JOHANNES KEPLER
UNIVERSITY LINZ

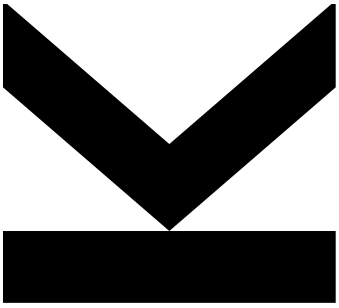
Submitted by
Michael Jäger, BSc.

Submitted at
Institute for System Software

Supervisor
o.Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck

February 2020

A Truffle-based Compiler for IEC Language



Masterthesis

to obtain the academic degree of

Diplom-Ingenieur

in the Masterprogram

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

IEC 61499 is a event-driven system architecture model to describe the behaviour of distributed applications within industrial control and measurement systems. It uses function blocks to model these applications. This work shows how to execute a basic function block, that is capable to exchange events and data with linked ones, on Graal VM. The language implementation framework Truffle allows to execute any language modelled as an Abstract Syntax Tree (AST) on top of Graal VM.

Although Truffle is invented for dynamic languages, also statically typed languages, like IEC 61499 Structured Text, which is used for function blocks, can benefit from runtime optimizations. This thesis shows how to parse a function block and map it to an AST that is executable on Graal VM. It also shows how to map the semantics of IEC 61499 Structured Text to Java, including statements, control structures, variables and datatypes. This work demonstrates how to implement a communication interface that executes the AST from an external event and sends back an output event after the execution.

Contents

1 Introduction

This chapter states the driving forces behind this thesis. It shortly explains the motivation to use IEC 61499 and what the goals are. Furthermore, the scope of what is supported from IEC 61499, will be described. The reader should get an overview of the structure of this thesis in the introduction.

1.1 Motivation

IEC 61499 [1] is a standard developed by the International Electrotechnical Commission. It describes how to model distributed industrial processes by using function blocks, which represent a reusable piece of software with a user-defined function. It has an internal state machine, input and output values and its own memory. Whereas the old standards IEC 61131-3 [2] produces large monolithic software solutions, IEC 61499 can build exchangeable components, which make the applications easier to maintain. Single or composed function blocks are executable components in a distributed system. With IEC 61499 a composition of executable components leads to an application. It can be compared with the Microservice Architecture [3] from Software Engineering.

In order to be executable, every component needs hardware with a specific runtime, which must meet the requirements of the execution model from the IEC 61499 standard. Since a function block is event-controlled, the internal state machine must behave exactly the same on all hardware devices. It is essential that every component can communicate with every other, even if different hardware is used. If a function block should be executed on a new hardware, the specific runtime must be implemented first. So it would be a big advantage, if an IEC 61499 function block could be executed on existing Virtual Machines, like the Java Virtual Machine (JVM) [4]. The framework Truffle [5], which is included in the Graal VM [6], provides a means to implement new languages, which are executed on the JVM. This is a great benefit, because Graal VM already implements features like memory management or runtime optimizations. It also addresses several hardware target architectures.

1.2 Goals

One goal of the thesis is to generate a compiler frontend that can parse a function block, which is written in IEC 61499 Structured Text (ST) [1]. The frontend contains a parser, a scanner and a symbol table. An attributed grammar (ATG) must be written, to generate the parser and scanner with the tool Coco/R [7]. The parser should perform type checks and automatic casts, with the help of the symbol table.
The frontend should build an Abstract Syntax Tree (AST). The Nodes of the AST are supposed to be implemented as Truffle Nodes, which are executable on the Graal VM.

As Truffle was designed to support dynamic languages such as Java-Script, R or Ruby, another goal is to fit the execution model of a function block into the framework.
At the end of this work, a function block should be executable on Graal VM and should be able to communicate with other function blocks on different runtimes.

1.3 Scope

This work does not provide a complete IEC 61499 implementation. It focuses on implementing a working prototype that is able to execute one single basic function block within a distributed IEC 61499 application. Truffle IEC is able to provide a communication interface to exchange data and events with other function blocks. Other function block types and real-time behaviour is neglected.

Our prototype supports all primitive datatypes and their arithmetic and relational operations. The implementation is capable of dealing with arrays, namespaces and user defined enums, but it does not support the datatype **STRUCT**, which is used to declare user-defined structures.

1.4 Structure of the Thesis

After the introduction, *Chapter 2* discusses basic knowledge to understand the IEC 61499 standard. It also describes the technologies that are used to implement this thesis, like Coco/R or Truffle.

Chapter 3 describes the architecture of the implemented system. It explains the components that are needed to provide an IEC 61499 implementation with Truffle, that is capable of exchanging data and events with other devices in a network.

The processing of a function block written in Structured Text is described in *Chapter 4*. This chapter gives an overview of the compiler frontend and explains how to deal with LL(1) conflicts, namespaces and enums.

Chapter 5 explains how the processed function block is represented as an AST.

The execution of the function block is presented by *Chapter 6*. This section contains also a description for exchanging data and events.

Chapter 7 presents an evaluation of this work. It shows the runtime performance of Truffle IEC compared to another IEC 61499 implementation and it demonstrates working examples.

Chapter 8 shows other implementations of IEC 61499 that are related to Truffle IEC and discusses similar compiler technologies.

This thesis finishes with resulting future work in *Chapter 9* and a summary and conclusion in *Chapter 10*.

2 Background Knowledge

This chapter provides basic knowledge to understand the technologies used in this thesis. It will describe the system architecture of IEC 61499 and the execution model for a function block. It also gives a short introduction to virtual machines, especially the Graal VM and Substrate VM. Also the concepts of dynamic compilation will be explained as well as the tool Coco/R for parser and scanner generation.

2.1 From IEC 61131-3 to IEC 61499

The standard IEC 61131-3 is commonly used to model industrial processes, measurements and control systems and to describe languages for Programmable Logic Controllers (PLC) [2]. The System Architecture is illustrated in Figure 1. It uses linked function blocks to build a program. A IEC 61131-3 function block can execute exactly one algorithm and multiple programs can be executed on a processing facility, which is called resource. Every program is executed by its own thread on the resource, which is depicted as task in Figure 1. Communication between programs is only possible, with global variables or communication function blocks. Using global variables as communication between programs is a bad practice. It will produce large monolithic applications, that are hard to maintain. Scaling is also challenging with this system model. If parts of a program are to be executed on another resource, it must be split into several parts. This is a complicated task, because the system needs to behave exactly the same after splitting. Therefore, developing IEC 61131-3 applications is more focused on managing resources, rather than concentrating on development tasks. IEC 61499 was invented by the International Electrotechnical Commission (IEC) [1] so that the developer can focus more on development again, instead of managing resources.

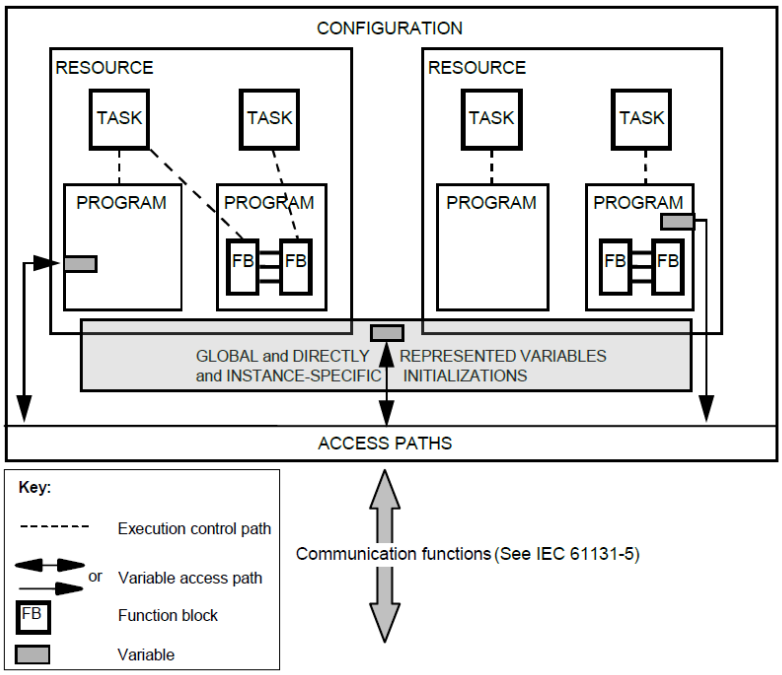


Figure 1: IEC 61131-3 System Architecture [2]

2.1.1 IEC 61499 System Architecture

The IEC 61499 standard is an architectural model to describe the behaviour and structure of *distributed* industrial measurement and control systems. It is a further development of the IEC 61131-3 standard and describes how to define industrial systems with function blocks. Furthermore, the function blocks can be distributed over multiple devices, while still forming a single application. The following models will explain how it is possible to realise that.

The **function block model** [1] [8] [9] says that a function block is a functional unit of software. It has a type name and an instance name. Respectively, there a three different types of function blocks, either a basic, a composite or a service interface function block. The functionality of a basic function block will be described in 2.1.2, as this work will focus on this type. A composite function block is a network containing other function blocks. With service interfaces it is possible to access communication services or other hardware from the devices [1] [8].

A network of connected function blocks is called an IEC 61499 application. The way they are modelled, is defined in the **application model** [1] [8] (Figure 2). The function blocks are linked with data and event connections. In contrast to IEC 61131-3, global variables are prohibited. The whole behaviour of the application is defined with function blocks of different types. In other words, an application is a set of linked function blocks, that form a control system. This model focuses only on the application design

and not on the hardware beneath it. So it is possible to design an application, without considering the hardware. This leads to the next model.

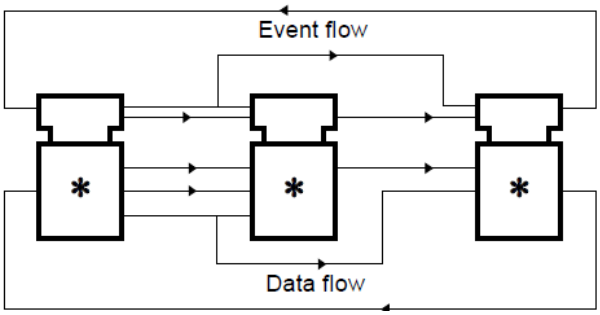


Figure 2: IEC 61149 application model [1]

The **system model** (Figure 3) describes the relationship between distributed devices in industrial control systems. It defines which devices in a network are available and how they are linked with each other [1] [8].

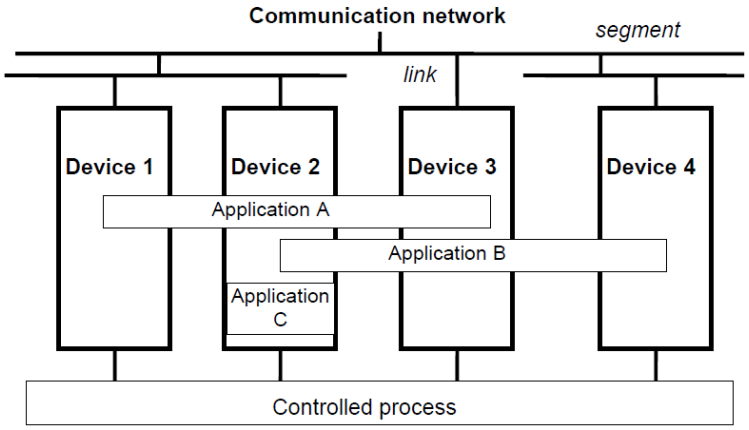


Figure 3: IEC 61149 system model [1]

For the description of the behaviour of linked devices, the **device model** [1] [8] (Figure 4) is responsible. A device is a physical environment that can have multiple resources, which are able to execute the function blocks. A resource is not necessarily bound to a physical device (e.g. CPU). It is just a logical execution unit in a device for function blocks. The device provides a *process interface* to access input and output data of the device (e.g. sensor or display). It also offers a *communication interface* which handles the exchange of data and events with other linked devices. These interfaces can be accessed by several resources on the device.

The **resource model** [1] [8] (Figure 5) defines the relationship between different function blocks within a resource. A resource is responsible to execute the function blocks in the right order, therefore it needs a scheduling function. Another requirement is

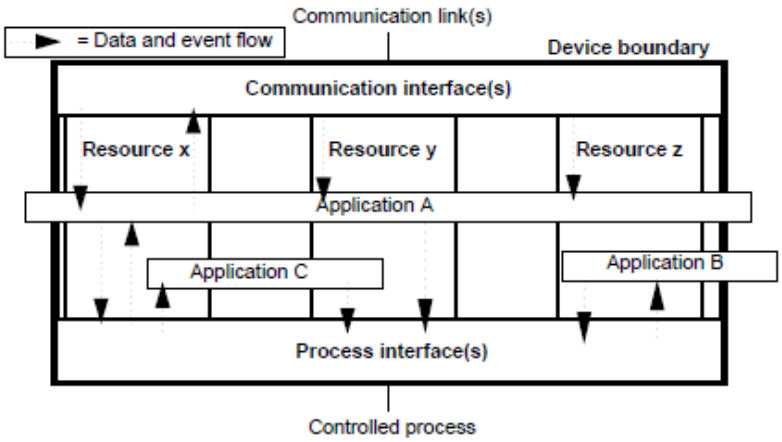


Figure 4: IEC 61149 device model [1]

independence from other resources on the same device, which means, that it must be possible to start, stop or reset it without affecting other resources on the same device. With service interface function blocks it is possible to communicate with the interface of the resource, respectively with the device. They also have a connection to the scheduling function. The algorithmic function blocks are implemented by the developer and are connected to the service interface function blocks. According to the standard, the resource must provide the implementation for the data and event flows.

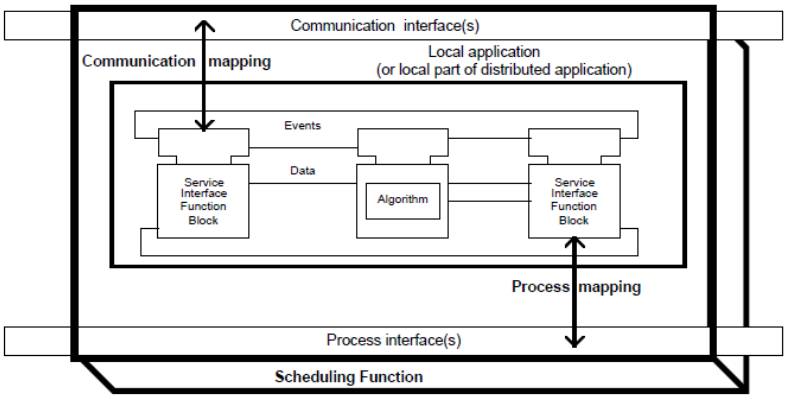


Figure 5: IEC 61149 resource model [1]

The **distribution model** (Figure 6) defines, how an application is distributed over the resources, and which parts of an application are mapped to which resource. It can be seen as a link between application and system model [1] [8].

2.1.2 Basic Function Block

A basic function block, illustrated in Figure 7, is based on an event-driven software architecture and has the following characteristics:

- Several algorithms can be executed within the function block. Regarding the IEC 61499 standard [1], an algorithm can be implemented in any language, as it is an

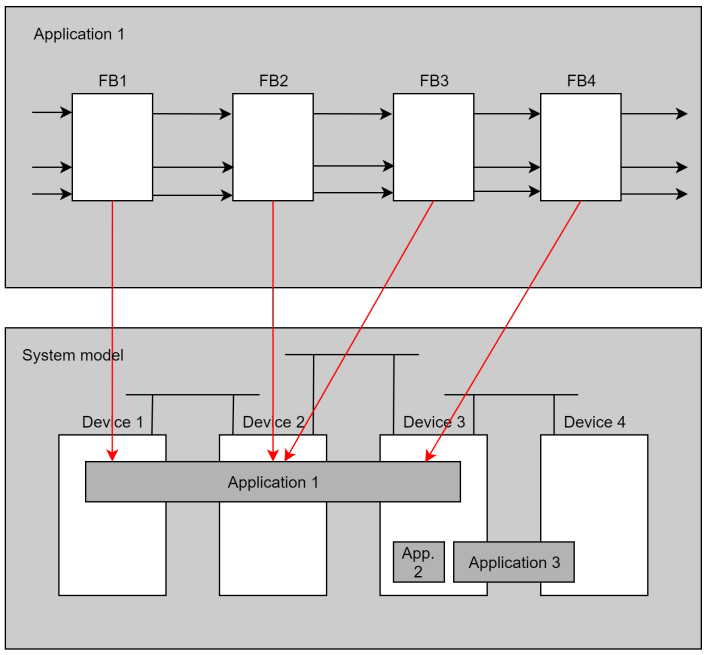


Figure 6: IEC 61149 distribution model [1]

encapsulated functionality. In most implementations of IEC 61499, Structured Text (ST) from IEC 61131-3 [2] is used. This work will also focus on ST.

- Input variables are only writeable from linked function blocks, whose outputs are connected to the inputs.
- In contrast to input variables, output variables are only readable from linked function blocks, whose inputs are connected to the outputs.
- Internal variables are only visible inside the function block.
- All variables can be written and read by all algorithms.
- A function block has event inputs and outputs. An input event can be bound to input variables and can execute one specific algorithm. An output event can be bound to output variables. This binding means that only the specified variables are written or read by the event.
- A function block has an internal state machine, the Execution Control Chart (ECC), which can change its state after events are received.

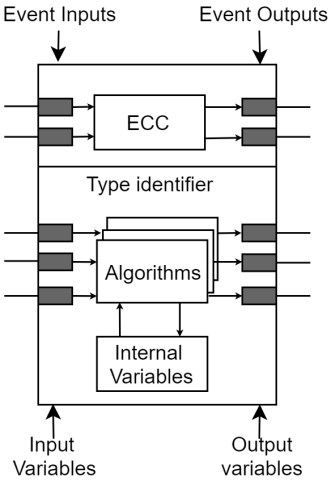


Figure 7: Basic Function Block [1]

2.1.3 Execution of a Basic Function Block

The execution of a basic function is explained with the following numbers, which refer to Figure 8 [9].

1. The values from another function block are available at the function blocks data input.
2. The input event, which is connected to the input variables, arrives at the function block.
3. The execution control signals the scheduling function of the resource that an event occurred and that input data is ready.
4. After a certain time, which depends on the workload of the resource, the algorithm starts to execute.
5. The algorithm, which is chosen by the input event performs its tasks. It may change the values of the internal and output variables of the function block.
6. After the algorithm is executed, it signals the scheduling function of the resource that output data is available.
7. The scheduling function asks the execution control to generate the corresponding output event. Depending on the state of the execution control chart, the right event is selected.
8. The selected output event is generated from the execution control. This signals other function blocks, which are connected to the output data of the function block, that valid output data is available.

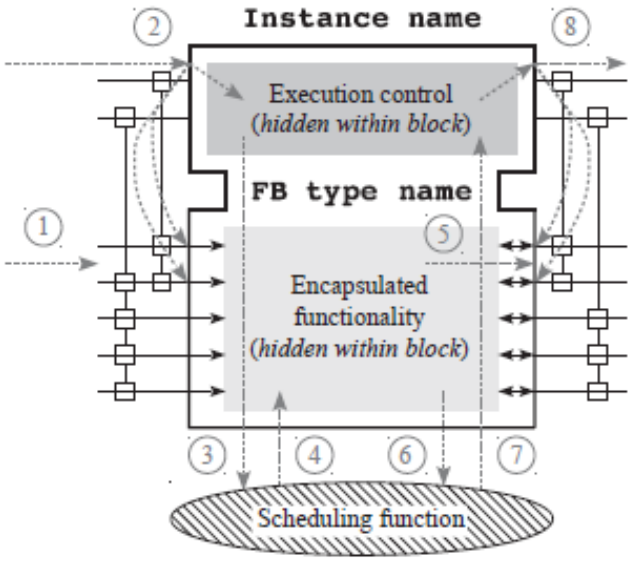


Figure 8: Execution of a basic function block [1]

2.1.4 Execution Control Chart

Which algorithms are to be executed, is determined by the execution control chart (ECC). Every basic function block has exactly one ECC. This is a finite state machine, which has states and transitions. It has an initial state, which can only switch to another state after an event arrives.

