & & a_{M-1,N-1} * b \end{pmatrix}$$

  The multiplication `b * a` is defined in analogy.

- ***Point-wise operations.*** The point-wise vector multiplication `a .* b` of a vectors `a` and a vector `b` is defined by

$$\bar{a} \texttt{ .* } \bar{b} = (a_0 * b_0, \ldots, a_{N-1} * b_{N-1})$$

and similarly for matrices

$$\bar{\bar{a}} * \bar{\bar{b}} = \begin{pmatrix} a_{0,0} * b_{0,0} & & \ldots \\ & \ldots & \\ \ldots & & a_{M-1,N-1} * b_{M-1,N-1} \end{pmatrix}$$

Addition $\bar{a}$ `+` $\bar{b}$ and subtraction $\bar{a}$ `-` $\bar{b}$ are defined likewise.

- ***Transpose operator.*** The transpose operator `a^t` on a matrix `a` is defined by

$$\bar{\bar{a}}\texttt{\^{}t} = \begin{pmatrix} a_{0,0} & & a_{N,0} \\ & \ldots & \\ a_{0,M} & & a_{N,M} \end{pmatrix}$$

where `a` is of type $T[M, N]$. In case of vectors the transpose operator is defined by

$$\bar{a}\texttt{\^{}t} = \begin{pmatrix} a_{0,0} \\ \ldots \\ a_{N,0} \end{pmatrix}$$

hence `a^t` is of type $T[N, 1]$ if `a` is of type $T[N]$.

- ***Diagonal operator.*** The diagonal operator `D` generates a quadratic diagonal matrix $\texttt{D}(x)$ of size N such that

$$\texttt{D}(x) = \begin{pmatrix} x & & 0 \\ & \ldots & \\ 0 & & x \end{pmatrix}$$

The dimension N is determined by the context.

- ***All operator.*** The all operator `A` generates a vector $\texttt{A}(x)$ of size N such that

$$\texttt{A}(x) = \begin{pmatrix} x \\ \ldots \\ x \end{pmatrix}$$

The dimension N is determined by the context.

Vectors considered as matrices. For notational convenience vectors `a` of type $T(M)$ will be considered as a matrix of type $T(1, M)$ in case of matrix multiplication. To give an example, let `a` be a vector of type $T(M)$ and `b` be a matrix of type $T(M, N)$. Then

$$\bar{a} * \bar{\bar{b}}$$

will be well defined since `a` is considered as a matrix of type $T(1, M)$. On the other hand $\bar{\bar{b}} * \bar{c}$ is not well defined for a vector `c` of type $T(N)$. We have to use the transpose of `c` in the multiplication

$$\bar{\bar{b}} * \bar{c}\texttt{\^{}t}$$

Note that, in consequence, we have that $\bar{a} * \bar{b}$ and $\bar{a} * \bar{b}\texttt{\^{}}t$ are equal if considered as matrices. This, however, does not cause inconsistencies since the first term denotes a vector, the second a matrix.

Slices. Slices of vectors and matrices may be used. The general definition of a vector slice

$$\bar{v}[i..j] = \{v_i, v_{i+1}, \ldots, v_j\}$$

provided that $\bar{v}[i..j] = \{v_0, v_{i+1}, \ldots, v_n\}$ such that $0 \leq i$, $i \leq j$, and $j \leq n$. Similar $\bar{m}[i_1..j_1, i_2..j_2$ denotes the submatrix as defined by the bounds $i_1 \leq j_1$, $i_2 \leq j_2$

We use the same notation within *synERJY*. The restriction (at present) is that the bounds as well of the size of slized vectors and matrices can be computed at compile time.

For typing, the slice $\bar{v}[i..j]$ is of $T[j-i]$ if $\bar{v}$ is of type $T[n]$, and $\bar{m}[i_1..j_1, i_2..j_2]$ is of type $T[j_1 - i_1, j_2 - i_2]$ if $\bar{m}$ is of type $T[m, n]$.

We consider $\bar{v}[i..i]$ as being equivalent to $\bar{v}[i]$, as well as $\bar{m}[i..i, j..j]$ as being equivalent to $\bar{m}[i, j]$. In the same way, the slice $\bar{m}[i_1..i_1, i_2..j_2]$ corresponds to $\bar{m}[i_1, i_2..j_2]$, etc. .

12 Execution and Exceptions

12.1 Reactive Configuration Classes

The compiled code starts up with executing the method `main` of the *configuration class*, i.e. the within an application unique class with a method `main`.

A configuration class cannot be abstract. This implies that the last statement of the constructor is of the form `active { ...}` if the configuration class is reactive. We speak of the *reactive executable* if referring to the reactive code in the constructor of a configuration class.

One instant of the reactive executable is executed if the system method `instant` is called. The method `instant` has no parameters. It returns an integer as an exception. If the result is 0 no exception is raised. Typically the body of the method `main` has a last statement often being of the form

```
while ( instant() == 0 ) { };
```

In a reactive configuration class, a special constant `timing` of type `time` may be declared, e.g.

```
static final time timing = 250msec;
```

The constant specifies the timing requirement for an instant to take place, i.e the maximal amount of time an instant may take till all computations at that instant are completed. Further it states after which amount of time the next instant should begin. We speak of the timing of the execution cycle.

The timing is used in the simulator for an exact timing of the instants in the simulator of the *synERJY* tool set. If the timing of the execution cycle on the target architecture coincides with the declared cycle time, the execution of an *synERJY* program on the simulator is in one-to-one correspondence to execution of the same program on the target architecture.

12.2 Exception Handling

Exception handling of *synERJY* considerably differs from the exception handling of JAVA™. This is a consequence of *synERJY* programs being essentially reactive. Reactive programs are usually expected to run forever. Any kind of run-time exception compromises the correctness of the *whole* program, not only of some part. Therefore exceptions are handled top-level, in the main loop: corrections must be executed before running the next instant of the reactive system. In case of exception handling the tail of the method `main` is often of the form

```
while ( instant() == 0 ) { };
```

Exceptions are raised using assertions in , for instance, pre- or postconditions, or invariants. An assertion consists of a Boolean guard that if true raises an exception being an integer value.

Assertions:
Assertion[73]
Assertion[73] *Assertions*[73]
Assertion:
(*Expression*[122]) `then throw` (*Expression*[122]) ;
(*Expression*[122]) `else throw` (*Expression*[122]) ;

Given an assertion of either the form e_1 `then throw` e_2 or e_1 `else throw` e_2, the assertion is evaluated by first evaluating the Boolean condition e_1 and then by evaluating the integer expression e_2 given that the condition evaluates to true or false in an obvious way.

The following exceptions are predefined:

1 ↔ `null_pointer_exception`
2 ↔ `array_index_out_of_bounds_exception`
3 ↔ `array_negative_size_exception`
4 ↔ `out_of_memory_exception`
5 ↔ `throw_value_is_zero_exception`
6 ↔ `instant_caused_time_overflow_exception`
7 ↔ `class_cast_exception`

The numbers 8 to 99 are reserved for further system exception, and the numbers 100 to 999 are reserved for exceptions thrown by the runtime system. Users must use exceptions greater than 999.

13 Blocks and Statements

A block is a sequence of statements, and of local variable declarations statements within braces. Reactive statements are a new kind of statement for specifying synchronous behaviour (cf. Section 14.3).

Normal execution of statements is specified below. A statement completes normally if all computation steps are carried out successfully. One speaks of a *normal completion*. Normal completion may be prevented because of

- transfer of control due to a `break`, `continue`, or `return` statement,
- throwing an exceptions (*the exception of synERJY substantially differs from that of* JAVA^{TM})

In that case one speaks of an *abrupt completion*. If not specified otherwise, abrupt completion of a substatement results in an abrupt completion of the statement itself.

13.1 Blocks

A block consists of a sequence of statements and is of the form
{ $stmt_1$ ... $stmt_n$ }.

Block:
{ *BlockStatements*$^{74}{}_{opt}$ }
BlockStatements:
BlockStatement^{74}
BlockStatement^{74} *BlockStatements*^{74}
BlockStatement:
LocalVariableDeclarationStatement^{75} ;
Label$^{16}{}_{opt}$ *Statement*^{75} ;

Every statement may be labelled. Labels allow to refer to a statement.[11].*Every statement must be terminated by a semicolon.Local class declarations are not allowed within a block since dynamic creation of objects is not supported.*.

13.2 Local Variable Declarations

A local variable declaration statement declares variable. *The modifier* `final` *is not supported.*

[11] As a reminder: labels of *synERJY* are of the form "*Identifier*^{16}`::`"

LocalVariableDeclarationStatement:
PrimitiveType[21] *VariableDeclarator*[52] ;
ReferenceType[24] *VariableDeclarator*[52] ;
SignalType[27] *SignalDeclarator*[54] ;

A local variable declaration statement can be freely mixed with other statements within a block. Its scope is the rest of the block in which it appears. The name of a local variable may never be redeclared, no shadowing is allowed (cf. Section 6).

13.3 Statements

Statements conform with those of JAVA™ with a few exceptions.

Statement:
EmptyStatement[76]
Assignment[76]
ExpressionStatement[77]
IfThenStatement[77]
IfThenElseStatement[77]
SwitchStatement[78]
WhileStatement[79]
DoStatement[79]
ForStatement[80]
BreakStatement[81]
ContinueStatement[81]
ReturnStatement[81]
ThrowStatement[82]
AssertStatement[82]

The try and the synchronized statement are not supported. The assert statement is additional statement.

13.4 The Empty Statement

The empty statement does nothing. Unlike as in JAVA™ the empty statement can be made explicit.

