
BPMN2uku - An Eclipse Plugin for Generating ukuFlow's Code from Business Process Model Notation

Bachelor-Thesis von Hien Quoc Dang
Supervisor: Pablo Guerrero
November 2012



TECHNISCHE
UNIVERSITÄT
DARMSTADT



DVS

BPMN2uku - An Eclipse Plugin for Generating ukuFlow's Code from Business Process Model Notation

Vorgelegte Bachelor-Thesis von Hien Quoc Dang

Supervisor: Pablo Guerrero

1. Gutachten: Prof. Alejandro Buchmann Ph.D.

2. Gutachten: Pablo Guerrero

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 29th November 2012

(Hien Q. Dang)

Contents

1	Introduction	3
1.1	Problem Statement	3
1.2	Proposed Approach	4
1.3	Related Work	4
2	BPMN2uku Plugin Model and Design	6
2.1	Allowed Elements	6
2.2	Process Well-formedness	6
2.3	Script Syntax	7
2.3.1	UkuExpression	7
2.3.2	Sequence Flow Condition	8
2.3.3	Script Task	8
2.3.4	Scope Creation Script & Text Annotation	9
2.3.5	Event Based Script	9
3	Implementation	13
3.1	Features	13
3.2	Fetching Data from BPMN2 Diagram and Optimization	15
3.2.1	XML format	15
3.2.2	Fetching Diagram	16
3.2.3	Processing Mixed Gateways	16
3.3	Scripting	17
3.4	Validation	18
3.5	Conversion Process	20
3.6	Deployment	21
3.6.1	Windows	21
3.6.2	Mac OS	22
3.6.3	Linux	22
3.6.4	Deployment Process	22
3.7	Used Libraries	23
3.7.1	Java Compiler Compiler	23
3.7.2	JDOM 2	23
3.7.3	RXTX	23
3.8	Download	23
4	Evaluation	25
4.1	Cross Platform Test	25
4.2	Performance Test	25
5	Conclusions & Future Work	26
5.1	Conclusions	26
5.2	Future Work	26
	List of Figures	27
	Bibliography	28

1 Introduction

In this day and age, the usage of wireless sensor networks (WSNs) is continuously increasing. The demand for using WSNs appears in many areas, for example environmental, structural or industrial monitoring, smart buildings, military applications. Nevertheless, there is still a gap to be closed so that experts in these domains (the end users) can develop applications easily for WSNs. This is concerned with difficulties in programming a WSN, which requires a good knowledge of sensor device and wireless communication.

A solution usually makes use of middleware to build an abstract level on top of WSNs, which offers users an easier way to control a WSN and interact with it. For this purpose, the ukuFlow project aims at providing a platform which runs entirely on WSNs, performs different tasks and controls network behaviors. In ukuFlow, users can precisely define the behavior of the network in form of a business process, and let the platform execute it on the network [11]. However, the bytecode employed by ukuFlow for the representation of a correct process requires a corresponding tool for its mainstream usage.

In this work, we present an Eclipse plug-in called BPMN2uku, which provides support for validating and converting a BPMN2 process to ukuFlow's bytecode and therefore facilitating the usage of the ukuFlow engine.

The content of this report is structured as follows: In the rest of chapter 1 we describe the existing problems and our proposed approach as well as an overview of related works. In chapter 2, we discuss about constraints of a ukuFlow process and the BPMN2uku plugin model. The implementation and the algorithms for checking well-formedness of BPMN2 process are discussed in chapter 3. Chapter 4 describes some test cases and evaluations for BPMN2uku. Lastly, chapter 6 gives the conclusion and outline the future works.

1.1 Problem Statement

For using ukuFlow, users have to specify a workflow in the form of a business process which defines the desired activities of WSNs. This workflow must be represented in the bytecode format of ukuFlow. In comparison to the common XML representation of a business process using BPMN2, the bytecode format of the same process is much more compact. Due to the limited resource of a WSN node, the ukuFlow's bytecode format has advantage over XML representation.

Instead of text and XML tags, ukuFlow's bytecode uses codes to represent the BPMN2 elements, their attributes as well as the connections between them. The diagram's graphical information part is only useful for illustrating the process for end-users, thus it is not stored in ukuFlow's bytecode. By this way, the ukuFlow's encoding mechanism reduces size of a process about 98% (see the evaluation in chapter 4). Nevertheless creating a process directly in bytecode format is complicated and may be very time-consuming.

In this work we look at developing a tool which can automatically convert a BPMN2 process to ukuFlow's bytecode format, and thus fill the gap between designing a process and having the ukuFlow process in bytecode format.

The BPMN2 specification is used to describe a general business process and cover many types of processes. However, the number of elements which could appear in a BPMN2 process is overwhelming for its particular purpose in a WSN. For this reason, not all BPMN2 processes can be directly converted to ukuFlow's bytecode, since ukuFlow places additional restrictions on the accepted set of BPMN2 processes. As a result, the development tool should also be capable of checking the conformance of a BPMN2 process with an *ukuFlow process*. In the rest of this report, the term *ukuFlow process* is defined as a BPMN2 process, which can be converted to ukuFlow bytecode.

Another point in using ukuFlow is to enable the registration (i.e. uploading) of bytecode to the ukuFlow engine running on a sensor node, and deregistration later on. While without the proposed tool this is possible via the command line, the integration of this functionality into the development plugin greatly facilitates experimentation with and operation of an ukuFlow sensor network. The development tool should also provide users an easy way to manage sensor nodes if there are multiple sensors connect to the system.

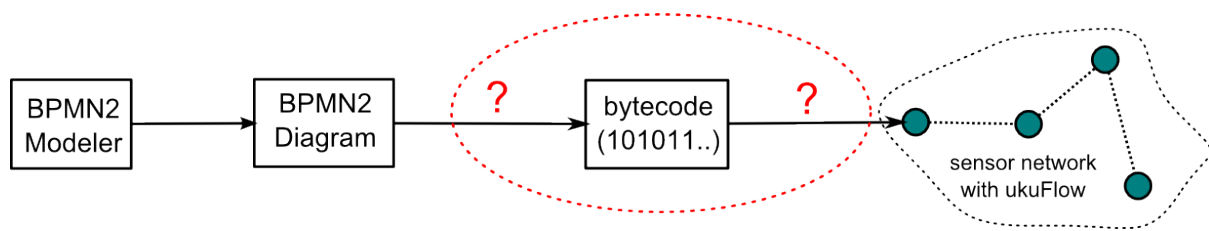


Figure 1.1: Gap between BPMN2 process and ukuFlow Engine

1.2 Proposed Approach

BPMN2 processes can be created easily by using a graphical IDE. The approach of this work is to create extension to one such IDE, namely Eclipse, which can integrate with the existing tool to assist users in designing *ukuFlow processes*, as well as to provide other functionalities to communicate with sensor nodes via serial port. The design and implementation of an Eclipse plugin for ukuFlow poses the following requirements:

- Check whether a BPMN2 diagram fulfills ukuFlow’s constraints (workflow’s constraints, script constraints etc.)
- Convert a BPMN2 diagram to ukuFlow’s bytecode
- Manage attached sensor nodes to the development computer, e.g., deal with node (dis)connections
- Enable registration and deregistration of a workflow into a ukuFlow engine.
- Support multiple platform OSs and provide an extensible design to accommodate for future ones.

Fortunately, an open-source BPMN2 plugin is already being developed, called BMPN2 Modeler. While it strives to provide the basic functionality (graphical representation of BPMN2 element’s and interconnection between them), and despite being in an incubation phase, it provides a starting point for this work.

For providing assistance to users in creating a correct ukuFlow process, in this work we implement a parser for the ukuFlow script task language, condition expressions and scope definitions. A grammar for the ukuFlow language is specified in extended *Backus – Naur* form (BNF) and ensures that all corrected expressions regarding this grammar can be converted to ukuFlow’s bytecode and all scripts, condition expressions in bytecode format must have an equivalent representation in the ukuFlow language. The parser is built on top of the ukuFlow grammar for syntactically verifying all integrated scripts and condition expressions.