Every state can have zero or more actions. An action consists of one or zero algorithms and one or zero output events, which are fired after the algorithm has executed. An action must have at least one algorithm or one output event. So it is possible to fire events without algorithms or to execute algorithms without firing an event.

A transition has a guard condition. This can be an event followed by an optional boolean condition. The guard condition has read only access to the variables of the function block.

If an event arrives, the ECC checks all outgoing transitions in the order in which they are declared. If the corresponding condition is fulfilled, the ECC stops evaluating the outgoing transitions and enters the next state. By entering this state, the bound actions are executed in the order in which they are declared. Afterwards, the ECC checks again the outgoing transitions in the new state. This is repeated until no more conditions fit [8].

2.1.5 Example Execution Control Chart

The example in Figure 9 will be discussed here. The state transitions are listed in Table 1.

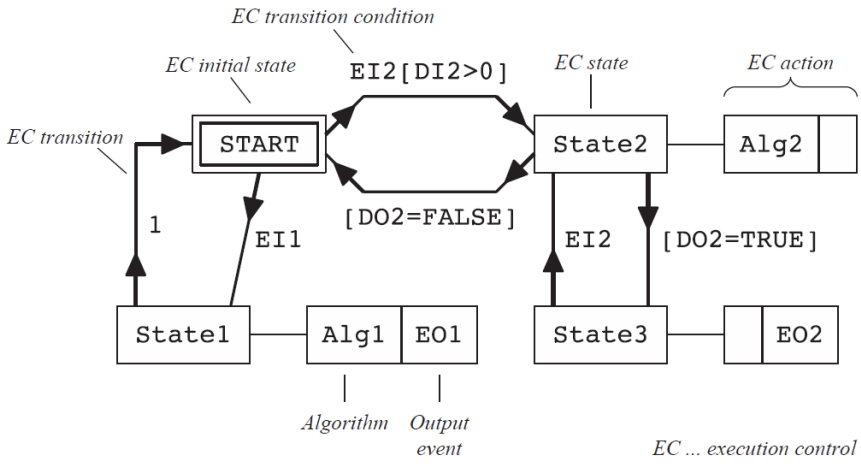


Figure 9: Execution control chart [8]

Transition	From State	To State	Condition
T1	START	State1	EI1
T2	State1	START	1
T3	START	State2	EI2 [DI2 >0]
T4	State2	State3	[DO2 = TRUE]
T5	State2	START	[DO2 = FALSE]
T6	State3	State2	EI2

Table 1: State transitions of Figure 9 [8]

First the function block has the state **START**. If an input event arrives at the function block, the outgoing transitions are checked. In this case it first checks **T1**.

If the input event is **EI1**, it will enter State1 and it will not continue evaluating **T2**. As it enters **State1**, it will execute **Alg1** and throw the output event **EO1**. After executing the action, it will check the outgoing transitions of **State1**, which is in this case only **T2**. The condition of **T2** is **1**, which evaluates to true. Therefore, it enters **START** again. The outgoing transitions of **START**, which are **T1** and **T2**, are false and the ECC stops.

If the input event is **EI2**, it will evaluate **T1** to false and check the next transition, which is **T3**. The condition of **T3** is **EI2 [DI2 >0]**, which is a short notation for **[event = EI2 ∧ DI2 >0]**. As the event is **EI2**, the ECC will evaluate **[DI2 >0]**. In this case it must compare the variable **DI2** with 0. If this comparison is true it will enter the next state, and execute its corresponding actions, otherwise the ECC will stay in the state **START**. It will perform state transitions until a holding state is found.

2.2 Virtual Machines and Dynamic Compilation

To execute a program, it can either be interpreted or compiled. A compiler translates the program's source code to machine code which fits the architecture of the corresponding hardware. An interpreter executes the program's source code (or intermediate representation (IR) of it) without translating it to machine code.

To interpret IR, a virtual environment is necessary, which is able to perform the defined IR operations. This is called a virtual machine (VM), which is an abstraction of a computer. It simulates a processor and all other hardware components. With this abstraction, it is possible to develop programs in a hardware-independent way. To speed up interpreting, a VM can pre-compile some parts of the program, which is called dynamic compilation. If this compilation happens during runtime, it is called Just-In-Time (JIT) compilation.

The Java Virtual Machine [4] (JVM) applies dynamic compilation for the language Java [10]. The JVM does not execute Java source code itself, it uses an IR of it, the so called Java bytecode in a class file. This class file is executed by the JVM.

If an IEC 61499 function block should be executed on the JVM, it has to be compiled to Java Bytecode. Also the behaviour of the IEC 61499 system architecture must be implemented. But this would be very challenging, because optimizing the source code of a function block needs a lot of effort. So it should be implemented on the Graal VM [6], which offers a framework for implementing new languages and a compiler that performs optimizations out of the box.

2.2.1 Graal VM

The Graal VM is a modified version of the Java HotSpot VM from Oracle [6]. It uses the Graal compiler next to all other components from the Java HotSpot VM. This compiler is written in Java and produces highly optimized code. It provides an infrastructure (Figure 10) to implement new languages which can use the services provided by the host VM, such as dynamic compilation, memory management and so on.

2.2.2 Truffle

The language implementation framework Truffle [6] is built on top of the Graal VM. It provides an interface to implement new languages as Abstract Syntax Tree (AST) interpreters [5]. Truffle ASTs are written in Java. This has several benefits, like Java data types, exception handling or the system library from Java. Every node of the AST must be implemented as Truffle node with a specific *execute* method. Listing 1 shows such a node. The guest language represented as AST, is then interpreted by

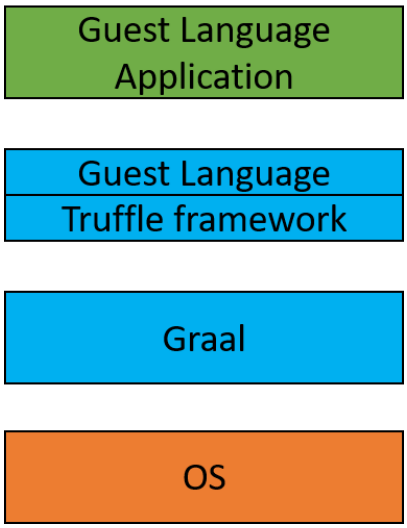


Figure 10: System architecture of a language implementation on top of Graal

Truffle. As interpreting an AST can be slow, truffle provides several mechanisms to optimize the execution.

A node can have multiple execute methods, depending on the data types of the operands. Normally a node has a specific operation for every possible operand type. The node in Listing 1 can handle the addition of either two *int* or *long* values.

With the Truffle DSL [11] it is possible to define different specialisations for specific operand types. For every node the DSL generates a new Truffle node, that has different states, depending on the specialization. For the example in Listing 1 the DSL processor will generate a node of type *AddNodeGen* inherited from the type *AddNode*. The generated node will have three different states: *uninitialized*, *long* and *int*. The DSL also generates code for handling the state changes. The first execution is performed in an uninitialized state. In this state the node will execute both child nodes and expects either two *long* or two *int* values as return type. Afterwards the node switches to the state depending on the return type of the child nodes.

It determines the right execute state of the node with guards that are integrated in the generated code. In the Truffle DSL, a guard is described by a method with the annotation *@Specialisation* specifying the expected operand types. If the guard is violated, the AST rewrites itself and replaces the node with a more generic node.

In Listing 1, for example, the specialisation of *executeInt* can be violated by an *ArithmeticException*. Then the node will rewrite itself with a more generic node, which is in this case a node that returns a *long* value. This is useful in a dynamic language where the type of the node is not statically known. This technique is called **tree rewriting** [5].

Dynamic languages can highly benefit from this approach, but for statically typed languages like IEC 61499 ST the advantage is limited. The rewrites are performed in interpreter mode. The nodes are profiled and analysed. The AST becomes stable, if the interpreter does not rewrite nodes any longer, which will trigger the Graal compiler. The compiler inlines all execute methods of the AST and optimizes the code. If, nevertheless, a rewrite is necessary, the VM switches back to interpreter mode and will perform de-optimizations. Afterwards, the VM starts profiling the AST again and will once more trigger the compiler if the AST becomes stable. Truffle also profiles branch probabilities in conditions or it will count the executions of a loop. This information is also used for the Graal compiler to optimize the code. From this kind of optimization IEC 61499 ST can benefit, too.

```
1  @NodeChildren(value = {@NodeChild("leftNode"),
2  @NodeChild("rightNode")})
3  public abstract class AddNode extends Node {
4      @Specialization(rewriteOn = ArithmeticException.class)
5      public int executeInt(int left, int right) {
6          return Math.addExact(left,right);
7      }
8      @Specialization
9      public long executeLong(long left, long right) {
10         return left + right;
11     }
12 }
```

Listing 1: Truffle node for addition

The usage of local variables within a function block can be optimized with Frames. A Truffle *Frame* is an object, provided by the framework, which holds an array containing the values of the local variables. The Graal compiler ensures that local variables can be accessed fast [6]. It performs an escape analysis and guarantees the program to be in static single assignment (SSA [12]) form. The SSA form allows the Graal compiler to perform standard optimizations, like constant folding or global numbering on the IR [6]. The usage of *Frame* is optional, but recommended to get optimized code. A function block needs a *MaterializedFrame* for its internal variables and it needs a *Frame* for every algorithm. Whereas a *MaterializedFrame* lives in the heap and is used for fields or global variables, *Frame* lives in the stack and can be used for local variables.

2.2.3 Substrate VM

With the Graal VM it is possible to compile Java Code ahead-of-time (AOT) to a stand-alone executable unit [13]. The so called native image contains the Truffle im-

plementation, the JDK and all libraries and it does not run on the JVM. Instead it is executed on the Substrate VM [14]. This VM is written in a subset of Java, as AOT compilation does not support the full semantics of Java. Development of Substrate VM is an ongoing effort. The current limitations can be looked up in the repository [15]. For example, dynamic class loading or finalizers are currently not supported by the Substrate VM. But it provides the necessary runtime components (like thread scheduling, garbage collector, the deoptimization, etc.) and has **faster startup time** and a **lower memory footprint** compared to other JVMs. The Substrate VM is also optimized to execute Truffle languages.

With the Graal VM [13] native image generator it is possible to deploy Substrate VM [13]. It uses a strict static code analysis and processes the application, the JDK and all dependencies. It then collects all used classes and the Graal compiler performs an AOT compilation of them. This can be done with a language implemented with Truffle. So it is possible to deploy a standalone VM for Truffle languages, that uses the Graal JIT compiler and all of its optimizations. And there is no need to install any Java runtime.

2.3 Coco/R

The ST file, which declares a IEC 61499 function block must be parsed first, to allow execution on Graal or Substrate VM. The tool Coco/R [7] [16] is used to generate a parser and scanner for ST. It produces a recursive descent parser and a scanner from an Attributed Grammar (ATG) file, which must be in Extended Back-Naur Form (EBNF) [17], to be processed by Coco/R.

In order to produce the scanner, the tokens must be declared as terminal symbols, with the available or self-defined character set in the section *TOKENS* in the ATG files. Terminal symbols are note derived to other symbols, because they are atomic. As can be seen in Listing 2 **ident** and **digit** are terminal symbols. **Ident** can be an arbitrary sequence of characters in the range from *a* to *z*, whereas **digit** ranges from character *0* and goes to character *9*.

In contrast to terminal symbols, nonterminal symbols will be further derived to other symbols with derivation rules, called productions. Productions can contain terminal symbols and nonterminal symbols. Listing 2 shows the production of the non terminal symbol *VarDecl*. It also contains the nonterminal symbol *Type*, which can be derived to an **ident** in the production of *Type*.

```
VarDecl = ident {"," ident} ":" Type ";"
        | ident "[" digit "]" ";".
Type = ident.
```

Listing 2: Simplified grammar of IEC 61499 ST

The parser is declared with EBNF productions in the corresponding section of the ATG file. It is possible to extend EBNF productions with semantic actions, which are executed by the parser. For example, the production of the variable declaration in Listing 2, could be extended with semantic actions to check the type, depending on the context of *VarDecl*. This is called attributed grammar and is illustrated in Listing 3. The code beetween **(.** and **.)** is directly inserted into the generated parser code. The attributes beetween < and > specify the parameters within the production. The parameters are then available in the inserted code and can be processed by the parser.

A so-called LL(1) parser just knows the next token. If two alternatives, within a production, are starting with the same terminal symbol, it cannot decide which alternative to choose. In this case the grammar is not LL(1). If the grammar is not LL(1), Coco/R can still resolve these conflicts with so-called resolvers. With Coco/R it is possible to make an arbitrary lookahead in the token stream. So the parser is able to look **k** symbols ahead and can then choose the alternative based on the lookahead. An example of an LL(1) conflict can be seen in the production *VarDecl* within Listing 2. Both alternatives are starting with an **ident**. To tackle this conflict, the resolver **IF(*isArray()*)** is placed before the alternative starts. This resolver checks if the next token after ident is the terminal symbol "[" and will enter the correct alternative. Thus, the resolver must be implemented in the corresponding language of Coco/R. The conflict can also be solved by factorizing the productions, but for demonstration purposes, the resolver is used in the example.

```
VarDecl<int typedef> =  IF(isArray()) ident "[" letter "]" ";"
                        | ident {"," ident} : Type<out int type>
                        (. if(typedef!=type) error(); .) .
```

Listing 3: Simplified grammar of IEC 61499 ST

An example of an LL(1) conflict in this work, solved by a resolver, is discussed in section 4.1.1.

3 Architecture

This chapter will describe the overall architecture of the implementation and the interaction between the components. It will explain how to integrate Graal into the IEC 61499 system model as device, which is able to perform communication to other devices with the IEC Runtime. This section declares how a function block represented as ST file can be parsed and executed on the Graal VM and how to handle IEC data types.

3.1 Graal VM as IEC Device

Figure 11 shows how to integrate Truffle into the IEC 61499 system model with the Graal VM as resource within the device. The IEC Truffle device has some limitations. It only supports the execution of a single basic function block on exactly one resource, which is mapped to one device. But it is possible to exchange data and events with other devices in the application. A simplified version of the communication protocol of IEC 61499 is implemented, to show that the event and the data flow between IEC Truffle and other devices is possible.

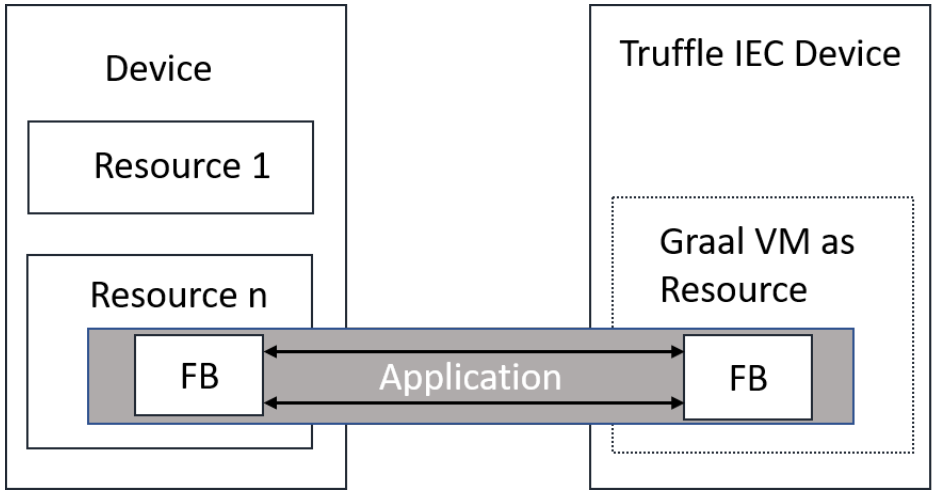


Figure 11: Architecture of IEC Truffle on the system view

3.2 Overview of the Components

Figure 12 shows an outline of the developed components in this work. First the function block represented as ST file, must be passed to the parser from the IEC runtime. The parser stores the ECC of the function block in the symbol table. This is necessary because the used names of the algorithms in the ECC states are declared afterwards in the ST file. During parsing, the algorithms are transformed to Truffle ASTs by the node builder. Afterwards, the node builder uses the collected information to verify the ECC and to merge the algorithm ASTs into one function block AST. This is explained in detail in Section 5.

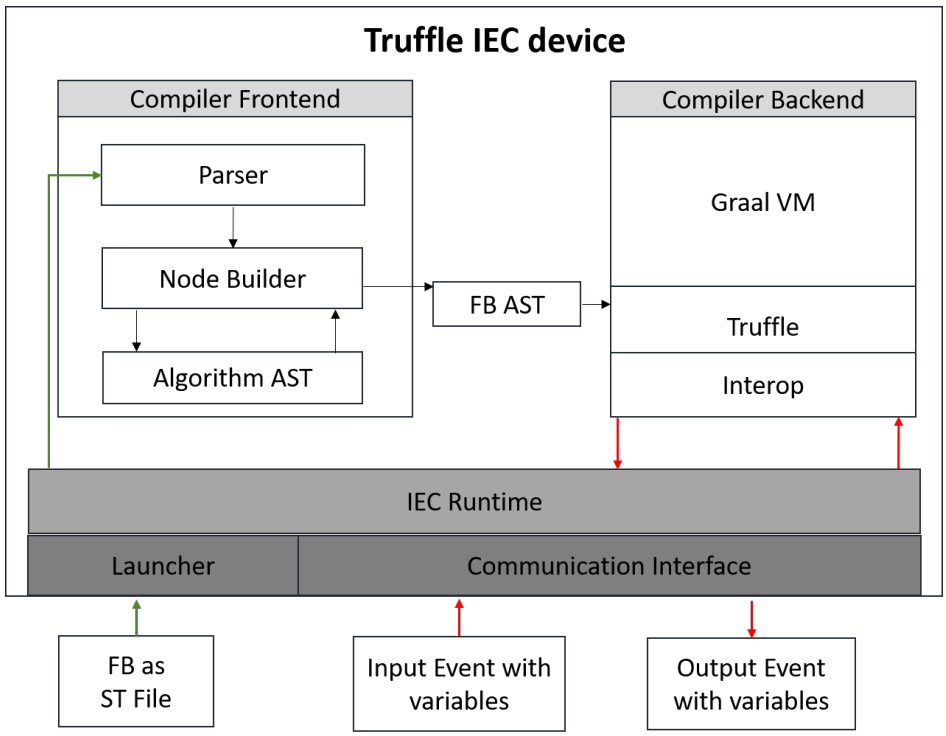


Figure 12: Overview of the components from the Truffle IEC device

The function block is then loaded into the compiler backend as Truffle AST. To execute the function block, it must be triggered with an event. The event and the data flow are marked as red arrows in Figure 12, while the loading of the ST file is marked as green arrows. As soon as an event arrives at the IEC Runtime, it will be decoded and delegated to the Truffle Interoperation Library [18] and the ECC will start to execute the corresponding actions.

3.2.1 Parser

The parser and scanner are generated by Coco/R [16] from an ATG file. As illustrated in Figure 13 the parser uses the scanner, node builder and symbol table as components.

The scanner processes the ST file and creates a token stream during the recursive descent of the parser. In parallel, the node builder creates the Truffle AST and performs the mapping from IEC data types to Java data types. Also the symbol table is built during parsing. The symbol table is necessary to perform error handling, type checks and automatic type casts for constants. Although IEC 61499 does not provide scopes, regarding to the standard [1], they must be considered in some cases. For example, every algorithm has its own scope. The handling of scopes within Truffle is done with the symbol table during node construction.

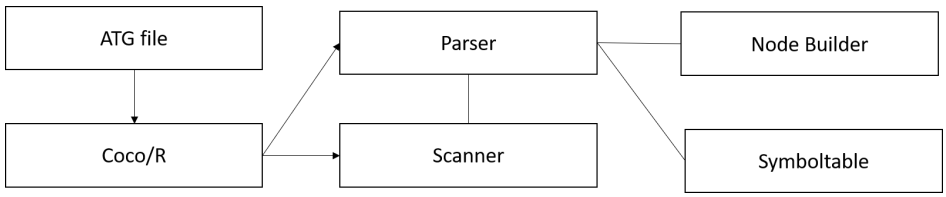


Figure 13: IECParser

3.2.2 IEC Runtime

The IEC Runtime ensures that a function block can be executed and controlled within the distributed application. It offers functionality that enables the interaction with other devices.

