Syntax and Semantics of PLCspecif

Dániel Darvas\textsuperscript{1,2}, Enrique Blanco Viñuela\textsuperscript{1}, István Majzik\textsuperscript{2}

Abstract

This technical report overviews PLCspecif, a formal complete behaviour specification method for PLC programs. The document provides details about the syntax and the semantics of PLCspecif.

Keywords: PLCspecif, formal specification, PLC, formal semantics, abstract syntax

\textsuperscript{1}CERN (EN/ICE), Geneva, Switzerland. E-mails: \{ddarvas, eblanco\}@cern.ch.
\textsuperscript{2}Budapest University of Technology and Economics, Department of Measurement and Information Systems, Budapest, Hungary. E-mails: \{darvas, majzik\}@mit.bme.hu.

This is an internal CERN publication and does not necessarily reflect the views of the CERN management.
## Contents

1 Introduction 3

2 Abstract Syntax 3
   2.1 Module Structure 3
   2.2 State Machine Modules 7
   2.3 Input-Output Connections 9
   2.4 Timer Modules 11
   2.5 Expressions 12

3 Concrete Syntax 16
   3.1 Module Structure 16
   3.2 State Machine Module 16
   3.3 Input-Output Connection Module 17
   3.4 Timers 17
   3.5 Expressions 17

4 Well-formedness Constraints 18
   4.1 Module Structure 18
   4.2 State Machines 20
   4.3 Input-Output Connection Modules 20
   4.4 Expressions 21

5 Informal Operational Semantics 23
   5.1 Definitions 23
   5.2 Execution of a Composite Module $m$ 23
   5.3 Execution of an Alternative Module $m$ 23
   5.4 Execution of a State Machine Module $m$ 24
   5.5 Execution of an Input-Output Connection Module $m$ 24
   5.6 Execution of an Empty Module $m$ 25
   5.7 Execution of a Timer Module $m$ 26

6 Formal Semantics 26
   6.1 Timed Automata 26
   6.2 Translation Algorithms 28
6.3 Utility Functions ......................................................... 43
6.4 Expressions .............................................................. 45

7 Examples ........................................................................ 47

7.1 R-S Flip-Flop .............................................................. 47
7.2 OnOff Module .............................................................. 47
1 Introduction

This technical report is dedicated to the detailed definition of the PLCspecif. The goal of PLCspecif is to provide a specification language for PLC modules, i.e. reusable, composable base components, each of them typically represented by one or some PLC functions or function blocks. PLCspecif is a complete formal behaviour specification language for PLC programs developed at CERN.

The structure of this report follows the general structure of defining models. First the abstract syntax is given (Section 2) then the concrete syntax (Section 3) and the necessary well-formedness constraints (Section 4). After, the semantics is described first in an informal way (Section 5) later in a formal way (Section 6). The report ends with some examples (Section 7).

Symbols and Abbreviations

0..1 Zero or one.
0..* Zero or more.
1..* One or more.

2 Abstract Syntax

This section overviews the abstract syntax of the PLCspecif formalism. The abstract syntax was designed as an Ecore metamodel, therefore the figures in this section are following the common EMF (Eclipse Modeling Framework) denotation. The concrete classes are represented by yellow boxes, while the abstract classes are shown as grey boxes. All objects have to be organised in a tree structure, the containment relations are defined using the composition arrows known from UML. If a collection is ordered, its name is prefixed by “O_”. As the abstract syntax is big, it is presented in parts. Each part is a different view of the abstract syntax model. The white boxes are representing classes semantically belonging to a different view of the model, but necessary to be shown to present some connections.

2.1 Module Structure

A PLCspecif specification is organized into modules. On a high level, each specification is a module. Each module is either a concrete representation of a behaviour (leaf module) or a composite module containing a set of submodules. The overview of the module structure can be seen in Figure 1.

– The **Module** is the most general representation of a module. A **Module** can be a composite (**CompositeModule**), an alternative (**AlternativeModule**) or a leaf (single, atomic) module (**LeafModule**).

– The **System** is a top-level container that can contain 1..* modules (**modules**). Its **topLevelModule** shows which module represents the top level of the specification. The modules present in **modules**, but not included in the module hierarchy of **topLevelModule** are not taking part in the specification, but can be included later.

– The composite module (**CompositeModule**) represents a composite module that is composed by several submodules. It can contain both composite and leaf modules. The **O_submodules** collection of the submodules is an ordered list of submodules. The order of the items determines the evaluation order of the submodules.

The alternative module (AlternativeModule) selects one of the two contained module for execution based on the given Boolean expression condition. If the condition is true, the moduleOnTrue module will be executed, otherwise the moduleOnFalse module.

The leaf module (LeafModule) is an abstract representation of a module containing specification for an “atomic” module of the specified system. Currently three different types of leaf modules are defined: state machines (StatemachineModule), input-output connection modules (IoConnectionModule), timer modules (TimerModule), and empty modules (EmptyModule).

A state machine module (StatemachineModule) is a module that specifies a submodule of the specified system using a state machine. The state machine modules are discussed in detail in Section 2.2.

An input-output connection module (IoConnectionModule) is a module that specifies a submodule of the specified system by defining connections between input and output variables. This module is typically used for representing modules with data-flow-like behaviour. The input-output connection modules are discussed in detail in Section 2.3.

A timer module (TimerModule) is a module that represents a PLC timer in the specified system. Three concrete timer module types are defined (TpTimer, TonTimer, ToffTimer), in accordance with the IEC 61131 standard [3]. The timer modules are discussed in detail in Section 2.4.

An empty module (EmptyModule) is a leaf module without any state machine or input-output connection (i.e. it can only contain input and output definitions).

After the high-level overview of the modules, this section is continued by overviewing the common details of the modules. Figure 2 shows the main concepts of the module structure.
– As the PLC code itself, the modules are working with variables too. A (non-composite) variable is defined as a **VariableDeclaration**. It contains the following information:

  – The name of the variable (**name**).
  – The type of the variable (**type**).
  – The direction of the variable (**direction**). If the direction is **INPUT**, the variable represents an input variable that cannot be modified in any of the modules. If the direction is **OUTPUT**, the variable represents an output variable that must be assigned in each PLC cycle. Currently other directions are not defined. A variable with direction **OUTPUT** can serve as an input of a submodule which reads the variable value computed by another module.
  – If the variable is internal or not (**internal**). An internal variable is defined to simplify the specification, therefore it is not obligatory to represent as a variable in the PLC program. A non-internal variable is either a real input of the PLC program or an output that has to be assigned by the PLC program.
  – If the variable is a parameter or not (**parameter**). A parameter variable is a special input variable whose value is assumed to be constant during the whole execution.
  – The modules that are assigning value to the variable (**modulesWriting**). Each variable can only be assigned in one single leaf module. However, the **modulesWriting** will contain all parents of this assigning leaf module, because on a higher hierarchy level these composite modules are also assigning values to the same variable.
  – The modules that are reading the value of the variable (**modulesReading**).

– A type (**Type**) can be either a primitive type (**PrimitiveType**) or an enumeration type (**EnumType**).

– A **PrimitiveType** is a reference to one of the defined primitive types (**PrimitiveTypes**): **BOOL**, **INT16**, **INT32**, **FLOAT16**, **TIME32**.
  
  * **BOOL** is a 1-bit data type to represent logic values true or false.
  * **INT16** is a 16-bit data type to represent signed integer values in the range of \([-2^{15};2^{15} - 1]\).
Figure 3: Variables model

* **INT32** is a 32-bit data type to represent signed integer values in the range of \([-2^{31}, 2^{31} - 1]\).

* **FLOAT** is a 16-bit data type to represent signed floating-point numeric values. Its precision is implementation-dependent.

* **TIME32** is a 32-bit data type to represent signed time values in the range of \([-2^{31} \text{ ms}, 2^{31} - 1 \text{ ms}]\). Its precision is 1 ms. (In accordance with IEC 61131-3[3].)

  - The enumeration type (**EnumType**) has a name (**name**) and 1..* enumeration literals (**literals**). An enumeration literal (**EnumLiteral**) describes a string and an integer value. See Figure 3 for details.

  - Each module has a set of input and output variables to handle (**handledInputVariables** and **handledOutputVariables**). The module can read any of the **handledInputVariables** and the module must assign a value to all variables in **handledOutputVariables**. In case of a composite module, the input reading and output writing will be performed by the submodules.

  - For each handled input variable of a composite module, there can be zero, one, or many submodules that handle this input variable. It is suggested to have exactly one submodule handling each input variable.

  - For each handled output variable of a composite module, there should be exactly one leaf module that handles this output variable. The predecessor composite modules of this leaf module can also be contained in the **handledOutputVariables**. It is not allowed to assign value to a variable in multiple leaf modules, except if they are in different branches of alternative modules (thus they cannot be executed at the same time).

  - Each module can contain input definitions. This is a set of expressions (**VariableDefinitionExpression**) connected to variables defining their values (**0_inputDefinitions**). The variable should be an internal variable (**internal=true**) and should be defined in the module where its **VariableDefinitionExpression** is contained. It behaves like a "local variable" that is only visible to the module defining it and to its submodules. At the beginning of the execution of the defining module, the value of the "local variable" is computed using the defined expression. After that, the
value does not change until the next execution. The list of input definitions is ordered
and the order determines the evaluation order of the named input expressions.

- Each module can contain output definitions. This is a set expressions connected to vari-
  ables defining their values (\texttt{outputDefinitions}), defining an assignment rule for a
  handled output variable. The list of output definitions is ordered and the order determines
  the evaluation order of the named output expressions.

- Each leaf module can define event inputs (events). An event input\footnote{Also called “event” where it cannot cause confusion.} (Event) is a special
  Boolean internal variable. Its definition may contain handled input variables and internal
  variables assigned in the input definitions section in the module or in one of its parent
  modules. An event input is \textit{enabled} if the defining Boolean expression is evaluated to true
  at the evaluation of the defining module. (The evaluation orders are clarified later.) An
  event input \textit{triggers} if it is enabled and if there is no event enabled with higher priority
  in the leaf module. A leaf module cannot contain multiple events with the same priority
  number. The higher priority number means lower priority and the priority numbers are
  non-negative, thus the highest priority is expressed by 0. Events cannot be defined in
  composite modules.

Currently event inputs cannot be used for I/O connection modules.

- To avoid the “spaghetti constructions”, a leaf module cannot have output variables
  (output) as inputs by default. However, these connections can be explicitly allowed by
  “allowed submodule connections” (\texttt{AllowedSubmoduleConnection}). It refers to some
  output variables (\texttt{readVariables}) that can be read by another module. The reader mod-
  ule is also defined (\texttt{readerModule}). These allowed connections should be defined in a
  composite module that contains all involved submodules.

  - The \texttt{variableState} of the \texttt{AllowedSubmoduleConnection} defines what state of the
    variable should be read. If the \texttt{variableState} is \texttt{FRESH}, the read value should be
    assigned to the variable in the same execution cycle, i.e. the module assigning the
    read variables should be executed before the execution of the reading module. If the
    \texttt{variableState} is \texttt{STALE}, then the execution order of the reading and the assigning
    module should be the opposite.

2.2 State Machine Modules

One of the possibilities to capture the behaviour of a leaf module in the PLCspecif formalism is

to use state machines. Figure\cite{fig:StateMachine} shows the main concepts of the state machines.

- A leaf module containing a state machine is defined by a \texttt{StatemachineModule}. It
  contains the following information:

  - The transitions defined in the state machine (\texttt{transitions}),
  - The root composite state that (hierarchically) contains all the states defined in the
    state machine (\texttt{rootState}),
  - The initial state of the state machine (\texttt{InitialState}).