In addition to script validation, a ukuFlow process is required to be well-formed (the well-formedness of a process is defined in Section 2.2), therefore this work is also concerned with validating the structure and the connectivity of BPMN2 processes. An algorithm for quickly checking the well-formedness is implemented by using the inductive definition of well-formed process.

The communication between sensor nodes and the development tool is realized by using an existing Java API, nevertheless, the mechanism for detecting all connected sensor nodes is not available and must be integrated into this tool. This detecting function is handled differently depending on the IDE’s underlying operating system.

1.3 Related Work

This work is the first attempt at offering a domain expert-friendly tool that generates ukuFlow bytecode, and thus is unprecedented. However, in the sense of providing a more comfortable environment for working with wireless sensor network, there are several projects which use Eclipse plugin to reach this goal.

YETI is an Eclipse plugin, which was developed by Burri et al. to support TinyOS development[1]. With this plugin installed, the Eclipse platform now become a IDE for *nesC* (programming language for TinyOS) developers. An abstract level for compiling and flashing command is built in the plugin, which assists users with a graphical wizard to define the *make* options. YETI also provides other features and functionalities, which can be used to speed up a time to develop an application for WSN or facilitate the newcomers in TinyOS.

BPMN2uku on the other hand aims to support ukuFlow’s end-users to develop applications in form of the BPMN2 processes without requiring knowledge of C or nesC.

As mentioned, ukuFlow requires that the deploying process must be well-formed. The validation mechanism of BPMN2uku is implemented using a recursive function (which is based on the inductive definition of well-formedness). There have been several proposals for verifying the well-formedness of a BPMN2 process. The approach of Nicola et al.[8], which defines some well-formedness rules for a process, requires that a process must be transformed into a sequence of characters, then the program checks if this sequence of characters satisfies all the rules. The checking process is done by XSB (a software which extends Prolog).

Forster et al. also propose a method for checking process by using pattern. This is already implemented as an Eclipse plugin. For verifying a specific process, user has to create a pattern for it. The program checks whether the process and its pattern match together [3]

A closely related project is Scope for WSN, which is developed by TU Darmstadt[14]. This project describes an approach for creating and working with scope in a wireless sensor network. Syntax of scope creation command is validated using JavaCC. There are also some similar points for creating a scope in ukuFlow's workflow and in the scope application.

2 BPMN2uku Plugin Model and Design

A BPMN2 process can claim to be conformant with a ukuFlow process if and only if it fulfills all constraints described in this chapter.

Firstly, a ukuFlow process *is* a BPMN2 process, therefore it must fully match all conformance points of the BPMN2 specification (described in [10]). Simultaneously, a BPMN2 process must also fulfill additional requirements to become an ukuFlow process. These requirements could be classified in three main categories: *Allowed Elements*, *Well-formedness Diagram* and *Script Syntax*.

2.1 Allowed Elements

As stated above, there are unconventional BPMN2 elements for a ukuFlow process and they should not be used for creating one. A list of currently supported BPMN2 elements and their descriptions are found in Table 2.1. A process which claims to be *fully compliant* with a ukuFlow process must consist only of the supported BPMN2 elements.

Processes which contain other elements, are *partially compliant* with ukuFlow if the elimination of all unsupported elements results in a *fully compliant* process. That is, a fully compliant process could be constructed by removing all unsupported elements from a partially compliant process.




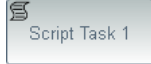







Element Name	Illustration	Description
Start Event		Initiation point of a ukuFlow process
End Event		End point of a ukuFlow process
Sequence Flow		Transition from an element to another
Script Task		Describe a task for the ukuFlow engine
Receive Task		Describe the event, which should be received from other sensors
Catch Event with Timer Definition		Specify the time out parameter for an <i>Event Based Gateway</i>
Text Annotation		Create a scope in the ukuFlow WSNs
Parallel Gateway		Indicate the parallel branching or parallel merging point in an ukuFlow process
Inclusive Gateway		Indicate the inclusive branching or inclusive merging point in an ukuFlow process
Exclusive Gateway		Indicate the exclusive branching or exclusive merging point in an ukuFlow process
Event Based Gateway		Indicate the branching point, which chooses the outgoing path based on the received events

Table 2.1: Allowed Elements in an ukuFlow process

2.2 Process Well-formedness

A correct process for the ukuFlow engine is required to be well-formed. The well-formedness of a process is concerned with the structure of the process and connectivity of elements and could be divided in four main restrictions: 1) Process has one Start Event with no incoming and one outgoing sequence flow, 2) Process has one End Event with no outgoing

and one incoming sequence flow, 3) Each activity(Receive Task, Script Task, Catch Event etc.) has one incoming and one outgoing sequence flow, 4) Each diverging gateway must match with one converging gateway and vice-versa
 The fourth rule can be specified in more detail of matched gateways as follows: All flows, which are forked from a parallel gateway, an inclusive or an exclusive gateway, must join at a converging gateway with the same type. The flows from an event based gateway must join at an exclusive converging gateway.

Rules 1 - 3 describe directly how is the properties of an element, therefore these rules are suitable to integrated in the implementation of the plugin. However, rule 4 is informal could not be used inand needs to be redefined so that it could be implemented and integrated in the plugin. The algorithm for checking well-formedness and its implementation are discussed later on.

2.3 Script Syntax

In this project, syntax of all scripts and condition expressions is defined using Extended Backus Naur forms. Currently, there are 3 types of scripts (in *ScriptTask*, *ReceiveTask* and *TextAnnotation*) and one type of condition expression (in the *sequence flow*) that user could enter to an ukuFlow process. These scripts again could contain different types of commands and expressions. In this Section, the syntax of *UkuExpression* is introduced firstly, which is used in definitions of scripts and condition expression. It is worth to note that an *UkuExpression* is not a complete expression in a script syntax but part of a complete expression.

2.3.1 UkuExpression

```

Number = [0-9]+ | 0x [0-9a-fA-F]+ | 0b [0-1]+
identifier = [a-zA-Z, _][a-zA-Z0-9_]*
Variable = '$' identifier
TerminationTerm = Number | Variable | RepositoryField

/* definition of UkuExpression */
ukuExpression = logicalOR
logicalOR = logicalAND ('OR' logicalAND)*
logicalAND = equalityExp ('AND' equalityExp)*
equalityExp = relationalExp (equalityOP relationalExp)*
relationalExp = additiveExp (relationalOP additiveExp)*
additiveExp = multiplicativeExp (additiveOP multiplicativeExp)*
multiplicativeExp = unaryExp (multiplicativeOP unaryExp)*
unaryExp = TerminationTerm | 'NOT' unaryExp | '(' ukuExpression ')'
```

Listing 2.1: UkuExpression in Extended Backus Naur Form

UkuExpressions are similar to the computational expressions in high-level programming language, they are constructed as infix expressions. Operators in *UkuExpressions* are depicted by special symbols and keywords. While computing an *UkuExpression*, the operator with higher priority operators are evaluated before the lower one. Parentheses can be used to force the evaluation order or to describe nested expressions. A list of operators ordered by their priorities are shown in the table 2.2. Except the logical operators have two representations using symbol and keyword, the others have just one representation as a symbol.

Operands of *UkuExpressions* could be numbers, variables, reserved keywords or other *UkuExpressions* (nested expression):

The reserved keywords are used for accessing resources of a sensor node. Except for keyword *NODE_ID*, which is used for getting the static *ID* of the current sensor node, the other keywords in table 2.4 contain dynamic values of sensor nodes. For example: *SENSOR_TEMPERATURE_RAW* is a keyword that holds the raw value of temperature sensor.