First, the function block must be loaded into the Graal VM as an AST. The component launcher from the IEC Runtime can handle that. It starts the runtime and creates an IEC language instance, a Truffle context. To get access to it, the Truffle Interoperation Library [18] has been introduced by the Graal developers. With this cross-language interoperability, it is possible to access the members of a program, written in the guest language, from the host language or from any other language installed on Graal. This mechanism is used in this work to manipulate the AST with Java, so that the execution model of an IEC 61499 function block will be fulfilled. After the context has been created, the launcher triggers the parser to build the AST from the source file which is loaded into the context and is ready for execution.

The IEC Runtime provides an interface, which can be used by the communication interface, to access and execute the function block. The communication interface provides an implementation for handling the network traffic between devices. It supports serialization for the incoming and outgoing messages, which must be mapped to the corresponding commands from the IEC Runtime. The implementation of the communication interface can be written in any language which is installed on the Graal VM. In this work, Java is used.

3.3 Data Types

In order to perform operations with Truffle, the data types of IEC 61499 must be mapped to Java. Table 2 shows how to map IEC 61499 data types to Java primitives. Since the IEC 61499 standard defines, that overflows are not allowed, this must be considered in this work. It has an impact on the mapping of the data types, especially on unsigned ones. So its necessary to add this overflow checks to all operations within the Truffle nodes. But the standard does not define the behaviour of an overflow. So in this work, a *Java Runtime Exception* is thrown that will abort the execution of the

function block. In real industrial systems this might not be the desired behaviour.

IEC Type	**Bits**	**Range**	**Java datatype**
BOOL	1	true or false	boolean
SINT	8	$-2^{8-1}...2^{8-1}-1$	byte
INT	16	$-2^{16-1}...2^{16-1}-1$	short
DINT	32	$-2^{32-1}...2^{32-1}-1$	int
LINT	64	$-2^{64-1}...2^{64-1}-1$	long
USINT	8	$0...2^{8}-1$	byte
UINT	16	$0...2^{16}-1$	short
UDINT	32	$0...2^{32}-1$	int
ULINT	64	$0...2^{64}-1$	long
REAL	32	single precision	float
LREAL	64	double precision	double
STRING	8	not defined	String
CHAR	8	not defined	char

Table 2: Mapping from IEC datatype to Java datatype

An IEC data type can be mapped directly, if the range of the values is the same as in Java. It is an advantage, if an IEC data type can be mapped directly to a Java data type, because the operations can be defined with simple Java operations and no boxing into Java objects is required.

IEC **LINT**, for example, fits perfectly into Java **long**. Listing 4 shows how the addition of two **LINT** values within a Truffle node is performed. This means that for the addition of two **LINT** values, the IECAddNode can switch into the state of an **LINT** execution, guarded by the parameters of the method. The node is therefore stable and Truffle will trigger the Graal compiler and execute the operation in machine code. The overflow check is performed with the *Math.addExact* method, from the Java system library. Only if an overflow happens, Graal will switch back to interpreter mode and stops the execution, after an exception has been thrown. As ST is a strictly typed language no other state changes will occur and therefore all calculations can be done in compiled code. This will lead to a higher performance.

```
1 public abstract class IECAddNodeSigned extends IECBinaryNode {
2     @Specialization
3     public long executeLINT(long left, long right) {
4      return Math.addExact(left,right);
5     }
6 }
```

Listing 4: Truffle node for addition with IEC LINT

3.3.1 Unsigned Data Types

Unfortunately, unsigned data types have other ranges, in some cases also other behaviour, than Java primitives. Therefore, unsigned operations need their own Truffle nodes. So its possible to perform different overflow checks, like in Listing 5. Also for relational operations special treatment is needed. The Java system library provides all necessary operations, for comparing and calculating unsigned values.

```
1  public abstract class IECAddNodeUnsigned extends IECBinaryNode {
2      @Specialization
3      public long executeULINT(long left, long right) {
4          long result = left + right;
5          if ((left >>> 63 == 1 || right >>> 63 == 1)
6                  && result < Long.MAX_VALUE)
7                  throw new ArithmeticException();
8          return result;
9      }
10 }
```

Listing 5: Truffle node for unsigned addition with IEC LINT

4 Compiler Frontend

This chapter will provide an overview of the Compiler Frontend and will explain how to process a function block written in ST. An overview of the IEC 61499 ST grammar and its LL(1) conflicts is given. The structure of the symbol table is described in this section. It also explains how to insert the ECC, enums and namespaces into the symbol table

4.1 Grammar

The EBNF grammar for IEC 61499 Structured Text is defined in the official standard [1]. Listing 6 shows a simplified version of it. Some parts are skipped, other parts are simplified. It just shows the declaration of a basic function block, so that the reader gets an overview of how this could look like. However, a more detailed version than Listing 6 is implemented, which still does not support the full semantics of IEC 61499, like service or composite function blocks. But it supports the semantics of namespaces, enum definitions and a basic function block. The attentive reader should have noticed that the productions **VarDecls**, **VarDeclInit**, **Expression** and **Statement** are not declared in Listing 6. This is because variable declarations, expressions and statements are part of the IEC 61131-3 grammar [2]. So the IEC 61499 grammar includes some parts of the 61131-3 grammar.

The production **FbDecl** defines the declaration of a function block. Every function block must start with the keyword **FUNCTION_BLOCK**, followed by an identifier, which is the name of the function block. Afterwards the interface will be declared with the production **FbInterface**. It contains the declarations of the event and the variable inputs and outputs. The productions **EventIn** and **EventOut** are defining zero or more events, represented by an identifier which is the name of the event. It can be followed by the keyword **WITH**. The identifiers after the keyword are representing variables. These are the variables which must be written after an event arrives or is sent. The variable names must be verified, after parsing, because they are not declared at this part in the grammar. After the **FbInterface** production, zero or more internal variables can be declared with the production **VarDecls**. These variables are only visible inside the function block. **VarDecls** is followed by **EccDecl**, which declares the ECC. Every ECC can have zero or more states, as shown in the production **EccState**. The state has an identifier and can be followed by zero or more actions. These are the actions which are executed when the ECC enters the corresponding state. The actions in **EccAction** are containing the executed algorithm names and the corresponding output events as identifiers. Normally the algorithm name is the first identifier followed by an arrow which points to the output event. If only one identifier is present, it is either the algorithm name or the output event. After the states are declared, also the

state transitions must be defined. This happens in the production **EccTrans**. Every transition has a guard condition, which must be fulfilled for a successful state change. The production **GuardCond** declares, that either an event or a boolean expression can trigger the state change. This identifier in **GuardCond** must be checked by the symbol table, whether it is an event or not. After the ECC is declared, zero or more algorithms will follow. After the algorithms have been processed, the ECC actions must be verified using the symbol table, because some algorithm names might have been assigned to a state.

```
1  FbDecl       = "FUNCTION_BLOCK" ident
2                  FbInterface {VarDecls} EccDecl [AlgoDecl {AlgoDecl}]
3                 "END_FUNCTION_BLOCK" .
4
5  FbInterface = EventIn EventOut
6                 ["VAR_INPUT"  {VarDeclInit";"} "END_VAR"]
7                 ["VAR_OUTPUT" {VarDeclInit";"} "END_VAR"].
8
9  EventIn      = "EVENT_INPUT"
10                {ident ["WITH" ident {"," ident}]";"}
11                "END_EVENT".
12
13 EventOut     = "EVENT_OUTPUT"
14                {ident ["WITH" ident {"," ident}]";"}
15                "END_EVENT".
16
17 EccDecl      = "EC_STATES"       {EccState} "END_STATES"
18                "EC_TRANSITIONS" {EccTrans} "END_TRANSITIONS".
19
20 EccState     = ident (";"|":" EccAction {"," EccAction}";").
21
22 EccAction    = ident ["->" ident ] ";".
23
24 EccTrans     = ident "TO" ident ":=" GuardCond ";".
25
26 GuardCond    = ident ["[" Expression "]"] | "[" Expression "]".
27
28 AlgoDecl     = "ALGORITHM" ident "IN" "ST" ":"
29                {VarDecls} Statement {Statement}
30                "END_ALGORITHM".
```

Listing 6: Simplified grammar of IEC 61499 ST

4.1.1 LL(1) Conflicts

Unfortunately the grammar has some LL(1) conflicts. Most of them are solved with factorization. But the conflict in the *CaseSel*, depicted in Listing 7, needs a resolver.

At the beginning of the repetition *Statement* in *CaseSel*, an identifier can either represent the beginning of a Statement (i.e., a Variable) or a constant name as the beginning of the next *CaseSel*.

```
1  CaseStatement = "CASE" Expression "OF"
2                  CaseSel {CaseSel} ["ELSE" {Statement}]
3                  "END_CASE".
4
5  CaseSel       = ConstExpr {"," ConstExpr } ":" Statement
6                  {IF(isStat()) Statement}.
7
8  Statement     = [
9                       Variable ":=" Expression
10                      |
11                      SelStatement
12                      |
13                      IterStatement
14                 ]
15                 ";".
```

Listing 7: CaseStatement in the IEC 61499 ATG

As shown in line 35 of Listing 8, **y** is a constant name, but it could also be a variable name. The parser does not know that, because it is an *ident* in both cases. This conflict is resolved by **IF(isStat())**, which returns true if the next token is not a constant name. Listing 9 shows the resolver. It first checks if the current lookahead token is in a legal start set of *Statement*, which is:

First(Statement) :={ **ident**, **";"**, **IF**, **CASE**, **EXIT**, **CONTINUE**, **FOR**, **WHILE**, **REPEAT** }

If the lookahead token is a legal start of *Statement* and if is not an *ident*, the resolver returns true and the parser enters the *Statement* production. Otherwise, the resolver needs to search the symbol table with the current lookahead token (i.e., the ident). If the symbol table returns a namespace element, the resolver needs to perform a **k** symbol lookahead with the method *getQualifiedName*, which resolves the namespace name. In **a.b.y**, for example, **a** and **b** are namespace elements and **y** is a variable. The current lookahead token is in that case just **a**, which is a namespace element. So the *getQualifiedName* method builds the fully qualified name, by looking ahead until **y** is reached. The resulting fully qualified name is then returned to the resolver. The resolver searches the symbol table again and checks if the returned element is a constant or an enum. If the check succeeds, the resolver returns false and the parser will enter

CaseSel rather then *Statement*.

```
FUNCTION_BLOCK Basic1
    EVENT_INPUT
        ADD_IN WITH num1, num2;
        SUB_IN WITH num1, num2;
    END_EVENT
    EVENT_OUTPUT
        ADD_OUT WITH result;
        SUB_OUT WITH result;
    END_EVENT
    VAR_INPUT
        num1 : INT := 1;
        num2 : INT := 1;
    END_VAR
    VAR_OUTPUT
        result : INT;
    END_VAR
    EC_STATES
        START;
        DO_ADD : DO_ADD -> ADD_OUT;
        DO_SUB : DO_SUB -> SUB_OUT;
    END_STATES
    EC_TRANSITIONS
        DO_SUB TO START := TRUE;
        START TO DO_SUB := SUB_IN;
        START TO DO_ADD := ADD_IN;
        DO_ADD TO START := TRUE;
    END_TRANSITIONS
    ALGORITHM DO_ADD IN ST:
        VAR
            CONSTANT y : INT := 0;
        END_VAR;
        CASE result OF
            1:
                result := num1 + num2;
            y:
                result := 1;
            ELSE
            result := 0;
        END_CASE;
    END_ALGORITHM
    ALGORITHM DO_SUB IN ST:
        result:= result - num1 - num2;
    END_ALGORITHM
END_FUNCTION_BLOCK
```

Listing 8: Basic function block with case statement in Structured Text

```
boolean isStat() {
    if (isfirstStat(la.val)) {
        if(la.kind != _ident)
            return true;
        Obj o = tab.find(la.val);
        if(o.kind == Obj.Kind.NameSpace){
            String qualName = getQualifiedName();
            o= tab.find(qualName);
        }
        if(o.kind == Obj.Kind.Con || o.type.kind == Struct.Kind.Enum)
            return false;
    }
    return true;
}
```

Listing 9: LL(1) resolver for case selection

4.2 Symbol Table

To resolve LL(1) conflicts, a symbol table is used. It is also used to build and verify the ECC as AST and to generate the universe with predefined data types and functions. All declared names are entered into the symbol table and are looked up during parsing while the AST is built. The symbol table is also used to deal with namespaces and enums and to perform type checks and implicit casts.

The symbol table is built up with 4 classes, as illustrated in Figure 14 (Tab, Scope, Struct and Obj) :

With the class **Tab** it is possible to insert all declared names and to open and close scopes. **Tab** is initialized as field in the parser. Before parsing starts, the universe which contains basic data types and operations, is build up in the symbol table. It can build up the data structure of the function block by inserting the elements with their corresponding kinds.

The classes **Scope**, **Struct** and **Obj** are holding all information and are used as nodes in the data structure.

Scope is responsible for keeping the declared elements of a function block within separate scopes. The only possibility for scopes in an IEC 61499 function block are algorithms. So every algorithm can declare its own variables, which are not visible from other algorithms in the same function block. Line 30 in Listing 9, for example, shows the declaration scope of variables for the **DO_ADD** algorithm. The algorithms

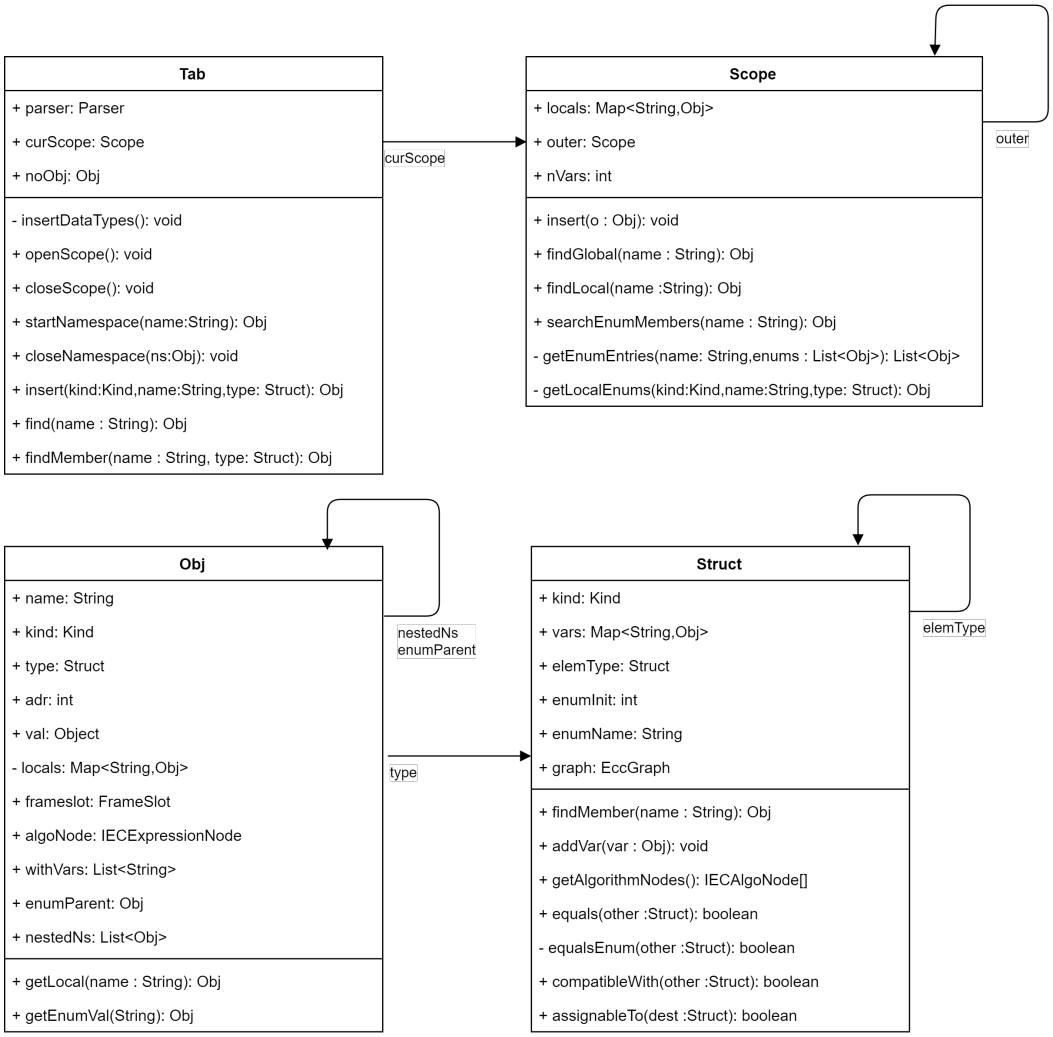


Figure 14: Class diagram of the symbol table

have access to their outer scopes, which are in that case the input and output variables. Every scope holds a map of all locals, objects and a reference to its outer scope. The outermost scope holds the universe. In contrast to other programming languages, the outermost scope is not used to store global variables, because IEC 61499 does not allow global variables. But it can be used to store namespaces, functions, other function blocks, or enums. **Scope** also provides methods to search through its members and it holds a map of all objects declared in the scope, called locals.

Obj represents a declared name in the program. An object can have one of the following kinds: *Constant*, *Enum Constant*, *Variable*, *Type*, *Namespace*, *Algorithm*, *Input Variable*, *Output Variable*, *Input Event* or *Output Event*. It also has a type, which can be a basic data type, function block or enum. The objects are stored in the different scopes of the symbol table. Every object stores a Truffle *FrameSlot*, which is needed for Truffle to specialize reads and writes. Truffle provides a *Frame* for guest languages, to store local variables in it, which are addressed by the *FrameSlot*. Every scope has its own Truffle *Frame*. The field for the Truffle *FrameSlot* is just used if the object is a variable. If the object is an event it also stores an array of strings, which contains the bound variable names. These names must be verified later, as they are declared afterwards in the ST file. Moreover, a namespace object has its own fields, to deal with different scopes by using namespaces. An object can also have local elements. For example, if it is an algorithm, it can have its own variable declarations. The local elements are stored in the field *locals*, which is a map where the name of the element is used as the key. If the object is a constant, its value is also stored in the Obj node.

Struct describes the type and the structure of an object. Like Obj, a Struct can have several kinds which are: *Fb49*, *Fb31*, *None*, *Enum*, *Arr* or one of the basic data types like *BOOL* or *INT*. It also has additional fields depending on its kind. If the structure is of the kind *Fb49* it holds an instance of the ECC graph, the algorithms, events and variables. If it is an array it also holds the element type of it, which is another Struct. Therefore, it is possible to build multi-dimensional arrays. In case of an enum, also the enumeration names must be held within the structure. The class **Struct** also provides methods for checking the type compatibility with other structs. For example, if it is an array it must also check the corresponding element types in it.