  The abstract state (\texttt{AbstractState}) is a general concept of states. An abstract state
  is either a composite state (\texttt{CompositeState}) refined by other states, or a single state
  that is not further refined (\texttt{SingleState}). For each abstract state the \texttt{containerState}
reference stores the CompositeState object that contains the given abstract state, except for the root state of the state machine. For the root state of the state machine, the containerState is undefined and the parentModule contains a reference to the containing StatemachineModule.

- A composite state (CompositeState) represents a state that is refined by several states. The states contained in the composite state are defined in containedStates. Any abstract state can be contained in at most one composite state, therefore the states have a tree structure in a state machine module, where the root of the tree is the root state.

- A single state (SingleState) represents a state that cannot be further decomposed into multiple states. A single state is either a basic state (BasicState) or a pseudo state, e.g. a state representing history.

- A basic state (BasicState) is a single state of the state machine that is not further refined. The initial state of a state machine can only be a BasicState, meaning that the initial state is this referenced state and its container composite states.

- A deep history state (DeepHistoryState) represents a pseudo state that stores a state configuration of its container composite state. A deep history state should not have any outgoing transition. The defaultState reference defines which state (BasicState) is stored by default. This state should be in the same composite state where the history state is.

- Note: shallow history state is not defined, its usage is not permitted.

- A transition (Transition) represents a potential state change in the state machine. It contains the following information:
  - The source state (from) that can be any state,
  - The target state (to) that can only be a single state (basic or history state),

Figure 4: State machine structure
– A guard (guard) that is a Boolean expression. If this Boolean expression is evaluated to false, the transition is not enabled. It is optional to have a guard attached to a transition. Having no guard defined is equivalent to a guard having the constant “true” expression.

– An event (event). If an event is attached to a transition, it can only fire if the connected event triggers. It is optional to have an event attached to a transition. A transition is called event-triggered, if its event reference is set to an event input. In other cases (i.e. if event is null), the transition is non-event-triggered.

**Differences to the UML State Machines / Harel’s Statecharts**

It is not a goal to define the syntax or semantics of the current state machine formalism based on the UML State Machines or Harel’s Statecharts. However, because the similarities are important, it is useful to emphasize the differences between these formalisms. In the state machine formalism defined above:

– Parallel regions are not allowed.

– In UML State Machines, all composite states should contain an “initial” pseudostate to define the initial state configuration. In PLCspecif, the initial state configuration is defined globally.

– The initial state of the whole state machine should be a basic (non-composite) state.

– Actions (e.g. entry, exit, …) are not allowed. Therefore it is not possible to define variable assignments inside the state machines. These are decoupled into a separate output definition block.

– At every moment, exactly one basic state can be active in each state machine.

– The usage of shallow history states is not permitted.

**2.3 Input-Output Connections**

State machines are suitable for modules that are stateful and the state to be stored can conveniently represented by a handful of states in the specification. However, if the state can only be described by some integers or real numbers (e.g. storing previous measurements or value requests), state machines are inappropriate and we suggest the usage of input-output connection modules. The idea of the input-output connection module is inspired by Function Block Diagrams (FBDs) and similar data-flow-like formalisms. It graphically defines how the outputs of the module should be assigned based on the current inputs and outputs from the previous cycles. Figure 5 shows the main concepts of the input-output connection modules.

– An input-output connection module (IoConnectionModule) represents the connections between the input and output variables. It has to assign values to each handled output variables exactly once, using the values of the handled input variables and the locally defined internal input variables. The IoConnectionModule consists of pins, data edges and block instances that are explained in the following.

– A block definition (BlockDefinition) represents a block that can be instantiated in an IoConnectionModule.
Figure 5: Input-output connection module structure

- Two types of block definition exist: built-in block definition and custom block definition.

- The built-in block definition (`BuiltInBlockDefinition`) represents a block that is part of the development environment, the built-in libraries or the operating system. For example, the system function blocks (SFB) and system functions (SFC) of the Siemens PLCs can be represented as built-in block definitions. The implementation of the built-in block definitions are not known, only their interface. The `BuiltInBlockDefinition.blockRef` textual field defines which built-in block is represented by this `BuiltInBlockDefinition`. The format of this field is implementation- and manufacturer-specific.

- The custom block definition (`CustomBlockDefinition`) represents a block of which interface and implementation are both known. For example, a comparison block ("out = (in1 < in2)") or a scaling block ("out = (in1-122)*(111-112)/(121-122)+112") is represented by a custom block definition. To define the outputs, the block contains exactly one block output assignment (`BlockOutputAssignment`) for each output in `O.outputDefinitions`. A block output assignment defines an expression (`expression`) and refers to the output that is defined (`outputPin`). The collection `O.outputDefinitions` is ordered and this order determines the order of execution.

- Every block definition contains exactly one input or output block pin definition (`BlockInputPinDefinition` or `BlockOutputPinDefinition`) for each input and output. The definitions of the input pins are stored in `BlockDefinition.inputs`, the output pins are stored in `BlockDefinition.outputs`. A block pin definition describes the name and the type of the input/output. The block pin definitions in `inputs` and `outputs` define the interface of the block defined by the `BlockDefinition`. The `BlockOutputAssignments` of the `CustomBlockDefinition` are also connected to the corresponding `BlockOutputPinDefinitions`. The evaluation order of the outputs is determined by the order in `CustomBlockDefinition.O.outputDefinitions`. 
– The I-O connection module contains instances of blocks (blockInstances). Each block instance is an instance of a block defined by a BlockDefinition.

– The block instance contains a set of input pins (0_inputs) and output pins (0_outputs). The “pins” are defined in the following.

– The module contains pins (Pin) that are either data producer (ProducerPin) or data consumer (ConsumerPin) “locations” (basically a pin is an entity that can be the source or the target of a data edge). A data producer pin (ProducerPin) can be an input variable (InputVariablePin), a constant (ConstantPin) or the output pin of a block instance (BlockOutputPin). A data consumer pin (ConsumerPin) can be an output variable to be assigned by the module (OutputVariablePin) or the input pin of a block instance (BlockInputPin).

The type of the pin is determined by the constant, variable or block that it represents. The type of the pin is given by the getType operation.

– The type of a ConstantPin is determined by ConstantPin.constant.type.

– The type of an InputVariablePin is determined by InputVariablePin.variable.type.

– The type of an OutputVariablePin is determined by OutputVariablePin.variable.type.

– The type of a BlockInputPin is determined by BlockInputPin.pinDefinition.type.

– The type of a BlockOutputPin is determined by BlockOutputPin.pinDefinition.type.

– The module contains data edges (DataEdge) that are the definitions of connections between “pins”. The data edges can be considered as “typed wires”. A data edge connects exactly one data producer pin (source) with several data consumer pins (targets). The type of the data edge, the producer pin and the consumer pins should match.

2.4 Timer Modules

– A timer module (TimerModule) is a module that represents a PLC timer in the specified system. Three concrete timer module types are defined (TpTimer, TonTimer, ToffTimer), in accordance with the IEC 61131 standard [3]. Figure 6 shows the metamodel of the timer modules.

– Each timer module can have 0..* defined events to reset the timer (resetTriggers). If any of these events triggers, the timer stops (see the semantics definition for details).

– A delay timer (DelayTimer that is either a TonTimer or a ToffTimer) has exactly one Boolean expression (expressionToDelay) whose value will be delayed (will be used as the input of the timer module), in accordance with the semantics of the TON and TOFF timers.

– A TpTimer can have 0..* set events (setTriggers). If any of these events trigger, the timer starts (see the semantics definition for details).
2.5 Expressions

This section describes the expression system that is used in the specification language. The high-level concepts can be seen in Figure 7.

Definition 1 (Dynamic type) The dynamic type of an expression is the type of its evaluated value.

For example, the dynamic type of a comparison is always Boolean, while the dynamic type of an addition depends on its arguments.

Definition 2 (Comparable types) Two types are comparable if they can be compared to each other. For PLCspecif it is defined as follows:

<table>
<thead>
<tr>
<th></th>
<th>BOOL</th>
<th>INT16</th>
<th>INT32</th>
<th>FLOAT16</th>
<th>TIME32</th>
<th>enum e</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>+</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>INT16</td>
<td>–</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>INT32</td>
<td>–</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>FLOAT16</td>
<td>–</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>TIME32</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>+</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Definition 3 (Promoted type) The promoted type of two given types is a type that can contain both type’s value range.

For PLCspecif the promoted types are defined as follows (– meaning that no promoted type exists):

<table>
<thead>
<tr>
<th></th>
<th>BOOL</th>
<th>INT16</th>
<th>INT32</th>
<th>FLOAT16</th>
<th>TIME32</th>
<th>enum e</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>BOOL</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>INT16</td>
<td>–</td>
<td>INT16</td>
<td>INT32</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>INT32</td>
<td>–</td>
<td>INT32</td>
<td>INT32</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>FLOAT16</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>FLOAT16</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>TIME32</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>TIME32</td>
<td>–</td>
</tr>
<tr>
<td>enum f</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>e, if e = f</td>
</tr>
</tbody>
</table>

Definition 4 (Compatible types) Two types are compatible if there exists a promoted type for them (see Definition 3).
Every expression is an **Expression** instance. This class represents any expression-like object. There are 4 main subtypes of the **Expression** class:

- A constant (**Constant**) is an unmodifiable, unnamed, typed value.
- The value reference (**ValueRef**) is a general concept of all the possible values that can be atomic elements of the expressions (leaves of the expression tree) except constants. See Figure 8 for the metamodel of value references.
  - A variable reference (**VariableRef**) is a reference to a variable (**VariableDeclaration**) in the expression tree.
  - A producer pin reference (**ProducerPinRef**) is a reference to a producer pin (**ProducerPin**) of an input-output connection module in the expression tree. Only for use in input-output connection modules.
  - A state reference (**InState**) is an expression in a state machine module that is true iff the referenced state (**AbstractState**) is active at the moment of evaluating the value of the **InState** instance. Only for use in state machine modules.
  - A timer output reference (**TimerOutputRef**) is a reference to the Boolean output of the timer module in which it is used. Only for use in timer modules.
  - A block input pin definition reference (**BlockInputPinDefRef**) is a reference to a block input pin definition to define the outputs of a block (not a block instance!). Only for use in block definitions.
- An operation (**Operation**) is a logic or arithmetic expression containing one or multiple values and operations. See Figure 9 for the metamodel of operations.

The following operation subtypes are defined:

- A logic operation (**LogicOperation**) is an n-ary operation that is evaluated to a Boolean value. It can have 1..* arguments (**arguments**) and a logic operator (**operator : LogicOperator**) that can be \( \lor, \land, \rightarrow, \text{XOR} \). If a logic operation has only one argument, then it is always evaluated to the value of this argument.
- A comparison (**Comparison**) is a binary operation that compares two values and can be evaluated to a Boolean value. It has exactly 2 arguments (**leftArgument**, **rightArgument**) and a comparison operator (**operator : ComparisonOperator**) that can be \( =, \neq, <, >, \leq, \geq \).
- An arithmetic operation (**ArithmeticOperation**) is a binary operation that represents an arithmetic operation. Its dynamic type depends on its arguments. It has exactly 2 arguments (**leftArgument**, **rightArgument**) and an arithmetic operator (**operator : BinaryArithmeticOperator**) that can be \( +, -, \times, \div \).
- A negation (**Negate**) is an unary operation that negates a Boolean value and can be evaluated to a Boolean value. It has exactly 1 argument (**argument**).
- A rising edge (**RisingEdge**) is an unary operation that is evaluated to true if and only if its expression (**expression**) is evaluated to true, but it was evaluated to false at the last evaluation.
- A falling edge (**FallingEdge**) is an unary operation that is evaluated to true if and only if its expression (**expression**) is evaluated to false, but it was evaluated to true at the last evaluation.
Figure 7: Overview of the expressions

- An if-then-else operation (**IteOperation**) is a ternary operation. Based on a condition (**condition**) it returns either the value of **onTrue** or the value of **onFalse**. Its dynamic type depends on its **onTrue** and **onFalse** arguments. The type of the **onTrue** and **onFalse** arguments should be compatible. The dynamic type of the expression is the promoted type for the two arguments.