Variables in *UkuExpressions* are denoted by a sequence of characters, starts with a dollar symbol (\$) and followed by a string (variable's name). Variable's name is a sequence of literals (both lower and upper cases) and numbers and must begin with a literal. The reason of using dollar symbol is that in some expressions both strings and variables are allowed, so a dollar symbol makes it possible for the parser to distinguish between them. Variables are not only used in *UkuExpressions* but also in other expression, they can be created and updated in *computational function statements* and *local function statements*. Variables are used for storing temporary results for later usage, once created, a variable is available for the whole process (i.e. an activity can use variables, which are created in other activities).

A number in an *UkuExpression* could be represented in 3 forms: decimal, hexadecimal and binary. The hexadecimal representations of a number must start with *0x* ('zero x') followed by a sequence of digits from 0 to 9 and literal from

Operators	Meaning
* % /	multiplicative
+ -	additive
< > >= <=	relational
= = ! =	equality
AND &&	logical and
OR	logical or
NOT ~	logical not

Table 2.2: Operators and their priority

A to F. The binary representation must start with *0b* ('zero b') and followed by a sequence of digits of 0 and 1. Decimal representation is a sequence of digits from 0 to 9. Literals in all of these representations are not case sensitive.

2.3.2 Sequence Flow Condition

In a ukuFlow process, gateways are used to control the flow of the token. While parallel diverging gateway multiplies the token and put one token in each outgoing path, the exclusive and inclusive diverging gateways have to evaluate the conditions of outgoing path. Only outgoing paths whose condition is evaluated to *true*, are taken as the next destination of the token. When designing an ukuFlow process, all sequence flows which come from an inclusive or exclusive diverging gateway must have a condition (or be the default sequence flow). That is also specified as a requirement for a BPMN2 process [10].

In ukuFlow process, the condition in a sequence flow is an UkuExpression. The evaluation of conditions is similar to C programming language: In the ukuFlow engine, condition expression is computed to a number, a result of zero means it is evaluated to *false*, and *true* otherwise. For example, the expression $3+1$ (will be evaluated to true) or $SENSOR_TEMPERATURE_CELSIUS > 23$ are both valid expressions.

2.3.3 Script Task

Script Task is a type of activity in a ukuFlow process. It is used to specify the some tasks (at least one) that a user wants to execute in WSNs. Script Task could have multiple task, each of them is described in a statement. There are three types of tasks (i.e statements), which are *Scope Function Statement*, *Local Function Statement* and *Computation Statement*.

```
(computationStatement | localFunctionStatement | scopeFunctionStatement)+
```

Listing 2.2: E-BNF of a Script Task

The syntax and purpose of each statement in script task are explained in the following sections:

Computation Statement : The computation statements are used to assign a new value for an existing variable or to create and initiate a new variable. The syntax is similar to a simple assignment operator (=) in C, which takes the value on the right side of the statement and stores it to the variable on the left side. Expressions on the right side must be an UkuExpression.

The variable on the left side is, in this case, unambiguous and takes a new value, therefore it should not have \$ as the prefix, just the variable name need to be specified.

Examples for correct computation statements:

- $var1 = NODE_ID + 1;$
- $var2 = SENSOR_TEMPERATURE_CELSIUS / \$ var1.$

```
<variable> = <expression >;
```

Listing 2.3: Syntax of Computation Statement

Local Function Statement : Local function statements are used for executing commands locally. Besides, the result of executing command could be stored in a variable. Syntax of local function statements starts with the keyword

local, then a variable and an equality sign if users wants to save the result to a variable (it is optional), followed by the function name and a list of parameters, and a semicolon at the end. Parameters can be numbers, variables, strings or reserved keywords. It is allowed to have a large number of parameter although in a real application the amount of parameters will be rather small. An example for a correct local function statement which make all LEDs of the current sensor node blink *n* times, where *n* equals the node's id:

```
local blink NODE_ID;
```

```
local [<variable> =] <function_name> [parameter_1] [parameter_2]... ;
```

Listing 2.4: Syntax of Local Function Statement

Scope Function Statement : The purpose of this statement is to execute a command in a group of sensor nodes (a Scope, see 2.3.4). The statement assumes that the related scope is created some where in the process (by specifying scope creation script in a text annotation see 2.3.4).

The scope function statement always starts with @ symbol followed by the scope's name, then the name of the executing function, list of parameters split by whitespace characters, and lastly, a semicolon at the end. Parameters of scope function statement could be numbers, variables, strings or reserved keywords (see 2.4). The number of parameters is not limited by script syntax, but depends on each particular function, it may require a certain amount of parameters. For example, function *blink* requires only one parameter to indicate how many times the sensor's LEDs should blink:

```
@firstFloor blink 3;
```

```
@<scopename> <function_name> [parameter_1] [parameter_2]... ;
```

Listing 2.5: Syntax of Scope Function Statement

2.3.4 Scope Creation Script & Text Annotation

A scope in WSNs is a group of sensor nodes which have some common properties or functionalities[14].

With the ukuFlow engine, users have the ability to create and work with scopes. As described above, *Scope Function Statements in Script Task* are used for executing commands in a scope of WSNs. The definition of a new scope is achieved by designating the a script in Text Annotation (TextAnnotation is an allowed elements in ukuFlow process). Each script must be put in a separated Text Annotation.

The syntax of scope definitions always starts with the keyword *SCOPE* followed by the name of the creating scope (i.e a string without whitespace characters). The next parameter is a number in the range of 0 and 65535 which describes the time-to-live (in second) of the scope. This parameter is optional and has a default value of 60 (seconds). That means if user does not specify the time-to-live for a scope, the plugin will take the default value. The last parameter is an UkuExpression, which specifies the condition of a sensor node in the scope. Any sensor node in the deployed WSNs belongs to this scope only if it could evaluate the expression to *true*.

```
SCOPE <scope_name> [time_to_live] ( <ukuExpression> )
```

Listing 2.6: Syntax of a Scope Definition

2.3.5 Event Based Script

Event Messages

Event messages are data packets which are utilized by the ukuFlow event manager for informing other sensor nodes about the occurred events. Event messages are used by the ukuFlow engine exclusively for making decisions at an event-based branching point (will be described in next section). The structure of event messages consists of six fields which could be accessed in the expressions of event-based scripts by calling the specific keywords (cf. Table 2.3).

Keyword	Value Range	Description
TYPE	0...1	type of messages
NODE	0...65535	node id
SENSOR	0...255	sensor type
MAGNITUDE	-32768...32767	sensor value
TIME	$0 \dots 2^{64} - 1$	timestamp
SCOPE	0...255	scope id

Table 2.3: Event Message's Fields [11]

Event Based Script

Besides the static branching condition where the condition can be evaluated by the sensor node itself, ukuFlow also supports dynamic branching condition, where decisions comes from the networks (i.e event message from other sensor nodes). Therefore, condition of each outgoing path is not an UkuExpression but the description of desired messages. The corresponding outgoing path of a condition is taken only if the sensor node receives messages that match the description.

ukuFlow processes make use of the Event Based Gateways to designate a dynamic branching. In comparison to other types of gateways (i.e inclusive, exclusive) where the conditions are placed in outgoing sequence flows, conditions for each outgoing path must be specified in a *Receive Task* which connects directly to the Event Based Gateways (see figure 2.1). The condition expressions specifying in Receive Tasks must follow the syntax of an Event Based Script

Except outgoing paths with condition (message description), an Event Based Gateway could also have outgoing path with a timeout parameter (and no condition). We call it *timeout path*. The timeout parameter specifies the maximum time, that one should wait for detecting an event received from the network. When the waiting time expires, if none of the event conditions of other outgoing paths was satisfied by the arrived events, token flows to the *timeout path*. This feature of ukuFlow's event based gateway ensures that the token is not completely blocked when the desired messages never arrive. The *timeout path* acts like a default path for Event Based Gateway if no other path is taken after a certain time.