EmptyStatement:
`nothing`

13.5 The Assignment Statement.

There are several assignment statements.[12]

Assignment:
LeftHandSide[76] *AssignmentOperator*[76] *Expression*[122]
LeftHandSide:
Identifier[16]
FieldAccess[107]
ArrayAccess[108]
VectorOrMatrixAccess[109]
AssignmentOperator: one of
`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
`:=`

The result of the left hand side of an assignment operator must be a variable. This may be a named variable, such as a local variable or a field of the current object or class or a computed variable, as can result from an *ArrayAccess*[108] or *VectorOrMatrixAccess*[109]. *In contrast to* JAVA™, the variable may not result from a *FieldAccess*[107] except if the variable is of the form "`this`....".[13]

The expression on the left hand side must have the same type as the expression on the right hand side.

Simple assignment `=`. First the left hand side is evaluated. If the computation fails to evaluate to a variable, evaluation completes abruptly, not evaluating the right hand expression. Otherwise, the expression is evaluated. If evaluation completes abruptly, evaluation of the assignment statement completes abruptly, and no assignment occurs. Otherwise, the value obtained for the right hand side is assigned to the left hand side.

Compound Assignment. All compound assignment operators must have operands of primitive type, except for `+=`, where operands may be of type `String`.

A compound assignment *x op*`=` *e* evaluates like *x* `=` *x op e*.

[12]Note that assignment is a statement unlike to JAVA™ where assignment is an operator.

[13]to support good software engineering: attributes should in principle be changed by setter methods. The value of attributes may be read if made public.

13.6 Expression Statement

Certain kinds of expressions may be used as statements.

ExpressionStatement:
Assignment[76]
IncrementDecrementExpression[112]
MethodInvocation[101]
ClassInstanceCreationExpression[99]

The method invoked must be a data method.

An expression statement is executed by evaluating the expression. Values is discarded. An expression statement completes normally if and only if evaluation of the expression completes normally.

13.7 The `if` Statements

The if statement allows conditional execution of a statement or a conditional choice of two statements.

The then- and else-branches of the if statement must always be blocks (*Block*[74]).

IfThenStatement:
`if (` *Expression*[122] `)` *Block*[74]

IfThenElseStatement:
`if (` *Expression*[122] `)` *Block*[74] `else` *Block*[74]

The expression must be of type `boolean`.

An if statement is executed by first evaluating the expression. execution of the if statement completes abruptly if the evaluation of the expression does. Otherwise, if the expression evaluates to true, the then branch is executed, or, if the expression evaluates to false, the if-then statement terminates normally, or, for the if-then-else statement, the else branch is executed. The if statement completes normally if the then or else branch completes normally.

13.8 The `switch` Statement

The switch statement transfers control depending on the value of an expression.

SwitchStatement:
`switch` (*Expression*[122]) *SwitchBlock*[78]
SwitchBlock:
{ *CaseList*[78]$_{opt}$ *Default*[78]$_{opt}$ }
CaseList:
Case[78]
Case[78] *CaseList*[78]
Case:
SwitchLabels[78] *BlockStatements*[74]
SwitchLabels:
SwitchLabel[78]
SwitchLabel[78] *SwitchLabels*[78]
SwitchLabel:
`case` *SwitchId*[78] :
`case` *SwitchRange*[78] :
SwitchRange:
SwitchId[78] .. *SwitchId*[78]
SwitchId:
Literal[17]
Identifier[16]
Default:
`default:` *BlockStatements*[74]

The expression must be of an integral type. The switch labels must have the type of the expression. In *synERJY*, the `default` must be the last case.

When the switch statement is executed the expression is evaluated. Execution of the if statement completes abruptly if the evaluation of the expression does. Otherwise, the value of the expression is compared with the switch labels.

Then, if either a literal or the value of an identifier matches the value of the expression, of if the value of the expression is in a switch range, all statements after the matching switch label in the switch block, if any, are executed in sequence.

Otherwise, if there is a default label, then all statements after the default label are executed in sequence.

If neither of these cases apply, the switch statement completes normally.

The switch statement completes normally if all the statements to be executed complete normally. If any of these statement completes abruptly because of a break statement the switch statement also completes normally.

Otherwise the switch statement completes abruptly if one of these statements does.

13.9 The `while` Statement

The while statement executes an expression and a statement repeatedly until the value of the expression is false.

WhileStatement:
`while (` *Expression*[122] `)` *Block*[74]

The expression must have type `boolean`.

Execution of a while statement first evaluates the expression. If the evaluation of the expression completes abruptly, the while statement completes abruptly. Otherwise, if the expression evaluates to true, the block is executed and the while statement is executed again, or, if the expression evaluates to false the while statement completes normally.

If the block completes abruptly because of

- a `break`, the while statement immediately completes normally.
- a `continue`, the while statement is immediately executed again.

Otherwise if the execution of the block completes abruptly, the while statement completes abruptly.

13.10 The `do` Statement

The do statement executes a Statement and an Expression repeatedly until the value of the Expression is false.

DoStatement:
`do` *Block*[74] `while (` *Expression*[122] `)`

The expression must have type `boolean`.

Execution of a do statement first evaluates the block. If the evaluation of the block completes normally the expression is evaluated. If the evaluation of the expression completes abruptly, the do statement completes abruptly. Otherwise, if the expression evaluates to true, the do statement is executed again, or, if the expression evaluates to false the do statement completes normally.

If the block completes abruptly because of

- a `break`, the do statement immediately completes normally.

- a `continue`, the expression is immediately evaluated.

Otherwise if the execution of the block completes abruptly, the do statement completes abruptly.

13.11 The `for` Statement

The for statement executes some initialization code, then executes an expression, a block, and some update code repeatedly until the value of the expression is false.

ForStatement:
`for (` *ForInit*$^{80}{}_{opt}$ `;` *ForExit*$^{80}{}_{opt}$ `;` *ForUpdate*$^{80}{}_{opt}$ `)` *Block*^{74}
ForInit:
Expression^{122}
LocalVariableDeclarationStatement^{75}
ForExit:
Expression^{122} *ForUpdate:*
Expression^{122}

The expression must have type `boolean`.

The for statement is executed by first executing the forinit clause. If this completes abruptly the for statement completes abruptly. The scope of a local variable declared in the forinit clause is the whole for statement.

The the iteration step is performed that consists of the following substeps:

- If present the expression is evaluated. If the evaluation of the expression completes abruptly, the for statement completes abruptly.Otherwise the next substep is executed.
- If the value of the expression is true, or if the expression is not present, the block is executed. If the block completes normally the next substep is executed. Otherwise, if the block completes abruptly
  - because of a break statement the for statement completes normally, or
  - because of a continue statement the next substep is executed.

  If the value of the expression is false the for statement completes normally.

- If the update clause is present the expression is evaluated discarding its value. If the evaluation completes normally the for statement is executed again. Otherwise the for statement completes abruptly.

13.12 The `break` Statement

A break statement transfers control out of an enclosing statement. A break statement may occur only in a while, do, or for statement.

BreakStatement:
`break`

The break statement attempts to transfer control to the innermost enclosing switch, while, do, or for statement which then immediately completes normally.

13.13 The `continue` Statement

A continue statement may occur only in a while, do, or for statement. statement.

ContinueStatement:
`continue`

The continue statement attempts to transfer control to the innermost enclosing while, do, or for statement which then immediately ends the current iteration and begins a new one.

13.14 The `return` Statement

A return statement returns control to the invoker of a method.

ReturnStatement:
`return` $Expression^{122}_{opt}$

A return statement must be contained in the body of a method of of a constructor, but not within an instance initializer or a static initializer. A return statement always completes abruptly attempting to transfer control to the invoker of the method or constructor that contains it. If it has an expression the value of the expression becomes the value of the method invocation. If the evaluation of the expression completes abruptly, the return

statement completes abruptly. If the evaluation of expression completes normally the return statement completes abruptly by transferring the control to the invoker returning the value of the expression.

A return statement without expression must be in the body of a method being declared using the keyword `void`. A return statement with an expression must be in the body of a method being declared to have a value of the same type as the expression has.

13.15 The `throw` Statement

ThrowStatement:
`throw` `(` *Expression*[122] `)` `;`

13.16 The `assert` Statement

AssertStatement:
`assert` `{` *Assertions*[73]$_{opt}$ `}`

14 Reactive Blocks an Reactive Statements

14.1 Reactive Blocks

The reactive statements define the synchronous sublanguage of *synERJY*. A reactive block consists of a sequence of reactive statements and is of the form { $rctstmt_1$, ..., $rctstmt_n$ }.

ReactiveBlock:
`{` *ReactiveBlockStatements*[82] `}`
ReactiveBlockStatements:
ReactiveBlockStatement[82]
ReactiveBlockStatement[82] *ReactiveBlockStatements*[82]
ReactiveBlockStatement:
ReactiveLocalSignalDeclaration[83]
Label[16]$_{opt}$ *ReactiveStatement*[83] `;`

A reactive statement may be labelled.[14] Every reactive statement must be terminated by a semicolon.

[14] As a reminder: labels of *synERJY* are of the form "*Identifier*[16] `:`"

14.2 Reactive Local Signal Declarations

A local signal declaration statement declares signal variable that must be initialized. For notational convenience, the initialisation may be dropped.

ReactiveLocalSignalDeclaration:
    SignalDeclaration[54] **;**

A local signal declaration statement can be freely mixed with other statements within a reactive block. Its scope is the rest of the block in which it appears. The name of a local variable may never be redeclared, no shadowing is allowed (cf. Section 6).

14.3 Reactive Statements

The reactive statements are as follows

ReactiveStatement:
    ReactiveEmptyStatement[84]
    ReactiveExpressionStatement[84]
    ReactiveIfThenStatement[85]
    ReactiveIfThenElseStatement[85]
    ParallelStatement[85]
    LoopStatement[86]
    EmitStatement[86]
    HaltStatement[86]
    NextStatement[86]
    CancelStatement[87]
    AwaitStatement[86]
    SustainStatement[88]
    ActivateStatement[89]
    AutomatonStatement[90]
    NextStateStatement[91]
ReactiveStatements:
    ReactiveStatement[83]
    ReactiveStatement[83] *ReactiveStatements*[83]

Reactive statements may be completed instantaneously, i.e. at the same instant they start to be evaluated, or at later instants (an instant being an execution step of the reactive machine, cf. Section 1.1).