Figure 15 shows the symbol table of the ST file from Listing 8. The symbol table represents the program after the parser processed line 30. In this state, the parser has just added the constant y to the symbol table. Scope 2 is the current scope. It is the scope of the algorithm **DO_ADD**. The outer scope is scope 1. It belongs to the function block *Basic1* which has a struct as its type. This struct has the kind FB49 and holds the corresponding events and variables from the function block. Every event

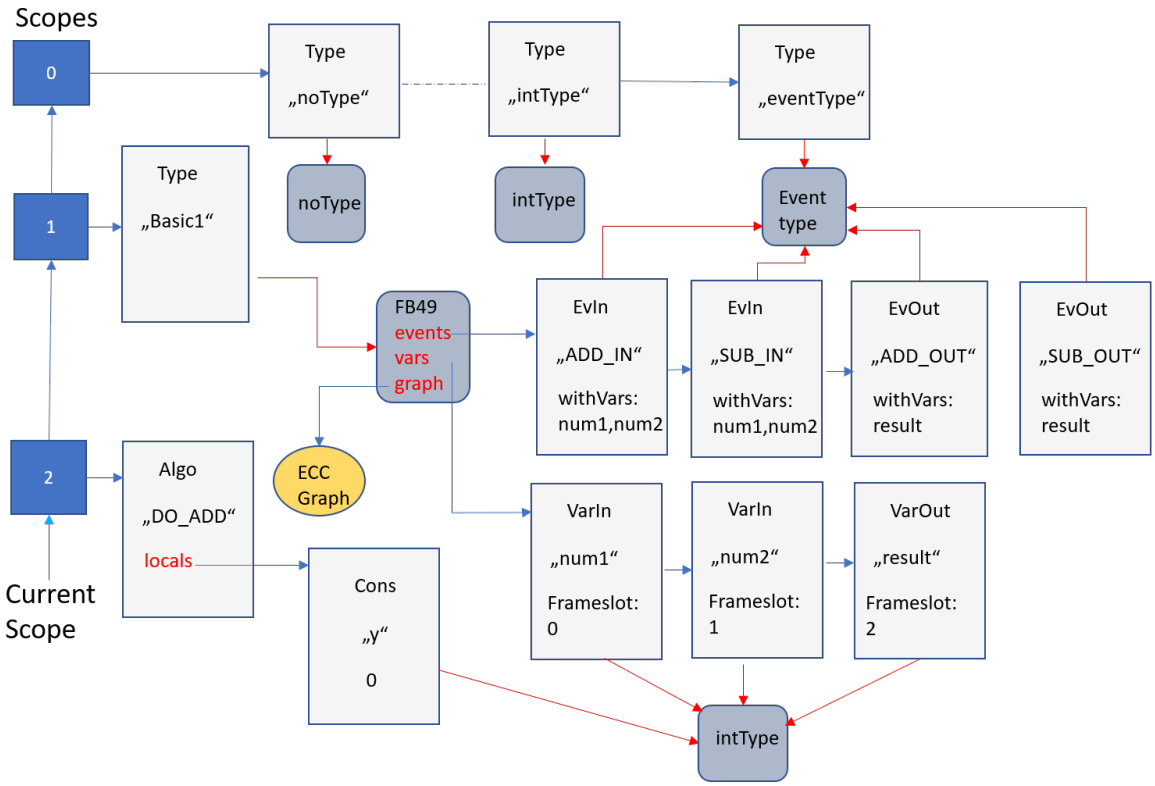


Figure 15: Symbol table of Listing 8 (light grey boxes are Obj nodes, dark grey boxes are Struct nodes, and blue boxes are Scope nodes)

object holds an array of strings in the field *withVars*, as one can see in the example Listing 8 in line 3. The event *ADD_IN* is bound to the variables *num1* and *num2*. This information is now stored in the object and must be verified after the parser has collected the input variables. Since these are not the only names that need to be verified, the verification will take place during the ECC construction. The names could also be verified after the input variables have been declared. But in order to get a cleaner and simpler software architecture, there is just one verification phase at the end of parsing. The outermost scope 0 holds the elements of the universe, which means all predefined basic data types. In addition, there is a struct *noType* for avoiding Java null pointer exceptions and a predefined event type.

4.3 ECC

The field graph is attached to a struct node to add the execution control chart to the symbol table. The ECC is represented with 5 classes in the symbol table as illustrated in Figure 16:

EccGraph: It is used to hold the states and transitions and must be created, when the parser enters the **EccDecl** production. This can be done in the attributed grammar file as shown in line 1 of Listing 10. The object *fb*, which represents a function block

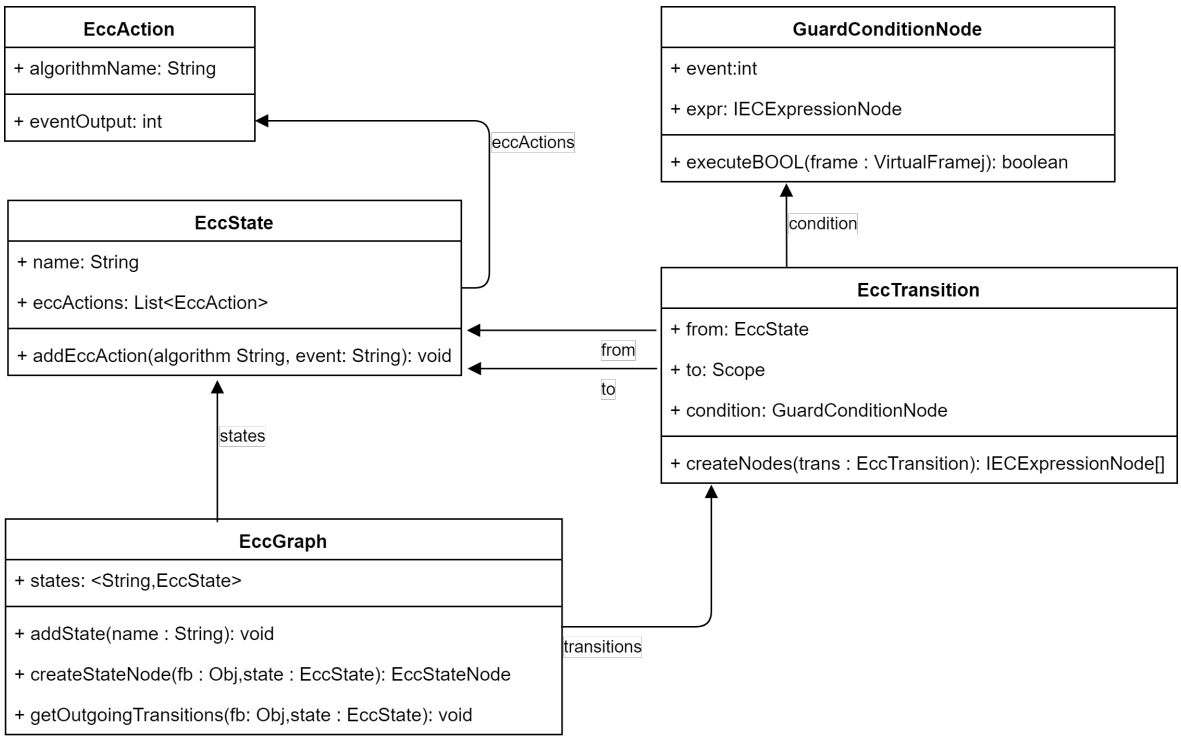


Figure 16: UML diagram of the ECC representation in the symbol table

object in the symbol table, is passed as a parameter to the production. An **EccGraph** object is created and installed in the designated field of the *fb* object.

EccState: The **EccGraph** object is passed to the **EccState** production as a parameter. The corresponding states are added to the **EccGraph** object. It is intended that state names are not inserted into the symbol table as declared names, because due to the IEC 61499 standard it is allowed to use variables which have the same name as a state name. This means that states cannot be accessed within an algorithm. But the state name must be checked in the *addState* method of the **EccGraph**, because a state cannot be declared twice.

EccAction: In order to construct a state successfully, the corresponding actions, which are fired after the state is entered, must be added to the graph. In line 15 of Listing 10, the production **EccAction** uses an **EccAction** object as an output parameter. To construct an EccAction object correctly, the class provides a static *addEccAction* method from **EccState**. This method checks whether the String algorithm or event is null or not. If the *String* event is null, the action does not know if the first identifier is an event or an algorithm. So with the creation of the object it just can check the symbol table if the declared name is an event, because algorithms are not declared yet. If it is an event, the field event in the **EccAction** object is set to the corresponding name. Otherwise, the algorithm name field is set with the name. It must be checked later, if this is really a name for an algorithm.

EccTransition: The transitions are collected in the production *EccTrans*. The two identifiers in the production are representing the from and the to state. It must be checked whether the states have been declared in the graph or not. If they are declared, the transition is added to the **EccGraph** object, passed by the parameter. Furthermore, the corresponding **GuardConditionNode** is added to the transition.

GuardConditionNode: The production **GuardCond** on line 29 of Listing 10 shows the creation of the corresponding node. In contrast to the other classes of the ECC, the **GuardConditionNode** can be created directly as Truffle Node, as the used events within the node are already declared at this point during parsing. So the node builder first checks if there is an ident declared. If so, it must check if the declared identifier is found in the symbol table as an Input Event. Otherwise, an error message must be thrown by the parser. It must also be checked if the **IECExpressionNode** object, which is returned by the production **Expression** is a boolean expression. This can be done by checking if the type of the node is boolean.

```
1  EccDecl<Obj fb>     =                  (.fb.type.g = new EccGraph();.)
2  "EC_STATES"      {EcState<fb.type.graph>} "END_STATES"
3  "EC_TRANSITIONS" {EcTrans<fb.type.graph>} "END_TRANSITIONS".
4
5  EccState<EccGraph graph>     =
6  ident                                  (.EccState s =
7                                           new EccState(t.val);.)
8  (";"|":" EccAction<out EccAction a>    (.s.addAction(a);.)
9  {
10       "," EccAction<out a>             (.s.addAction(a);.)
11 }
12 ";")                                   (.graph.addState(s);  .)
13 .
14
15 EccAction<out EccAction a>   =        (.String algo=null, ev=null;.)
16 ident (.algo=t.val;.) ["->" ident     (.  ev=t.val;.) ] ";"
17                                        (.a = EccAction.create(algo,ev).)
18 .
19
20 EccTrans<EccGraph graph>     =        (.EccState f =null, to = null;.)
21 ident                                 (.f   = graph.getState(t.val);.)
22 "TO" ident                           (.to = graph.getState(t.val);.)
23 ":=" GuardCond<out Node g>
24                                        (.graph.addTrans(from,to,g).) ";".
25
26 GuardCond<out Node g>          =      (.String ev=null;.)
27     (
28        ident                           (.ev=t.val;.)
29        ["[" ExpressionGuardCond<out Node e> "]"]
30        |
31        "[" Expression<out Node e> "]"
32     )
33                                        (.g=builder.createGuard(ev,e);.)
34     .
```

Listing 10: Attributed grammar for ECC declaration

4.4 Enums

Listing 11, lines 2 to 5, shows how to declare an enum as a new data type in an ST file. The behaviour of enums is declared in the IEC 61131-3 standard [2] and was taken over into IEC 61499. An enum must be declared with its name and its values, represented as identifier list in the type declaration section of the program. Additionally, it is possible to declare an initialization value for the enum type. If no initial value has been assigned, the first identifier is used. To declare a variable as enum, it must be declared in the corresponding variable declaration section in a function block, as can be seen in

lines 11-16 of Listing 11. It must be declared as a variable with the user-defined enum type.

```
1  TYPE
2  TEnum1 : ( AAA, BBB, CCC );
3  TEnum2 : ( CCC, DDD, EEE ) := EEE;
4  TEnum3 : ( FFF, EEE, GGG ) := EEE;
5  TEnum4 : ( HHH, III, JJJ, EEE ) := TEnum4#EEE;
6  END_TYPE
7
8  FUNCTION_BLOCK Basic1
9  ...
10     VAR
11     ev1 : TEnum1;
12     ev2 : TEnum2;
13     ev3 : TEnum3;
14     ev4 : TEnum4;
15     ev5 : TEnum2 := DDD;
16     ev6 : TEnum4 := TEnum4#JJJ;
17     (* AAA : TEnum1; This declaration would signal an
          error *)
18     END_VAR
19 ...
20     ALGORITHM evalEnum IN ST :
21     CASE ev2 OF
22     TEnum2#EEE:
23     ev1 := AAA;
24     DDD:
25     ev2 := TEnum2#CCC;
26     ELSE
27     ev2 := DDD;
28     END_CASE;
29     END_ALGORITHM
30 ...
31 END_FUNCTION_BLOCK
```

Listing 11: Enum Declaration

Enum values can be directly accessed by their name, but only if the name is unique within the ST program. Otherwise the value must be accessed with its fully qualified name. The case statement in line 21 shows an example for the enum variable *ev2*. In line 25 the value *CCC* is assigned to *ev2*. As *CCC* is also declared in TEnum1 it must be accessed with its fully qualified name which is *TEnum2#CCC*. The value *DDD* can be assigned to *ev2* with its short name, because *DDD* only occurs in *TEnum2* and is therefore unique, as displayed in line 27. This check for uniqueness must be implemented in the symbol table. It must be checked whether the enum access is valid

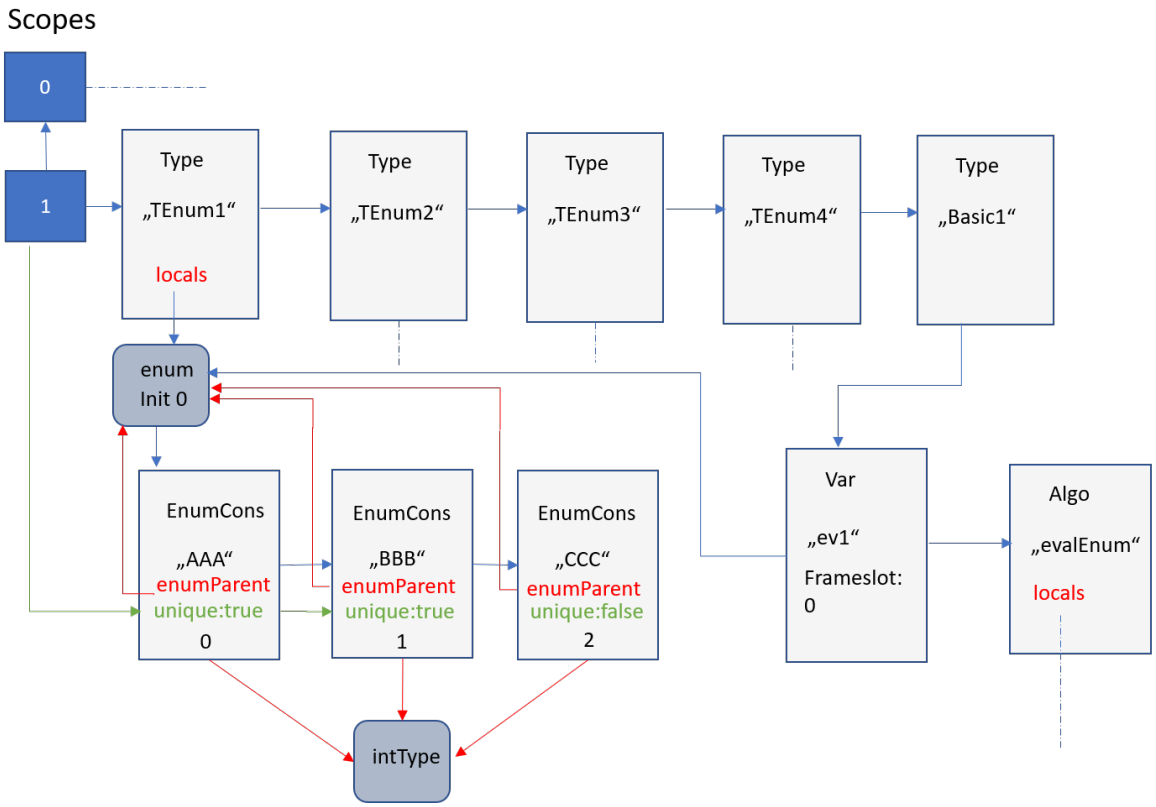


Figure 17: Symbol table of Listing 11

or not. As the IEC 61131-3 standard [2] only defines the behaviour and not the implementation, it is free to choose how to implement enums. In this work, enums are implemented as *Obj* nodes of kind *Type* and their corresponding *Struct* is an enum kind.

A resulting issue is represented in line 17. The declared variable **AAA** is already a unique enum value and therefore ambiguous. The official standard does not define how to handle that kind of situation [2]. In this work, we assume, that it is not allowed to declare a variable that has the same name as a unique enum value. Another possibility to handle that would be to force the function block developer to use the fully qualified name of the enum in that case.

The symbol table of Listing 11 is displayed in Figure 17. First, the user-defined enum types must be collected by the symbol table. They are inserted in the same scope as the function block, regarding to the example scope 1. Scope 0 is reserved for the universe. Every enum structure stores its initial value as a number. This number represents the index of the enum member. For example, the object *TEnum1*, which was declared in line 2, has the value **AAA** as its initial value as this is the first identifier. Therefore, index 0 is stored. If a new enum variable should be declared, like in line 11 of Listing 11, the parser needs to look up the initial value in the symbol table. This is just the case if the parser has not collected an initial value during variable declaration, like the

variable *ev1* in line 11.

The symbol table must also check whether an enum value is unique or not, unless it is not declared with its fully qualified name. In order to handle that, *Scope* holds a map with all enum entries, which needs to be maintained during the declaration of the enum types, depending on the uniqueness of one entry. So if a declared enum entry is not unique, like the value *CCC* of type *TEnum2* in line 3, it must be marked as not unique and deleted from the scope's enum map. But it is not deleted from the *TEnum2* object members.

If a declared name is used outside the variable declaration section, it must be searched in the symbol table. If an enum entry is accessed with its short name, like in line 23, the symbol table is searched for **AAA**. So first the symbol table checks the locals of the current scope. If no local object is found, the symbol table checks the members of the enum objects in the scope. The entries are cached in a separate map. The found object has a reference to its enum parent, to enable type checks. If the enum entry is accessed with its fully qualified name, like in line 25 **TEnum2#CCC**, the symbol table first searches **TEnum2** in the current scope and afterwards looks for **CCC** in the object members.

4.5 Namespaces

A namespace is a language element combining different entities of a program to one common scope. It starts with the keyword **NAMESPACE**, followed by an identifier representing its name and ends with the keyword **END_NAMESPACE**. Namespaces can be nested. Listing 12 shows an example of a function block, which uses namespaces. **TEnum1** in line 2 is declared in the global namespace. The elements of the global namespace can be accessed without a prefix. Line 2 shows the declaration of a nested namespace. The elements of **TEnum1** can be accessed from the global namespace with the prefixes **a.b.c**. Inside the namespace **a.b.c** the elements are accessed without prefix.

```
1  TYPE
2  TEnum1 : ( AAA, BBB, CCC );
3  END_TYPE
4
5  NAMESPACE a.b.c
6      TYPE
7      TEnum2 : ( CCC, DDD, EEE ) := EEE;
8      TEnum3 : ( FFF, EEE, GGG ) := EEE;
9      TEnum4 : ( HHH, III, JJJ, EEE ) := TEnum4#EEE;
10     END_TYPE
11 END_NAMESPACE
12
13 NAMESPACE d
14     TYPE
15     TEnum5 :(KKK,MMN,LLL) := MMN;
16     END_TYPE
17 END_NAMESPACE
18
19 NAMESPACE a.b
20     FUNCTION_BLOCK Basic1
21         VAR
22         ev1 : TEnum1;
23         ev2 : c.TEnum2;
24         ev3 : c.TEnum3 := c.TEnum3#FFF;
25         ev4 : c.TEnum4 := c.TEnum4#JJJ;
26         ev5 : d.TEnum5 := d.LLL;
27         out: BOOL;
28         END_VAR
29         ALGORITHM nsAlgo IN ST :
30         out := (ev1 = AAA);
31         out := (ev2 = c.TEnum2#EEE);
32         out := (ev3 <> c.TEnum3#EEE);
33         out := (ev4 <> TEnum1#CCC);
34         out := (ev5 = d.LLL);
35         END_ALGORITHM
36     END_FUNCTION_BLOCK
37 END_NAMESPACE
```

Listing 12: Namespace Declaration

4.5.1 Declaration of Namespaces

The declaration of a namespace is defined in the grammar of IEC 61499 ST, as shown in Listing 13. By looking at the grammar, the Obj *ns* is created in the method *startNs* from the symbol table. This object is needed later for closing the namespace by the method *closeNs*.