The *timeout* parameter must be described in an *Intermediate Catch Event* with *Timer Definition* which connects directly to the Event Based Gateway (see figure 2.1)

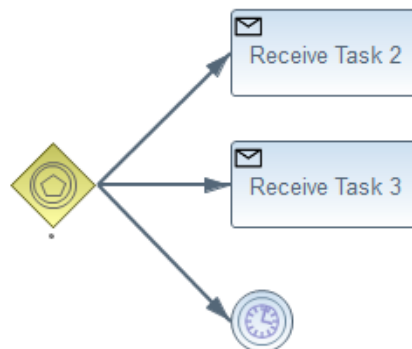


Figure 2.1: Event Based Gateway

For sufficiently specifying the conditions for Event Based Gateway, ukuFlow provides five types of expressions which could be classified into two main groups: *Event Generator* (EG) Expressions and *Event Filter* (EF) Expressions.

The EG Expressions can be used to specify when and at which sensor nodes should the event messages be generated as well as what kind of sensor's data should be attached to the messages. On the other hand, EF Expressions are used to specify the constraints to filter out the received messages.

The syntax and semantics for each type of Event Based condition expression are described as follows:

Periodic Event Generator: This command is used to periodically generate events in a scope. It begins with the keyword *PERIODIC_EG* followed by the sensor type (see table 2.4), then a symbol \wedge (stands for delta) and the period time in second. As the last parameter are two symbols *@s* followed by a string representing the scope's name.

```
PEG : PERIODIC_EG <SENSOR_TYPE> ^<PERIOD> @s <SCOPE>;
PERIOD : [0-9]+
```

Listing 2.7: Syntax for Periodic Event Generator

Distribution Event Generator: Distribution EG generates event messages within a scope. It differs from Periodic EG in that the frequency for generating events is specified by a function. Syntax of Distribution EG starts with keyword *DISTRIBUTION_EG*, followed by sensor type, then the symbol ^, a function and list of parameters to describe the distribution function, then the scope specifying by @s and name of the scope.

```
DEG : DISTRIBUTION_EG <SENSOR_TYPE> ^<DISTRIBUTION> @s <SCOPE>;
DISTRIBUTION : <function_name> [<parameter>]*
```

Listing 2.8: Syntax for Distribution Event Generator

Patterned Event Generator: This command is also similar to other event generators, but it uses a pattern to define the frequency of generating event messages.

Syntax of Patterned EG adopts the keyword *PATTERNED_EG* to indicate the start of expression. Then it is followed by the sensor type, symbol ^ and the pattern description, and lastly is the scope description. The pattern description consist of two parts, a sequence of 1 and 0 and a number. Each character in the 1-0 sequence describes a time slot, where 1 means generate an event, 0 means do nothing. The second part of a pattern specifies time magnitude (in seconds) for each time slot.

```
PAEG : PATTERNED_EG <SENSOR_TYPE> ^<PATTERN> @s <SCOPE>;
PATTERN : [0,1]+ <TIME>
TIME : [0-9]+
```

Listing 2.9: Syntax for Patterned Event Generator

Simple Event Filter: By applying constraints to restrict the range of fields in a event message, Simple EFs are used to filter out the received event messages. The constraints are built by using comparison operator to limited the accepted range of each field in event messages. On top of that, logical expressions and parentheses could also be used for specifying nested and complicated constraints.

Supported logical operators are: *AND*, *OR* and *NOT*.

Supported comparators are: == (equal), != (not equal), > (greater than) < (less than), >=(greater or equal) and <=(less or equal).

```
SEF : SIMPLE_EF {[<CONDITION>]+} {<SOURCE>}
SOURCE : <variable>|<PEG>|<DEG>|<PAEG>|<SEF>|<CEF>
CONDITION : (<EVENT_FIELD> <operator> <VALUE>)|(<VALUE> <operator> <EVENT_FIELD>)
```

Listing 2.10: Syntax for Simple Event Filter

Composite Event Filter: ukuFlow also enables users to specify conditions where the decisions depend on not only one event but the combination of several events.

Composite EF expressions start with keyword *COMPLEX_EF* followed by a logical condition expression, then two optional parameters: *<COMPOSITION_POLICY>* and *<EVICTION>*. Operands of logical expressions can be identifiers (i.e variable), Event Generators or Event Filters. It is allowed to have nested logical expression for specifying complicated conditions.

For example : *COMPLEX_EF (PERIODIC_EG SENSOR_CO ^10 @s room110 AND PERIODIC_EG SENSOR_CO2 ^10 @s room200)*

```
CEF : COMPLEX_EF <OPERATION> | <COMPOSITION_POLICY> | <EVICTION>
```

Listing 2.11: Syntax for Composite Event Filter

TOP expression

As mentioned, the Event Filter expressions are built by using Event Generators or other Event Filters. It could make the expression for complicated filter become very big and hard to maintain. Therefore, we enable users to encapsulate an event expression (i.e Event Generator or Event Filter) through a simple identifier. This is done by specifying an assignment expression, where the identifier is on the left side and the Event expression is on the right side:

`<variable> = <event_expression>`

That is, users can specify several assignment expression on a event-based script, but in the end it should have one ultimate event expression, which will be evaluated when when ukuFlow engine processes the corresponding Event Based gateway. For this purpose, we introduce the TOP expression. It is similar to an assignment expression but instead of identifier, the keyword *TOP* must be specified on the left side. Each Event Based script must have one and only one *TOP* expression.

`TOP = <event_expression>`. The syntax of a complete Event Based script in eBNF forms are summarized in 2.12.

```
[<identifier> = <PEG> | <DEG> | <PAEG> | <CEF> | <SEF>]*  
TOP = <identifier>|<PEG>|<AEG>|<SEF>|<CEF>
```

Listing 2.12: Event Based Script Syntax

Keyword	Available in
SENSOR_LIGHT_PAR_RAW	TmoteSky
SENSOR_LIGHT_TSR_RAW	TmoteSky
SENSOR_TEMPERATURE_RAW	TmoteSky, Z1
SENSOR_TEMPERATURE_CELSIUS	TmoteSky, Z1
SENSOR_TEMPERATURE_FAHRENHEIT	TmoteSky, Z1
SENSOR_HUMIDITY_RAW	TmoteSky
SENSOR_HUMIDITY_PERCENT	TmoteSky
SENSOR_ACCM_X_AXIS	Z1
SENSOR_ACCM_Y_AXIS	Z1
SENSOR_ACCM_Z_AXIS	Z1
SENSOR_VOLTAGE_RAW	TmoteSky
SENSOR_CO2	<i>External sensor</i>
SENSOR_CO	<i>External sensor</i>
NODE_ID	TmoteSky, Z1

Table 2.4: Reserved Keywords as Predefined Constants [12, 2]

3 Implementation

3.1 Features

The BPMN2ukuFlow plugin is built on the Eclipse platform and integrates with BPMN2 Modeler plugin. Therefore BPMN2 Modeler should be installed together with BPMN2uku plugin. Users should also have some initial expertise in creating a BPMN2 diagram using BPMN2 Modeler plugin and understand the syntax ukuFlow's constraints which are required for issuing a valid ukuFlow process.