14.4 The Reactive Empty Statement

The reactive empty statement is made explicit to denote its including block as reactive.

ReactiveEmptyStatement:
`nothing`

The statement completes normally and instantaneously.

14.5 The Reactive Expression Statement

Within a reactive block, assignments may be executed, variables may be incremented or decremented, and methods may be called.

ReactiveExpressionStatement:
Assignment[76]
IncrementDecrementExpression[112]
ReactiveMethodInvocation[105]

The statement completes normally if either of the statements completes normally. If it completes, the statement completes instantaneously, except if the method is reactive. Then the statement completes if the method completes. The method invoked must be either a data method or a reactive method, but not a node. A node may only be invoked within a flow context (cf. Section 14.18).

14.6 The Reactive `if` Statements

The if statement allows conditional execution of a statement or a conditional choice of two statements.

The reactive if statement is of either of the form

`if (`*bexpr*`) {`
then − branch
`}`

or

`if (`*bexpr*`) {`
then − branch
`} else {`
else − branch
`}`

with the expression being a Boolean expression, and with the then and else branch each being a list of reactive statements.

ReactiveIfThenStatement:
`if (` *Expression*[122] `)` *ReactiveBlock*[82]
ReactiveIfThenElseStatement:
`if (` *Expression*[122] `)` *ReactiveBlock*[82] `else` *ReactiveBlock*[82]

An reactive if statement is executed by first evaluating the expression. execution of the if statement completes abruptly if the evaluation of the expression does. Otherwise, if the expression evaluates to true, the then branch is executed, or, if the expression evaluates to false, the if-then statement terminates normally, or, for the if-then-else statement, the else branch is executed. The if statement completes normally if the then or else branch completes normally.

The evaluation of the expression is instantaneous. If the evaluation of the expression completes normally the then- or else-branch is started instantaneously.

14.7 The Parallel Statement

The parallel statement concurrently evaluates its branches.

ParallelStatement:
`[[` *ParallelStatements*[85] `]]`
ParallelStatements:
ReactiveStatements[83]
ReactiveStatements[83] `||` *ParallelStatements*[85]

The parallel statement completes normally if all its branches have completed normally. Note that the parallel statement synchronizes on completion. It only completes if all its branches have completed. The parallel statement completes abruptly if one of its branches completes abruptly.

14.8 The `loop` Statement

The body of the loop statement is started immediately when the loop statement is started. If the body completes abruptly the loop statement completes abruptly. If the body of the loop statement completes normally the loop statement is restarted instantaneously at the same instant. Hence the loop statement never completes if it does not complete abruptly.

LoopStatement:
`loop` *ReactiveBlock*[82]

14.9 The `emit` Statement

Signals are emitted using the emit statement.

EmitStatement:
`emit` *Identifier*[16]
`emit` *Identifier*[16] `(` *Expression*[122] `)`

The emit statement is evaluated instantaneously, and the signal is updated. In case that the signal is valued, the expression is evaluated at first. If the evaluation of the expression completes abruptly the evaluation of the emit statement completes abruptly. Otherwise the emit statement completes normally and instantaneously.

14.10 The `halt` Statement

The halt statement never completes. It keeps control forever.

HaltStatement:
`halt`

14.11 The `next` Statement

If evaluated the next statement keeps control till the next instant, i.e. if evaluation starts at one instant the next statement completes normally only at the next instant.

NextStatement:
`next`

14.12 The `await` Statement

The await statements waits till some condition becomes true.

AwaitStatement:
`await` `next`$_{opt}$ *Expression*[122]
`await` `next`$_{opt}$ `when` *WhenElseClause*[87]

First the expression is evaluated. If the evaluation of the expression completes abruptly the await statement completes abruptly. If the evaluation of the expression completes normally then either, if its value is true, the await statement completes instantaneously, or, if its value is false, the await statement keeps control in that the evaluation of the await statement is restarted at the next instant.[15]

14.13 The `cancel` Statement

The cancel statement allows to preempt evaluation. The most general form of the cancel statement has the pattern

```
cancel strongly_opt  next_opt  {
    body
}       when ( expression_1 ) { when_branch_1 }
   ...
   else when ( expression_m ) { when_branch_n }
   else { else_branch }
```

The modifiers `strongly` and `next` or optional. The body and the branches each consist of a list of statements.

CancelStatement:
`cancel strongly`$_{opt}$ `next`$_{opt}$ *ReactiveBlock*[82]
`when` *WhenElseClause*[87]

WhenElseClause:
WhenPart[87]
WhenPart[87] `else when` *WhenElseClause*[87]
WhenPart[87] `else` *ReactiveBlock*[82]

WhenPart:
`(` *Expression*[122] `)` *ReactiveBlock*[82]
`(` *Expression*[122] `)`

The behaviour of the cancel statement substantially depends on the modifiers used. There are four cases:

(i) *No modifier used*: If the evaluation of the cancel statement starts at an instant the reactive body is evaluated till it cannot be evaluated further at that instant. Then the when branches are evaluated successively at the same instant by evaluating the expression. If the evaluation completes normally, and if the expression's value is true,

[15] The semantics is that of the immediate await of Esterel [2].

the respective branch is evaluated at the same instant. The cancel statement completes normally (at that instant or at later instants) if the branch completes normally. If the expression evaluates to false the next branch is evaluated. If for none of the when branches, the expression evaluates to true, then, given an unconditional else branch, the else branch is evaluated instantaneously, otherwise evaluation of the cancel statement continues at the next instant by further evaluating the reactive body, and checking the condition of the when branches as described above.

(ii) If the modifier `next` is used, the cancel statement behaves as described in (i), but the conditions of the when branches are only checked at the next instant after evaluation of the cancel statement starts.

(iii) If the modifier `strongly` is used the behaviour of (i) is modified in that first the condition of the when branches are checked *before* the evaluation of the reactive body takes place.

(iv) If both the modifiers `strongly` and `next` are used the behaviour is that of (iii) but conditions are checked only at the next instant after evaluation of the cancel statement starts.

In all cases it holds that the evaluation of the reactive body completes abruptly if the evaluation of the reactive body, of the Boolean expressions or of the branches completes abruptly.

14.14 The `sustain` Statement

The sustain statement may be either of the form

```
sustain {| body |};
sustain {  body  };
```

The body consists of a list of flow constraints in the first case, and of a list of statements that evaluate instantaneously in the second case.

SustainStatement:
`sustain` *FlowContext*[92]
`sustain` *ReactiveBlock*[82] *the block must be instantaneous*

The behaviour of `sustain { block };` is equivalent to

```
loop {
    body
    next;
};
```

It is required that the body is "instantaneous, i.e. if evaluation of the body is started at an instant evaluation terminates at the same instant.

14.15 The `activate` Statement

The activate statement allows to "down-sample" evaluation of its body.

ActivateStatement:
`activate` `next`f_{opt} *ReactiveBlock*[82]
`when` `(` *Expression*[122] `)`

Evaluation of the activate statement starts by evaluating the expression. Evaluation completes abruptly if evaluation of the expression completes abruptly. If evaluation of the expression completes normally and the value is true, the body of the activate statement is evaluated. If the value is false, the body is not evaluated. The activate statement completes abruptly if the body does so. The activate statement completes normally if and only if it body completes normally.

If the modifier `next` is used the activate statement behaves as described above with the exception that the body is evaluated unconditionally at exactly the instant the evaluation of the activate statement starts.

14.16 The Automaton Statement

The automaton statement allows to specify automata in textual form. The general pattern is

```
automaton {
   init { init-branch }
   ...
   state s1
     entry  { entry-block }
     during { during-block }
     do     { do-block }
     exit   { exit-block }
   when (expression) { when-branch }
   else when (expression) { when-branch }
   ...
   else { else-branch }
```

```
    ...
};
```

All blocks and branches must be instantaneous except for the do block. The statement to be executed as the last one must be a next state statement (*NextStateStatement*[91]). In particular, if the last statement of a branch is a conditional all the branches of the conditional must end with a next state statement. Further every such conditional must have an else branch. The intention is that whenever a state is left, control should pass to another state.

Syntactically, automata are specified according to the following rules:

AutomatonStatement:
`automaton` { *AutomatonBody*[90] }
`automaton` *StringLiteral*[19]
AutomatonBody:
InitDefinition[90] *StateDefinitions*[90]
InitDefinition:
`init` *ReactiveBlock*[82] `;`
StateDefinitions:
StateDefinition[90]
StateDefinition[90] *StateDefinitions*[90]
StateDefinition:
`state` *Identifier*[16] *StateBody*[90] `when` *WhenElseClause*[87] `;`
StateBody:
StateEntry[90]$_{opt}$
StateDo[90]$_{opt}$
StateDuring[90]$_{opt}$
StateExit[90]$_{opt}$
StateDo:
`do` *ReactiveBlock*[82]
StateEntry:
`entry` *ReactiveBlock*[82] *the block must be instantaneous*
StateDuring:
`during` *ReactiveBlock*[82] *the block must be instantaneous*
`during` *FlowContext*[92]
StateExit:
`exit` *ReactiveBlock*[82] *the block must be instantaneous*

Evaluation of an automaton starts by evaluating the body the init definition. If the branch completes normally it completes instantaneously, and the state is entered that is specified by the next state statement executed as the last statement.

Entering a state at an instant the entry block is executed. If the entry block completes normally it completes instantaneously. Then the do block is evaluated. If evaluation of the do block terminates, the when clauses are evaluated successively. If the condition of a when clause evaluates to true, the exit block, and subsequently the respective reactive block is executed (resulting in the evaluation of a next state expression). If all the conditions evaluate to false, control remains with the entered state.

If being in a state, execution of the do block continues and the during block is executed in parallel, and then the clauses are excuted as stated above.