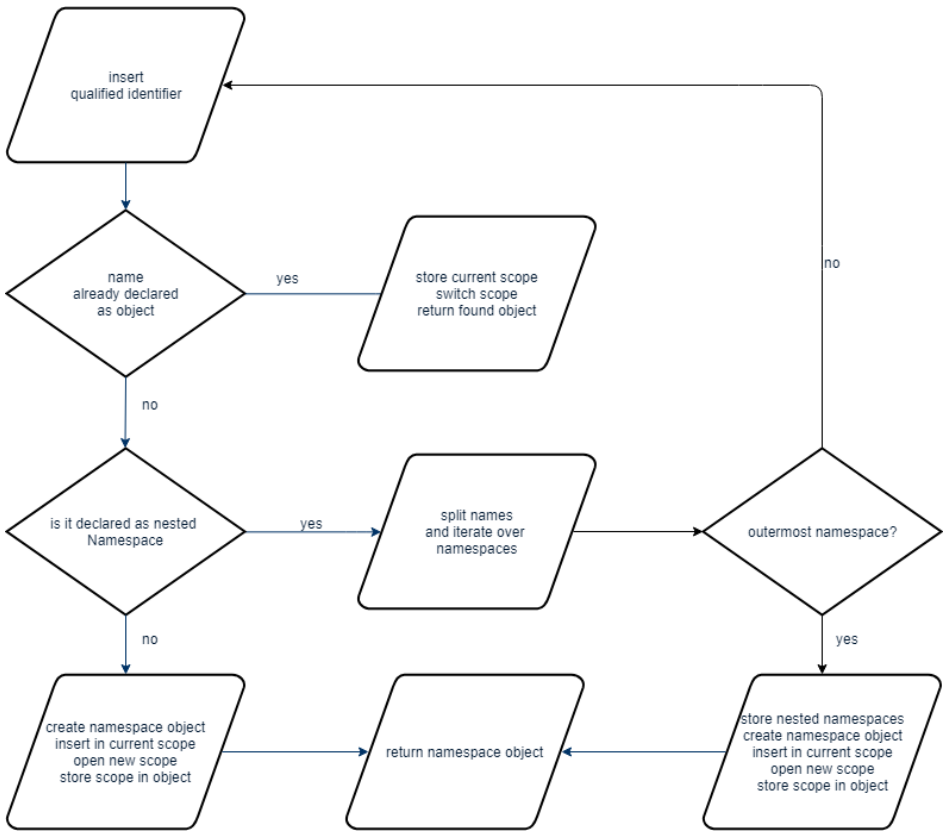


Figure 18: Creation of a namespace object

```
1 NamespaceDecl
2 =
3 "NAMESPACE"
4 QualIdent<out String q>       (.Obj ns = tab.startNs(q); .)
5 {NamespaceElement}            (.tab.closeNs(ns);         .)
6 "END_NAMESPACE"
7 .
```

Listing 13: Attributed grammar for namespace declaration

Figure 19 shows how the insertion of a new namespace element, which is declared as qualified identifier, into the symbol table works. When a new namespace is declared, it must first be checked if the namespace already exists. If so, the previously defined namespace is returned by the *startNs* method and the symbol table switches to the scope of this namespace.

Otherwise, the next step is to check whether the declared name is a nested namespace or not. If it is a single namespace, a new namespace object is created and inserted into the current scope. Also a new scope is opened and the current scope is stored to the namespace object. An example for a single namespace, the namespace **d**, is shown in line 13 of Listing 12.

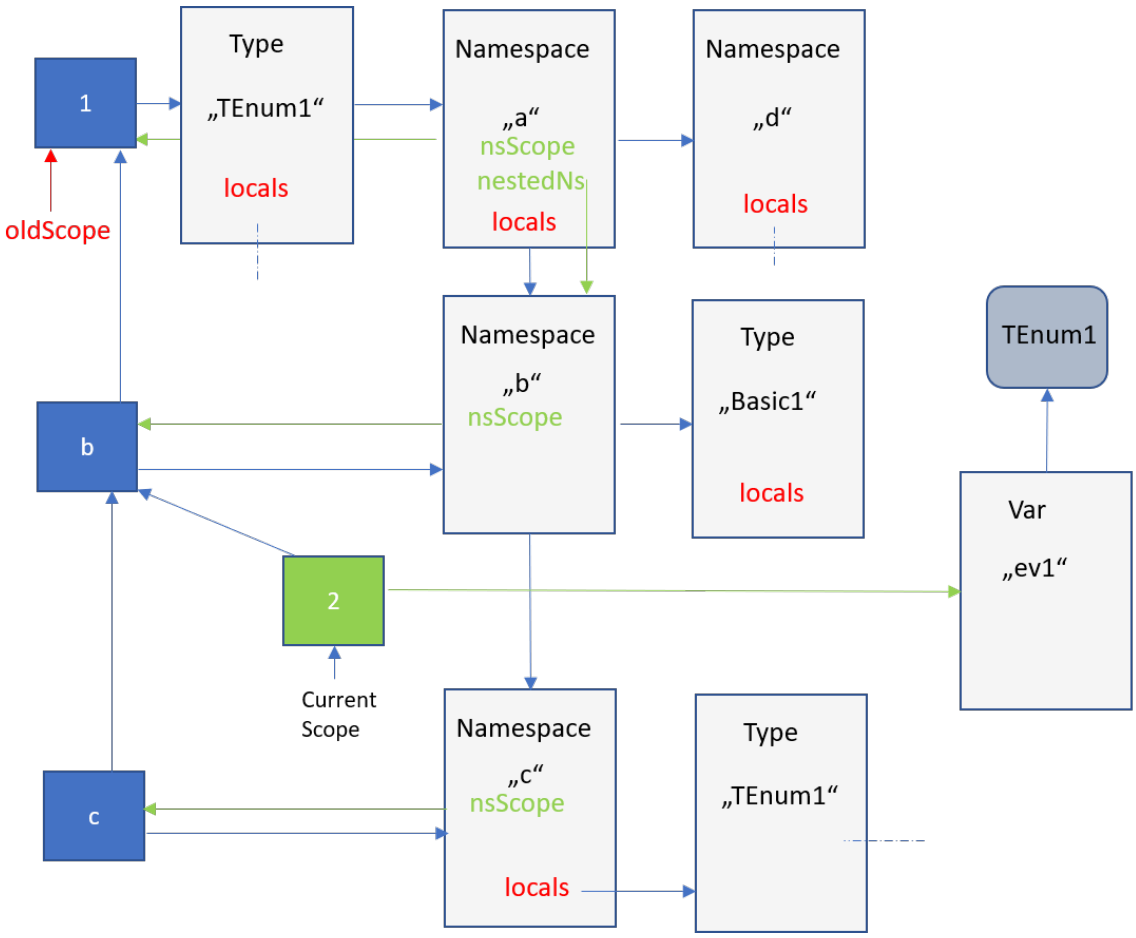


Figure 19: Symbol table of Listing 12 in line 22.

The other case is a nested namespace, like in line 5 of Listing 12. The namespace is declared as a qualified identifier, namely **a.b.c**. As **a.b.c** is a nested namespace, the individual elements, which are **a**, **b** and **c**, must be created in the right order. First **a** is created. As **a** is the outermost namespace, it must be later inserted into the current scope of the symbol table. Also the nested elements must be inserted into the symbol table and must be linked to the outermost scope. In the example, **a** has the field *nestedNs* which stores the references to the namespaces **b** and **c** as linked list. This field is used only in the outermost namespace to close it correctly. The scope of every namespace is stored in the field *nsScope*, so that it is possible to reopen a existing scope if the namespace is declared twice.

For every namespace element a new scope must be opened. This means that the scopes must be closed also in reverse order. So the first element must store the nested namespace in the right order. This is implemented with a simple ordered list, that is stored in the outermost namespace element. This is necessary, because a nested namespace is just closed with one single **END_NAMESPACE** tag and therefore it is ambiguous how many scopes must be closed.

The resulting symbol table, after the parser passed line 22 in Listing 12, is depicted

in Figure 19. As the namespaces **a**, **b** and **c** have been declared already from line 5 to line 11 the declaration of the nested namespace **a.b** on line 19 requires to use existing namespace objects. This means that first the namespace **a** is searched within the symbol table and afterwards the namespace object **b**. When **b** has been found, the symbol table switches to the scope stored in the field *nsScope* in object **b**. The namespace object **b** needs to be installed to namespace **a** in the field *nestedNs*, which is a simple linked list. The object *Basic1*, which represents the function block is added to scope **b**. The production where the function block has been inserted opens a new scope (2) with the scope **b** as its outer scope. This is a temporary scope for adding the declared names within the function block, which are later added to the function block's locals.

As shown in the grammar in Listing 13, the method *closeNs* is called with the previously defined namespace object as a parameter for closing the namespace. The implementation of this method can be seen in Listing 14. First, it is checked whether the object *ns* is a nested namespace or not. This is simply achieved by checking if the field *nestedNs* is null or not. If this field is not null, the tab closes the scopes in reverse order. Afterwards, the field is set to null again. This is necessary because it might be possible to use the namespace **a** again without nesting and in that case just one scope needs to be closed.

```
1  public void closeNamespace(Obj ns) {
2  if(ns.nestedNs==null) {
3      ns.locals = curScope.locals();
4      closeScope();
5      return;
6  }
7
8  for(int i=ns.nestedNs.size()-1; i >= 0;i--) {
9      Obj current = ns.nestedNs.get(i);
10     current.locals=curScope.locals();
11     closeScope();
12 }
13 ns.nestedNs=null;
14 ns.locals = curScope.locals();
15 closeScope();
16 }
```

Listing 14: Closing a namespace in the symbol table

5 Building the AST

This chapter explains in a bottom-up way, how to map an IEC 61499 basic function block to an abstract syntax tree. It starts by explaining how the Truffle nodes are constructed. Furthermore, it states how the execution control chart is modelled within the AST. Also the mapping of the internal variables is depicted in this section.

5.1 ATG for Statement and Expression Nodes

The behaviour of an algorithm from a function block is defined as a list of statements, as seen in line 29 of Listing 6 . Listing 15 shows how the statements are declared as ATG. A statement can be a variable assignment, as shown from line 5 to line 6 in the grammar depicted at Listing 15. Another possibility for statements are control structures like selection or iteration statements.

```
1 Statement<out IECStatementNode s>
2 =[
3       SelStatement<out s>
4     | IterStatement<out s>
5     | Variable<out String n>      (.Obj var = tab.find(n);.)
6     ":=" Expression<out Node e>   (.s=builder.createWriteVar(var,e);.)

7  ] ";" .
```

Listing 15: Attributed grammar of statement production

Listing 16 shows how expressions are declared as ATG. The expression subtree starts with the production *Expression* at line 4 as its root node and the leaves are in the *PrimaryExpr* production at line 6. The attributed grammar is simplified to provide better readability. The main difference to the grammar used in the implementation of this work is that some parts of the ATG are truncated. In the implementation, all expression productions are calling the node builder to create the corresponding expression subtree which is returned. This is only demonstrated in the production *Expression* on line 1, so that the reader can get an idea of how the node construction is done within an ATG file.

```
1  Expression<out Node e>  =
2   XorExpr<out Node l>       (.e=l;.)
3  {"OR" XorExpr<out Node r> (.expr=builder.createOrNode(l,r);.)
4  }.
5
6  PrimaryExpr<out Node e,Struct t>
7  =
8  Constant<out e>
9  |
10 IF(isQualEnumIdent())QualEnumVal<out String enumVal>
11                            (.e=builder.createEnumNode(enumVal);.)
12 |
13 Variable<out String qualVar>
14                            (.e=builder.createReadNode(qualVar);.)
15 |
16 '(' Expression<out e> ')'
17 .
```

Listing 16: Attributed grammar of statements and expression

5.2 IECExpressionNode

An **IECExpressionNode** implements the Truffle class **Node**. It must provide an execute method for every IEC data type, which is called by Truffle. Every expression node holds a *Struct* object, which represents its type. This type information is used during the node construction to perform type checks. Every instance of an expression node needs its own type checks. Figure 20 shows the resulting variations of an **IECExpressionNode**.

5.2.1 IECLiteralNode

	Literal	**Typed Literal**
DecLiteral	986	INT#-123
BinLiteral	2#1111_1111	INT#2#010101_01
OctLiteral	8#377	SINT#8#01234
HexLiteral	16#01BC	SINT#16#59AB
CharLiteral	not allowed	CHAR#'A'
StringLiteral	'string'	STRING#'OK'

Table 3: Different kinds of literals

An IECLiteralNode is declared as a leaf of an expression production, as shown on the ATG in line 8. ST provides two different kinds of literals, namely typed and untyped literals. Table 3 shows the different kinds of literals. Untyped literals are declared as tokens in the scanner. Typed literals are declared within productions in the ATG,

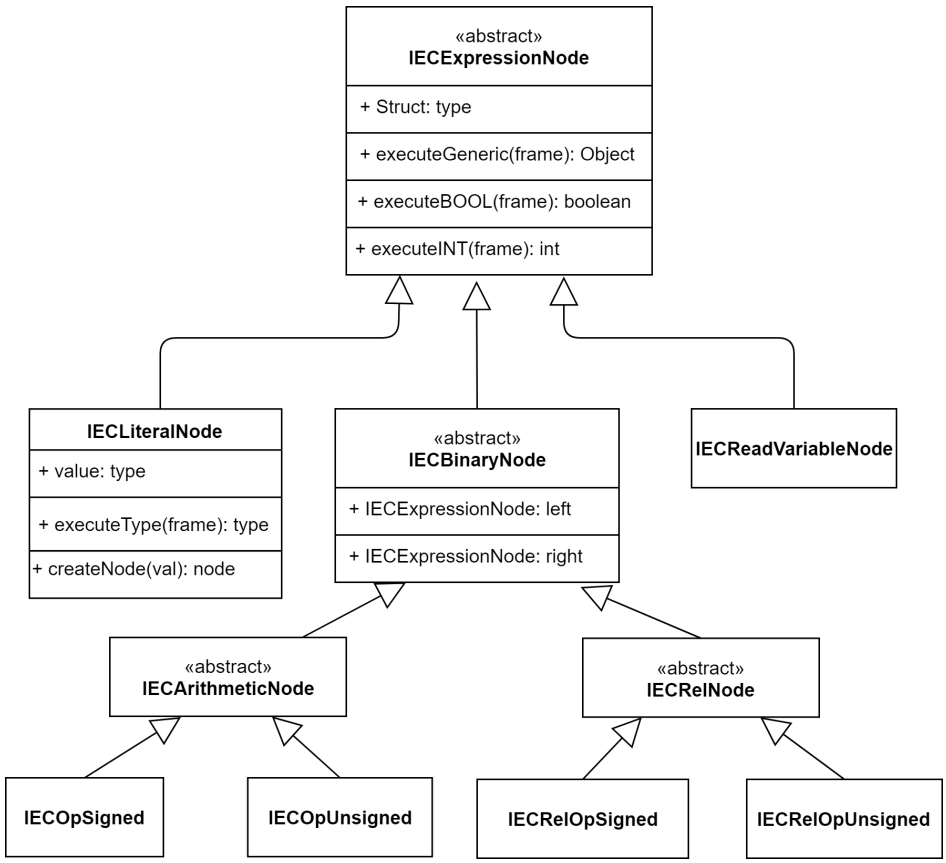


Figure 20: Class hierarchy of IECExpressionNode

shown in Listing 17. To create a literal node, the parser first enters the production *Constant*. Lets take the example for a simple typed decimal literal, like **INT#-123** from Table 3. It has four tokens, namely **INT** , # , **-** and **123**. With these tokens, the parser will enter the *IntLiteral* production, in line 8, and will collect the kind of the scalar type name, which is in that case *INT*, represented by a constant in the parser. The node builder is now able to create the right node, based on the previously collected information and the current token. The builder must check if a scalar type name has been collected. If so, it must also check whether the value fits in the data type or not. If necessary, an error has to be signaled. In the case of an untyped literal, the builder checks which type fits. It starts with the smallest data type and continues until the biggest data type is reached. The literal *986* from Table 3, for example, of the type *INT*. The builder first checks if it fits into *SINT*. In our case, this check fails and the builder continues to check the next bigger type. As the next bigger type is **INT** and **986** fits into **INT**, the builder finishes the node construction. For every data type an **IECLiteralNode** is created. It stores the value of the literal with the Java data type that is mapped from the IEC data type. Every literal node provides a static create method, which is responsible for range checks and for parsing the string value to a Java primitive. A literal node also overwrites the execute method from the **IECExpressionNode** in the corresponding type.

```
1  Constant<out Node con> =
2   NumericLiteral<out con> | CharLiteral<out con> .
3
4  NumericLiteral<out IECExpressionNode con>
5  =
6   RealLiteral<out con> | IntLiteral<out con> | BoolLiteral<out con>.
7
8  IntLiteral<out Node con> (.int k=-1;boolean m=false;.)
9  =
10  (
11     ScalarTypeName (.k=t.kind;.) "#"
12     ( BinLiteral|OctLiteral|HexLiteral |["+"|"-" (.m=true;.)]
          DecLiteral)
13     |(BinLiteral|OctLiteral|HexLiteral|DecLiteral)
14  )
15  (.con=builder.createIntNode(t,k,m);.)
16  .
```

Listing 17: Attributed grammar of Constant

5.2.2 IECReadVariableNode

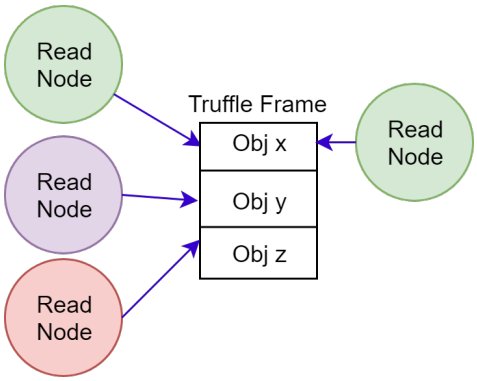


Figure 21: IECReadVariableNodes and Truffle Frame

This node is designed to read variable values from a Truffle *Frame*. It is created in the production *PrimaryExpr* in the ATG from Listing 16 in lines 11 and 14. The node builder searches the variable name in the symbol table. The returned object serves as the identifier for a Truffle Frameslot. The Frameslots are created during the variable declaration of the function block. If no object has been found in the symbol table, an error has to be signaled. During execution, the node reads the value from the Frameslot based on its type and calls the execute method with the correct return value type. Like in the example in Figure 21, we assume that Obj x is of type **BOOL**. This means that the node calls the *executeBOOL* method, which returns the Java data type boolean, as boolean is mapped to IEC **BOOL**.

5.2.3 IECBinaryNode

An IECBinaryNode has a left and a right expression node as its children. It serves as an abstract base class for arithmetic and relational operation nodes. All operations are implemented with Java syntax, based on the data type and the kind of the operation. Also the special treatment of unsigned data types needs to be considered.

5.2.4 IECArithmeticNodeNode

Arithmetic nodes represent the following operations:

- (+) Addition
- (-) Subtraction
- (*) Multiplikation
- (/) Division
- **(MOD)** Modulo
- (**) Power

For every operation, a specific node is created. For example, the node addition is created as *IECAdditonNodeSigned* or *IECAdditionNodeUnsigned*, based on the type. A binary node, like the addition node, needs its left and right child for creation. So the left and the right child must be checked if their types are compatible. If their types are not equal the builder checks if an implicit upcast is possible. If not, the parser needs to signal an error. The IEC 61131-3 standard defines rules for implicit upcasts, illustrated in Figure 22. These rules are supported by this implementation of IEC 61499. For user-defined data types, such as enums, arithmetic operations are not allowed, due to the standard. In general, arithmetic operations are only allowed for numeric data types.

```
1 @NodeInfo(shortName = "+")
2 public abstract class IECAddNodeSigned extends IECBinaryNode {
3     @Specialization
4     protected long addLINT(long left, long right) {
5         return Math.addExact(left, right);
6     }
7 }
```

Listing 18: Addition of two LINT values within IECAdditionNodeSigned

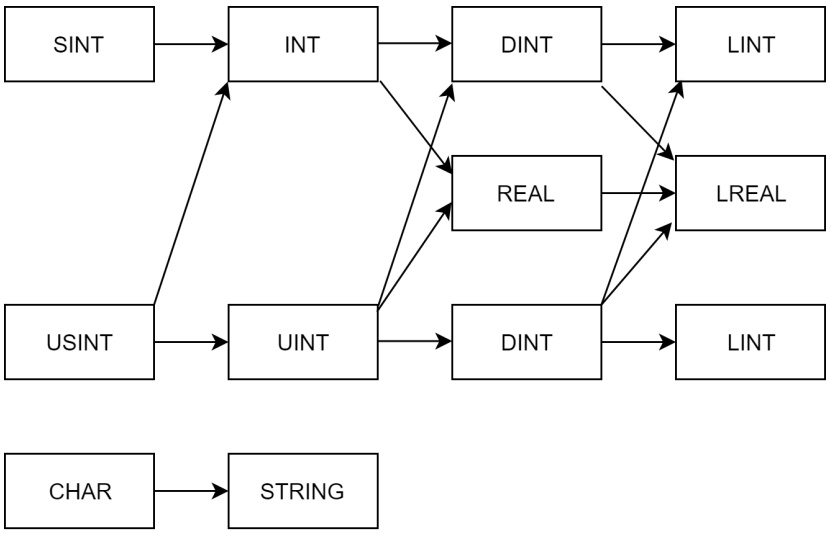


Figure 22: Implicit cast rules from the IEC 61131-3 standard [2]

Figure 23 shows the creation of an addition node for an expression containing a **LINT** and a **USINT** value, which could be **LINT#3 + USINT#2**. The **LINT** literal node contains the value 3 stored as Java long and the **USINT** literal node stores the value 2 represented as Java byte. The node builder converts the **USINT** literal node to a **LINT** literal node to ensure type safety. The addition node will now execute the *addLINT* method, showed in Listing 18, during runtime. Another possibility to implement that, would be to write an extra specialisation for the addition of **LINT** and **USINT** within the addition node. But this would lead to a mess of code and as IEC 61499 ST is strictly typed, the casts are performed in the compiler frontend, which happens in the loading phase of a function block. As a consequence, the construction of the AST will be slower, but the execution will be faster, as there are no dynamic state changes of the arithmetic nodes.