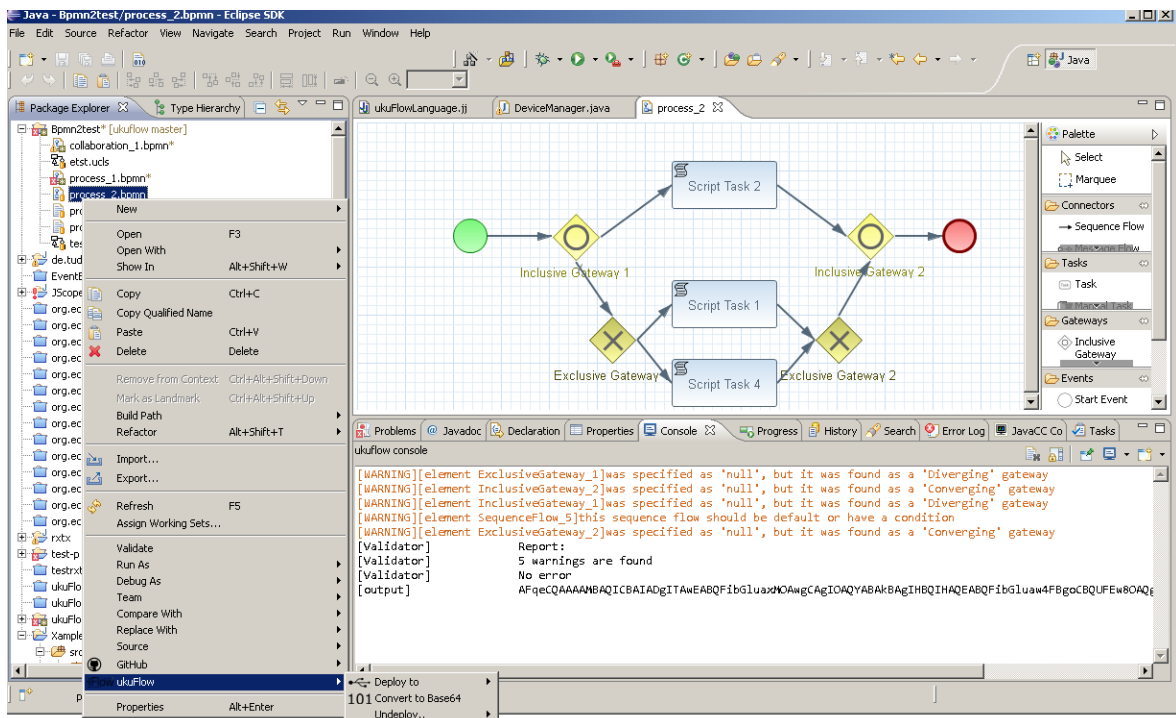


Figure 3.1: Overview of BPMN2uku plugin

At the GUI part, BPMN2uku provides an additional menu to the pop-up menu of the Eclipse platform, which contains up to four options: *Deploy*, *Undeploy*, *Validate* and *Convert to Base64*. These options represent the four core functions of the BPMN2uku plugin.

In addition, a console for ukuFlow is also attached to the Eclipse platform, which are used for the following purposes:

- Displaying feedbacks of validation process
- Displaying feedbacks from the node during deployment
- Alerting the user if something happen with the connectivity to sensor node

When users choose to validate a BPMN2 diagram, the plugin will check whether this BPMN2 diagram is suitable to be an ukuFlow's workflow. If it is not suitable, all errors and warnings are displayed in the ukuFlow console. Each error message contains the id of the faulty element and information about the error, so that the user is able to fix the diagram. The validation mechanism and ukuFlow's constraints are described on section 3.4

A preference page for ukuFlow was also designed where user can specify additional setting information for BPMN2uku. It is located under: "Window\Preference\ukuFlow"

The conversion command also validates the selected BPMN2 process, if no error is found during the validation process, it jumps to next step for converting the ukuFlow process to ukuFlow bytecode. Output of the conversion process is a

vector of bytes, which is encoded to base64 format and then stored in a file. The output file has the same name with the selected BPMN2 process but with *.uku64* extension. By default, output file is located under the same folder as of the original BPMN2 diagram file. User can change the destination folder to a subfolder by specifying the subfolder's name on the *Output Folder's Name* field of ukuFlow's preference page.

The bytecode data of a process often contains '0' bytes, which is one of the control characters of transmission and leads to interruption while it is transmitted to the sensor node. On the other hand base64 encodes the output vector to a string which does not contain the special control character¹.

The deployment option is only visible if users right-click on a BPMN2 diagram or an ukuFlow's bytecode file (*.uku64*). A list of available serial ports together with the sensor node identifier will be displayed, so that one can be chosen for deployment.

After uploading the process's bytecode to sensor node, the plugin keep connecting to corresponding serial port and listen to feedbacks from sensor node. Each message is displayed in the ukuFlow's console in following form:

- `[timestamp][portname] message`

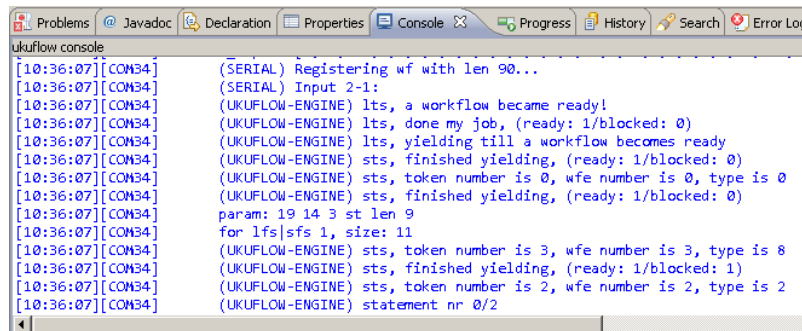


Figure 3.2: Console while Listening to the Serialport

During the deployment (undeployment) and listening process, the serial port of the utilized sensor node is blocked and only after a timeout occurs it is released and made available for other user's actions. The default timeout for deployment (undeployment) and listening process is 10 seconds, but it could be adjusted by entering a new value in range from 5 seconds to 300 second on *Connection timeout* field on the ukuFlow's preference page (see figure 3.3).

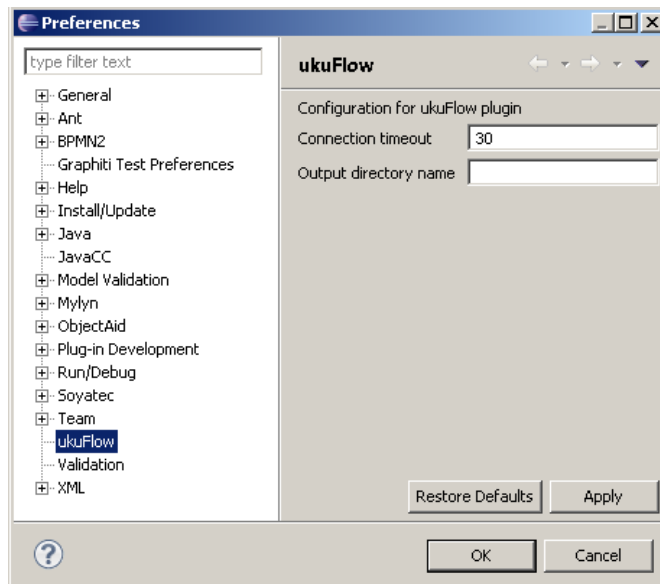


Figure 3.3: ukuFlow's Preference Page

The *Undeployment* function works similarly to the deployment but instead of uploading the whole bytecode data to the sensor node, it extracts id of the selected process and send it together with the deregistration command into the sensor node.

¹ <https://tools.ietf.org/html/rfc4648>

The sensor node detection function (i.e DeviceFinder), deployment and undeployment functions are all implemented in Java and require no installation of other software or libraries like Contiki or TinyOS.