The automaton statement completes abruptly, if an abrupt completion occurs when evaluating any of its components. There is a particular next state statement `next state exit` that when executed causes the automaton to complete normally at the instant this particular statement is executed.

The behaviour of states is closely related with that of a cancel statement for which the modifier `next` but not `strongly` has been used.

14.17 The `next state` Statement

The next state statement can only be used within the automaton statement. It must be last statement executed by a branching in the automaton (cf. Section 14.16).

NextStateStatement:
`next state` *Identifier*[16]
`next state exit`

If evaluated the control is passed to the state specified. In case of statement `next state exit` the respective automaton completes normally at the instant the statement is called. A next state statement always completes normally and instantaneously.

14.18 Flow contexts.

A flow context consists of a sequence of local signal declarations, *flow equations*, *state variable equation*,of invocations of data methods of type `void`, or of invocations of nodes (but not of reactive methods).

FlowContext:
{| *FlowStatements*[92] |}
FlowStatements:
FlowStatement[92]
FlowStatement[92] *FlowStatements*[92]
FlowStatement:
SignalDeclaration[54] ;
FlowEquation[92] ;
StateVariableEquation[92] ;
MethodInvocation[101] ;
NodeInvocation[106] ;
FlowEquation:
Identifier[16] *VectorOrMatrixAccess*[109]$_{opt}$:= *Expression*[122] ;
StateVariableEquation:
Identifier[16]' := *Expression*[122] ;
(the identifier is primed)

Note that flow statements cannot be labelled.

Evaluation of a flow context proceeds as flows The order of presentation of the statements is irrelevant except for the following condition:

- A locally declared signal has to be declared before it is used.

Hence, given a flow context such as

```
{|  y := 2 * x;
     Signal<int> x = new Signal<int>();
     x := 0 -> pre(x) + 1;
|}
```

(with `y` being declared globally) will be rejected while

```
Signal<int> x = new Signal<int>();
y := 2 * x;
x := 0 -> pre(x) + 1;
```

is accepted. The order of evaluation is determined by the use of data, and will be ordered by the compiler accordingly. In case of the example, the equation will be evaluated in the following order

```
Signal<int> x = new Signal<int>();
x := 0 -> pre(x) + 1;
y := 2 * x;
```

since the value of `x` is used in the second equation. Note that in case of

```
Signal<int> x = new Signal<int>();
x := 0 -> pre(x) + 1;
y := 2 * pre(x);
```

any order will do: there is no direct use of a variable in the same instant, but only access to the previous values.

A flow equation s `:=` e constrains the value of a signal to be equal to that of the expression on its right hand side. The expression must be of a flow type. The type of the signal on the left hand side must be compatible with the type of the expression on the right hand side, e.g. `Signal`$\{C\}$`<`T`>` and $T\{C\}$.

Evaluation of a flow equation s `:=` e starts with evaluation of the flow expression e. If the evaluation of the expression completes abruptly, the evaluation of a flow constraint terminates abruptly. If evaluation of the expression completes normally it completes instantaneously. The value of the flow variable s is constrained to that value, and the flow equation completes normally and instantaneously.

Data method may be used to generate side effects within a flow context. Data methods complete instantaneously if they complete normally.

Note that, within a flow context, all expressions are typed to be of a flow type (*FlowType*[29]). In particular, all (data) expressions of type T are considered to be of type $T\{true\}$. The semantics of expressions of flow types is explained in Section 8.

15 Specification of Precedences

Time races may occur if, for instance, a valued signal is updated at the same instant twice and no order of the respective emit statements or flow equations can be determined from the program structure. This typically happens if a valued signal is updated in two parallel branches. Then a program is rejected with an error message.

The other case is that, e.g. method calls compete for resources in terms of read-write or write-write conflicts. A programmer may want to resolve such conflicts by prescribing the order of evaluation by specifying precedences.

We distinguish two categories of time race conflicts, those confined to a class, and those which can occur system-wide. There are different kinds of precedences covering these cases.

PrecedenceSpecification:
`precedence` { *Precedences*[94]$_{opt}$ };
Precedences:
Precedence[94]
Precedence[94] *Precedences*[94]
Precedence:
ClassPrecedences[95]
ApplicationPrecedence[96]

On a class level , we may have time races between methods, assignments, emit statements, flow equations, and access to variables. These are resolved either by

- by imposing a precedence order on method signatures or variables. Then, if involved in a time race, a method call of a method being smaller according to the precedence order will be evaluated before a method call of a method being greater according to the order.
- A finer precedence order on individual method calls, assignments, emit statements, and flow equations can be imposed by using labels and by imposing a precedence order on labels. Then, e.g. a method call with a label being smaller according to the precedence order on labels will be evaluated before a method call with a label being greater in the order, if the two method calls are engaged in a time race.

The rules of specifying such precedences are:

ClassPrecedences:
    Commutative[95]
    ClassPrecedences[95] `<` *Commutative*[95]
Commutative:
    SignatureOrLabel[95]
    Commutative[95] $\sim$ *SignatureOrLabel*[95]
SignatureOrLabel:
    Signature[95]
    Signature[95] `*`
    LabelSequence[95]
Signature:
    FieldName[36]
    `(` *FieldName*[36] `=` `)`
    MethodName[36] `(` *TypeParameters*[25]$_{opt}$ `)`
LabelSequence:
    Label[16]
    Label[16] *LabelSequence*[95]

So, for instance,

- `f(int) < g(int,boolean)` states that the method `f` with parameter of type `int` is executed before the method `g` with parameters of type `int` and `boolean`.
- `(x=) < y` states that all assignments to the variable `x` must be executed prior to accessing the value of the variable `y`.

  Note that the "signature" `(x=)` covers all the different assignment operators.
-

These rules apply only when the respective entities are engaged in a time race.

Since methods are private by definition in reactive classes, and since a reactive method may occur only in one non-abstract reactive class, time races involving methods are confined to classes. Hence on the level of applications only time races between emit statements and flow equations may occur. We believe that, outside of an object, no distinction should be made between any emittance or constraint of a signal within the object, i.e. emit statements

and flow equations of a signal within an object should be handled uniformly if observed on the outside. The precedence imposed should hence be specified in terms of the objects involved. The format chosen in *synERJY* is

`for` s `:` o_1 `<` o_2 `<` `...` `<` o_n

meaning that every update of a signal s in object o_1 should be scheduled before any update of the signal s in object o_2, etc. .

ApplicationPrecedence:
    `for` *Identifier*[16] `:` *PrecedenceOnObjects*[96]
PrecedenceOnObjects:
    FieldNameOrThis[96] `<` *FieldNameOrThis*[96]
    FieldNameOrThis[96] `<` *PrecedenceOnObjects*[96]
FieldNameOrThis:
    `this`
    FieldName[36] (reference to a reactive object)

16 Expressions

Kinds of expressions. The nature of an expression depends on its context.

- An expression within a flow context (*FlowContext*[92]) or a clock expression (*Clock*[27]) is called a *flow* expression. Its type is a signal type (*SignalType*[27]), and its values are references to flows.
- Otherwise, the type of an expression is a primitive type (*PrimitiveType*[21]) or reference type (*ReferenceType*[24]).

However, since operations on primitive or reference types lift to flows (cf. Section 8), these operators are only discussed for primitive or reference types.

Values and types. An expression can evaluate to a value, to a variable, or to nothing as a result, and its evaluation may have a side effect.

The result of an expression is nothing only if it invokes a method declared `void`. A method invocation can only occur in an expression statement (*ExpressionStatement*[77]).

If an expression denotes a variable, its value is used if required for further computation.

An expression has a type if its result is a value or a variable. The type is determined at compile time. The type of an expression is compatible with the type of its value.

All simple types and all operators on simple types are imported from C. Hence the behaviour of all these operators is inherited from C.

Completion. Evaluation of an expression may complete *normally* or *abruptly*. In the following situations an exception may be thrown causing abrupt completion :

- If sufficient memory is not available when creation a class or array instance, or concatenating strings. The error code for the exception `out_of_memory_exception` is thrown.
- If the dimension of an array is less than zero when creating an array instance,the error code for the exception `array_negative_size_exception` is thrown.

- If the value of an object reference is null when accessing a field the error code for the exception `null_pointer_exception` is thrown.
- If the target reference is null when invoking an instance method the error code for the exception `NullPointerException` is thrown.
- If the value of the array reference expression is null when accessing an array the error code for the exception `NullPointerException` is thrown.
- If the value of the array index expression is negative or greater than or equal to the length of the array when accessing an array the error code for the exception `ArrayIndexOutOfBoundsException` is thrown.
- If a cast is found to be not allowed at run time the error code for the exception `ClassCastException` is thrown.

Evaluation order. Expressions and lists of expressions evaluate from left to right. All operands are evaluated before the operator is applied. Only exceptions are the operators `&&`, `||`, and `? :`. The order of evaluation may be explicitly specified using parenthesis. There is an implicit precedence of operators that is specified by the order operators are introduced below.

16.1 Primary Expressions

Primary expressions that are not composed of other expressions by applying operators with one exception: expressions in parenthesis are considered as primary expressions.

Primary:
- *Literal*[17]
- `this`
- *ClassInstanceCreationExpression*[99]
- *ArrayCreationExpression*[100]
- *MethodInvocation*[101]
- *FieldAccess*[107]
- *ArrayAccess*[108]
- *VectorOrMatrixAccess*[109]
- *SensorOrSignalAccess*[109]
- *CastExpression*[110]
- *PrimaryFlowExpression*[111]

(*Label*[16]$_{opt}$ *Expression*[122])

The type of literals is defined as usual, e.g. Boolean literals are of type `boolean`.

The keyword `this` may be used only in the body of an instance method, instance initializer or constructor, or in the initializer of an instance variable of a class. In any of these cases it refers to the object in context of which it is used. The type of `this` is the class in the declaration of which it is used.

16.2 Class Instance Creation Expressions

An class instance creation expression creates a new instance of a class the type of which is that class.