- An element-of operation (**ElementOfOperation**) is a logic operation. It is evaluated to true iff the value of the given expression (**expression**) is one of the constants in **values**. Its dynamic type is the dynamic type of **expression**. The type of each item in **values** should be the same as the dynamic type of **expression**.

- An expression in disjunctive normal form (**DnfExpression**) represents an AND/OR table. It consists of one or many **DnfClause**. Each **DnfClause** consists of a logic expression (**expression**) describing one column of an AND/OR table. The operator of **expression** should be **AND**. The expression is evaluated to a Boolean value.

- A switch-case table (**SwitchCaseTable**) provides a representation for complex expressions. It consists of 1..* rows (**rows**). Each row (**SwitchCaseRow**) contains a logic condition (**condition**) and a value expression (**value**). If the logic condition of a row **r** is evaluated to true, the switch-case table will be evaluated to the value given in this row **r**. The conditions of the rows must be mutually exclusive and in any case at least one of them should be evaluated to true (in other words: always exactly one row’s condition should be evaluated to true). The operator of each expression should be **AND**. The type of the **value** for each row should be compatible. The dynamic type of the expression is the promoted type of the **values** for each row.

- An uninterpreted expression (**UninterpretedExpression**) is an expression that cannot be described using the types explained before. For example, it can hold manufacturer-specific expressions. Currently this type is only used for creating example models for development purposes.

- It is possible to build temporal logic expressions using **Expressions**, however it is depends on the further usage, therefore it is not defined here.
Figure 8: Value references in the expression metamodel

Figure 9: Operations in the expression metamodel
3 Concrete Syntax

This section presents some high-level concepts for the concrete syntax of PLCspecif. At this stage of the project, it is not a goal to define a precise concrete syntax for each element. Instead, we focus on the key points in order to be able to present examples.

3.1 Module Structure

The structure of a module is represented by a table such as the one in Figure 10. The representation of the core logic part depends on the concrete type of the module.

<table>
<thead>
<tr>
<th>Element Description</th>
<th>Concrete Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module.name</td>
<td></td>
</tr>
<tr>
<td>Module.description</td>
<td></td>
</tr>
<tr>
<td>Assigned inputs:</td>
<td></td>
</tr>
<tr>
<td>• Module.handledInputs</td>
<td></td>
</tr>
<tr>
<td>Assigned outputs:</td>
<td></td>
</tr>
<tr>
<td>• Module.handledOutputs</td>
<td></td>
</tr>
<tr>
<td>Input definitions:</td>
<td></td>
</tr>
<tr>
<td>• Module.O.inputDefinitions</td>
<td></td>
</tr>
<tr>
<td>Event definitions:</td>
<td></td>
</tr>
<tr>
<td>• Module.events</td>
<td></td>
</tr>
<tr>
<td>Core logic</td>
<td></td>
</tr>
<tr>
<td>Output definitions:</td>
<td></td>
</tr>
<tr>
<td>• Module.O.outputDefinitions</td>
<td></td>
</tr>
<tr>
<td>Invariant properties:</td>
<td></td>
</tr>
<tr>
<td>• Module.invariantProperties</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: Concrete syntax of a module

3.2 State Machine Module

The main elements of a state machine module are represented by the following symbols.

<table>
<thead>
<tr>
<th>Element Description</th>
<th>Concrete Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicState s (not initial)</td>
<td>S</td>
</tr>
<tr>
<td>BasicState s (initial)</td>
<td>S</td>
</tr>
<tr>
<td>CompositeState s (with substates s1, s2)</td>
<td>S1  S2</td>
</tr>
<tr>
<td>Transition (with trigger ev and guard g)</td>
<td>@ev [g]</td>
</tr>
<tr>
<td>DeepHistoryState</td>
<td>H*</td>
</tr>
</tbody>
</table>
3.3 Input-Output Connection Module
The main elements of an input-output connection module are represented by the following symbols.

<table>
<thead>
<tr>
<th>Element</th>
<th>Concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputVariablePin or ConstantPin $v$</td>
<td>$v$</td>
</tr>
<tr>
<td>OutputVariablePin $v$</td>
<td>$v$</td>
</tr>
<tr>
<td>DateEdge (type=BOOL)</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>DateEdge (type $\neq$ BOOL)</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>BlockInstance (name=$v$, inputs: $in1$, $in2$, outputs: $out$)</td>
<td>$\begin{array}{c} \text{in1} \ \text{in2} \end{array}$ $\begin{array}{c} V \ \text{out} \end{array}$</td>
</tr>
</tbody>
</table>

3.4 Timers
The timer modules are represented using simple tables, such as the one in Figure 11:

**TON timer**
- Expression to delay: $\langle\text{TonTimer.expressionToDelay}\rangle$
- Delay length: $\langle\text{TonTimer.delayTime}\rangle$
- Reset events: $\langle\text{TonTimer.resetTriggers}\rangle$

Figure 11: Concrete syntax of a timer module

3.5 Expressions
The concrete syntax of the expressions follows their common representation. Here we focus on the syntax of the particular operations that are not common in arithmetics.

- Rising edge $r$: $\text{rising_edge}(\langle r.\text{argument} \rangle)$
- Falling edge $f$: $\text{falling_edge}(\langle f.\text{argument} \rangle)$
- If-then-else operation $i$: $\langle i.\text{condition} \rangle ? \langle i.onTrue \rangle : \langle i.onFalse \rangle$

**AND/OR Tables**
The AND/OR tables (or DNF expressions) are represented by a table in the following way:

<table>
<thead>
<tr>
<th>$v_1$</th>
<th>$\alpha_{1,1}$</th>
<th>$\ldots$</th>
<th>$\alpha_{1,m}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_2$</td>
<td>$\alpha_{2,1}$</td>
<td>$\ldots$</td>
<td>$\alpha_{2,m}$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\ddots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$v_n$</td>
<td>$\alpha_{n,1}$</td>
<td>$\ldots$</td>
<td>$\alpha_{n,m}$</td>
</tr>
</tbody>
</table>
Here each \( v_i \) represents a variable (or expression), and each \( \alpha_{i,j} \) is a value in the value range of \( v_i \) or the “don’t care” symbol “\( \cdot \)”. Let us denote the current value of variable \( v_i \) by \( \llbracket v_i \rrbracket \). The meaning of the AND/OR table is the following Boolean expression:

\[
\bigvee_{j=1}^{m} \left( \bigwedge_{i=1}^{n} \left( (\alpha_{i,j} = \cdot) \lor \left( \llbracket v_i \rrbracket = \alpha_{i,j} \right) \right) \right)
\]

**Switch-Case Tables**

The switch-case table expressions are represented by a table in the following way:

<table>
<thead>
<tr>
<th>( P_1 )</th>
<th>( \ldots )</th>
<th>( P_m )</th>
<th>( \text{Value} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha_{1,1} )</td>
<td>( \ldots )</td>
<td>( \alpha_{1,m} )</td>
<td>( \text{val}_1 )</td>
</tr>
<tr>
<td>( \alpha_{2,1} )</td>
<td>( \ldots )</td>
<td>( \alpha_{2,m} )</td>
<td>( \text{val}_2 )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \ldots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( \alpha_{n,1} )</td>
<td>( \ldots )</td>
<td>( \alpha_{n,m} )</td>
<td>( \text{val}_n )</td>
</tr>
</tbody>
</table>

Here each \( P_i \) represents a variable (or expression), and each \( \alpha_{i,j} \) is a value in the value range of \( v_j \) or the “don’t care” symbol “\( \cdot \)”. The expression described by the switch-case table is defined below.

Let \( \gamma_i \) denote the satisfaction of the conditions of row \( i \) for \( i = 1, \ldots, n \):

\[
\gamma_i = \bigwedge_{j=1}^{m} \left( (\alpha_{i,j} = \cdot) \lor (P_j = \alpha_{i,j}) \right)
\]

Then if \( \gamma_i \) is true, the current value of the switch-case table is \( \text{val}_i \).

We require that the conditions for each row (including the don’t care symbols) is mutually exclusive (\( \forall i, j \in \{1, \ldots, n\} : i \neq j \Rightarrow (\gamma_i \land \gamma_j) \)) and that the row conditions together cover all possible cases (\( (\bigvee_{i=1}^{n} \gamma_i) = \text{true} \)).

### 4 Well-formedness Constraints

A valid model should be an instance of the metamodel introduced in Section 2 but also it should satisfy the well-formedness rules of PLCspecif. The well-formedness rules that are not implicitly included in the previous Ecore metamodels are described in this section.

#### 4.1 Module Structure

**WF-M-1** The `VariableDeclaration.name` is unique, i.e. there is no other `VariableDeclaration` with the same name that is defined in the same module or in one of the predecessors or ancestors.

**WF-M-2** If the `VariableDeclaration.direction` is `INPUT`, the `VariableDeclaration.modulesWriting` should be empty.

**WF-M-3** If the `VariableDeclaration.direction` is `OUTPUT`, the `VariableDeclaration.modulesWriting` should contain at least one module. It can contain also the ancestors of the writing module.
WF-M-4 If VariableDeclaration.internal=false, then the variable should be defined in the topmost module (i.e. that has no ancestors).

WF-M-5 Internal input variables should be defined in their defining modules. If for a VariableDeclaration v v.internal=true and v.direction=INPUT, there should be exactly one VariableDefinitionExpression in the O_inputDefinitions of the module defining v where variable=v.

WF-M-6 Only the internal input variables’ values can be defined.

The Module.O_inputDefinitions can only contain VariableDefinitionExpressions with variable v where v.direction=INPUT and v.internal=true.

WF-M-7 Output definitions can only define output variables.

The Module.O_outputDefinitions can only contain VariableDefinitionExpressions with variable v where v.direction=OUTPUT.

WF-M-8 The Module.handledOutputVariables can only contain variables with direction=OUTPUT. (Note: the handledInputVariables can contain both INPUT and OUTPUT variables.)

WF-M-9 The Event.triggerExpression should only contain ValueRef elements of type VariableRef elements. The VariableRef can only refer to variables that are in the handledInputVariables or definedVariables (only for internal input variables) of the module containing the event.

WF-M-10 — (deleted)

WF-M-11 The Module.handledOutputVariables can only contain VariableDeclarations with direction=OUTPUT.

WF-M-12 Only necessary allowed submodule connections are permitted.

For each AllowedSubmoduleConnection asc the following should be true: asc.readerModule.handledInputVariables contains each item in asc.readVariable.

WF-M-13 A variable with direction=OUTPUT can only be contained in Module.handledInputVariables if there is an AllowedSubmoduleConnection for that module (as readerModule) and variable (as readVariable).

WF-M-14 In every collection LeafModule.events the event’s priority should be a unique, non-negative number.

WF-M-15 — (merged with WF-M-6)

WF-M-16 If there exists an AllowedSubmoduleConnection asc with asc.variableState=FRESH and asc.readVariable=v1,v2,... and asc.readerModule=r then the modules assigning values to variables v1,v2,... should all be executed before the execution of module r.