The implementation of BPMN2uku plugin was divided into three main steps. In the first step, a basic structure of BPMN2uku plugin was built, which runs on Eclipse and can be extended by adding the implementation of other functionalities. Also a very first version of fetching function is built which reads and converts a BPMN2 process to Java objects for processing later on. The details implementations are found in section 3.2.2

In the second step, a grammar for almost all expressions in ukuFlow language (except Event-Based script) are defined, together with that, a script parser for this grammar is implemented by using JavaCC API. The plugin are now able to validate script tasks, condition expressions of static branching points and scope creation expressions. Valid scripts are also converted to Java objects by using the parser which could be converted to bytecode format later on. The implementation of these aspects is discussed in section 3.3

Within this step, the deployment function for uploading ukuFlow's bytecode together with the support function for finding all available devices are also implemented and are discussed in section 3.6

Third (and final) step is concerned with the validity of a BPMN2 process regarding its components and structure and the conversion function. In this step, a mechanism for collecting and reporting error was built and an algorithm for checking well-formedness was developed and implemented into the plugin. The parser for ukuFlow language was also completed with the parser for the Event-Based script.

Also, the implementation of the conversion function using the visitor pattern [4], an ukuFlow process could now completely be converted to bytecode and made ready for its deployment.

3.2 Fetching Data from BPMN2 Diagram and Optimization

3.2.1 XML format

The BPMN2 processes are stored in XML format which holds one root element with tag name "bpmn2:definitions". The root element consists of two child elements with tag name "bpmn2:process" and "bpmndi:BPMNDiagram". First element holds the description and properties of each component and connection in the process, second element holds the graphical diagram information (i.e location, height, width, etc.) which is needed for illustrating the element in GUI. Since the bytecode does not relate to graphical representation of a process, the second child of a process is ignored, just the first one needs to be fetched and converted to Java objects.

<pre><bpmn2:process id="process_2" name="Default Process"> <bpmn2:startEvent id="StartEvent_1"> <bpmn2:outgoing>SequenceFlow_2</bpmn2:outgoing> </bpmn2:startEvent> <bpmn2:sequenceFlow id="SequenceFlow_1" sourceRef="ScriptTask_1" targetRef="InclusiveGateway_2"/> <bpmn2:endEvent id="EndEvent_1"> <bpmn2:incoming>SequenceFlow_3</bpmn2:incoming> </bpmn2:endEvent> </bpmn2:process></pre>	<p>Elements Connections Properties</p>
<pre><bpmndi:BPMNDiagram id="BPMNDiagram_1" name="Default Process Diagram"> <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="process_2"> <bpmndi:BPMNShape id="BPMNShape_1" bpmnElement="StartEvent_1"> <dc:Bounds height="36.0" width="36.0" x="100.0" y="100.0"/> </bpmndi:BPMNShape> <bpmndi:BPMNShape id="BPMNShape_2" bpmnElement="EndEvent_1"> <dc:Bounds height="36.0" width="36.0" x="500.0" y="100.0"/> </bpmndi:BPMNShape> </bpmndi:BPMNPlane> </bpmndi:BPMNDiagram></pre>	<p>Diagram Information</p>

Figure 3.4: XML file: Process Information & Diagram Information

Each element of a BPMN2 process is described in a XML element and has a unique id as an attribute. This id is used to refer to the id holder. For example: to specify that a sequence flow with id "sequenceflow_1" is connected directly to a start event as an outgoing path, the id "sequenceflow_1" must be specified in the description of start event:

```
<bpmn2:startEvent ... >
  <bpmn2:outgoing>sequenceflow_1</bpmn2:outgoing>
</bpmn2:startEvent >
```

Listing 3.1: XML description of a Start Event

This attribute is important for the fetching program to link the element together and construct from the XML data a complete process using Java objects.

3.2.2 Fetching Diagram

The fetching process make use of JDOM2 library [7] to read the XML file and built up a new representation as a JDOM tree.

By following the structure of the XML file, the fetching program visits the JDOM tree and finds the element with tag name "bpmn2:process". Each BPMN2 element or connection is described in a child element of "bpmn2:process". They are classified based on their XML tag name from that an instance of the appropriate Java class is created. All appropriate classes must inherit from class *UkuEntity*, which has some general attributes and functions that all other elements must have. By using constructor- and setter- methods, all supported attributes and properties of a XML element are fetched and put into the Java object. A table of XML tag and its appropriate Java class could be found at 3.1. Each child element within the *ukuFlow process* has a unique attribute called **ID**. The IDs are used to refer to other elements, especially to describe how an element is connected to other elements. Attributes are classified using the attribute's name. Only supported attributes are processed and added to the Java object, the others are ignored and a warning message will be added to the error manager.

After the fetching process, all supported properties of a XML element are transferred into a Java object and a list of *UkuEntity* objects is returned as the result.

The parser functions are implemented in class *BPMN2XMLParser.java* under package *de.tudarmstadt.dvs.ukuflow.xml* and appropriate Java classes for all BPMN2 elements are stored in the package *de.tudarmstadt.dvs.ukuflow.xml.entity*

XML Element Tag	Java Class
sequenceFlow	UkuSequenceFlow
startEvent	UkuEvent
endEvent	UkuEvent
scriptTask	UkuExecuteTask
parallelGateway	UkuParallelGateway
exclusiveGateway	UkuExclusiveGateway
inclusiveGateway	UkuInclusiveGateway
eventBasedGateway	UkuEventGateway
textAnnotation	UkuScope

Table 3.1: XML Element and Java Class

3.2.3 Processing Mixed Gateways

A gateway in BPMN2 process could be *Converging*, *Diverging*, *Mixed* or *Unspecified*. However, gateways in a *ukuFlow*'s process should be either *Diverging* or *Converging*. The *ukuFlow* engine can also be extended to support Mixed Gateway, but since a mixed gateway could be efficiently split into two simpler gateways in software level, we decided to add a feature to BPMN2uku, which splits every mixed gateway in the process. This split function is applied to all mixed gateways right after the fetching process.

Firstly, we have to define, what is a Mixed gateway as well as Converging and Diverging gateways. In BPMN2 Editor, users are able to specify the direction for gateways, even if the provided specification is wrong. In most cases, the exact types of gateway could be determined by counting number of incoming and outgoing paths (see table 3.2). Still, it is impossible to determine if the gateway has just one outgoing and one incoming path. In this case, users have to manually specify type for the gateway and the program adopts this type for the gateway.

Incoming	Outgoing	Derived Type
n	1	Converging
1	n	Diverging
n	n	Mixed (<i>needs to be processed</i>)
1	1	(<i>needs to be specified</i>)

Table 3.2: Auto-determine Gateway Type (n = 2,3...)

Once all mixed gateways are determined, they are split into two connected gateways (one Converging and one Diverging) using following process:

For each mixed gateway,

1. Create two new gateway instance of the mixed gateway's class
2. All incoming sequence flows, which point to the mixed gateway, will be redirected to first new gateway (the converging gateway)
3. All outgoing sequence flows of the mixed gateway will be linked to the other gateway (the diverging gateway)
4. Create a new sequence flow, which connects the two new gateways
5. Remove the original Mixed Gateway from the workflow

By replacing one mixed gateway with two simpler gateways, the process actually increases the size of a workflow. However, adding the capability of processing mixed gateways into the ukuFlow engine requires more resources and is also hard to debug. The options to let ukuFlow engine or software process mixed gateway is slightly equal regarding sensor node' resources. An illustration for split process of an mixed gateway is shown in 3.5.

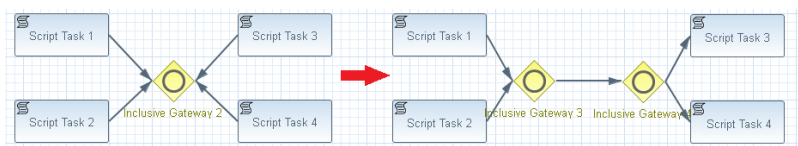


Figure 3.5: Workflow Enhancement for Mixed Gateway

3.3 Scripting

As mentioned, an ukuFlow process can contain several scripts and condition expressions. The grammar for each type of expressions in a ukuFlow process is already described in e-BNFs in section 2.3. In this project, we use JavaCC API to build the parser for scripts from the extended BNFs. Based on the structure of the expression, we have built 2 script parsers. One parser is responsible for validating Event-Based scripts. The second parser is implemented to handle the other type of script and condition expression. As described above, the script task, scope creation and static branching condition can contain UkuExpressions in their expressions, using only one parser for these three types of scripts to make the implementation of UkuExpression reusable.