ClassInstanceCreationExpression:
`new` *ClassType*[25] `(` *Arguments*[99]$_{opt}$ `)`
`new` *ClassType*[25] `()` `{` *AnonymousMethods*[99]$_{opt}$ `}`
Arguments:
Expression[122]
Expression[122] `,` *Arguments*[99]
AnonymousMethods:
MethodDeclaration[55]
MethodDeclaration[55] `;` *AnonymousMethods*[99]

A new instance may be either created by calling a constructor of some class T optionally extended by a class body. The constructor must be accessible. The constructor may be extended only if T is non-final. One speaks of an *anonymous class* if the option is taken. The anonymous class is a subclass of T which must not be final.

An anonymous class can never be abstract or static. It is implicitly final. An anonymous class has never has an explicit constructor but an compiler generated *anonymous constructor*.

A class is is said to be *instantiated* if some instance of it has been created. Abstract classes cannot be instantiated except if all its abstract methods are implemented by the optional class body of an anonymous class.

Which constructor of used is used for instantiation is determined by its signature.

The type of the class instance creation expression is the class type being instantiated. It may be an anonymous class type. in that case the respective anonymous constructor has been called.

Evaluation of a class instance creation expression has the following steps:

- Sufficient space is allocated. If sufficient space is not available the error code for the exception `out_of_memory_error` is thrown, and the class instance creation expression completes abruptly.
- New fields that are created for the new instance are set to a default.
- The arguments of the constructor are evaluated from left to right. If the evaluation of some argument completes abruptly the expressions on the right are not evaluated,and the class instance creation expression completes abruptly.
- The constructor is selected according to constructor signature and the constructor is evaluated according to explicit constructor invocation (*ConstructorInvocation*[60]) as explained in Section 16.2.
- The result of the class instance creation expression is a reference to the newly created instance.

If all steps complete normally the class instance creation expression completes normally.

Note that there are no member classes or local classes.

16.3 Array Creation Expressions

An array instance creation expression creates a new array. Arrays are one- or two-dimensional.

ArrayCreationExpression:
    OneDimensionalArrayCreationExpression[100]
    TwoDimensionalArrayCreationExpression[100]
OneDimensionalArrayCreationExpression:
    `new` *PrimitiveType*[21] `[` *Expression*[122] `]`
    `new` *ClassOrInterfaceType*[25] `[` *Expression*[122] `]`
TwoDimensionalArrayCreationExpression:
    `new` *PrimitiveType*[21] `[` *Expression*[122] `,` *Expression*[122] `]`
    `new` *ClassOrInterfaceType*[25] `[` *Expression*[122] `]`

The components have a type being either a primitive type, a class type or an interface type.

Evaluation of an array creation expression first evaluates, from left to right, the expressions that determines the dimensions that must be of integral type. If evaluation of one of the expressions completes abruptly, the

expressions on the right are not evaluated, and the array creation expressions completes abruptly. If the value of a dimension is less than zero the error code for the exception `negative_array_size_exception` is thrown.

Next, space is allocated according to the computed dimensions. If this fails the error code for the exception `out_of_memory_error` is thrown, and the array creation expression completes abruptly.

Then, the array is created with the specified dimension, and all its components are initialized by a default value.

If completion is normal the value of an array creation expression is a reference to the array created.

16.4 Vector and Matrix Creation

Vectors and Matrices are created using array creation expressions for one- and two-dimensional arrays. one should note however that the dimensions should be equal to those specified in the type, e.g.:

```
int[3]     x = new int[3];
float[3,2] y = new float[3,2]
```

16.5 (Data) Method Invocation Expressions

A (data) method invocation expression invokes a class or an instance method.

MethodInvocation:
MethodName[36] `(` *Arguments*[99]$_{opt}$ `)`
Primary[98] `.` *MethodName*[36] `(` *Arguments*[99]$_{opt}$ `)`
`super` `.` *MethodName*[36] `(` *Arguments*[99]$_{opt}$ `)`

Determining class (interface) and signature. When evaluating a method invocation expression at first the declaration of a method has to be determined that is invoked. There are several cases:

- If the method is invoked using a *MethodName*[36] then
  - if the method name is a simple name, i.e. an identifier, and if the identifier appears within the scope of a method declaration with that name, then there must be an enclosing type declaration with the respective method being a member. Let T be the innermost such type declaration. The class or interface to search is T.

  - If it is a qualified name of the form $X.y$ with X being a class Name, then the name of the method is y and the class to search is the one named by X.
  - In all other cases, the qualified name has the form $X.y$ with X being a field name. Then the name of the method is x and the class or interface to search is the declared type of the field named by the X.

- If method invocation is of the form is $X.y$ with X being a *Primary*[98], then the name of the method is x and the class or interface to be searched is the type of X.

- If method invocation is of the form is `super`.x, then the name of the method is x and the class to be searched is the superclass of the class whose declaration must contain the method invocation. The direct superclass should neither be an interface nor `Object`.

If no such method declaration can be determined the method invocation expression completes abruptly.

Applicability. Given that the declaration referred to in a method invocation expression has been determined, the next step is to check which method declarations are *applicable* – i.e. whether the number of arguments coincides with that of the formal parameters, and whether the types of the formal parameters and the arguments are compatible – and *accessible* (cf. Section 6.5). Note that there exists only one such method since no implicit conversion of types is allowed. It is called the *compile-time declaration* for the method invocation. If no such declaration is found the method invocation expression completes abruptly.

Appropriateness. The next step is to check whether the chosen method is *appropriate*:

- If an instance method is invoked using a *MethodName*[36] being an *Identifier*[16], then:
  - The invocation should not appear within a static context.
  - Otherwise, the invocation should be directly enclosed by the innermost enclosing class C of which the method is a member.

- If the method is invoked using a *MethodName*[36] of the form *TypeName*[36].*Identifier*[16], then the compile-time declaration must be `static`.
- If the method invocation is of the form `super`.x, then:
  - If the method is abstract, a compile-time error occurs.
  - If the method invocation occurs in a static context, a compile-time error occurs.
- If the compile-time declaration for the method invocation is `void`, then the method invocation must either occur in an expression statement (*ExpressionStatement*[77]) or in the *ForInit*[80] or *ForUpdate*[80] part of a for statement (*ForStatement*[80]), or a compile-time error occurs.

Compile time information. The following compile-time information is then associated with the method invocation for use at run time:

- The name of the method.
- The qualifying type of the method invocation.
- The number of parameters and the types of the parameters, in order.
- The result type, or void, as declared in the compile-time declaration.
- The invocation mode, computed as follows:
  - If the compile-time declaration has the static modifier, then the invocation mode is static.
  - Otherwise, if the compile-time declaration has the private modifier, then the invocation mode is nonvirtual.
  - Otherwise, if the method invocation is of type M3, then the invocation mode is super.
  - Otherwise, if the compile-time declaration is in an interface, then the invocation mode is interface.
  - Otherwise, the invocation mode is virtual.

If the compile-time declaration for the method invocation is not void, then the type of the method invocation expression is the result type specified in the compile-time declaration.

Runtime evaluation. Runtime evaluation consists of several steps:

Computing the target reference. There are several cases to consider, depending on the cases of the *MethodInvocation*[101]:

- If a method is invoked using a *MethodName*[36], then:
  - If the method name is an *Identifier*[16], then:
    - If the invocation mode is static, then there is no target reference.
    - Otherwise, let T be the enclosing type declaration of which the method is a member, then the target reference is `this`.
  - If the method name is a qualified name of the form *TypeName*[36].*Identifier*[16], then there is no target reference.
  - If the method name is a qualified name of the form *FieldName*[36].*Identifier*[16], then:
    - If the invocation mode is static, then there is no target reference.
    - Otherwise, the target reference is the value of the expression FieldName.
- If method invocation is of the form is $X.y$ with X being a *Primary*[98], then:
  - If the invocation mode is `static`, then there is no target reference. The expression X is evaluated, but the result is then discarded.
  - Otherwise, the expression X is evaluated and the result is used as the target reference.
- If method invocation is of the form is `super`.x,, then the target reference is the value of `this`.

Evaluation of arguments. The argument expressions are evaluated from left to right. If the evaluation of any argument expression completes abruptly, then no part of any argument expression to its right appears to have been evaluated, and the method invocation completes abruptly for the same reason.

Checking accessibility. Since there is no dynamic loading in *synERJY* accessibility is checked at compile time.

Locating the method to invoke. The strategy for method lookup depends on the invocation mode. Let T be the enclosing type declaration of method m. If m is an instance method then, if the target reference is `null` the error code for the `null_pointer_exception` will be thrown, otherwise the target reference is said to refer to a target object and will be used as the value of the keyword `this` in the invoked method.

static No target reference is needed and overriding is not allowed. Method m of class T is invoked.

nonvirtual Overriding is not allowed. Method m of class T is invoked.

interface or virtual Dynamic method lookup is used starting from the actual runtime class R of the target object.

super Dynamic method lookup is used starting from T.

If the dynamic method lookup start from class S, then if the class S contains a declaration for a non-abstract method named m with the same signature as required by the method invocation, then:

- If the invocation mode is `super` or `interface`, then this is the method to be invoked, and the procedure terminates.
- If the invocation mode is `virtual`, and the declaration in S overrides $X.m$, where Let X be the compile-time type of the target reference of the method invocation, then the method declared in S is the method to be invoked, and the procedure terminates.

Otherwise, if S has a superclass, this same lookup procedure is performed recursively using the direct superclass of S in place of S.

That the above procedure will always find a non-abstract, is guaranteed by the compile time analysis.

16.6 Reactive Method Invocation Expression

A reactive method invocation expression invokes a reactive method declared in the same class the method is invoked from.

ReactiveMethodInvocation:
MethodName^{36} `(` *Arguments*$^{99}{}_{opt}$ `)`

Method invocation is executed at compile time.