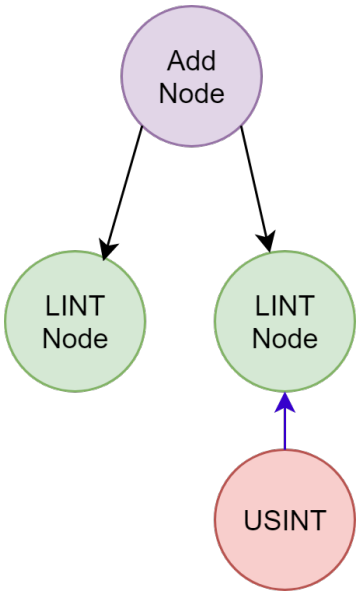


Figure 23: Implicit type cast of an addition node

5.2.5 IECRelationalNode

Relational nodes represent the following operations:

- **(NOT)** Logical Not
- **(OR)** Logical or
- **(XOR)** Logical xor
- **(AND)** Logical and
- (=) equal
- (<>) not equal
- (<) less
- (>) greater
- (<=) less equal
- (=>) greater equal

They act in a similar way as arithmetic nodes. A relational node always returns a boolean value in its execute method. Therefore, on node creation the builder must check if the corresponding types of the node's children are comparable. For comparison, also the rules for implicit casts are valid. Logical relational operations (NOT,OR,XOR,AND) are only available for boolean expressions, whereas non logical operations (<,<=,>,=>) are valid for arithmetic expressions. For strings and enums, only the operations equal and not equal are allowed.

5.3 IECStatementNode

Statements are implemented as *IECStatementNode* as shown in Listing 19. This node implements the Truffle class Node. It provides the method *executeVoid*, which is executed by the Truffle Framework. It is an abstract base node for all IEC statements.

```
1 @NodeInfo(language = "IEC")
2 public abstract class IECStatementNode extends Node{
3     public abstract void executeVoid(VirtualFrame frame);
4 }
```

Listing 19: IECStatementNode

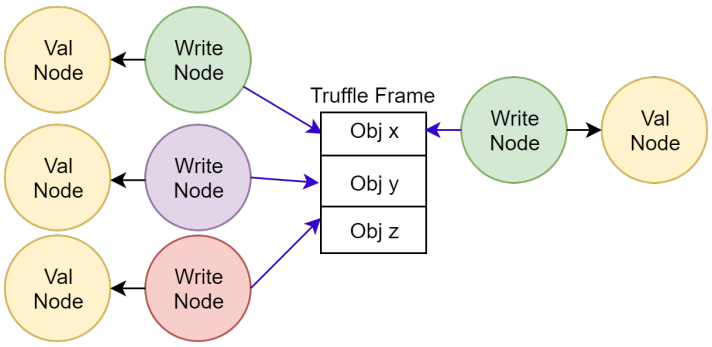


Figure 24: IECWriteVariableNodes and Truffle Frame

5.3.1 IECWriteVariableNode

This node is designed to store variables from a function block into a Truffle *Frame*. A *WriteVariableNode* is created within an assign statement, shown in line 6 of Listing 16. The node references an object from the symbol table, which is used as an identifier within the Truffle Frame as depicted in Figure 24. It has an *IECExpressionNode* as its child, which holds the expression that should be written to the Frame, illustrated as the yellow *ValNode* in Figure 24. If a variable assignment happens, first the declared name is searched or created in the symbol table and its corresponding object is used as a unique identifier for the Frameslot. In line 6 of Listing 16, the production *Expression* is entered after the **":="** token. The resulting *IECExpressionNode* is needed for creating the node, as this is the value, that should be written to the *Frame*. After the expression node is collected, a new *WriteVariableNode* can be created, with the resulting expression node as its child. During runtime, Truffle first executes the write node. The write node executes the value node and writes the corresponding value to the Truffle Frame.

5.3.2 IECIfNode

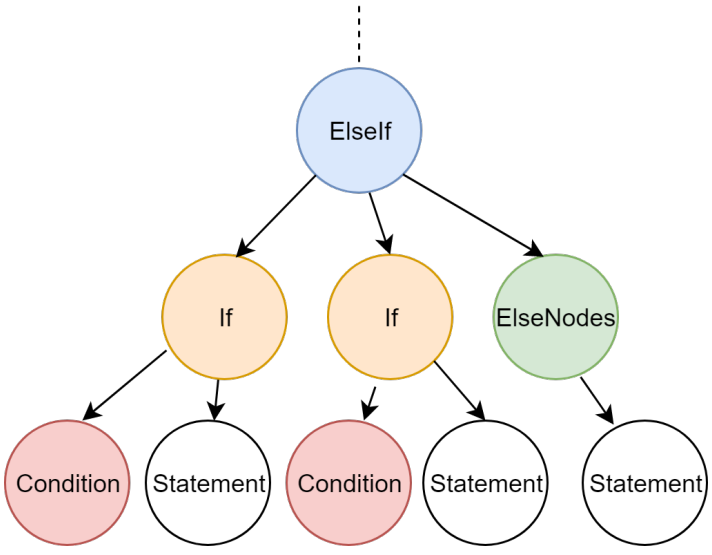


Figure 25: IECIfNode

Figure 25 illustrates how the *IECElseIfNode* is constructed. The nodes *if* and *else* are its children. First it executes the first *IECIfNodes* condition. If this node evaluates to true, then the statement nodes are executed. If the *IECIfNode* is false, the node checks the next if node. If all *ifNodes* evaluated to false, the *IECElseIfNode* checks whether an *elseNode* is appended or not. If so, the else node is executed too. Otherwise, the execution of the *IECElseIfNode* stops.

As Truffle is able to optimize conditional statements, based on profiling information, the profiler is used in the *IECIfNode*. Listing 20 shows how the *IECIfNode* is implemented. The condition profiler counts the executions of the if branch at runtime, shown on line 10, as long as Graal is in interpreter mode. After the AST has become stable, Graal switches to compiler mode. Based on the calculated probabilities, Graal only executes the hot branch, in compiler mode. If these assumptions are violated, Graal must switch back to interpreter mode. This is possible, because Graal adds assertions to machine code [6].

```
1  public class IECIfNode extends IECStatementNode {
2  @Child private IECExpressionNode conditionNode;
3  @Children private IECStatementNode[] thenNodes;
4
5  private final ConditionProfile condition = ConditionProfile.
       createCountingProfile();
6
7      @ExplodeLoop
8      @Override
9      public boolean executeBOOL(VirtualFrame frame) {
10         if (condition.profile(evaluateCondition(frame))) { {
11             for (IECStatementNode thenNode : thenNodes) {
12                 thenNode.executeVoid(frame);
13             }
14           }
15     }
16 }
```

Listing 20: IECIfNode using a ConditionProfiler

5.3.3 IECCaseNode

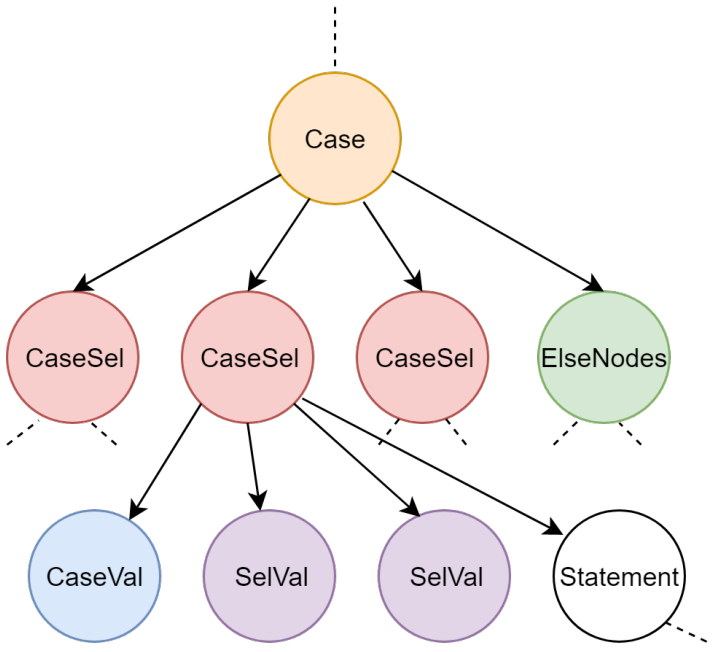


Figure 26: IECCaseNodes with CaseSelectionNodes

The case statement, which is declared in the grammar from Listing 7 is represented as *IECCaseNode*. It has an array of case selection nodes, as shown in red in Figure 26. The case selection node children are one case value and an array of case selection values. When the case node executes, it iterates over its case selection children and executes them. The case selection nodes are iterating over their selection values,

which are compared to the case value. If one comparison returns true, the case node stops iterating and the corresponding statements of the case selection are executed. Otherwise, the case node will check for an else node, which might exist.

5.3.4 Loop Nodes

To implement a loop with Truffle, two components are necessary: a condition node and a loop body node. The loop body node needs to implement the Truffle loop node interface. With Truffle loop nodes, the runtime can perform additional optimizations, like on-stack-replacement for loops (OSR) [6]. With OSR the loop node will trigger the JIT compiler after a certain amount of repetitions.

Figure 27 shows how a loop statement is implemented in an abstract way. Based on the type of the loop, the nodes have their own loop body nodes, which must override the *executeRepeating* method from the Truffle class *RepeatingNode*. Within this method, the loop semantics from ST can be mapped to Java control structures. In case of the *IECWhileNode*, the implementation of the *executeRepeating* method is shown in Listing 21. The semantics of break and continue statements are implemented as Java exceptions in Truffle. So if a break statement is executed at runtime, it must throw a break exception and the current loop node returns false and leaves the execute method, as shown in line 12 in the source code. The implementation of a *IECRepeatNode* is similar to the *IECWhileNode*, except that the condition is checked after the statement nodes have been executed once. This is equal to a *do while* loop in Java.

```
1  @Override
2  public boolean executeRepeating(VirtualFrame frame) {
3      if (!evaluateCondition(frame)) {
4          return false;
5      }
6      for (IECStatementNode bodyNode : bodyNodes) {
7          try {
8              bodyNode.executeVoid(frame);
9          } catch (IECContinueException e) {
10             continueTaken.enter();
11             return true;
12         } catch (IECBreakException e) {
13             breakTaken.enter();
14             return false;
15         }
16     }
17     return true;
18 }
```

Listing 21: ExecuteRepeating method from the IEC while loop body

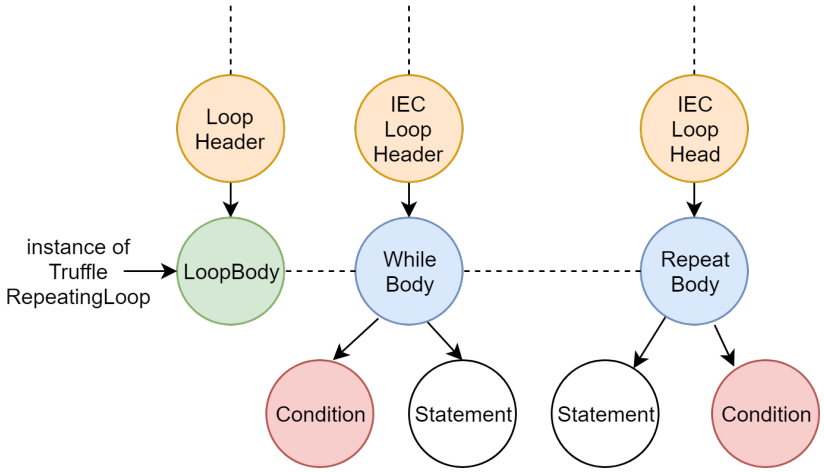


Figure 27: IECLoopNode instances based on Truffle RepeatingNode

5.4 ECC Graph Node

The idea of this work is to implement the state chart from the ECC as Truffle nodes. The node builder needs to create Truffle nodes from the ECC, that is stored in the symbol table. How the ECC is inserted into the symbol table was described in Section 4.3. After the parser has collected the last algorithm, shown in line 2 of Listing 6, the node builder can construct the ECC for the function block. The declarations collected during parsing can now be verified by the node builder. The nodes are created in the following order:

1. The states which have been collected in the symbol table are created as **IEC-StateNodes**, illustrated as light red and green nodes in Figure 28. They have an array of *EccActionNodes* and an array of the current outgoing transition nodes as children. Listing 22 shows the method *createStateNodes*, which is called by the node builder after parsing of every state. First the *EccActionNodes* are created for the corresponding state. The actions are stored as *EccAction* in the symbol table. To create an action node, the algorithm object has to be searched in the symbol table with the *EccAction* algorithm name. At this point, the builder can signal an error if the algorithm name does not exist in the symbol table and is therefore not declared at all. Otherwise a new *EccActionNode* is created for the actual state.

2. The next step is to create the outgoing transition nodes of a state. The method *getOutgoingTransitions* in Listing 22 shows how the outgoing transitions are constructed. The transitions are stored in the function block object from the symbol table, which is passed as a parameter to the method *getOutgoingTransitions*.

3. Since the method does not return a node object, the *EccTransition* array is converted to an *EccTransitionNode* array with the method *createNodes*. This method creates new nodes with a *GuardConditionNode* as their child. As already men-

tioned, the *GuardConditionNodes* have been declared on the fly during parsing. Also the next state is added as a child, namely the *IECStateNode* **to**.

4. The last step is to create an *EccNode* with its child state nodes as a child. The first state in the array is set as the current state.

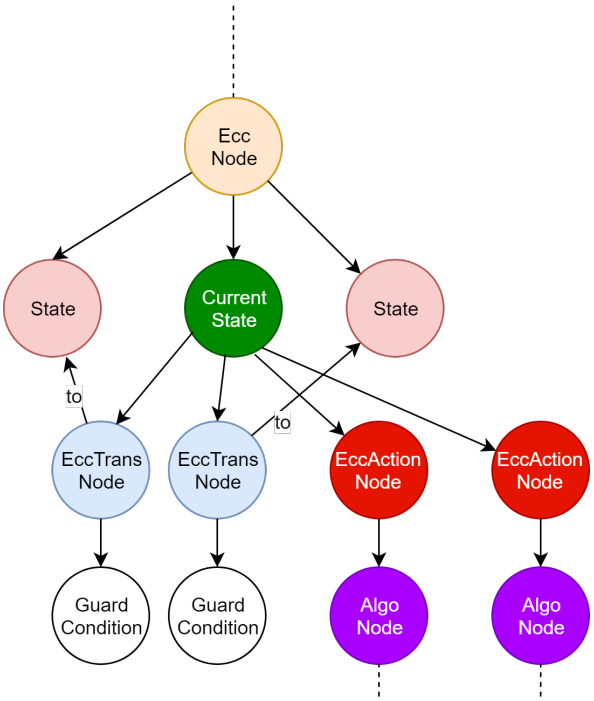


Figure 28: Execution Control Chart from a function block as AST

```
1  public EccStateNode createStateNodes(Obj fb,EccState state) {
2      List<EcActionNode> nodes = new ArrayList<EccActionNode>();
3      for(EccAction action:state.eccActions) {
4          Obj algo  = fb.type.findMember(action.algorithmName);
5          if(algo == noObj || algo.kind != Obj.Kind.Algo) {
6              throw new IECParseException();
7          }
8          nodes.add(new EccActionNode(algo.algoNode, action.eventOutput.
               name));
9      }
10     EcTransition[] trans = getOutgoingTransitions(state);
11     return new EccStateNode(name, nodes.toArray(new EcActionNode[0]),
           EccTransition.createNodes(trans));
12 }
13
14
15 private EcTransition[] getOutgoingTransitions(Obj fb, EccState state)
       {
16     return Arrays.asList(fb.transitions)
17             .stream()
18             .filter(trans -> trans.from == state)
19             .collect(Collectors.toList())
20             .toArray(new EcTransition[0]);
21     }
```

Listing 22: Creation of a EccStateNode and its current outgoing transitions

5.4.1 Function Block as an AST

Figure 29 illustrates an IEC 61499 function block, represented as an AST. It has an array of **WriteVariableNodes** and one **EccGraphNode** as its children. The function block node is implemented as Truffle root node. This is the node which is called to execute a function block, either on initialization or when an event arrives. It has its own Truffle frame, which stores the values of the internal variables. In contrast to algorithm nodes, the function block node needs a Truffle materialized frame. With this kind of frame, it is possible to store the internal variables in heap memory and to access them across algorithms. The Frame acts as internal memory instance for the function block.

The node has two different execute methods, as showed in Listing 23. One method for the initialization, namely *executeInit*, and the method *executeEvent*, which is called when an event arrives. Therefore, the node can have two different states. On initialization, all internal variables are written to the frame. If an event arrives the node switches to execution state and executes the graph node. The variables belonging to

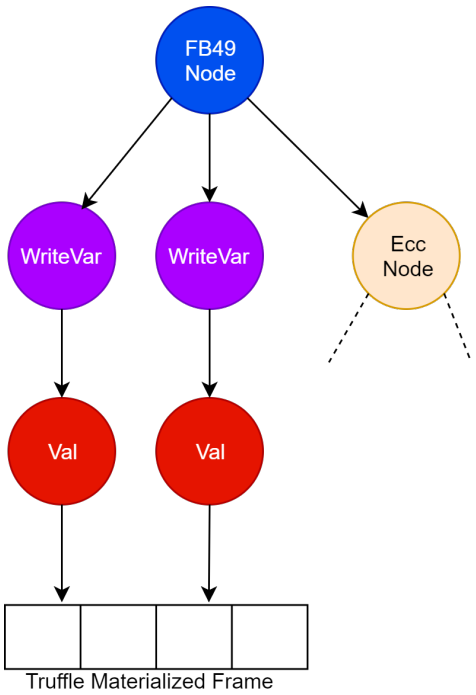


Figure 29: IEC 61499 function block represented as AST

the event are written before the node executes.

```
1  public abstract class IECFb49Node extends RootNode {
2      @Child private EccGraph graph;
3      @Children private WriteVariableNode[] writeVariables;
4      protected MaterializedFrame frame;
5
6      @Specialization(guards = "frame.getArguments().length == 0 ")
7      public void executeInit(VirtualFrame frame) {
8          writeVariables(frame);
9      }
10
11     @Specialization(guards = "isEvent(frame.getArguments())")
12     public void executeEvent(VirtualFrame frame) {
13         graph.execute(frame);
14     }
15 }
```

Listing 23: IEC function block root node

6 Execution of a Function Block

This chapter describes how to execute a function block represented as an AST. It explains how it is loaded into the runtime and how it is possible to receive and send events to other function blocks within a distributed IEC 61499 application.