By integrating Java code within the grammar specification in .jj file, the parser built by JavaCC can not only validate a user's script but also be able to convert it appropriate Java objects.

Parsing UkuExpressions

Firstly a set of Java classes is implemented, which represents for all types of supported operand and operations. Whenever an expression in the script matches with a BNF rule, the parser generates an instance of the corresponding Java class. For example the expression " $\$var1 + NODE_ID$ " is recognized as a binary expression and therefore an instance of the corresponding class `BinaryNumericalExpression` which extends from the `NumericalExpression` class (see 3.6), is created. The operands $\$var1$ and $NODE_ID$ are matched with classes `UkuVariable` and `UkuRepositoryField`, which are subclasses of `PrimaryExpression`.

Parsing nested expressions results an `UkuExpression` where its operands are again `UkuExpressions`. That is, the parser program will build up a tree for each `UkuExpression` where every leaf is a `PrimaryExpression` term (i.e number, variable, reserved keywords) and every node is an operation (i.e `LogicalExpression`, `NumericalExpression`). A graphical representation of a tree that is built from an `UkuExpression` is shown in figure 3.7.

Parsing Scripts

The implementation of parsers for scripts also follow the principle of parsing `UkuExpression`. For each type of expression and command there must be an corresponding Java class. An instance of the corresponding class is created when a Backus-Naur forms is matched.

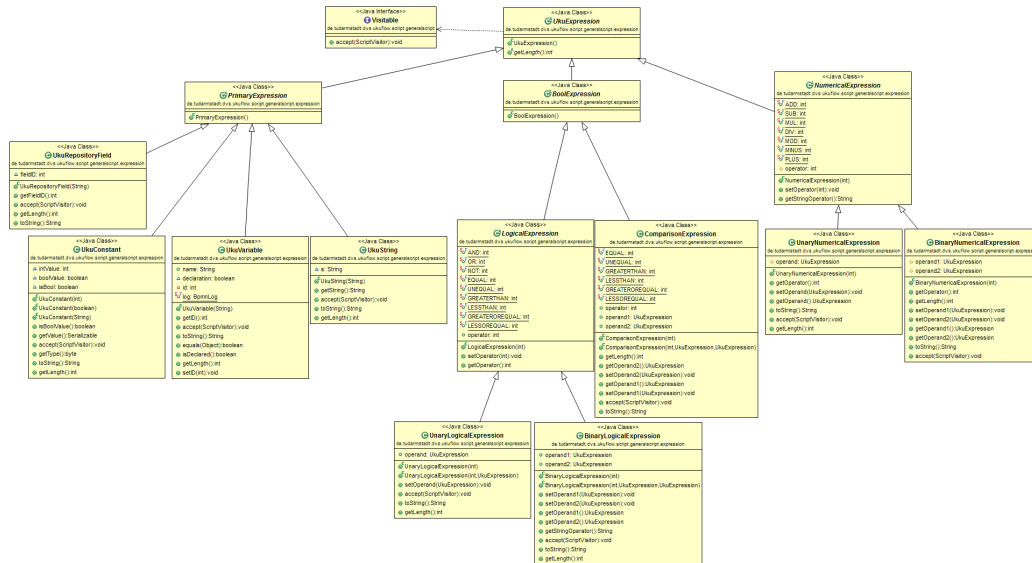


Figure 3.6: Class Diagram of UkuExpression

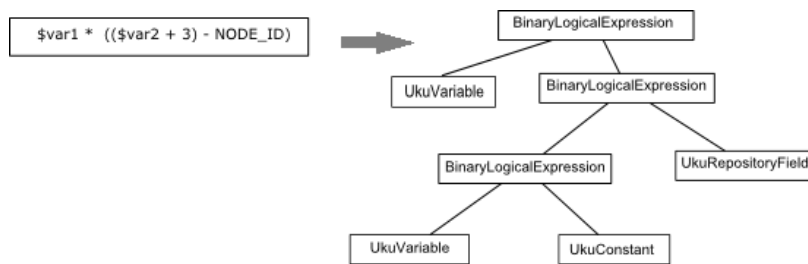


Figure 3.7: Example of parsing an UkuExpression

Script Type	Output Format (Java form)
TextAnnotation	UkuScope
Condition Expression	UkuExpression
Receive Task(i.e Event Based)	EEventBaseScript
Task Script	List of TaskScriptFunction

Table 3.3: Output format of Scripts

3.4 Validation

An ukuFlow's workflow is a BPMN2 with some restrictions. For example the script of script-task, condition within sequence flow and event base script have to match the ukuFlow's syntax (described in 3.3). That means before converting to ukuFlow's bytecode, the process needs to be verified whether it is compatible with a ukuFlow workflow.

Syntax errors of scripts are found by JavaCC during the fetching phase. The other type of errors would be found by running the validation function, which are implemented in class UkuProcessValidation under package `de.tudarmstadt.dvs.ukuflow.validation`. Errors are discovered in different phases, therefore the error- and warning-manager is implemented using the singleton pattern[4], which It allows to have only one instance of the error manager in the whole program. Now if the program found any error or warning, it just has to be added to this instance.

A correct process for ukuFlow engine is required to be well-formed.

A BPMN2 process is well-formed if it fulfills the following conditions: 1) there is one Start Event with no incoming and 1 outgoing sequence flow. 2) There is one End Event with no outgoing and one incoming sequence flow. 3) Each activity (Receive Task, Script Task, Catch Event etc.) has exactly one incoming and one outgoing sequence flow. 4) Each

diverging gateway must match with a converging gateway and vice-versa

The flows, which are forked from a parallel gateway, an inclusive or an exclusive gateway, must join at a converging gateway with the same type. The flows from an event based gateway must join at an exclusive converging gateway.

Batch Validation

The batch validation verifies a process by looking at each entity individually without considering the role of this entity in the whole process. In the implementation, the batch validation function travels through all entities of the process, performs the validity check by applying some rules (which rules should be applied depends on the type of entity). An entity is valid if all rules for this entity are passed. A process passes the batch validation if all entities are valid.

Batch validation could check the first three conditions of well-formedness, which are described above. In addition, it also checks some extended conditions for a ukuFlow’s workflow. The conditions are listed as follows:

1. A task script should have script, in other words, empty script is not allowed
2. A receive task must have script, empty script is not allowed
3. Sequence flow, which has source element is an inclusive or exclusive gateway, must have condition or be default sequence flow.
4. Sequence flow, which has source is the other element, should neither have condition nor be default sequence flow.

Sequential Validation and Well-formedness Check

Using the same idea as described in [8], we could also redefine the well-formedness of a business process by using the definition of well-formed subprocess.

A subprocess is a sequence of connected BPMN2 elements, which does not contain Start Event and End Event. A

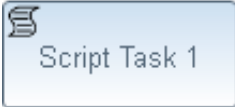
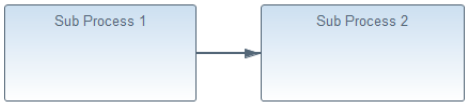
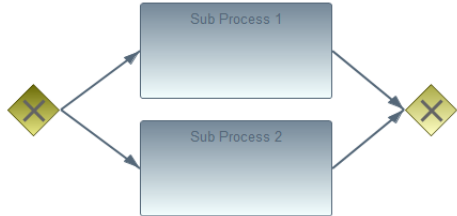
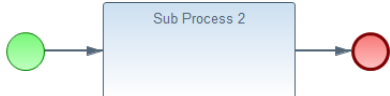
Description	Illustration
1. Rule: An activity is a wellformed sub-process	sub-process: 
2. Rule: Connecting the most-left element of a well-formed sub-process with the most-right element of another wellformed sub-process results a well-formed sub-process	sub-process: 
3. Rule: Sub-process starts with diverging gateway and ends with matched a converging gateway and each path between them is a wellformed sub-process, is wellformed	sub-process: 
A well-formed process is defined as a start event connect to a wellformed sub-process and this sub-process is connected to an end event	well-formed process: 

Table 3.4: Wellformed Process and Sub-process

inductive definition of wellformed subprocess is specified and illustrated in the first 3 rules in the table 3.4. While rule 1 and 2 make sure that every elements are connected and all elements can be reached, the third rule says that each

diverging gateway must match with a converging gateway. The inductive definition make sure that all subprocesses, which connect two matched gateways must be wellformed.