WF-M-17 If there exists an AllowedSubmoduleConnection with variableState=STALE and readVariable=v1,v2,... and readerModule=r then the modules assigning values to variables v1,v2,... should all be executed after the execution of module r.

WF-M-18 If a variable v is a parameter (parameter=true), then it should be input variable (direction=INPUT).
WF-M-19 The AlternativeModule.condition should only contain VariableRef elements referring to parameter variables (parameter=true).

WF-M-20 The AlternativeModule.condition should have a Boolean dynamic type.

4.2 State Machines

WF-SM-1 The expression Transition.guard should have Boolean dynamic type.

WF-SM-2 The expression Transition.guard should contain only ValueRef elements of type VariableRef.

WF-SM-3 — (deleted)

WF-SM-4 — (deleted)

WF-SM-5 The source of a Transition cannot be a PseudoState.

WF-SM-6 In each run-to-completion step, only finite number of transitions can fire, i.e. it is impossible to have an infinite run-to-completion step.

WF-SM-7 For each active BasicState all non-event-triggered outgoing Transitions should be mutually exclusive, i.e. it is not possible to have nondeterminism in a run-to-completion step.

WF-SM-8 For each active BasicState all event-triggered outgoing Transitions should be mutually exclusive, i.e. there is no nondeterminism in the selection of firing event-triggered transition.

4.3 Input-Output Connection Modules

WF-IO-1 For each BlockInstance bi, for each BlockInputPinDefinition bpd in bi.definition.inputs, there should be exactly one BlockInputPin bip in bi.0.inputs where bip.pinDefinition=bpd.

WF-IO-2 The number of elements in BlockInstance.0.inputs should be the same as the number of elements in BlockInstance.definition.inputs.

WF-IO-3 For each BlockInstance bi, for each BlockOutputPinDefinition bpd in bi.definition.outputs, there should be exactly one BlockOutputPin bop in bi.0.outputs where bop.pinDefinition=bpd.

WF-IO-4 The number of elements in BlockInstance.0.outputs should be the same as the number of elements in BlockInstance.definition.outputs.

WF-IO-5 In a CustomBlockDefinition cbd for each BlockOutputPinDefinition bpd in cbd.outputs there should be exactly one BlockOutputAssignment boa in 0.outputDefinitions where boa.outputPin=bpd.

WF-IO-6 The number of elements in CustomBlockDefinition.0.outputDefinitions should be the same as the number of elements in CustomBlockDefinition.outputs.

WF-IO-7 For each variable v in IoConnectionModule.handledOutputVariables there should be exactly one OutputVariablePin ovp in one of the IoConnectionModule.blockInstances where ovp.variable=v.
WF-IO-8 The `BlockOutputAssignment.expression` can only contain `ValueRef` elements of type `BlockInputPinDefRef` and the `BlockInputPinDefRef.pin` should be in one of the `BlockDefinition`'s inputs collection.

WF-IO-9 No cyclic block dependence.

Considering an `IoConnectionModule` as a hypergraph with `Pins` as nodes and `DataEdges` as hypergraphs, the graph should be directed acyclic.

WF-IO-10 — (deleted)

WF-IO-11 For any `OutputVariablePin ovp` in the pins of an `IoConnectionModule icm`, `ovp.variable` should be in `icm.handledOutputVariables`.

WF-IO-12 For each `DataEdge de` the `de.type` should be the same as the type of the type of `de.source` and the type of all the `de.targets` elements. (The type of a `BlockInputPin` or `BlockOutputPin bp` is `bp.pinDefinition.type`. The type of an `OutputVariablePin ovp` is `ovp.variable.type`. The type of an `InputVariablePin ivp` is `ivp.variable.type`. The type of a `ConstantPin cp` is the type of the `cp.constant`.)

WF-IO-13 For each `ConsumerPin cp` exactly one `DataEdge de` should exists where `de.targets` contains `cp`. (Included also in the Ecore metamodel.)

4.4 Expressions

WF-EX-1 The expression of a `DnfClause` (`DnfClause.expression`) can only contain logical expression in form of “α ∧ β ∧ ...” where α, β,... are logical expressions not containing Boolean operators other than negation.

WF-EX-2 The `LogicOperation.arguments` can only contain expressions with `BOOL` dynamic type.

WF-EX-3 The `Comparison.leftArgument` and `Comparison.rightArgument` expressions should be comparable types.

WF-EX-4 The `ArithmeticOperation.leftArgument` and `ArithmeticOperation.rightArgument` should be compatible types.

WF-EX-5 The `Negate.argument` should be an expression with `BOOL` dynamic type.

WF-EX-6 The `IteOperation.condition` should be an expression with `BOOL` dynamic type.

WF-EX-7 The `IteOperation.onFalse` and `IteOperation.onTrue` arguments should be compatible types.

WF-EX-8 The expression tree of `BlockOutputAssignment.expression` can only contain `BlockInputPinDefRef` value references (i.e. the occurrence of `TimerOutputRef`, `VariableRef`, ... are forbidden).

WF-EX-9 The expression tree of `VariableDefinitionExpression.expression` cannot contain `BlockInputPinDefRef` value references.

WF-EX-10 The expression tree of `VariableDefinitionExpression.expression` cannot contain `ProducerPinRef` value references if the `VariableDefinitionExpression.variable` is not defined in an `IoConnectionModule`. 
WF-EX-11 The expression tree of VariableDefinitionExpression.expression cannot contain InState value references if the VariableDefinitionExpression.variable is not defined in an StatemachineModule.

WF-EX-12 The expression tree of TimerModule.delayTime can only contain VariableRef and TimerOutputRef value references (i.e. the occurrence of BlockInputPinDefRef, ... are forbidden).

WF-EX-13 The expression tree of AlternativeModule.condition can only contain VariableRef value references (i.e. the occurrence of BlockInputPinDefRef, TimerOutputRef, ... are forbidden).

WF-EX-14 The expression tree of Transition.guard can only contain VariableRef value references (i.e. the occurrence of BlockInputPinDefRef, TimerOutputRef, ... are forbidden). (See also WF-SM-2.)

WF-EX-15 — (deleted)

WF-EX-16 The expression tree of DelayTimer.expressionToDelay can only contain VariableRef value references (i.e. the occurrence of BlockInputPinDefRef, InStateRef, ... are forbidden).

WF-EX-17 The DnfClause.expression.operator should be AND.

WF-EX-18 The SwitchCaseRow.condition.operator should be AND.

WF-EX-19 For each constant with enumeration type, the Constant.value equals to one of the EnumLiteral.stringValue in the EnumTypeDef of the constant. (The constant’s value should be one of the corresponding enumeration’s literals.)

WF-EX-20 All variables referred in an Event.triggerExpression by a VariableRef should be contained in the container module’s handledInputVariables or definedVariables.

WF-EX-21 All variables referred in a Module.O.inputDefinitions.expression by a VariableRef should be contained in the container module’s handledInputVariables or definedVariables.

WF-EX-22 All variables referred in a Module.O.outputDefinitions.expression by a VariableRef should be contained in the container module’s handledInputVariables or definedVariables, or it has to be an internal variable set part of handledOutputVariables in one of the contained submodules.

WF-EX-23 All variables referred in a TimerModule.delayTime by a VariableRef should be contained in the container module’s handledInputVariables or definedVariables.

WF-EX-24 All variables referred in a DelayTimer.expressionToDelay by a VariableRef should be contained in the container module’s handledInputVariables or definedVariables.

WF-EX-25 The variable defined in a Module.O.inputDefinition should be contained in the container module’s definedVariables.

WF-EX-26 The variable defined in a Module.O.outputDefinition should be contained in the container module’s handledOutputVariables.
WF-EX-27 A variable $v$ defined in a Module.O_inputDefinition can only refer to internal input variables that are defined in an ancestor module or in the same module, but before the definition of $v$.

WF-EX-28 A variable $v$ defined in a Module.O_outputDefinition can only refer to internal output variables that are defined in the same module before the definition of $v$, or in a previously executed module.

WF-EX-29 In an ElementOfOperation $e$ the dynamic type of $e.expression$ should be the same as the values of each constant in $values$.

5 Informal Operational Semantics

To facilitate the understanding of the semantics of PLCspecif, we first discuss the semantics informally. Then the next section will provide the formal semantics.

5.1 Definitions

An event input $e_1$ has a higher priority than event input $e_2$, if the priority number assigned to $e_1$ is smaller than the priority number assigned to $e_2$. All priority numbers are non-negative integers. The highest possible priority is expressed by priority number 0.

An event input is enabled iff the corresponding trigger condition is evaluated to true. An event input is triggered if the event input is enabled and this is the highest priority enabled event input in the module (i.e. it has the lowest priority number).

The firing of a transition $t$ is enabled iff (1) the state $t.from$ is active (or one of its substates if $t.from$ is a CompositeState), (2) $t.guard$ is evaluated to true, (3) the state machine of $t$ is the current state machine under execution. If a trigger event is connected, an enabled transition cannot fire until this event is triggered.

5.2 Execution of a Composite Module $m$

1. (External inputs.) At the beginning of a PLC cycle, the current values of the variables in $m.handledInputVariables$ are known. These values do not change until the end of the PLC cycle (Step 5) in accordance with the PLC semantics.

2. (Local input definitions.) The values of the internal variables are computed based on the definitions in $m.O_inputDefinitions$. The calculation order is the order of items in $m.O_inputDefinitions$.

3. (Submodule execution.) For each module in $m.O_submodules$, the submodules are executed (recursively) based on the rules presented here. The execution order is the explicitly defined order of submodules in $m.O_submodules$.

4. (Output assignments.) The values of $m.O_outputDefinitions$ are computed and the computed values are assigned to the defined output variables. The calculation order is the order of items in $m.O_outputDefinitions$.

5. (End of cycle.) End of the PLC cycle. GOTO Step 1 on a new PLC cycle.

5.3 Execution of an Alternative Module $m$

1. (External inputs.) At the beginning of a PLC cycle, the current values of the variables in $m.handledInputVariables$ are known. These values do not change until the end of the PLC cycle (Step 5) in accordance with the PLC semantics.
2. **(Local input definitions.)** The values of the internal variables are computed based on the definitions in \(m.0.inputDefinitions\). The calculation order is the order of items in \(m.0.inputDefinitions\).

3. **(Submodule execution.)** If the defined condition is true, the \(moduleOnTrue\) module is executed, else the \(moduleOnFalse\) module is executed.

4. **(Output assignments.)** The values of \(m.0.outputDefinitions\) are computed and the computed values are assigned to the defined output variables. The calculation order is the order of items in \(m.0.outputDefinitions\).

5. **(End of cycle.)** End of the PLC cycle. GOTO Step 1 on a new PLC cycle.

### 5.4 Execution of a State Machine Module \(m\)

1. **(External inputs.)** At the beginning of a PLC cycle, the current values of the variables in \(m.handledInputVariables\) are known. These values do not change until the end of the PLC cycle (Step 5) in accordance with the PLC semantics.

2. **(Local input definitions.)** The values of the internal variables are computed based on the definitions in \(m.0.inputDefinitions\). The calculation order is the order of items in \(m.0.inputDefinitions\).

3. **(State machine execution.)**
   
   (a) Firing all non-event-triggered transitions. All transitions in \(m.transitions\) not having a connected trigger event, with guard evaluated to true and with active input state are exhaustively fired. After this step, no more transitions are enabled without trigger event.
   
   (b) Evaluation the \(triggerExpressions\) of all events in \(m.events\).
   