6.1 Interaction between IEC Runtime and AST

To execute a function block AST, it must be triggered somehow. This is possible with a Truffle polyglot context [18], which allows evaluating Graal guest languages. The Truffle interoperation library specifies the message protocol between the host and the guest language. In case of this work, the host language is Java and the guest language is IEC 61499 Structured Text. With the interoperation library, it is possible to import and export elements implemented in ST to Java.

Figure 30 shows the necessary components to execute a function block represented as an AST on top of the Graal VM. The **FB Executable Object** extends the Truffle Object interface from the interoperation library. This means that it as an entity that can be shared across other languages. The **FB Executable Object** can execute the function block's root node with different options, depending on the parameters in the execute method. It is also able to read and write the materialized frame of the node. The **IEC Runtime** creates a polyglot context, in our case an IEC context. With the bindings of the polyglot context it is possible to access the executable Truffle object from a function block.

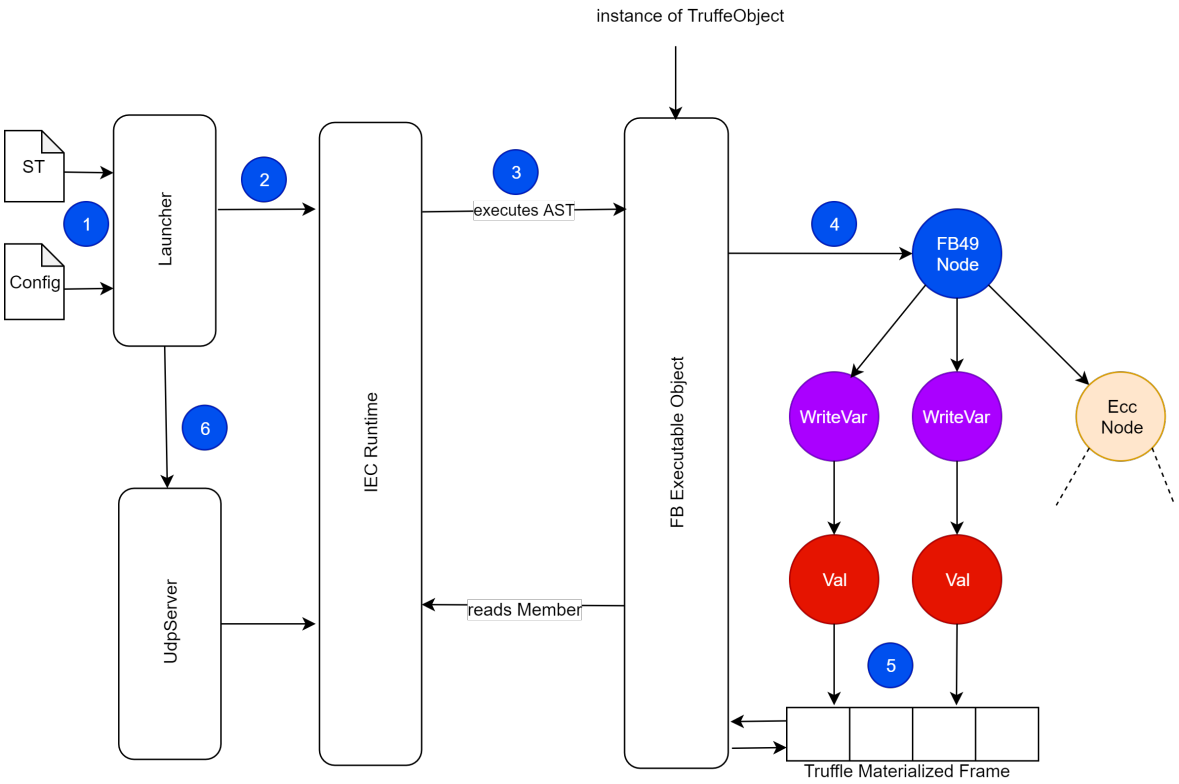


Figure 30: IEC Runtime overview

6.1.1 Load Function Block into Graal VM

The component **Launcher** from Figure 30 is the main entry point. The workflow of loading a function block is marked with the blue filled numbered circles and is explained in the following steps:

1. The launcher loads the configuration file, which holds the corresponding settings for the network communication. It also loads the ST file, which represents the function block into memory.

2. It creates an instance of an IEC runtime. The runtime creates a Truffle polyglot context for the IEC 61499 ST language. With this context it is possible to evaluate the source code of the ST file.

3. In this step, the runtime evaluates the context with the given ST function block. It triggers the parser to build the abstract syntax tree from the source and executes the root node without parameters.

4. As seen in Listing 23 in line 6, the node executes the *executeInit* method, because no parameters have been passed.

5. The **WriteVariable** nodes are writing the initial value of the variables to the Truffle frame. The initial values have been collected while parsing the variable declaration section of the function block.

6. With the loaded settings the launcher starts the **UdpServer**, which listens to incoming events and delegates them to the IEC runtime.

6.1.2 Receive Event and execute Function Block

After the function block has been loaded to the runtime, it must interact with linked devices to form a distributed IEC 61499 application. Figure 31 shows the workflow, after an event has arrived.

1. A UDP package arrives at the **UdpServer**. This package consists of exactly one event and can include multiple variable values. The **UdpServer** must decode the event ids and the values and names of the variables.

2. After the package has been decoded, the **UdpServer** asks the **IEC Runtime** to write variables and execute the function block with an event.

3. Before an event can be executed, the runtime must write the values of the bound variables to the memory of the function block. To access the memory, the Truffle frame must be accessed. This can be done with the **FB Executable Object**.

4. After the values have been written to the function block, it is ready to execute. For execution, the **IEC Runtime** asks the **FB Executable Object** to execute the AST with the given event.

5. The **FB Executable Object** calls the root node of the AST, with the event as argument.

6. As seen in Listing 23 on line 11, the FB node executes the *executeEvent* method. The *IECFb49Node* executes its child now, namely the *EccGraph* object graph.

7. The *EccGraphNode* executes the outgoing transitions of the current state, illustrated as the green node in Figure 31 with the name "Current State". There can be at least one outgoing transition in every state. The transitions are declared as node children, namely guard condition nodes, in the state node.

8. The first guard condition node is evaluated. It is checked if the event which is passed equals the event which is assigned to the node. Also the optional boolean condition in the guard is checked.

9. If the previously evaluated guard has evaluated to true, then the ECC executes the next node. The next state is also stored in the guard condition node. After the ECC has left the current state, the event is no longer passed as an argument to the other guards. If the guard has evaluated to false, the resulting guard condition nodes are evaluated until one of them is true. If none is true the ECC will stop.

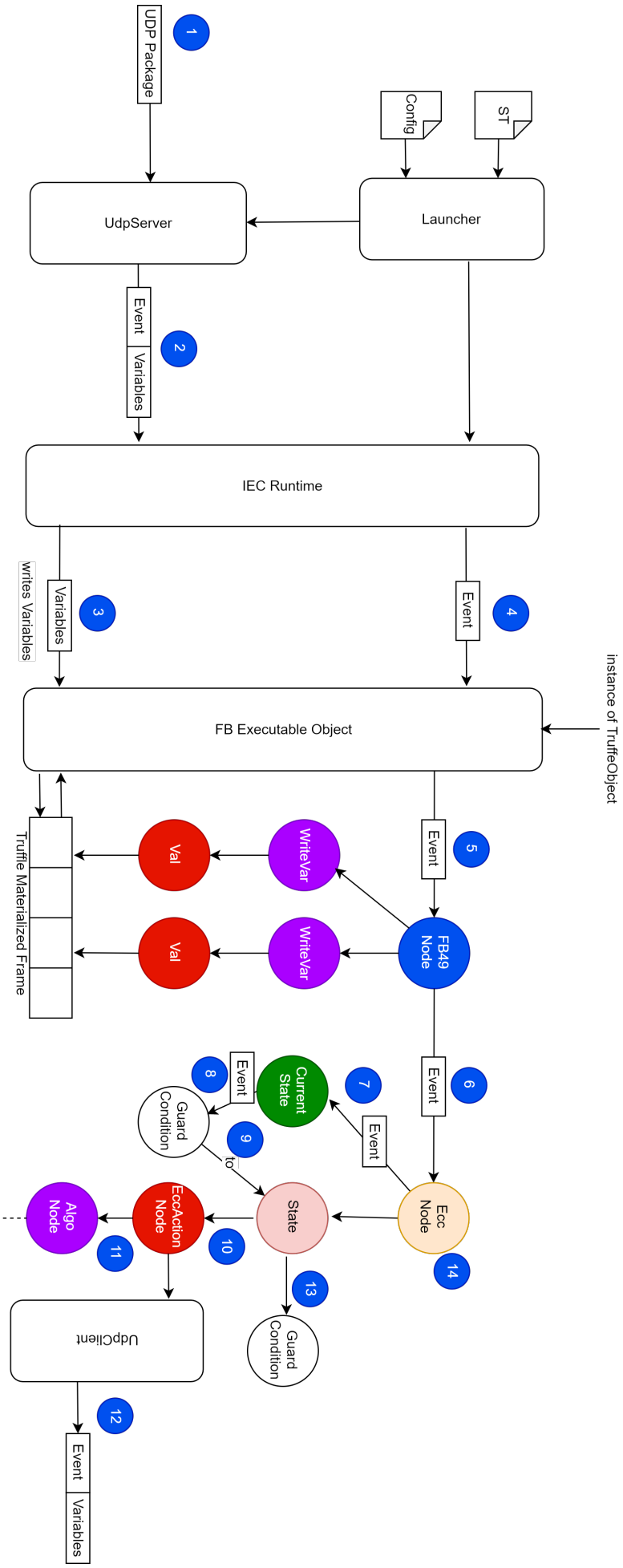


Figure 31: IEC Runtime event workflow

10. After entering a new state, the first thing which happens is to execute its action nodes. In case of Figure 31, there is just one action node, but in other examples there might be more.

11. The action node executes its assigned algorithm.

12. After the algorithm has been executed, the action node sends the bound output event with its corresponding variables to the linked IEC device. For this task, the action node has the developed component **UdpClient** installed. The client performs decoding of the event and variables and sends it to the other device.

13. After sending the event, the last executed state also executes its children for the outgoing transitions, like in step 8, except that the event is no longer passed to the arguments of the execute method from the nodes.

14. If no more state changes are performed, the ECC node replaces the "Current State" node with the last processed state. At this point, the ECC stops the execution.

6.2 Network Communication

To form a distributed IEC 61499 application, it is necessary to exchange events and data with other devices. The component **UdpServer**, illustrated in Figure 31, handles incoming events. It is important that the events are processed in the same order in which they arrive at the device. To ensure this behaviour, two different threads and a shared concurrent data structure are used. The **UdpReceiver** thread listens to incoming UDP messages and adds every message to the shared data structure. The **UdpProcessor** thread takes the messages from the data structure and executes the IEC runtime. The chosen data structure is a *BlockingQueue* [19] from the Java system library. It ensures that the **UdpProcessor** thread waits until a new event has been added to the queue by the **UdpReceiver**. If the queue is full, the **UdpReceiver** thread waits until space is available. The two threads are started by the launcher. Appropriate termination of the threads is ensured by the runtime.

A further requirement is that an event which appears during the execution of a function block will not get lost. This requirement can only be achieved if the queue is not full. If the execution time of the function block is higher than the sending interval time between two events, the queue will get full. As a consequence, the **UdpReceiver** waits until the processor finishes the execution of the function block. Events which occur in the meantime cannot be added to the queue and will get lost. To handle that, an adequate size of the queue must be chosen. However, IEC 61499 does not define how to treat that case, so it is the responsibility of the implementer to choose an adequate

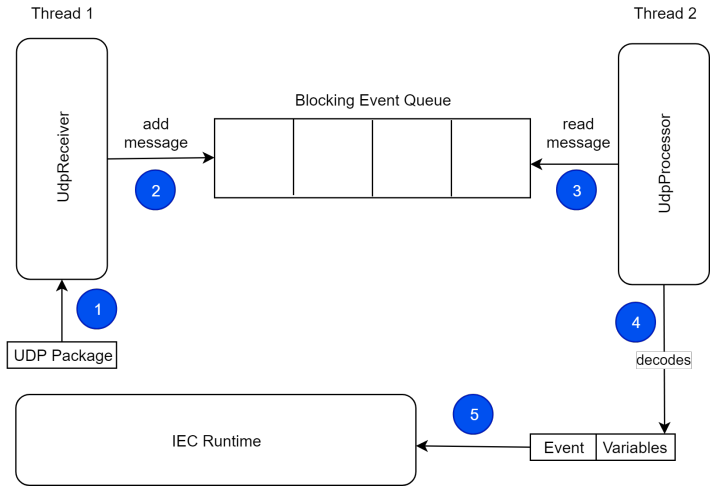


Figure 32: Handling of incoming events

sending interval for events. A common variation in other implementations of IEC 61499 is to provide feedback to the language implementer of the maximum execution time for a function block. But this is out of scope of this thesis and might be a topic for future work as real-time execution is a complicated task that requires a lot of engineering effort.

The processing of the UPD packages, illustrated in Figure 32, happens in the following order:

1. The thread **UdpReceiver** opens a **DatagramSocket** [20] at the defined port and receives UDP packages. The receive function from the socket will block until a new package arrives.

2. If a message has arrived at the socket, the **UdpReceiver** adds the message as byte array to the event blocking queue. If the queue is full the thread waits at this point until space is available.

3. The **UdpProcessor** thread takes elements from the queue, if the queue is not empty. Otherwise, it blocks at this position and waits for a message.

4. If a message has been taken out of the queue, the byte array is decoded into one event and multiple variables. The decoding is defined in the ISO/IEC 8824 [21] and ISO/IEC 8825 [22] standard.

5. The runtime first writes the values of the variables to the Truffle materialized frame. In the second step it executes the function block AST with the decoded event as parameter.

7 Evaluation

This chapter shows the code size of our implementation and how it was tested. It also demonstrates a working example in a distributed IEC 61499 application. The implementation is compared to another IEC 61499 implementations in terms of performance.

7.1 Code Size

The tool **CodeMR** [23] was used to evaluate the metrics. Table 4 shows the lines of code (LOC) of the different components of our implementation.

The LOCs in the **Compiler Frontend** are accumulated without the ATG file. They contain the **Node Builder**, the **Symbol Table** and the classes **Parser** and **Scanner** which are generated by Coco/R. Since the grammar is very extensive, the **Parser** and **Scanner** make up most of the code in the **Compiler Frontend**.

The component **Truffle Nodes** contains all statement, expression and literal nodes. About 70 percent of the component's code is used for the expression nodes as every operation needs to be defined for every data type.

The **Runtime** contains the classes which are exported from the IEC 61499 ST Truffle context. The implementation of the Truffle context is also part of this component. It also provides an interface to load a function block, send events and perform read and write operations on it.

The implementations of the network communication is located in the component **Server**. It contains the subcomponents **UdpServer** and **UdpClient** and implementations for decoding and encoding network messages.

The main entry point of the IEC compiler is implemented in the component **Launcher**. It contains the implementation starting the compiler with a function block represented as an ST file or a string.

Component	**LOC**
ATG file	2071
Compiler Frontend	5426
Truffle Nodes	14005
Runtime	1663
Server	1417
Launcher	313
Total (without ATG)	24895

Table 4: Code sizing of the different components of this work

7.2 Testing

Different components have been tested in order to guarantee a correct implementation. The scanner needs to process 15 different tokens. For every token at least 10 positive and 10 negative tests have been written. In total, there are 348 test cases which have been executed successfully. The tool **Robot Framework** [24] has been used to write the test cases. With this tool it is possible to use tabular test data syntax, as shown in Figure 33, which can avoid repetitive test code.

TestIdentifier		
Compare Token	abc	${_Identifier}
Compare Token	A_BCD	${_Identifier}
Compare Token	IW215Z	${_Identifier}
Compare Token	QX75	${_Identifier}

Figure 33: Robot Framework Test Cases

773 test function blocks have been provided by *nxtControl GmbH* to test the **Parser** and the **Runtime**. A truncated example of such a function block under test is shown in Listing 24. All tests follow the same pattern. First, the function block is parsed. If the parser signals an error, the test fails. Otherwise, the test will continue and the runtime will execute the function block. Statements which are to be tested within the function block must be verified somehow. This is demonstrated with a simple example in line 14 of Listing 24. The value **TRUE** is assigned to the variable **bool1** and is later checked by a conditional statement as can be seen in line 16. All these checks are stored to an boolean output variable which is checked by the test suite after the function block has been executed.

All other components have been tested by manually written unit tests.

```
1  FUNCTION_BLOCK Test
2      VAR_INPUT
3          QI : BOOL;
4      END_VAR
5      VAR_OUTPUT
6          QO : BOOL;
7      END_VAR
8      VAR
9          out0 , out1 , out2 , out3 : BOOL;
10         bool1 : BOOL;
11         int1 : INT;
12     END_VAR
13     ALGORITHM test0001 IN ST :
14         bool1 := TRUE;
15         int1 := 20;
16         out0 := (bool1 = TRUE);
17         out1 := (int1 > 19);
18         out2 := (int1 < 21);
19         out3 := (int1 = 20);
20         QO := out0 AND out1 AND out2 AND out3 ;
21     END_ALGORITHM
22 END_FUNCTION_BLOCK
```

Listing 24: Example for a function block under test.

7.3 Example

We will now present an example of a function block used in an industrial system that is used throughout this section to demonstrate its functionality. The given system can be used for a vibration analysis which measures the vibration levels of industrial machinery and uses the information to display the health of the machine to the user. To interpret the vibration signal, it must be transferred from the time domain to the frequency domain with a Fast Fourier Transformation (FFT). As this is a computational-intensive task, the FFT is outsourced to the Truffle IEC device, which provides runtime optimizations. First the FFT function block has been developed as a basic function block which was afterwards integrated into the distributed system.

7.3.1 FFT Basic Function Block

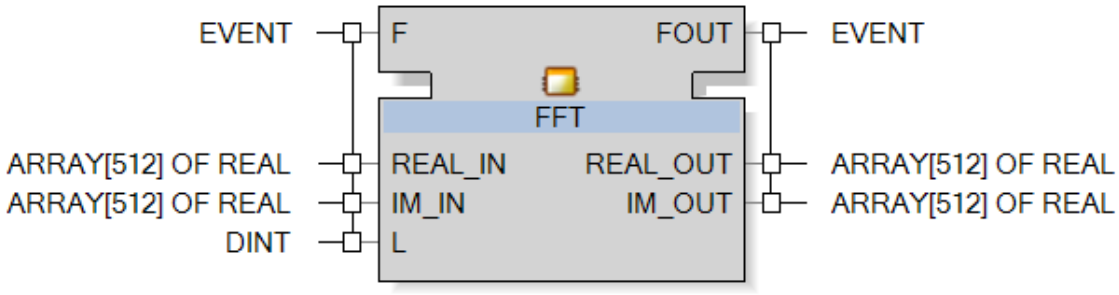


Figure 34: FFT basic function block developed in *nxtStudio*

The function block performs a FFT and is depicted in Figure 34. It is used to transform a signal from its time domain to the frequency domain.

The function block uses three input variables, which are written to its memory after the input event **F** has arrived:

- **REAL_IN** represents the real values of an arbitrary signal.
- **IM_IN** stands for the imaginary components of an arbitrary signal.
- **L** is used to define the length of the signal.

After the input variables have been written to the memory, the algorithm is executed, which sends the output event **FOUT** and its bound variables:

- **REAL_OUT** represents the calculated real values depending on the input signal.
- **IM_OUT** stores the calculated imaginary components depending on the input signal.

The algorithm implements the Cooley-Tukey FFT algorithm [25] in Structured Text and has a runtime complexity of $O(n \log n)$. It was developed in the IDE *nxtStudio* by *nxtControl* [26] and can be executed on the Ecort runtime from *nxtStudio* and on Truffle IEC. It can also be executed stand-alone or integrated into a distributed IEC 61499 application.