Applicability. Since reactive methods are always private, applicability has to be checked at first – i.e. whether the number of arguments coincides with that of the formal parameters, and whether the types of the formal parameters and arguments are compatible. Note that at most one method with the same number of parameters can be declared. If no such method declaration exists, the compiler yields an error message.

Compile time evaluation. Essentially, the reactive method is statically linked in that the method invocation expression is replaced as follows:

- for each parameter, a freshly created local variable of the same type is declared that is initialised with the respective argument.
- The declarations are followed by the body of the method invoked, but with all occurrences of the formal parameters being syntactically replaced by the respective freshly created local variable.

16.7 Node Invocation Expression

A node invocation expression invokes a noded declared in the same class the node is invoked from.

NodeInvocation:
MethodName^{36} (*Arguments*$^{99}_{opt}$)

Node invocation is executed at compile time.

Applicability. Since nodes are always private, applicability has to be checked at first – i.e. whether the number of arguments coincides with that of the formal parameters, and whether the types of the formal parameters and arguments are compatible. Note that at most one node with the same number of parameters can be declared. If no such node declaration exists, the compiler yields an error message.

Compile time evaluation. Node invocation behaves like invocation of a reactive methods except that a *base clock* is implicitly set in that

- all formal signal parameters are replaced by local variables that inherit the clock of the respective arguments.

In that clocks of a node are "relative", being only determined by each node invocation.

While invoking the nodes the clocks of arguments and formal parameters are checked to guarantee consistent clocks. In particular,

- all signals of type `Sensor<`T`>`, `Signal<`T`>`, or `DelayedSignal<`T`>` must have the same clock, and
- all the other formal parameters (that have "derived" clocks) must have corresponding (derived) clocks.

Note that, for proper typing with clocks, all the clocks should be uniquely determined by the clocks specified for the formal parameters. Hence nodes are not allowed to access the signals that are specified as a class member. Clocks of these are "absolute" in contrast to the clocks of a node that are relative, hence a clock check would fail.

The clock calculus is explained at length in the *synERJY* Language Introduction [3].

16.8 Field Access Expressions

A field access expression accesses a field on an object or array. The object or array are referenced by either the value of the expression or by the keyword `super`.

FieldAccess:
Primary[98] `.` *Identifier*[16]
`super .` *Identifier*[16]

The primary must be a reference type. The identifier must be name a field and must be unambiguous in the entity referenced by the primary expression or by `super`.

The field access expression is evaluated as follows:

- If the field is static final, the result is the value of the respective class variable in the class or interface that is the type of the primary expression.
- If the field is static but not final, the result is the respective class variable in the class or interface that is the type of the primary expression.
- If the field is not static and the value of the primary expression is the null pointer the error code for the exception `null_pointer_exception` is thrown.

- If the field is not static but final the result is the value of the respective instance variable in the class or interface that is the type of the primary expression.

- If the field is neither static nor final the result is the respective instance variable in the class or interface that is the type of the primary expression.

The keyword `super` may only be used in an instance method, instance initializer or constructor, or in the initializer of an instance variable of a class. The keyword `super` refers to the direct superclass, otherwise the same rules apply as for the primary expressions.

16.9 Array Access Expressions

An array access expression refers to the component of an array.

ArrayAccess:
OneDimensionalArrayAccess[108]
TwoDimensionalArrayAccess[108]
OneDimensionalArrayAccess:
ExpressionName[36] `[` *Expression*[122] `]`
Primary[98] `[` *Expression*[122] `]`
TwoDimensionalArrayAccess:
ExpressionName[36] `[` *Expression*[122] `,` *Expression*[122] `]`
Primary[98] `[` *Expression*[122] `,` *Expression*[122] `]`

A array access expression has two constituents, the *array reference expression* on the left – that must be of array type –, and the *index expression(s)* within the square brackets – that must be of integral type. The result of a array reference expression is a variable of the type of the array components. This variable is never final, even if the array reference stems from a final variable.

An array access expression is evaluated from left to right. If one of the expressions completes abruptly, the expressions on the right of the respective expression are not evaluated, and the array access expression completes abruptly. Otherwise

- If the value of the array reference expression is `null`, then the error code of the exception `null_pointer_exception` is thrown.

- Then, if the value of an array index expression is out of bounds, i.e. smaller than zero or greater than the specified dimension, the error code of the exception `array_index_out_of_bounds_exception` is thrown.
- Otherwise, the value of the array access expression is the variable of determined by the array referenced by the array reference expression and by the index values obtained from evaluating the array index expression(s), and the array access expression completes normally.

16.10 Vector and Matrix Access Expressions

An array access expression refers to the component of an array.

VectorOrMatrixAccess:
VectorAccess[109]
MatrixAccess[109]
VectorAccess:
ExpressionName[36] `[` *IndexOrSlice*[109] `]`
Primary[98] `[` *IndexOrSlice*[109] `]`
MatrixAccess:
ExpressionName[36] `[` *IndexOrSlice*[109] `,` *IndexOrSlice*[109] `]`
Primary[98] `[` *IndexOrSlice*[109] `,` *IndexOrSlice*[109] `]` *IndexOrSlice:*
Expression[122]
Expression[122] `:` *Expression*[122]

A vector/matrix access expression has two constituents, the *vector/matrix reference expression* on the left – that must be of vector/matrix type –, and the *index expression(s)* within the square brackets – that must be of integral type or a pair of integral types separated by a colon. The result of a vector/matrix reference expression is a variable of the type of the vector/matrix components. This variable is never final, even if the array reference stems from a final variable.

16.11 Sensor or Signal Access Expressions

The sensor or signal access expressions access the *presence*, the *value*, or the *(a)wait for signal* time of the sensor or signal specified by the identifier.

SensorOrSignalAccess:
`?` *Identifier*[16]

`$` *Identifier*[16]
`@` *Identifier*[16]

Remember that sensors or signals are always private members of a class hence the identifier unambiguously refers to a sensor or signal. Sensor or signal access expressions complete normally.

The present operator `?` The result of `?`*s* is `true` at an instant if the sensor or signal *s* is updated at that instant, otherwise `false`.

The value operator `$` The value operator is defined for valued sensors and valued signals. The result of `$`*s* is the value of the signal after it has been updated for the last time.

The (a)wait for signal operator `@` The result of `@`*s* is the amount of real time that has passed since the signal has not been present.

16.12 Cast Expressions

The cast expression converts a value of a primitive type to a similar value of another primitive type. Or it checks whether a reference type is compatible with a specified type.

CastExpression:
`(` `(` *PrimitiveType*[21] `)` *Primary*[98] `)`
`(` `(` *ReferenceType*[24] `)` *Primary*[98] `)`

The type of a cast expression is the type specified by the primitive or the reference type on the left. The result of a cast expression is a value, even if the primary evaluates to a variable.

If the evaluation of the primary expression completes abruptly, the evaluation of the cast expression completes abruptly. Otherwise the the value of the primary expression is converted to the type specified by the cast operator. If the cast is not permitted at run time, the error code of the exception `class_cast_exception` is thrown. Otherwise, the cast expression completes normally.

Note that cast expressions are primary in synERJY.

16.13 Primary Flow Expressions

Flow operators only apply to flow expressions, i.e. expressions the type of which is of the form $T\{C\}$ with T being primitive.

PrimaryFlowExpression:
    TimeShiftExpression[111]
    UpSamplingExpression[111]
TimeShiftExpression:
    `pre` `(` *Expression*[122] `)`
UpSamplingExpression:
    `current` `(` *Expression*[122] `)`

The time shift operator `pre`. The type of the operand expression must be a flow type. The type of the time shift expression is that of the operand expression.

The evaluation of a time shift expression completes abruptly, if its operand expression does so. Otherwise it completes normally.

The value of a time shift expression is a reference to a flow obtained by shifting the flow, in time (cf. Section 8), e.g.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
`pre`(d)	δ			d_1	d_2			...
d	d_0			d_1	d_2			...
ν_1	.	.		.	.		.	...
ν_0	.	.	.	.	.	.	.	...

Remark. We include the flows d and `pre`(d) in one flow diagram since they have the same clocks.

The flow operator `current`. The type of the operand expression must be a flow type. The type of the up-sampling expression is $T\{C'\}$ with the operand expression being of type $T\{C\}$ and the expression C being of type `boolean`$\{C'\}$.

The evaluation of a up-sampling expression completes abruptly, if its operand expression does so. Otherwise it completes normally.

The value of a up-sampling shift expression is a reference to a flow obtained by up-sampling the flow, in time (cf. Section 8), e.g.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
`current`(d)	d_0	d_0		d_1	d_2		d_2	...
ν_0	.	.	.	.	.	.	.	...

where

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
d	d_0			d_1	d_2			...
ν_1	.	.		.	.		.	...
ν_0	.	.	.	.	.	.	.	...

16.14 Prefix/Postfix Operators

Prefix and postfix expressions are the numeric increment and decrement and the cast from a primitive type to a flow expression type.

PrefixPostfixExpression:
Primary[98]
IncrementDecrementExpression[112]
IncrementDecrementExpression:
`++` *Identifier*[16]
`--` *Identifier*[16]
Identifier[16] `++`
Identifier[16] `--`

The increment and decrement operators `++` and `--`. The identifier must be a variable of numeric type. The type of the respective expression is that of the variable. The increment operator `++` adds 1 to the value of the variable and the decrement operator `--` subtracts 1 from the value of the variable. The operators are native.

The increment and decrement expressions always complete normally.

16.15 The Down-sampling Operator `when`

The down-sampling operator down-samples a flow (cf. Section 8).

DownsamplingExpression:
PrefixPostfixExpression[112]
PrefixPostfixExpression[112] `when` *PrefixPostfixExpression*[112]

The type of the left operand expression must be a flow type. The type of its right operand expression must be a Boolean flow type. The clocks of the respective flow types types must be equal. The type of the down-sampling

expression is $T\{C\}$ if the flow expression is of the form E `when` C, and if the type of E is of the form $T\{C'\}$ for some Boolean flow expression `C'`.