   (c) Firing at most one event-triggered transition. The event with the lowest priority number will trigger out of the events where the \(event.triggerExpression\) is evaluated to \textit{true}. If there is no transition triggered to this event, the active state of the state machine does not change in this step.
   
   (d) (Same as 3.a.) Firing all non-event-triggered transitions. All transitions in \(m.transitions\) not having a connected trigger event, with guard evaluated to true and with active input state are exhaustively fired. After this step, no more transitions are enabled without trigger event.

4. **(Output assignments.)** The values of \(m.0.outputDefinitions\) are computed and the computed values are assigned to the defined output variables. The calculation order is the order of items in \(m.0.outputDefinitions\).

5. **(End of cycle.)** End of the PLC cycle. GOTO Step 1 on a new PLC cycle.

### 5.5 Execution of an Input-Output Connection Module \(m\)

1. **(External inputs.)** At the beginning of a PLC cycle, the current values of the variables in \(m.handledInputVariables\) are known. These values do not change until the end of the PLC cycle (Step 5) in accordance with the PLC semantics.

2. **(Local input definitions.)** The values of the internal variables are computed based on the definitions in \(m.0.inputDefinitions\). The calculation order is the order of items in \(m.0.inputDefinitions\).
3. **(Input-output connection module execution.)** This step consists of assigning current values to each pins and data edges. These current values are only valid for the current execution of the input-output connection module $m$.

(a) **(Computing input pins.)** For each `InputVariablePin` the current value is the value of the corresponding input variable. The current values of the `ConstantPins` are known a priori.

(b) **(Computing data edges.)** For each `DataEdge` $d$ where the actual value of $d.source$ is known, let the current value of $d$ be the current value of $d.source$.

(c) **(Computing consumer pins.)** For each `ConsumerPin` $cp$ where the actual value of the data edge $cp.source$ is known, let the actual value of $cp$ be the actual value of $cp.source$.

(d) **(Computing block output pins.)** For each `BlockInstance` $bi$ where the actual value is known for every `BlockInputPin` in $bi.inputs$, the actual values are computed.
   
i. If $dbi$ is defined by a `CustomBlockDefinition`, the current output values are computed based on the $bi.definition.0.outputdefinitions$. The computation order is the order of elements in $bi.definition.0.outputdefinitions$. The computed values are the current values of the corresponding `BlockOutputPins`.
   
   ii. If $dbi$ is defined by a `BuiltinBlockDefinition`, the computation of the current output values is implementation-specific. The computed values are the current values of the corresponding `BlockOutputPins`.

(e) If there exists any `OutputVariablePin` in $m.targets$ that has no current value, GOTO 3.b (Computing data edges). (As the hypergraph of the pins and data edges is directed acyclic and all consumer pins are connected to a data edge, this loop terminates.)

4. **(Output assignments.)** The values of $m.0.outputDefinitions$ are computed and the computed values are assigned to the defined output variables. The calculation order is the order of items in $m.0.outputDefinitions$.

5. **(End of cycle.)** End of the PLC cycle. GOTO Step 1 on a new PLC cycle.

5.6 **Execution of an Empty Module $m$**

1. **(External inputs.)** At the beginning of a PLC cycle, the current values of the variables in $m.handledInputVariables$ are known. These values do not change until the end of the PLC cycle (Step 5) in accordance with the PLC semantics.

2. **(Local input definitions.)** The values of the internal variables are computed based on the definitions in $m.0.inputDefinitions$. The calculation order is the order of items in $m.0.inputDefinitions$.

3. —

4. **(Output assignments.)** The values of $m.0.outputDefinitions$ are computed and the computed values are assigned to the defined output variables. The calculation order is the order of items in $m.0.outputDefinitions$.

5. **(End of cycle.)** End of the PLC cycle. GOTO Step 1 on a new PLC cycle.
5.7 Execution of a Timer Module $m$

1. **(External inputs.)** At the beginning of a PLC cycle, the current values of the variables in $m.\text{handledInputVariables}$ are known. These values do not change until the end of the PLC cycle (Step 5) in accordance with the PLC semantics.

2. **(Local input definitions.)** The values of the internal variables are computed based on the definitions in $m.\text{O_inputDefinitions}$. The calculation order is the order of items in $m.\text{O_inputDefinitions}$.

3. The output of the timer is computed based on the semantics defined in IEC 61131-3 [3].

4. **(Output assignments.)** The values of $m.\text{O_outputDefinitions}$ are computed and the computed values are assigned to the defined output variables. The calculation order is the order of items in $m.\text{O_outputDefinitions}$.

5. **(End of cycle.)** End of the PLC cycle. GOTO Step 1 on a new PLC cycle.

6 Formal Semantics

The formal semantics of PLCspecif is defined as a construction of a variant of the widely-known timed automata formalism. Timed automata has a well-defined formal semantics, and it is a relatively high-level formalism, thus it is convenient for describing the formal semantics of PLCspecif. Furthermore, the semantics of a specification defined as an automaton is close to the control flow graph of its implementation, helping to design a code generator that follows the formal semantics.

6.1 Timed Automata

Timed automaton (TA) [1] is essentially a set of locations, clocks, and edges (transitions), each having a source and target location, and a clock constraint. To make the usage of the formalism more convenient, we extended the TA definition with data variables that can be used in conditions of edges, also firing transitions represented by the edges can assign new values to data variables. The semantics definition is based on the timed automata defined below, that is an extension of the timed automata formalism defined in [1]. These extensions do not increase the expressive power of the formalism, but simplifies the discussion. (The coloured parts show the extensions in the definition.)

**Definition 5 (Timed automaton with data variables)** A timed automaton (TA) is a tuple $(L,l_0,C,A,E,I,V,v_0,U)$ where:

- $L$ is a set of locations,
- $l_0 \in L$ is the initial location,
- $C$ is the set of clocks,
- $A$ is a set of actions, co-actions and the internal $\tau$-action,
- $E \subseteq L \times A \times B_{\text{bool}}(C) \times B_{\text{bool}}(V) \times (V \rightarrow (B'(V) \cup \{\cdot\})) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset (if $(l,a,\text{cg},\text{vg},m,r,l') \in E$ then it is marked as $l \xrightarrow{a,\text{cg},\text{vg},m,r,l'} l'$, where $a \in A$ is the action, $\text{cg} \in B_{\text{bool}}(C)$ is the guard on clocks, $\text{vg} \in B_{\text{bool}}(V)$ is the guard on data variables, $m: V \rightarrow (B'(V) \cup \{\cdot\})$ is change of the data variables (where $\cdot$ means “no change”), and $r \in 2^C$ is the set of clocks to reset),
– \( I: L \rightarrow B_{\text{bool}}(C) \) assigns invariants to locations,

– \( V \) is the set of data variables,

– \( v_0: V \rightarrow D \) assigns initial values to the variables \((D \) is the set of possible variable values\),

– \( U \subseteq L \) is the set of urgent locations.

N.b.: \( B_{\text{bool}}(C) \) is a clock constraint of form \( x \sim n \) or \( x - y \sim n \) for \( x,y \in C, \sim \in \{\leq, <, =, >, \geq\} \) and \( n \in \mathbb{N} \). \( B'(V) \) is an expression on data variables in \( V \) and/or constants. \( B_{\text{bool}}(V) \subset B'(V) \) is a Boolean expression using data variables in \( V \) and/or constants.

The semantics defined for this definition is the following:

**Definition 6 (Operational semantics of timed automaton with data variables)** Let \( \langle L, l_0, C, A, E, I, V, v_0, U \rangle \) be a timed automaton. The semantics is defined as a labelled transition system \( \langle S, s_0, \rightarrow \rangle \), where

– \( S \subseteq L \times \mathbb{R}_\geq C \times (V \rightarrow D) \) is the set of states,

– \( s_0 = (l_0, u_0, v_0) \in S \) is the initial state,

– \( \rightarrow \subseteq S \times (\mathbb{R}_\geq 0 \cup A) \times S \) is the transition relation such that:

  – (delay transition) \( (l, u, v) \xrightarrow{d} (l, u + d, v) \) if \( \forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l) \) and \( l \notin U \),

  – (action transition) \( (l, u, v) \xrightarrow{a} (l', u', v') \) if there exists \( l \xrightarrow{a,cg, cg, m,r} l' \) in \( E \) such that \( u \in cg, v \in cg, v' = m(v), u' = [r \rightarrow 0]u, \) and \( u' \in I(l') \),

where for \( d \in \mathbb{R}_\geq, u + d \) maps each clock \( x \in C \) to the value \( u(x) + d \), and \( u' = [r \rightarrow 0]u \) denotes the clock valuation which maps each clock in \( r \) to 0 and agrees with \( u \) over \( C \setminus r \). The notation \( v \in cg \) means \( cg \) guards are evaluated to true having the \( v \) data variable values. \( v' \) will be the evaluation of \( m \) while having current data variable values as defined in \( v \).

For each variable \( i \in V \), the definition of \( v' = m(v) \) is the following:

\[
(m(v))(i) = v'(i) = \begin{cases} 
v(i) & \text{if } m(i) = \cdot \\
m(i) & \text{else}
\end{cases}
\]

If the timed automaton have only the internal action \( \tau \), then the semantics can be defined based on a Kripke structure instead of a labelled transition system (LTS). The set of states, initial states and transitions are the obvious translations of the ones defined for LTS. The labelling for each state \( s = (l, u, v) \) should (at least) contain all the atomic propositions that are true for the variables \( u, v \). Obviously, there can be a huge number of these atomic propositions.

As the formal semantics definition is presented as a pseudocode, we provide a (reduced) metamodel of the timed automaton corresponding to the definition above. The metamodel can be found in Figure 12.
6.2 Translation Algorithms

This section presents the precise pseudocodes of the transformation from PLCspecif modules to timed automaton. The algorithms are illustrated by non-formal figures to help the reader to understand the main ideas.

The following main algorithms are defined:

- General TA and variable definitions, as well as the platform model (handling input variables, scan cycle):
  TranslatePlatform(Module) – Algorithm 1

- General representation for any module:
  TranslateRec(Module, TALocation, TALocation, inout TA) – Algorithm 2

- Specific representation of composite modules:
  TranslateRecSpec(CompositeModule, TALocation, TALocation, inout TA) – Algorithm 3

- Specific representation of alternative modules:
  TranslateRecSpec(AlternativeModule, TALocation, TALocation, inout TA) – Algorithm 4

- Specific representation of state machine modules:
  TranslateRecSpec(StatemachineModule, TALocation, TALocation, inout TA) – Algorithm 5

- Specific representation of input-output connection modules:
  TranslateRecSpec(IoConnectionModule, TALocation, TALocation, inout TA) – Algorithm 6

- Specific representation of empty modules:
  TranslateRecSpec(EmptyModule, TALocation, TALocation, inout TA) – Algorithm 7

- Specific representation of TON timer modules:
  TranslateRecSpec(TonTimer, TALocation, TALocation, inout TA) – Algorithm 8

- Specific representation of TOFF timer modules:
  TranslateRecSpec(ToffTimer, TALocation, TALocation, inout TA) – Algorithm 9

- Specific representation of TP timer modules:
  TranslateRecSpec(TpTimer, TALocation, TALocation, inout TA) – Algorithm 10

The pseudocodes use the following utility functions, precisely defined later:

- allSubmodulesOf(Module) : set of Module – Algorithm ??
– allParentStatesOf(\textit{AbstractState}) : set of \textit{CompositeState} – Algorithm 12
– allPseudoStatesOf(\textit{AbstractState}) : set of \textit{PseudoState} – Algorithm 13
– basicStatesIn(\textit{StatemachineModule}) : set of \textit{BasicState} – Algorithm 14
– basicStatesOf(\textit{AbstractState}) : set of \textit{BasicState} – Algorithm 15
– historyStatesToUpdate(\textit{AbstractState}) : set of \textit{PseudoState} – Algorithm 16
– blockInstEvalOrder(\textit{IoConnectionModule}) : ordered list of \textit{BlockInstance} – Algorithm 17
Algorithm 1: TranslatePlatform

// This algorithm creates the TA, constructs the variable definitions, as well as the platform model (handling input variables, scan cycle).