7.3.2 Distributed Application

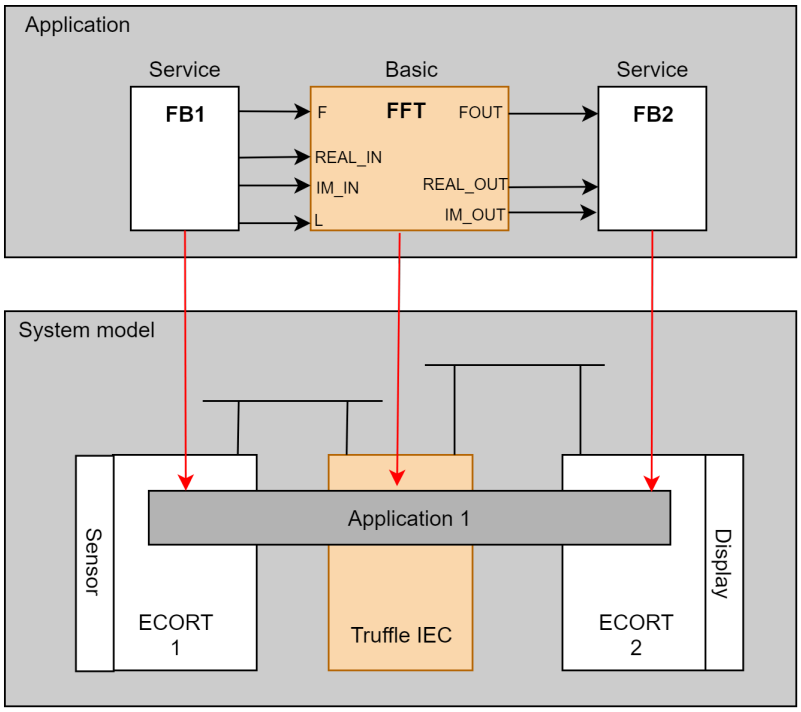


Figure 35: FFT basic function block embedded into the IEC 61499 system model

Figure 35 shows, on the basis of the IEC 61499 system model, how to form a distributed application containing a Truffle IEC device that executes a basic function block. The system consists of three function blocks, which are executed on three different linked devices, one Truffle IEC device and two *Ecort* runtime devices from *nxtControl* [26]. As Truffle IEC implements the same communication protocol like *Ecort*, it is capable of exchanging data with it. So different runtime systems can execute one IEC 61499 application.

The sensor of the machine is represented as a service interface function block (**FB1**) and the device **Ecort 1** can access it. The sensor will return a sample of discrete data, which represent a vibration signal. The signal must be sent in a periodic time interval $t_{FB1Intervall}$. In order to prevent undefined behaviour, $t_{FB1Intervall}$ must be chosen appropriately. So $t_{FB1Intervall} < t_{FFTexecution}$ must hold, where $t_{FFTexecution}$ is the execution time of the **FFT** function block.

The **Truffle IEC** device can receive the data samples from **FB1**, which is executed on the **Ecort** runtime, over the network as a UDP package. The runtime delegates the samples to the **FFT** function block, which processes them and sends an output event with the calculated values to the linked function block **FB2**.

The device **Ecort 2** receives the output event from the **FFT** function block as an input event with the corresponding data samples. The runtime of the device provides access

to a display, that is connected to the device. Alternatively, the display could have an embedded runtime system installed. The access to the hardware is provided by the service interface function block **FB2**.

7.4 Runtime Measurements

In order to compare the performance of **Truffle IEC** with the one of **Ecort**, the same sample function blocks have been executed on both runtimes and the execution times of the algorithms have been measured. To provide comparable results, the timestamps must be taken in the same execution steps on both runtimes. Before starting the measurement, the function block has to be loaded into the runtime. Afterwards, an event with bound variables is sent to the function block. The next step is to write the values of the variables to the internal function block memory. At this point the first timestamp (t_{start}) is taken and the function block starts to execute the ECC. After the ECC has stopped, the second timestamp is taken (t_{stop}). The resulting execution time (t_{exe}) can be calculated now ($t_{exe} = t_{stop} - t_{start}$). That corresponds with step 5 (t_{start}) to step 14 (t_{stop}) of the function block execution workflow illustrated in Figure 31.

[ms]	t_{EcortL}	t_{EcortH}	$t_{TruffleL}$	$t_{TruffleH}$	$t_{TruffleLc}$	$t_{TruffleHc}$
GCD	0.002	128	0.126	104	0.044	90
Prim	0.001	445	0.11	361	0.04	344
Loop	0.001	268	0.12	270	0.05	266
ECC Loop	0.002	397	0.15	452	0.08	434
FFT	0.009	423	0.133	402	0.052	380

Table 5: Execution times in milliseconds of various algorithms on different runtime systems

Table 5 shows the algorithms and the average execution time for 10 runs for every runtime with different input values. Every algorithm has been tested with input values that generate low computation effort, and input values that generate high computation effort. Since Truffle is slower on the first execution of an algorithm, the measurements $t_{TruffleLc}$ and $t_{TruffleHc}$ are representing a corrected average of the execution time. Whereas $t_{TruffleL}$ and $t_{TruffleH}$ calculate the average value of run 1 to 10, $t_{TruffleLc}$ and $t_{TruffleHc}$ are calculating the average value of run 1 to 11. More than 10 runs have shown no improvement of the performance.

The following algorithms have been evaluated:

- **GCD**: This algorithm calculates the greatest common divisor for two decimal input values.

- **Prim** checks, whether the input value is a prime number or not.
- **Loop** is a simple loop that increments a variable until the input value is reached.
- **ECC Loop** increments a variable with state changes instead of a loop statement. This means that the ECC performs state changes, which execute an algorithm that increments a counter that is checked in the state transition.
- **FFT** is the example function block from the last section.

For input values that produce low computation effort, Ecort is always faster than Truffle IEC, because Ecort immediately executes machine code. Truffle, however, performs the first executions in interpreter mode where profiling feedback is collected.

This can be seen in Figure 36, where the execution times from the algorithm **GCD** are displayed. In this example the values 26 and 2000000000 are used as input values, which will produce a lot of loop iterations. As shown in Figure 36, the first execution on Truffle is considerably slower, because it is executed in interpreter mode and profiling feedback is collected. But after the second run, Graal executes the algorithm faster than Ecort, because after the profiling feedback has been collected, it executes the algorithm in compiled mode.

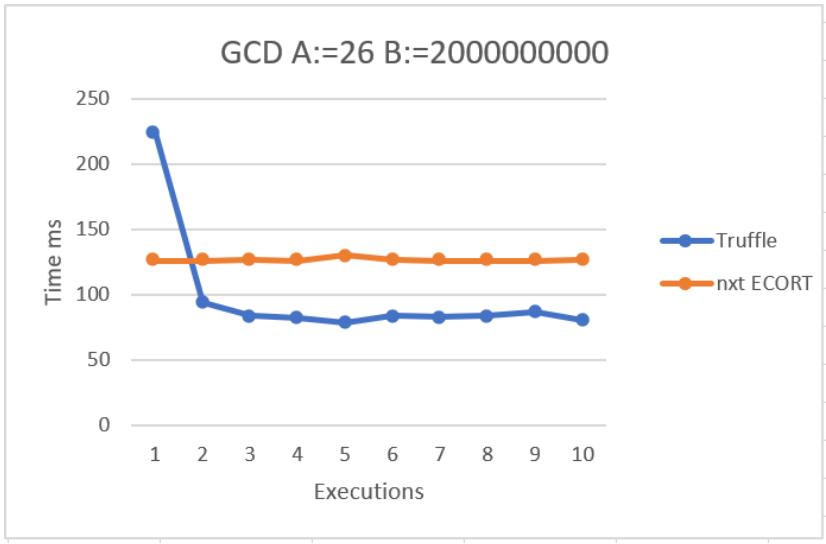


Figure 36: Execution of GCD algorithm on Truffle and nxtEcort

One performance issue is that if the function block performs state changes, also the AST structure is changing as shown in Figure 37. At (1) the FB49 Node executes the Ecc Node which executes the Guard Condition Node, at (2) of the current state. The new state is entered at (3), which means that the corresponding EccAction Node with its algorithm will be executed at (4). The Algo Node will be executed in compiled mode now, as there will be no node replacements in that node. This is guaranteed,

because ST is a statically typed language. When no more state changes occur, the Ecc Node needs to replace the Current State Node with the node of the new state at (5). Unfortunately, a node replacement forces Graal to switch to interpreter mode. This means that the ECC will always be executed in interpreter mode and therefore a function block with a lot of ECC state changes, like the **ECC Loop** from Table 5 is slower on Truffle IEC than on Ecort. That is also the reason why $t_{TruffleL}$ is slower than t_{EcortL}, as Ecort executes immediately compiled code.

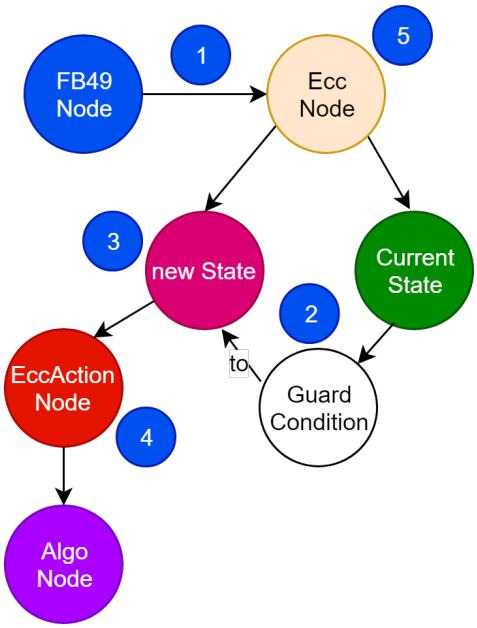


Figure 37: Execution Control Chart

We experimented with different approaches to execute the ECC in compiled mode, but they did not reduce the execution time. We tried to replace the ECC Node, which contains the logic to perform state changes, with a Truffle loop node to benefit from On Stack Re-placement (OSR). But it turned out that the mapping of the ECC logic is too complicated and leads to superfluous code which was executed 20 percent slower than the current version of the ECC node. Another approach was to store the current state into the Truffle Frame to avoid node replacements, but this also did not improve the performance, as the access to the frame is expensive.

8 Related Work

This chapter will describe two different IEC 61499 implementations, which are related to the IEC Truffle implementation. It also states a different technology than Truffle, to implement new languages in a hardware-independent way, namely LLVM.

8.1 nxtControl

This work is supported by the company *nxtControl GmbH* [26] which is part of Schneider Electric [27]. The company has developed several commercial products that form an IEC 61499 ecosystem. One component is the *nxtEcort* runtime system, which is able to process every function block type and establish network communication. The runtime works similar to Truffle IEC. It loads a function block application represented as ST, and the integrated compiler translates the source code into machine instructions of the target platform. Therefore, in contrast to Truffle IEC, the *nxtEcort* runtime executes the function block application as compiled code.

Beside the runtime, *nxtControl* has also developed the Integrated Development Environment (IDE) *nxtStudio*. It enables hardware-independent IEC 61499 engineering. It is possible to create new function blocks in ST or use existing ones from the system library which is called *nxtLib*. Using *nxtStudio*, the 61499 developer can engineer an IEC 61499 application either in a graphical notation or by writing ST code. *NxtStudio* then sends the developed application to the runtime where it is executed.

Another product by *nxtControl* is *nxtHMI*, which provides Human Machine Interfaces (HMI) and their corresponding runtime systems for IEC 61499 applications. *NxtHMI* offers a variety of visualisation devices, which can be used to monitor and control industrial processes.

8.2 4diac

4DIAC is an open source framework which provides a complete implementation of IEC 61499. It stands for **Framework for Distributed Industrial Automation and Control** and was founded by Alois Zoitl [8] [28].

Similiar to *nxtControl*, 4Diac offers an IDE, a runtime environment and a system library. The 4Diac IDE offers several features, like a system explorer, an application editor, a distribution editor or deployment on the runtime system. The real-time-capable runtime environment is called *4diac FORTE* and is able to execute all types of function blocks on different operating systems [28]. The runtime provides a communication infrastructure to communicate within a IEC 61499 network. The 4diac system

library is called *4diac LIB* and provides a set of reusable function blocks that can be executed on *4diac FORTE*.

8.3 LLVM

Unlike *nxtControl* or *4diac*, Low Level Virtual Machine (LLVM) [29] is not related to IEC 61499, but it is related to Truffle. The LLVM project provides an infrastructure for reusable compiler tools. It offers different frontends, an IR, an optimizer and code generators for hardware dependent target architectures.

The available frontends can parse different input languages and translate them into LLVM bitcode, which is the core of LLVM and is similar to RISC assembly byte code. The backends can generate machine code for the corresponding target platform from LLVM bitcode. It is also possible to interpret LLVM bitcode or to use a JIT compiler. For example, the Sulong project [30] aims at translating LLVM bitcode to Truffle ASTs in order to profit from the Graal VM toolchain.

Whereas Truffle can map the semantics of a language to Java syntax, LLVM frontends must map the semantics to LLVM bitcode, which is much more challenging as it does not provide high-level constructs. No IEC 61499 to LLVM compiler has been implemented so far, but this would be an interesting topic for future work.

9 Future Work

This section discusses some possibilities to improve and extend the current Truffle IEC implementation.

Completeness
To achieve completeness, it must be possible to execute more than one function block type. Therefore, also composite and service function blocks need to be implemented. Another issue is to ensure real-time behaviour and to implement the time data type.

Execution Control Chart
In terms of performance the ECC could be a bottleneck, so it would be great to optimize the ECC in various ways. For example, it would be possible to merge states that execute the same algorithm. Another common way in other IEC 61499 implementations is to generate the ECC as Structured Text code and optimize the generated code.

Multi Language Execution Environment
An interesting topic for future work would also be to use the existing polyglot features from Graal to implement an execution environment for function blocks, that allows the function block developer to write the algorithm in any language installed on Graal. This would also be permitted by the IEC 61499 standard as it does not define the language of the implemented algorithm.

10 Summary and Conclusion

This thesis described an IEC 61499 implementation on top of the Graal VM. It showed how to parse a function block with Coco/R and how to insert its elements into a symbol ta-ble. During the parsing of the function block, an AST has been built as executable Truffle nodes. With the help of the Truffle framework, it was possible to execute the nodes and to benefit from Graal's runtime optimizations and JIT compilation.

We showed that it is possible to embed a basic function block, that is executed on Truffle, in an IEC 61499 distributed application. In some cases, especially for algorithms with a high runtime complexity, it turned out that the function block is executed faster compared to common systems. A possible application scenario would be to outsource computational-intensive tasks to Truffle IEC.

11 Figures

12 Listings

Acknowledgements

This thesis was supported by the company nxtControl, which is part of Schneider Electric. Very special thanks goes to Walter Oberndorfer from nxtControl, who has made this work possible and was very involved and always provides valuable feedback, even if he was very busy.

I would also like to thank my supervisor Hanspeter Mössenböck for continuous constructive feedback during this work and Herbert Prähofer for providing ideas and literature about IEC 61499.

Another special thanks goes to Alois Zoitl, who is one of the main persons behind IEC 61499. After talking to him I was always sure that I implemented correctly regarding to the standard.

13 Bibliography

[1] *IEC 61499-1: Function blocks - Part 1: Architecture.* Ser. International standard, IEC 61131-3. International Electrotechnical Commission, 2012. [Online]. Available: https://webstore.iec.ch/publication/5506.

[2] *IEC 61131-3:Programming languages Automates programmables.* Ser. International standard, IEC 61131-3. International Electrotechnical Commission, 2013. [Online]. Available: https://webstore.iec.ch/publication/4552.

[3] J. Thönes, "Microservices", *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015, ISSN: 0740-7459. DOI: 10.1109/MS.2015.11.

[4] T. Lindholm, F. Yellin, and A. Buckley. (Feb. 8, 2019). The java virtual machine specification, [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se12/jvms12.pdf.

[5] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer, "Self-optimizing ast interpreters", in *Proceedings of the 8th Symposium on Dynamic Languages*, ser. DLS '12, Tucson, Arizona, USA: ACM, 2012, pp. 73–82, ISBN: 978-1-4503-1564-7. DOI: 10.1145/2384577.2384587. [Online]. Available: http://doi.acm.org/10.1145/2384577.2384587.

[6] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all", in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013, Indianapolis, Indiana, USA: ACM, 2013, pp. 187–204, ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581. [Online]. Available: http://doi.acm.org/10.1145/2509578.2509581.

[7] H. Mössenböck, "A generator for production quality compilers", in *Proceedings of the Third International Workshop on Compiler Compilers*, ser. CC '90, Schwerin, Germany: Springer-Verlag, 1991, pp. 42–55, ISBN: 0-387-53669-8. [Online]. Available: http://dl.acm.org/citation.cfm?id=111924.111928.

[8] A. Zoitl and R. W. Lewis, *Modelling Control Systems Using IEC 61499*, 2nd ed. 2014, ser. IET Control Engineering Series 95. The Institution of Engineering and Technology, 2014, ISBN: 1849197601, 9781849197601.

[9] D. Rubini, "Entwurf einer virtuellen maschine für die abarbeitung von iec 61499 basic funktionsblöcken", Diplomarbeit, Institut für Automatisierungs- und Regelungstechnik, TU Wien.

[10] J. G.B.J.G.S. G. Bracha and A. Buckley. (). The java java language specification, [Online]. Available: https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf.

[11] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger, "A domain-specific language for building self-optimizing ast interpreters", in *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2014, 2014, ISBN: 978-1-4503-3161-6.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph", *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991, ISSN: 0164-0925. DOI: 10.1145/115372.115320. [Online]. Available: http://doi.acm.org/10.1145/115372.115320.

[13] O. Labs. (2019). Graal vm native image, [Online]. Available: https://www.graalvm.org/docs/reference-manual/aot-compilation/.

[14] ——, (2019). Substrate vm, [Online]. Available: https://www.oracle.com/technetwork/java/jvmls2015-wimmer-2637907.pdf.

[15] ——, (2019). Substrate vm limitations, [Online]. Available: https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md.

[16] H. Mössenböck, *The compiler generator coco/r - user manual*, 2010.

[17] N. Wirth, "What can we do about the unnecessary diversity of notation for syntactic definitions?", *Commun. ACM*, vol. 20, no. 11, 822–823, Nov. 1977, ISSN: 0001-0782. DOI: 10.1145/359863.359883. [Online]. Available: https://doi.org/10.1145/359863.359883.

[18] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck, "High-performance cross-language interoperability in a multi-language runtime", in *Proceedings of the 11th Symposium on Dynamic Languages*, ser. DLS 2015, Pittsburgh, PA, USA: ACM, 2015, pp. 78–90, ISBN: 978-1-4503-3690-1. DOI: 10.1145/2816707.2816714. [Online]. Available: http://doi.acm.org/10.1145/2816707.2816714.

[19] Oracle. (2019). Interface blockingqueue, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html.

[20] ——, (2019). Java datagrammsocket, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/net/DatagramSocket.html.

[21] ISO Central Secretary, "ISO/IEC 8824-2", en, International Organization for Standardization, Geneva, CH, Standard, 2015. [Online]. Available: https://www.iso.org/standard/68351.html.

[22] ——, "ISO/IEC 8825-7", en, International Organization for Standardization, Geneva, CH, Standard, 2018. [Online]. Available: https://www.iso.org/standard/68345.html.

[23] (2020). Codemr, [Online]. Available: https://www.codemr.co.uk/.

[24] (2020). Robot framework, [Online]. Available: https://robotframework.org/.

[25] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series", *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965, ISSN: 00255718, 10886842. [Online]. Available: http://www.jstor.org/stable/2003354.

[26] (2020). Nxtcontrol gmbh, [Online]. Available: https://www.nxtcontrol.com/en/.

[27] (2020). Schneider electric, [Online]. Available: https://www.se.com/at/de/.

[28] (2020). 4diac, [Online]. Available: http://www.fordiac.org.

[29] (2020). Llvm, [Online]. Available: `http://llvm.org/`.

[30] M. Rigger, M. Grimmer, and H. Mössenböck, "Sulong - execution of llvm-based languages on the jvm: Position paper", in *ICOOOLPS '16*, 2016.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 04 Februar 2020

Michael Jäger, Bsc.