Down-sampling expressions are evaluated from left to right. If the left operand completes abruptly, the right operand is not evaluated, and the evaluation of the down-sampling expression completes abruptly. Then the right operand is evaluated. If its evaluation completes abruptly, the evaluation of the down-sampling expression completes abruptly. Otherwise the evaluation of the down-sampling expression completes normally.

The value of a down-sampling expression is a reference to a flow that is downs-sampled according to its clock. E.g. let the operands refer to the flows d and c

	1	*2*	*3*	*4*	*5*	*6*	...	
d	d_0	d_1		d_2	d_3		d_4	...
c	t	f		t	t		f	...
ν_0	.	.	.	.	.	.	.	...

Then the result of down-sampling is

	1	*2*	*3*	*4*	*5*	*6*	...	
d `when` c	d_0			d_2	d_3			...
ν_1	.	.		.	.		.	...
ν_0	.	.	.	.	.	.	.	...

16.16 Unary Operators

The unary operators are `+`, `-`, , and `!`. Unary operators group right-to-left, so that `-+x` means the same as `-(+x)`.

UnaryExpression:
- *DownsamplingExpression*[112]
- `+` *UnaryExpression*[113]
- `-` *UnaryExpression*[113]
- *UnaryExpression*[113]
- `!` *UnaryExpression*[113]

The result type always is the type of the operand.

If the operand is of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand flow.

The unary plus operator +. The unary plus operator is native. The operand expression must be a primitive numeric type or a primitive numeric flow type.

The result of the unary plus expression is the value of the operand or a reference to a flow.

The unary minus operator -. The unary minus operator is native. The operand expression must be a primitive numeric type or a primitive numeric flow type.

The result of the unary minus expression is the value obtained as a result of arithmetic negation.

The bitwise complement operator . The bitwise complement operator is native. The type of the operand expression must be a primitive integral type or a primitive integral flow type.

If evaluated the unary bitwise complement expression has a value being the bitwise complement of the value of the operand; note that x equals $(-x) - 1$.

Logical complement operator !. The type of the operand expression must be Boolean or a Boolean flow type.

The value of the unary logical complement expression is true if the operand value is false and false if the operand value is true.

16.17 Multiplicative Operators

The multiplicative operators `*`, `.*` (*point-wise multiplication of vectors and matrices*), `/` have the same precedence and are left-associative (except for floats and related types).

MultiplicativeExpression:
UnaryExpression[113]
MultiplicativeExpression[114] `*` *UnaryExpression*[113]
MultiplicativeExpression[114] `.*` *UnaryExpression*[113]
MultiplicativeExpression[114] `/` *UnaryExpression*[113]
MultiplicativeExpression[114] `%` *UnaryExpression*[113]

All the multiplicative operators are native. The operands must have the same primitive numeric types or the same primitive numeric flow types. The type of the expression is that of its operands.

If the operands are of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand flows.

The multiplication operator `*`. The operator performs multiplication, producing the product of its operands. Multiplication is a commutative operation if the operand expressions have no side effects. While integer multiplication is associative when the operands are all of the same type, floating-point multiplication is not associative.

If applied to vectors or matrices the multiplication operator denotes the *scalar product*, i.e.

- Given vectors a and b with the same dimension $T[n]$, the scalar product $a * b = \sum_{i=0}^{n} a[i] * b[i]$.
- Given matrices a and b with dimension $T[m, n]$ and $T[n, p]$, the scalar product $(a * b)[i, k] = \sum_{j=0}^{n} a[i, j] * b[j, k]$.

The point-wise multiplication operator `,*`. The operator performs point-wise multiplication of vectors and matrices. Point-wise multiplication is a commutative operation if the operand expressions have no side effects. While integer multiplication is associative when the operands are all of the same type, floating-point multiplication is not associative.

Point-wise multiplication is defined as follows

- $(a * b)[i] = a[i] * b[i]$ for vectors a and b with the same dimension $T[n]$.
- $(a * b)[i, j] = a[i, j] * b[i, j]$ for matrices a and b with dimension $T[m, n]$ and $T[n, p]$.

The division operator `/`. The operator performs division producing the quotient of its operands. The left-hand operand is the dividend and the right-hand operand is the divisor.

The operator is defined point-wise for vectors and matrices.

The remainder operator `%`. The operator yields the remainder of its operands from an implied division; the left-hand operand is the dividend and the right-hand operand is the divisor.

16.18 Additive Operators

additive operators `+`, `-` have the same precedence and are left-associative.

AdditiveExpression:
MultiplicativeExpression[114]
AdditiveExpression[116] `+` *MultiplicativeExpression*[114]
AdditiveExpression[116] `-` *MultiplicativeExpression*[114]

All the operators are native. The operands must have the same type which is the type of the expression.

If the operands are of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand flows.

The string concatenation operator `+`. The operands must have the type String.

The result is a reference to a newly created `String` object that is the concatenation of the two operand strings. The characters of the left-hand operand precede the characters of the right-hand operand in the newly created string.

The addition operator `+`. The operands must have the same primitive numeric types or the same primitive numeric flow types.

The binary addition operator performs addition when applied to two operands of numeric type, producing the sum of the operands.

Addition is point-wise for vectors and matrices.

The subtraction operator . The operands must have the same primitive numeric types or the same primitive numeric flow types.

The binary subtraction operator performs subtraction, producing the difference of two numeric operands.

Subtraction is point-wise for vectors and matrices.

16.19 Shift Operators

The shift operators include `<<`, `>>`, and `>>>`. They group left-to-right. The left-hand operand of a shift operator is the value to be shifted; the right-hand operand specifies the shift distance.

ShiftExpression:
AdditiveExpression[116]

ShiftExpression^{116} `<<` *AdditiveExpression*^{116}
ShiftExpression^{116} `>>` *AdditiveExpression*^{116}
ShiftExpression^{116} `>>>` *AdditiveExpression*^{116}

The arguments of shift operators must be of primitive integral type or of primitive integral flow type. All the shift operators are native.

If the operands are of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand flows.

The left shift operator `<<`. The value of n`<<`s is n left-shifted s bit positions.

The signed right shift operator `>>`. The value of n`<<`s is n right-shifted s bit positions with sign-extension.

The unsigned right shift operator `>>>`. The value of n`<<`s is n right-shifted s bit positions with zero-extension.

16.20 Relational Operators

The relational operators are `<`, `>`, `<=`, and `>=`.

RelationalExpression:
ShiftExpression^{116}
RelationalExpression^{117} `<` *ShiftExpression*^{116}
RelationalExpression^{117} `>` *ShiftExpression*^{116}
RelationalExpression^{117} `<=` *ShiftExpression*^{116}
RelationalExpression^{117} `>=` *ShiftExpression*^{116}

All the relational operators are native. The type of a relational expression is `boolean` or a Boolean flow type. The operands must be of the same primitive numerical type or of the same primitive numerical flow type. In the latter case, the result has the same clock as the operands.

If the operands are of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand flows.

The smaller than operator `<`. The value of x`<`y is true if the value of x is less than the value of y. Otherwise is false.

The smaller than or equal operator `<=`. The value of x`<=`y is true if the value of x is less than or equal to the value of y. Otherwise the value is false.

The greater operator `>`. The value of x`>`y is true if the value of x is greater than the value of y. Otherwise is false.

The greater than or equal operator `>=`. The value of x`>`y is true if the value of x is greater than or equal to the value of y. Otherwise the value is false.

16.21 Equality Operators

The relational operators are `==`, and `!=`. The operators group left-to-righ t.

EqualityExpression:
    RelationalExpression[117]
    EqualityExpression[118] `==` *RelationalExpression*[117]
    EqualityExpression[118] `!=` *RelationalExpression*[117]

The type of a equality expression is always `boolean`. The equality operators may be used to compare two operands of the same type. a`!=`b always has the same result as !(a`==`b).

If the operands are of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand flows.

The numerical equality operators `==` and `!=`. The operators are native. Both the operands must be of the same primitive numeric type or of the same primitive numeric flow type.

The operator `==` yields the value `true` if the operands evaluate to the same value, otherwise to `false`.

The Boolean equality operators `==` and `!=`. Both the operands must be of type `boolean` or of the same Boolean flow type.

The operator `==` yields the value `true` if both the operands evaluate to either `true` or `false`, otherwise to `false`.

The reference equality operator == and !=. Both operands must have the same reference type.

The result of the reference equality operator `==` is `true` if the operand values are both null or both refer to the same object or array; otherwise, the result is false.

The result of the reference equality operator `!=` is `false` if the operand values are both null or both refer to the same object or array; otherwise, the result is false.

16.22 Bitwise and Logical Operators

The bitwise operators and logical operators include the *and* operator `&`, the *exclusive or* operator `^`, and *inclusive or* operator `|`. These operators have different precedence, with `&` having the highest precedence and `|` the lowest precedence. Each of these operators groups left-to-right. Each operator is commutative if the operand expressions have no side effects. Each operator is associativ e.

AndExpression:
EqualityExpression[118]
AndExpression[119] `&` *EqualityExpression*[118]

ExclusiveOrExpression:
AndExpression[119]
ExclusiveOrExpression[119] `^` *AndExpression*[119]

InclusiveOrExpression:
ExclusiveOrExpression[119]
InclusiveOrExpression[119] `|` *ExclusiveOrExpression*[119]

The operands must have the same type. The type of the result is `boolean` or, if the operands are of a flow type, a Boolean flow type with the the same clock as the operands have.

If the operands are of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand flows.

The integer bitwise operators &, ^, and |. The operators are native. The operands are of primitive integral type or of primitive integral flow type.

The result value is the bitwise and, exclusive or, resp. inclusive or of the operand values.

The Boolean logical operators `&`, `^`, and `|`. The operands are of type `boolean` or of Boolean flow type.

For `&`, the result value is `true` if both operand values are `true`; otherwise, the result is `false`.

For `^`, the result value is `true` if the operand values are different; otherwise, the result is `false`.