Input: m :: Module; // The root module
Output: a :: TA; // The TA representation

1. $a \leftarrow$ new empty TA;

// Creating all variables
2. foreach $m' \in$ allSubmodules($m$) do
3. foreach $v \in m'.definedVariables$ do
4. $v' \leftarrow$ new TAVariable(name: $v$.name, type: $v$.type, default_value: translate($v$.defaultValue) ta: $a$);
5. mapping($v$, TAVariable) $\leftarrow v'$;

// This function keeps the mapping.
6. $I \leftarrow \{ \text{mapping}(v, \text{TAVariable}) \mid v$.direction = INPUT $\land \neg v$.internal $\land \neg v$.parameter$\}$; // set of input variables
7. $P \leftarrow \{ \text{mapping}(v, \text{TAVariable}) \mid v$.direction = INPUT $\land \neg v$.internal $\land v$.parameter$\}$; // set of parameter variables

// Creating TA locations
8. $l_0 \leftarrow$ new TALocation(ta: $a$);
9. $l_1 \leftarrow$ new TALocation(ta: $a$);
10. $l_{\text{init}} \leftarrow$ new TALocation(ta: $a$);
11. $l_{\text{end}} \leftarrow$ new TALocation(ta: $a$);
12. $a$.initialLocation $\leftarrow l_{\text{init}}$;

// Creating the transition representing the PLC input sampling
13. $t_0 \leftarrow$ new TATransition(from: $l_0$, to: $l_{\text{init}}$, clock_guard: $\emptyset$, data_guard: $\emptyset$, var_chng: $\langle \forall v \in P : v := $nondeterministic value$\rangle$, ta: $a$);
14. $t_1 \leftarrow$ new TATransition(from: $l_{\text{init}}$, to: $l_1$, clock_guard: $\emptyset$, data_guard: $\emptyset$, var_chng: $\langle \forall v \in I : v := $nondeterministic value$\rangle$, ta: $a$);

// Creating the transition representing the PLC cycle
15. $t \leftarrow$ new TATransition(from: $l_{\text{end}}$, to: $l_{\text{init}}$, clock_guard: $\emptyset$, data_guard: $\emptyset$, var_chng: $\emptyset$, ta: $a$);

// Translate modules recursively
16. TranslateRec($m$, $l_1$, $l_{\text{end}}$, $a$);
17. return $a$;

Figure 13: Illustration for function TranslatePlatform (Algorithm 1)
Algorithm 2: TranslateRec

// This algorithm translates the modules recursively. The representation will be inserted between the two given TA locations (l₁, l₅).
Input: m :: Module, l₁, l₅ :: TALocation
In-out : a :: TA;
// The TA representation

// Creating intermediate locations
l₄ ← new TALocation(ta: a);

// Translating input definitions
l₃ ← l₁;
foreach vde ∈ m.O.inputDefinitions do
    l₅ ← l₄;
    new TATransition(from: l₃, to: l₅, clock_guard: ∅, data_guard: ∅, var_chng:
        (mapping(vde.variable, TAVariable) := translate(vde.expression)), ta: a);
    l₃ ← l₅;

// Translating event definitions and their computations (only for LeafModules)
e if m is a LeafModule then
    l₃ ← l₄;
    foreach e ∈ m.events do
        v ← new TAVariable(name: e.name, type: Boolean, ta: a);
        h ← \bigwedge_{e' ∈ P} ¬ e'.triggerExpression; // Higher priority events are not active.
        new TATransition(from: l₃, to: l₅, clock_guard: ∅, data_guard: ∅, var_chng: (v := (e.triggerExpression ∧ h)), ta: a);
        mapping(e, TAVariable) ← v;
    l₃ ← l₄;
e else
    l₃ ← new TALocation(ta: a);
    new TATransition(from: l₂, to: l₃, clock_guard: ∅, data_guard: ∅, var_chng: ∅, ta: a);

// Module-specific translation (different for each type)
TranslateRecSpec(m, l₁, l₄, a);

// Translating output definitions
l₅ ← l₄;
foreach vde ∈ m.O.outputDefinitions do
    l₅ ← new TALocation(ta: a);
    new TATransition(from: l₅, to: l₄, clock_guard: ∅, data_guard: ∅, var_chng: (v := vde.expression), ta: a);
    l₅ ← l₄;
new TATransition(from: l₁, to: l₅, clock_guard: ∅, data_guard: ∅, var_chng: ∅, ta: a);
Figure 14: Illustration for function TranslateRec (Algorithm 2)
**Algorithm 3: TranslateRecSpec**

// This algorithm translates a CompositeModule module recursively. The representation will be inserted between the two given TA locations.

**Input:** $m :: CompositeModule$, $l_1,l_2 :: TALocation$

**In-out:** $a :: TA$;

// The TA representation
1. $l_{from} \leftarrow l_1$;
2. foreach $sm \in m.O_{submodules}$ do
3.   $l_{to} \leftarrow $ new TALocation($ta: a$);
4.   TranslateRecSpec($sm,l_{from},l_{to},a$);
5. $l_{from} \leftarrow l_{to}$;
6. new TATransition(from: $l_{from}$, to: $l_2$, clock_guard: $\emptyset$, data_guard: $\emptyset$, var_chng: $\emptyset$, ta: $a$);
Algorithm 4: TranslateRecSpec

// This algorithm translates an AlternativeModule module recursively. The representation will be inserted between the two given TA locations.
Input: m :: AlternativeModule, l₁, l₆ :: TALocation
In-out : a :: TA;
// The TA representation

// Creating intermediate locations
1 l₂ ← new TALocation(ta: a);
2 l₃ ← new TALocation(ta: a);
3 l₄ ← new TALocation(ta: a);
4 l₅ ← new TALocation(ta: a);

// If the condition is true...
5 g ← translate(m.condition);
6 new TATransition(from: l₁, to: l₂, clock_guard: ∅, data_guard: g, var_chng: ∅, ta: a);
7 new TATransition(from: l₃, to: l₆, clock_guard: ∅, data_guard: ∅, var_chng: ∅, ta: a);
8 TranslateRecSpec(translate(m.moduleOnTrue), l₂, l₃, a);

// If the condition is false...
9 new TATransition(from: l₁, to: l₄, clock_guard: ∅, data_guard: ¬g, var_chng: ∅, ta: a);
10 new TATransition(from: l₅, to: l₆, clock_guard: ∅, data_guard: ∅, var_chng: ∅, ta: a);
11 TranslateRecSpec(translate(m.moduleOnFalse), l₄, l₅, a);

Figure 16: Illustration for the function TranslateRecSpec for AlternativeModules (Algorithm 4)
Algorithm 5: TranslateRecSpec

// This algorithm translates a StatemachineModule module recursively. The representation will be inserted between the two given TA locations.

Input: m :: StatemachineModule, l₁, l₄ :: TALocation
In-out : a :: TA;                                   // The TA representation

// Creating intermediate locations
l₂ ← new TALocation(ta: a);
l₃ ← new TALocation(ta: a);

// Creating TA variables
activeState ← new TAVariable(type: enum of basicStatesIn(m), defaultValue: m.initialState, ta: a);
foreach s ∈ allPseudoStatesOf(m.rootState) do

if s is a DeepHistoryState then
  v ← new TAVariable(type: enum of basicStatesIn(m), defaultValue: s.defaultState, ta: a);
  mapping(s, TAVariable) ← v;

B ← false;

// Translating non-triggered transitions as TA transitions
foreach t ∈ m.transitions do

if t.trigger = Ø then

srcActive ← ∪ s' ∈ basicStatesIn(t.from) (activeState = literal(s'));

let g ← srcActive ∧ t.guard;       // t can fire if the source is active and guard is true
B ← (B ∨ g);                       // B is a symbolic expression (not evaluated in transformation time)

// After firing the target state will be activated
if ¬(t.to is a DeepHistoryState) then

vc ← (activeState := literal(t.to));

foreach h ∈ historyStatesToUpdate(t.to) do
  // o is the concatenation operator, e.g. (a₁, ..., aₙ) o (b₁, ..., bₘ) = (a₁, ..., aₙ, b₁, ..., bₘ).
  vc ← vc ∪ (mapping(h, TAVariable) := literal(t.to));                      // Saving history
else

vc ← (activeState := mapping(t.to, TAVariable));                           // Restoring state from history

t₁ ← new TATransition(from: l₁, to: l₃, clock_guard: Ø, data_guard: g, var_chg: vc, ta: a);

// Translating triggered transitions as TA transitions
B₂ ← false;

foreach t ∈ m.transitions do

if t.trigger ≠ Ø then

srcActive ← ∪ s' ∈ basicStatesIn(t.from) (activeState = literal(s'));

let g ← srcActive ∧ mapping(t.trigger, TAVariable) ∧ t.guard;       // t can fire if the source is active and guard is true

B₂ ← (B₂ ∨ g);                       // B₂ is a symbolic expression (not evaluated in transformation time)

// After firing the target state will be activated
if ¬(t.to is a DeepHistoryState) then

vc ← (activeState := literal(t.to));

foreach h ∈ historyStatesToUpdate(t.to) do
  // o is the concatenation operator, e.g. (a₁, ..., aₙ) o (b₁, ..., bₘ) = (a₁, ..., aₙ, b₁, ..., bₘ).
  vc ← vc ∪ (mapping(h, TAVariable) := literal(t.to));                      // Saving history
else

vc ← (activeState := mapping(t.to, TAVariable));                           // Restoring state from history

t₃ ← new TATransition(from: l₂, to: l₄, clock_guard: Ø, data_guard: g, var_chg: vc, ta: a);

// If none of the triggered transition could fire, skip this phase and the second ESF step

new TATransition(from: l₂, to: l₄, clock_guard: Ø, data_guard: ¬B₂, var_chg: vc, ta: a);

// When the "exhaustive stabilisation firing" step is over
new TATransition(from: l₁, to: l₃, clock_guard: Ø, data_guard: ¬B, var_chg: {}, ta: a);
new TATransition(from: l₃, to: l₄, clock_guard: Ø, data_guard: ¬B, var_chg: {}, ta: a);
Figure 17: Illustration for the function TranslateRecSpec for StatemachineModules (Algorithm 5)
Algorithm 6: TranslateRecSpec

// This algorithm translates a IoConnectionModule module recursively. The representation will be inserted between the two given TA locations.

Input: m :: IoConnectionModule, l₁, l₂ :: TALocation
In-out : a :: TA;

// The TA representation

// Translating block output pins to variables
foreach bi ∈ m.blockInstances do
  foreach p ∈ bi.O.outputs do
    v ← new TAVariable(name: p.name, type: p.type, ta: a);
    mapping(p, TAVariable) ← v;

// Mappings for input variable pins
foreach p ∈ m.pins do
  if p is a InputVariablePin then
    mapping(p, TAVariable) ← mapping(p.variable, TAVariable);

// Mappings for block input pins
foreach p ∈ m.pins do
  if p is a BlockInputPin then
    mapping(p, TAVariable) ← mapping(p.inEdge.source, TAVariable);
  if p is a ConstantPin then
    mapping(p, ·) ← p.constant.value;

// Transitions representing the calculation of block outputs
l ← l₁;
foreach bi ∈ blockInstEvalOrder(m) do
  foreach p ∈ bi.O.outputs do
    l' ← new TALocation(ta: a);
    vc ← (mapping(p, TAVariable) := outputDefExprInstance(p.pinDefinition.assignment, bi)); // instance of the meta-assignment (outputDefExprInstance translates the meta-expression to the instance level)
    new TATransition(from: l, to: l', clock_guard: ∅, data_guard: ∅, var_chng: vc, ta: a);
    l ← l';

// Translating output variable assignments
foreach p ∈ m.pins do
  if p is a InputVariablePin then
    l' ← new TALocation(ta: a);
    vc ← (mapping(p, TAVariable) := mapping(p.inEdge.source, TAVariable));
    new TATransition(from: l, to: l', clock_guard: ∅, data_guard: ∅, var_chng: vc, ta: a);
    l ← l';

new TATransition(from: l, to: l₂, clock_guard: ∅, data_guard: ∅, var_chng: ∅, ta: a);
Figure 18: Illustration for the function TranslateRecSpec for IoConnectionModules (Algorithm 6)
Algorithm 7: TranslateRecSpec

// This algorithm translates an EmptyModule module recursively. The representation will be inserted between the two given TA locations.
Input: m :: EmptyModule, l₁, l₂ :: TALocation
In-out : a :: TA; // The TA representation

```
new TATransition(from: l₁, to: l₂, clock.gaurd: ∅, data.gaurd: ∅, var.čhng: ∅, ta: a);
```

Figure 19: Illustration for the function TranslateRecSpec for EmptyModules (Algorithm 7)
Algorithm 8: TranslateRecSpec

// This algorithm translates an TonTimer module recursively. The representation will be inserted between the two given TA locations.
Input: m :: TonTimer, l₁, l₃ :: TALocation
In-out : a :: TA; // The TA representation

// Creating intermediate locations
l₂ ← new TALocation(ta: a); // Creating TA variables and clock
active ← new TAVariable(type: bool, defaultValue: false, ta: a); // true if timer is currently active
timerOutput ← new TAVariable(type: bool, defaultValue: false, ta: a); // to represent TimerOutputRef
clk ← new TAClock(ta: a);

// Creating intermediate locations
R ← ⋁ re ∈ m resets mapping(re, TAVariable); // reset requested if R = true
del ← m. expressionToDelay; // delay expression

// TA edges
new TATransition(from: l₁, to: l₂, clock_guard: {}, data_guard: ¬active ∧ del ∧ ¬R, var_chg: (active := true), clocks_to_reset: clk, ta: a); // activation of the timer
new TATransition(from: l₁, to: l₂, clock_guard: {}, data_guard: (active ∧ ¬del) ∨ R, var_chg: (active := false), ta: a); // deactivation of the timer
new TATransition(from: l₁, to: l₂, clock_guard: {}, data_guard: ¬((active ∧ ¬del) ∨ R) ∧ ¬¬active ∧ del ∧ ¬R, var_chg: {}); // no action

// Setting the timer output
new TATransition(from: l₂, to: l₃, clock_guard: clk ≥ m. delayTime, data_guard: active, var_chg: (timerOutput := true), ta: a);
new TATransition(from: l₂, to: l₃, clock_guard: clk ≥ m. delayTime, data_guard: ¬active, var_chg: (timerOutput := false), ta: a);
new TATransition(from: l₂, to: l₃, clock_guard: clk < m. delayTime, data_guard: {}, var_chg: (timerOutput := false), ta: a);

Figure 20: Illustration for the function TranslateRecSpec for TonTimers (Algorithm 8)
Algorithm 9: TranslateRecSpec

// This algorithm translates an ToffTimer module recursively. The representation will be inserted between the two given TA locations.
Input: m :: ToffTimer,l1,l3 :: TALocation
In-out : a :: TA; // The TA representation

// Creating intermediate locations
1. l2 ← new TALocation(ta: a); // Creating TA variables and clock
2. active ← new TAVariable(type: bool, defaultValue: false, ta: a); // true if timer is currently active
3. timerOutput ← new TAVariable(type: bool, defaultValue: false, ta: a); // to represent TimerOutputRef
4. clk ← new TAClock(ta: a); // Representing activation of the timer
5. \( R \leftarrow \bigvee_{r \in m.\text{resetTriggers}} \text{mapping}(\text{re}, \text{TAVariable}) \); // reset requested if \( R = \text{true} \)
6. de ← m.\text{expressionToDelay}; // delay expression

// TA edges
7. new TATransition(from: l1, to: l2, clock_guard: \( \emptyset \), data_guard: \( \neg \text{active} \land \neg \text{de} \land \neg R \), var_chng: \( \{ \text{active} := \text{true} \} \) ); // activation of the timer
8. new TATransition(from: l1, to: l2, clock_guard: \( \emptyset \), data_guard: \( (\text{active} \land \text{de}) \lor R \), var_chng: \( \{ \text{active} := \text{false} \} \) ); // deactivation of the timer
9. new TATransition(from: l1, to: l2, clock_guard: \( \emptyset \), data_guard: \( \neg((\text{active} \lor \text{de}) \lor R) \land \neg(\text{active} \land \neg \text{de} \land \neg R) \), var_chng: \( \{ \} \) ); // no action
10. new TATransition(from: l2, to: l3, clock_guard: \( \text{clk} \leq m.\text{delayTime} \), data_guard: \( \emptyset \), var_chng: \( \{ \text{timerOutput} := \text{de} \lor \text{active} \} \) ); // Setting the timer output
11. new TATransition(from: l2, to: l3, clock_guard: \( \text{clk} > m.\text{delayTime} \), data_guard: \( \emptyset \), var_chng: \( \{ \text{timerOutput} := \text{de} \} \) ); // Setting the timer output

\[ \begin{align*}
\text{l}_1 & \quad \left[ \text{active AND !de AND !R} \right] \\
& \quad \text{active} := \text{TRUE} \\
& \quad \text{reset} \\
\text{l}_2 & \quad \left[ \text{active AND de OR R} \right] \\
& \quad \text{active} := \text{FALSE} \\
& \quad \text{else} \\
\text{l}_3 & \quad \text{timerOutput} := \text{de OR (active AND clk <= ne)} \\
\end{align*} \]

Figure 21: Illustration for the function TranslateRecSpec for ToffTimers (Algorithm 9)
Algorithm 10: TranslateRecSpec

// This algorithm translates an TpTimer module recursively. The representation will be inserted between the two given TA locations.

Input: \( m :: \) TpTimer, \( l_1, l_3 :: \) TALocation

In-out: \( a :: \) TA;

// The TA representation

// Creating intermediate locations
1 \( l_2 \leftarrow \) new TALocation(ta: \( a \));

// Creating TA variables and clock
2 active \( \leftarrow \) new TAVariable(type: bool, defaultValue: false, ta: \( a \)); // true if timer is currently active
3 timerOutput \( \leftarrow \) new TAVariable(type: bool, defaultValue: false, ta: \( a \)); // to represent TimerOutputRef
4 clk \( \leftarrow \) new TAClock(ta: \( a \));

// Representing activation of the timer
5 \( R \leftarrow \) \[ S \land (\neg \text{active} \lor (\text{active} \land \text{clk} > \text{ne})) \]
6 \( \text{active} := \text{true} \) \]
7 \( \text{reset clk} \]

// TA edges
8 new TATransition(from: \( l_1 \), to: \( l_2 \), clock_guard: \( \emptyset \), data_guard: \( S \land \neg \text{active} \), var_chng: \( \langle \text{active} := \text{true} \rangle \), clocks_to_reset: \( \text{clk} \), ta: \( a \)); // starting the timer
9 new TATransition(from: \( l_1 \), to: \( l_2 \), clock_guard: \( \text{clk} > m.\text{delayTime} \), data_guard: \( S \land \text{active} \), var_chng: \( \langle \text{active} := \text{true} \rangle \), clocks_to_reset: \( \text{clk} \), ta: \( a \)); // re-starting the timer
10 new TATransition(from: \( l_1 \), to: \( l_2 \), clock_guard: \( \emptyset \), data_guard: \( R \), var_chng: \( \langle \text{active} := \text{false} \rangle \), ta: \( a \)); // deactivation of the timer
11 new TATransition(from: \( l_1 \), to: \( l_2 \), clock_guard: \( \emptyset \), data_guard: \( \neg R \land \neg (S \land \neg \text{active}) \), var_chng: \( \langle \text{active} := \text{false} \rangle \), ta: \( a \)); // no action
12 new TATransition(from: \( l_2 \), to: \( l_3 \), clock_guard: \( \text{clk} \leq m.\text{delayTime} \), data_guard: \( S \land \text{active} \), var_chng: \( \langle \text{timerOutput} := \text{true} \rangle \), clocks_to_reset: \( \text{clk} \), ta: \( a \)); // no action
13 new TATransition(from: \( l_2 \), to: \( l_3 \), clock_guard: \( \text{clk} \leq m.\text{delayTime} \), data_guard: \( \neg \text{active} \), var_chng: \( \langle \text{timerOutput} := \text{false} \rangle \), ta: \( a \)); // no action
14 new TATransition(from: \( l_2 \), to: \( l_3 \), clock_guard: \( \text{clk} > m.\text{delayTime} \), data_guard: \( \emptyset \), var_chng: \( \langle \text{timerOutput} := \text{false} \rangle \), ta: \( a \)); // no action

Figure 22: Illustration for the function TranslateRecSpec for TpTimers (Algorithm 10)
6.3 Utility Functions

Algorithm 11: allLeafmodulesOf

// Collects recursively all leaf modules of \( m \), including \( m \) if it is a LeafModule.
Input: \( m :: Module \)
Output: set of Module

1 if \( m \) is a LeafModule then return \( \{ m \} \);
2 else if \( m \) is a AlternativeModule then
3 return \( \{ \text{allLeafmodulesOf}(m.\text{onTrue}), \text{allLeafmodulesOf}(m.\text{onFalse}) \} \);
4 else if \( m \) is a CompositeModule then
5 \( SM \leftarrow \emptyset \);
6 foreach \( m' \in m.\text{submodules} \) do
7 \( SM \leftarrow SM \cup \text{allLeafmodulesOf}(m') \);
8 return \( SM \);
9 return error;

Algorithm 12: allParentStatesOf

// Collects all container CompositeStates of \( s \), not including \( s \) itself.
Input: \( s :: AbstractState \)
Output: \( PS :: \text{set of CompositeState} \)

1 \( PS \leftarrow \emptyset \);
2 \( s' \leftarrow s \);
3 while \( s'.\text{containerState} \neq \text{undefined} \) do
4 \( PS \leftarrow PS \cup s'.\text{containerState} \);
5 \( s' \leftarrow s'.\text{containerState} \);
Algorithm 13: allPseudoStatesOf

Input: $s : AbstractState$
Output: set of PseudoState

1 $PS \leftarrow \emptyset$;
2 if $s$ is a PseudoState then
3 \quad $PS \leftarrow \{s\}$;
4 else if $s$ is a CompositeState then
5 \quad foreach $s' \in s.containedStates$ do
6 \quad \quad $PS \leftarrow PS \cup \text{allPseudoStatesOf}(s')$;
7 return $PS$;

Algorithm 14: basicStatesIn

// Collects all basic states defined for a given StatemachineModule $m$.
Input: $m : StatemachineModule$
Output: set of BasicState

1 return basicStatesOf($m$.rootState);

Algorithm 15: basicStatesOf

Input: $s : AbstractState$
Output: set of BasicState

1 $AS \leftarrow \emptyset$;
2 if $s$ is a BasicState then
3 \quad $AS \leftarrow \{s\}$;
4 else if $s$ is a CompositeState then
5 \quad foreach $s' \in s.containedStates$ do
6 \quad \quad $AS \leftarrow AS \cup \text{basicStatesOf}(s')$;
7 return $AS$;

// If $s$ is a PseudoState, $AS$ remains empty.