For `|`, the result value is `false` if both operand values are `false`; otherwise, the result is `true`.

16.23 Conditional-And Operator

The conditional-and operator groups left-to-right. It is fully associative.

ConditionalAndExpression:
InclusiveOrExpression[119]
ConditionalAndExpression[120] `&&` *InclusiveOrExpression*[119]

The operands must either be of type `boolean`, or must be of the same Boolean flow type. The result must have the same type as the operands.

The left-hand operand expression is evaluated first; if its value is `false`, the value of the conditional-and expression is `false` and the right-hand operand expression is not evaluated. If the value of the left-hand operand is `true`, then the right-hand expression is evaluated and its value becomes the value of the conditional-and expression.

If the operands are of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand f lows.

16.24 Conditional-Or Operator

The conditional-or operator groups left-to-right. It is fully associative.

ConditionalOrExpression:
ConditionalAndExpression[120]
ConditionalOrExpression[120] `||` *ConditionalAndExpression*[120]

The operands must either be of type `boolean`, or must be of the same Boolean flow type. The result must have the same type as the operands.

The left-hand operand expression is evaluated first; if its value is `true`, the value of the conditional-and expression is `true` and the right-hand operand expression is not evaluated. If the value of the left-hand operand is `false`, then the right-hand expression is evaluated and its value becomes the value of the conditional-and expression.

If the operands are of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand flows.

16.25 Conditional Expression

The conditional operator `...?...:...` uses the boolean value of one expression to decide which of two other expressions should be evaluated.

The conditional operator groups right-to-left.

Conditionalexpression:
ConditionalOrExpression[120]
ConditionalOrExpression[120] `?` *Expression*[122] `:` *Conditionalexpression*[121]

The first operand must be of type `boolean` or of a Boolean flow type. The other operands must have the same type which is the type of the result as well. If the first operand is of a Boolean flow types, the other operands must be of some flow type with the same clock.

The conditional operator may be used to choose between second and third operands. If the first operand evaluates to `true`, the result is obtained by evaluating the second operand. Otherwise the result is obtained by evaluating the third operand.

If the operands are of flow type the result is a newly created flow where the operator is instant-wise applied to all the values of the operand flows.

16.26 Arrow Operator

The arrow operator groups left to right. It is associative.

ArrowExpression:
Conditionalexpression[121]
ArrowExpression[121] `->` *ArrowExpression*[121]

Operands must be of the same flow type which is the type of the result too.

The result references a new flow which is defined as follows: at the *first* instant the clock evaluates to true, the new flow has the value of of the flow referenced by the first operand at that instant. At *later* instants at which the clock evaluates to true, the new flow has the value of of the flow referenced by the second operand at that instant. E.g.

d `->` d'	d_0			d'_1	d'_2			.
d	d_0			d_1	d_2			.
d'	d'_0			d'_1	d'_2			.
c_2	t	f		t	t		f	.
c_1	t	t	f	t	t	f	t	.
c_0	t	t	t	t	t	t	t	.

Remark. All the flows are presented in one flow diagram for convenience.

16.27 Expression

An *expression* is any assignment expression.

Expression:
ArrowExpression[121]

16.28 Constant Expression

ConstantExpression:
Expression[122]

A compile-time constant expression is an expression denoting a value of primitive type or a String that is composed using only the following:

- Literals of primitive type and literals of type String
- Casts to primitive types and casts to type `String`
- The unary operators `+`, `-`, , and `!`
- The multiplicative operators `*`, `/`, and `%`
- The additive operators `+` and `-`
- The shift operators `<<`, `>>`, and `>>>`
- The relational operators `<`, `<=`, `>`, and `>=`
- The equality operators `==` and `!=`
- The bitwise and logical operators `&`, `^`, and `|`
- The conditional-and operator `&&` and the conditional-or operator `||`
- The ternary conditional operator `...?...:...`

- Simple names that refer to final variables whose initializers are constant expressions
- Qualified names of the form $TypeName^{36}$. $Identifier^{16}$ that refer to final variables whose initializers are constant expressions

17 Specification of Temporal Properties

17.1 Temporal Propositions

Temporal propositions may be expressed using CTL (Computation Tree Logic) or LTL (Linear Time Logic). We use NuSMV as a prover. For details of the logical syntax or its semantics we refer to `http://nusmv.irst.itc.it/`. There is one difference though: temporal quatifiers have a colon as postfix, e.g. `AX:` instead of `AX`. This is necessary to avoid confusion with class names and identifiers when parsing.

PropositionSpecification:
`proposition` { *Propositions*[124] }
Propositions:
Proposition[124]
Proposition[124] *Propositions*[124]
Proposition:
CtlProposition[124]
LtlProposition[125]
Literal[17]

17.2 Computation Tree Logic

CtlProposition:
Label[16] `ctl` *CtlFormula*[124]
Label[16] `ctl` *CtlFormula*[124] `fair` *CtlFormula*[124]
CtlFormula:
`AX:` *CtlFormula*[124]
`AF:` *CtlFormula*[124]
`AG:` *CtlFormula*[124]
`EX:` *CtlFormula*[124]
`EF:` *CtlFormula*[124]
`EG:` *CtlFormula*[124]
`A:` *CtlFormula*[124] `until` *CtlFormula*[124]
`E:` *CtlFormula*[124] `until` *CtlFormula*[124]
`(` *CtlFormula*[124] `)`
`true`
`false`
`?`*Identifier*[16]

`$`*Identifier*[16]
`!` *CtlFormula*[124]
CtlFormula[124] `&` *CtlFormula*[124]
CtlFormula[124] `|` *CtlFormula*[124]
CtlFormula[124] `xor` *CtlFormula*[124]
CtlFormula[124] `->` *CtlFormula*[124]
CtlFormula[124] `<->` *CtlFormula*[124]

17.3 Linear Time Logic

The linear time logic of NuSMV is extended by Past operators.

LtlProposition:
Label[16] `LTL` *LtlFormula*[125]
LtlFormula:
`X:` *LtlFormula*[125]
`G:` *LtlFormula*[125]
`F:` *LtlFormula*[125]
LtlFormula[125] `:U:` *LtlFormula*[125]
LtlFormula[125] `:V:` *LtlFormula*[125]
`Y:` *LtlFormula*[125]
`Z:` *LtlFormula*[125]
`H:` *LtlFormula*[125]
`O:` *LtlFormula*[125]
LtlFormula[125] `:S:` *LtlFormula*[125]
LtlFormula[125] `:T:` *LtlFormula*[125]
`(` *LtlFormula*[125] `)`
`true`
`false`
Identifier[16]
`!` *LtlFormula*[125]
LtlFormula[125] `&` *LtlFormula*[125]
LtlFormula[125] `|` *LtlFormula*[125]
LtlFormula[125] `xor` *LtlFormula*[125]
LtlFormula[125] `->` *LtlFormula*[125]
LtlFormula[125] `<->` *LtlFormula*[125]

17.4 Verification Constraints

Constraints are means to restrict the model used for verification (cf. the language introduction, Chapter 8). There are three kinds of constraints:

- Constraints of the form *`signal`* `:>` n - the range of the integer valued signal *signal* is from 0 to n,
- Constraints of the form `@`*`signal`* `:>` n - the range of the number of instants since the *signal* was present the last time is from 0 to n, and
- temporal constraints to be explained below

The corresponding rule is

ConstraintSpecification:
Identifier[16] `:>` *Literal*[17]
`@`*Identifier*[16] `:>` *Literal*[17]
TemporalConstraint[126]

17.5 Temporal Constraints

The idea is to restrict the behaviour of sensors of an application or within an object. The constraints generate a model which runs in parallel with the original model generated from the code. This "constraint model" is the weakest model to satify all the temporal constraints. A generation of such model does not work for general CTL formulas, but for those using universal quantification only, and which restricts negation to propositional formula. This logic is refered to as ACTL. We further restrict the logic in that all constraints are of the form ϕ `->` ψ, where only sensors may occur within the formula ψ.

TemporalConstraint:
ACtlFormula[126] `->` *ACtlFormula*[126]
ACtlFormula:
PropositionalFormula[127]
`AX:` *ACtlFormula*[126]
`AF:` *ACtlFormula*[126]
`AG:` *ACtlFormula*[126]
`A:` *ACtlFormula*[126] `until` *ACtlFormula*[126]
`(` *ACtlFormula*[126] `)`

PropositionalFormula:

- `true`
- `false`
- `?`*Identifier*[16]
- `$`*Identifier*[16]
- `!` *ACtlFormula*[126]
- *ACtlFormula*[126] `&` *ACtlFormula*[126]
- *ACtlFormula*[126] `|` *ACtlFormula*[126]
- *ACtlFormula*[126] `xor` *ACtlFormula*[126]
- *ACtlFormula*[126] `->` *ACtlFormula*[126]
- *ACtlFormula*[126] `<->` *ACtlFormula*[126]

References

[1] C. André. Representation and analysis of reactive behaviours: A synchronous approach, in: *Proc. CESA'96*, Lille, France, July 1996.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] R. Budde, A. Poigné, K.H. Sylla. *synERJY: Introduction to the Language*, `http://www.ais.fraunhofer.de/~ap/synERJY`, 2003 -2006

[4] R. Budde, A. Poigné, K.H. Sylla. *synERJY: User Manual*, `http://www.ais.fraunhofer.de/~ap/synERJY`, 2003 - 2006

[5] J. Gosling, B. Joy, G. Steele. *The* JAVA^{TM} *Language Specification*, Addison-Wesley, Nov. 1999

[6] N. Halbwachs. *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Dordrecht, 1993.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[8] D. Harel. Statecharts: A visual approach to complex systems, *Science of Computer Programming*, 8:231–274, 1987.

[9] P. Le Guernic, A. Benveniste, P. Bournaii, T. Gautier, Signal: a dataflow oriented language for signal processing, *IEEE-ASSP*, 34(2), 1986.