Algorithm 16: historyStatesToUpdate

Input: $s : AbstractState$ // Newly activated state
Output: $PS : set$ of PseudoState

1 $PS \leftarrow \emptyset$;
2 foreach $m \in \text{allParentStatesOf}(s)$ do
3 \quad foreach $s' \in m.containedStates$ do
4 \quad \quad if $s'$ is a DeepHierarchyState then
5 \quad \quad \quad $PS \leftarrow PS \cup s$;
Algorithm 17: blockInstEvalOrder

Input: \( m :: \text{IoConnectionModule} \)

Output: ordered list of BlockInstance

1 \( CP \leftarrow \emptyset; \) \hspace{1cm} // set of already computed ProcuderPins
2 \( L \leftarrow (); \) \hspace{1cm} // The input variable pins and constants are computed at the beginning of the execution
3 \hspace{1cm} \text{foreach} \{ p \in m.\text{pins} : p \text{ is a InputVariablePin} \lor p \text{ is a ConstantPin}\} \text{ do}
4 \hspace{2cm} CP \leftarrow CP \cup p;
5 \hspace{1cm} \text{while} \ |L| < |m.\text{blockInstances}| \text{ do}
6 \hspace{2cm} \text{foreach} bi \in m.\text{blockInstances} \text{ do}
7 \hspace{3cm} \text{if} \ \neg \text{contains}(L, bi) \text{ then}
8 \hspace{4cm} rdy \leftarrow \bigwedge_{ip \in bi.\text{inputs}} (ip.\text{inEdge.source} \in CP);
9 \hspace{4cm} \text{if} rdy \text{ then}
10 \hspace{5cm} \text{// Every input is ready, the block outputs can be computed}
11 \hspace{5cm} \text{// } \circ \text{ is the concatenation operator, e.g.}
12 \hspace{5cm} (a_1, \ldots, a_n) \circ (b_1, \ldots, b_m) = (a_1, \ldots, a_n, b_1, \ldots, b_m).
13 \hspace{4cm} L \leftarrow L \circ bi;
14 \hspace{4cm} CP \leftarrow CP \cup bi.\text{O.outputs};
15 \hspace{2cm} \text{return} L;

6.4 Expressions

This section presents the semantics of the expressions in a denotational, pseudo-formal way.

Evaluation of primitive types (value references and constants)

- Evaluation of a timer output reference
  \[ [r :: \text{TimerOutputRef}] = \text{timerOutput} \text{ (if used in a TimerModule)} \]

- Evaluation of a variable reference
  \[ [r :: \text{VariableRef}] = \text{current value of variable mapping}(r.\text{variable}, \text{TAVariable}) \]

- Evaluation of a state reference
  \[ [r :: \text{InStateRef}] = \text{activeState} \text{ (if used in a StatemachineModule)} \]

- Evaluation of a producer pin reference
  \[ [r :: \text{InputVariablePin}] = \text{current value of variable mapping}(r.\text{variable}, \text{TAVariable}) \text{ (if used in a IoConnectionModule)} \]
  \[ [r :: \text{BlockOutputPin}] = \text{current value of variable mapping}(r.\text{variable}, \text{TAVariable}) \text{ (if used in an IoConnectionModule)} \]
  \[ [r :: \text{ConstantPin}] = r.\text{constant.value} \text{ (if used in an IoConnectionModule)} \]

- Evaluation of a constant
  \[ [c :: \text{Constant}] = c.\text{value} \]
Evaluation of unary operations

- Evaluation of a negation
  \[ n \bowtie \text{Negate} = \neg \llbracket n.\text{argument} \rrbracket \] (if \( n.\text{argument} \) has a Boolean dynamic type)

- Evaluation of an edge
  \[ e \bowtie \text{RisingEdge} = \neg L \land \llbracket e.\text{argument} \rrbracket, \] where \( L \) denotes the last value \( \llbracket e.\text{argument} \rrbracket \) was evaluated to (if \( e.\text{argument} \) has a Boolean dynamic type)

\[ e \bowtie \text{FallingEdge} = L \land \neg \llbracket e.\text{argument} \rrbracket, \] where \( L \) denotes the last value \( \llbracket e.\text{argument} \rrbracket \) was evaluated to (if \( e.\text{argument} \) has a Boolean dynamic type)

Evaluation of binary operations

- Evaluation of a comparison
  \[ c \bowtie \text{Comparison} = (\llbracket c.\text{leftArgument} \rrbracket = \llbracket c.\text{rightArgument} \rrbracket) \]
  \[ c \bowtie \text{Comparison} = (\llbracket c.\text{leftArgument} \rrbracket \neq \llbracket c.\text{rightArgument} \rrbracket) \]
  \[ c \bowtie \text{Comparison} = (\llbracket c.\text{leftArgument} \rrbracket < \llbracket c.\text{rightArgument} \rrbracket) \]
  \[ c \bowtie \text{Comparison} = (\llbracket c.\text{leftArgument} \rrbracket > \llbracket c.\text{rightArgument} \rrbracket) \]
  \[ c \bowtie \text{Comparison} = (\llbracket c.\text{leftArgument} \rrbracket \leq \llbracket c.\text{rightArgument} \rrbracket) \]
  \[ c \bowtie \text{Comparison} = (\llbracket c.\text{leftArgument} \rrbracket \geq \llbracket c.\text{rightArgument} \rrbracket) \]

- Evaluation of an arithmetic operation
  \[ a \bowtie \text{ArithmeticOperation} = (\llbracket a.\text{leftArgument} \rrbracket + \llbracket a.\text{rightArgument} \rrbracket) \]
  \[ a \bowtie \text{ArithmeticOperation} = (\llbracket a.\text{leftArgument} \rrbracket - \llbracket a.\text{rightArgument} \rrbracket) \]
  \[ a \bowtie \text{ArithmeticOperation} = (\llbracket a.\text{leftArgument} \rrbracket \times \llbracket a.\text{rightArgument} \rrbracket) \]
  \[ a \bowtie \text{ArithmeticOperation} = (\llbracket a.\text{leftArgument} \rrbracket / \llbracket a.\text{rightArgument} \rrbracket), \] if at least one of the arguments has a dynamic type that is not integer
  \[ a \bowtie \text{ArithmeticOperation} = (\llbracket a.\text{leftArgument} \rrbracket \text{ div } \llbracket a.\text{rightArgument} \rrbracket), \] if both arguments have a dynamic type integer

Evaluation of complex operations

- Evaluation of an if-then-else operator
  \[ i \bowtie \text{IteOperation} = \begin{cases} \llbracket i.\text{onTrue} \rrbracket & \text{if } \llbracket i.\text{condition} \rrbracket \\ \llbracket i.\text{onFalse} \rrbracket & \text{if } \neg \llbracket i.\text{condition} \rrbracket \end{cases} \]
  (if \( i.\text{condition} \) has a Boolean dynamic type)

- Evaluation of an AND/OR-table (DNF-expression)
  \[ d \bowtie \text{DnfExpression} = \bigvee_{c \bowtie \text{DnfClause} \in d.\text{clauses}} \llbracket c.\text{expression} \rrbracket \]

- Evaluation of a switch-case table
  \[ s \bowtie \text{SwitchCaseTable} = \begin{cases} \llbracket s.\text{rows}[0].\text{value} \rrbracket & \text{if } \llbracket s.\text{rows}[0].\text{condition} \rrbracket \\ \llbracket s.\text{rows}[1].\text{value} \rrbracket & \text{if } \llbracket s.\text{rows}[1].\text{condition} \rrbracket \land \neg \llbracket s.\text{rows}[0].\text{condition} \rrbracket \\ \llbracket s.\text{rows}[2].\text{value} \rrbracket & \text{if } \llbracket s.\text{rows}[2].\text{condition} \rrbracket \land \neg \llbracket s.\text{rows}[1].\text{condition} \rrbracket \land \\ \text{...} \end{cases} \]

- Evaluation of “element of” operation
  \[ e \bowtie \text{ElementOfOperation} = (\llbracket e.\text{expression} \rrbracket \in e.\text{values}) \]
**RSFlipFlop**

This module represents a flip-flop with two reset inputs and an edge-driven set input.

**Assigned inputs:**
- AutoReset : BOOL  
  Reset request from logic
- ManualReset : BOOL  
  Reset request from operator
- Set : BOOL  
  Set request

**Assigned outputs:**
- Q : BOOL  
  Normal (positive) output
- notQ : BOOL  
  Negated output

**Input definitions:** —

**Event definitions:**
- @reset \( \leftarrow \) AutoReset OR ManualReset \( ( \text{pri}=1) \)  
  Resets the flip-flop
- @set \( \leftarrow \) rising_edge(Set) \( ( \text{pri}=2) \)  
  Sets the flip-flop

**Core logic (state machine)**

![Core logic diagram](image)

**Output definitions:**
- \( Q = \text{in\_state}(\text{On}) \)
- \( \text{not}Q = \text{NOT} \text{in\_state}(\text{On}) \)

**Invariant properties:**
- ALWAYS Q \( \neq \) notQ

---

### 7 Examples

#### 7.1 R-S Flip-Flop

Here we present a simple R-S flip-flop module as an example for this report. The specification of the flip-flop is presented in Figure 23. This flip-flop has two reset inputs and an edge-driven set input. If one of the reset inputs is true, the normal output (Q) of the module will become false. If there is a rising edge on the set input, the output will be set to true. The reset input has priority over the set input. Besides the normal output Q, its negated value should be also produced (notQ) that must always have a value different from the value of Q.

The general module structure can be observed in Figure 23. The header is followed by three blocks: (1) the input and event definitions, (2) the core logic definition, and (3) the output definition. Each element of the specification (e.g. the input and event definitions, invariant properties) can have a corresponding textual description to help the understanding the specification.

Figure 24 presents the corresponding TA. Note that some TA locations were merged if there was exactly one edge between them without any guard or variable assignment.

#### 7.2 OnOff Module

The example presented in Section 7.1 is obviously too small to properly motivate a formal specification and a code generation method. We have conducted an experiment with a much bigger, real PLC module from CERN’s UNICOS (Unified Industrial Control System) framework\(^4\). This object is called OnOff that is the representation of a valve, heater, or similar digital actuator. Currently, we are using a manual implementation of OnOff, developed during the last

---

\(\text{http://cern.ch/unicos/}\)
ten years. This implementation is significantly more complex than the R-S flip-flop example: the manual implementation has about 120 input/output variables and 600 lines of code.

With the aim of creating a formal specification for this module, we were able to capture the key parts in PLCspecif. This led to an increased understanding of the module, as for the specification the module has to be understood in detail. On the basis of the work with and the review of the specification, we have found several problems in our specification. Also, we have found some bugs in the current implementation just by creating a specification based on the legacy code. The specification of OnOff is much more complex than the example presented here, it contains 24 modules (leaf and composite), and 32 internal input/output definitions in addition to the module’s input/output variables. 9 of 24 modules are state machine modules, containing in total 31 states and 41 transitions. The specification of the OnOff module uses all features of the PLCspecif state machine module. Publication of this specification case study in detail is in progress, its authorization is pending.

Figure 24: TA describing the semantics of module RSFlipFlop
References