The last rule in the table 3.4 specify that a wellformed process starts with a start event, which is connected to a wellformed subprocess and end with a end event. Since checking number of start and end event is trivial, the last rule reduce problem of checking wellformedness of a process to checking wellformedness of the subprocess between the Start and End Event.

As the inductive definition of wellformed subprocess, the implementation of checking wellformedness is also done in a recursive function.

Basically the function will follow the workflow until it meets an element, that destroys the wellformedness of a subprocess. The recursive function will take an element as input parameter, and return the first element that destroy the wellformedness. It means that calling check function on a start event of a wellformed process must result the end event. A result of *null* or others element type is considered to be as non-wellformed. The pseudo code of this algorithms is shown in listing 3.2

There is an interesting case when the algorithm processes a diverging gateway. According to the 3. rule, there must be a matched converging gateway and all paths between them must be a well-formed sub-process. It means that if we travel to each outgoing path, it is expected to see a well-formed sub-process connected to a converging gateway. Therefore applying the wellformed check on each outgoing path must result a converging gateway, otherwise the process is not wellformed. Wellformed check must also return the same converging gateway for all outgoing paths. If it is the case, the last check is applied to confirm that this both gateway are matched. That is the number of incoming path of converging gateway must be equals to number of outgoing paths of diverging gateway. If any of these conditions is not fulfilled, it will raise an error to inform, that there are no matching gateway for the processed diverging gateway.

```
wellformedcheck(Element start)
    if (start==null) then return null;
    if (start==EndEvent) then return start;
    if (start==ConvergingGateway) return start;

    /* rule 1 & 2 */
    if (start==Activity) return then wellformedcheck(start.next);

    /* rule 3 */
    if (start==DivergingGateway) then {
        Set s;
        foreach(i=0 : start.getOutgoings().size()){
            Element t = wellformedcheck(start.outgoingpath[i])
            if(t == ConvergingGateway) then
                s.add(t);
            else
                return null;
        }
        if s.size() != 1 then // more than 1 converging for a diverging gateway
            return null;
        else
            return wellformedcheck(s.get(0).nextElement());
    }
```

Listing 3.2: Pseudo code for checking wellformedness

3.5 Conversion Process

After fetching and validating phase, a ukuFlow process is transformed into a list of linked Java objects. Make use of visitor pattern, conversion process visits each element in the process and converts it to bytecode. The output is an vector which is created by sequentially appending all elements' bytecode.

After visiting all element of a process, the final output vector is converted to *base64* format and stored in the a file with *.uku64* extension. In figure 3.8, a summary of conversion process from a BPMN2 process to ukuFlow bytecode.

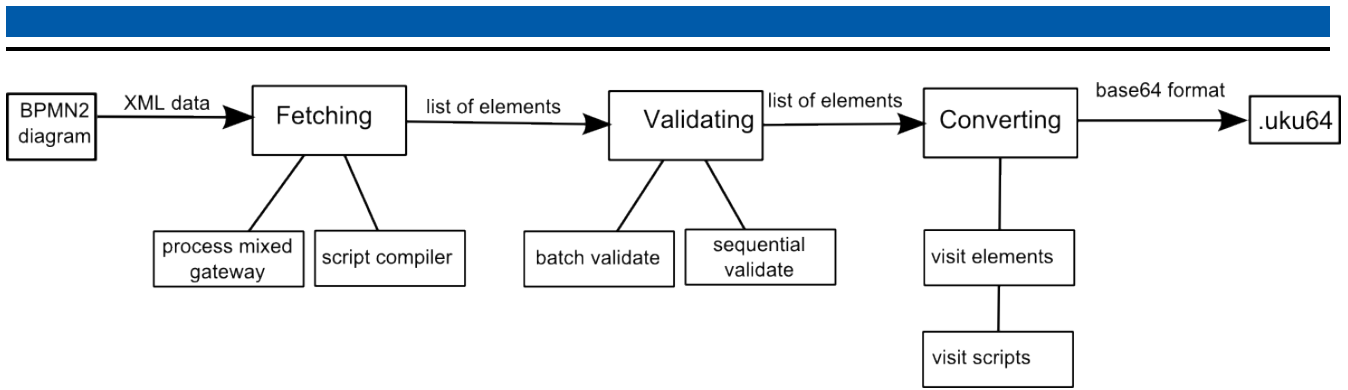


Figure 3.8: Conversion from BPMN2 to .uku64

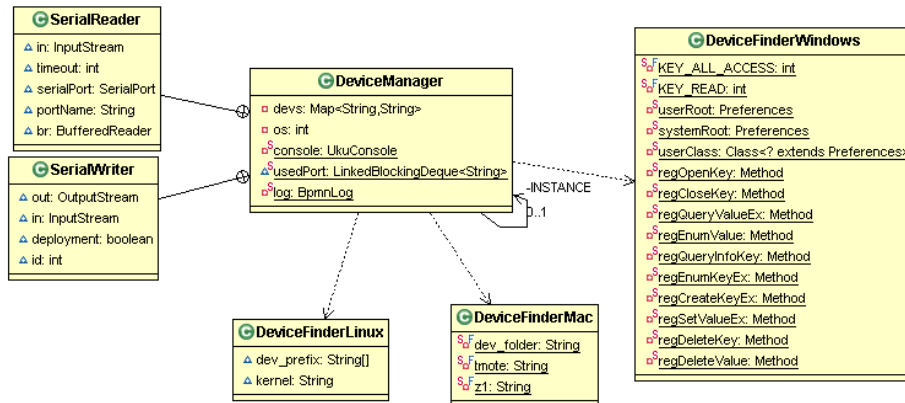


Figure 3.9: Class Structure of DeviceManager & Associated Classes

3.6 Deployment

Deploying a ukuFlow's process is actually uploading it to the serial port. For this purpose, RXTX library offers a java wrapper for communicating with serial port. RXTX can also provide a list of available serial ports but no information about which devices is connecting to these ports. The requirement is that only supported sensor devices are shown. For this purpose, a mechanism to detect sensor nodes currently connected to the IDE's computer must be implemented.

Each operating system manages the external serial devices and their communication ports in a different way, therefore it is necessary to determine which operating system the end-user is using. Java offers a class called System.java under package java.lang. Using this class, the name of the user's OS can be obtained by calling method getProperty(String) with parameter "os.name".

All the classes for finding sensor devices and deploying workflow are stored in package *de.tudarmstadt.dvs.ukuflow.deployment*. For each operating system, we have created a corresponding class, which has the purpose to find all connected and supported devices (*Tmote Sky* and *Zolertia*). Based on the detected OS, the *DeviceManager* will call the search function on appropriate class to get a map of device identification and its port.

Here is more detail on how the program search for device on each operating system

3.6.1 Windows

In Windows, the information about the devices and their serial port is stored in the *Windows Registry*. More specific for the *Zolertia* node, the device's information including the port number are stored in

"HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Enum\USB\VID_10C4&PID_EA60\DeviceID"

And the information about *Tmote Sky* devices is stored in

"HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Enum\FTDIBUS\VID_0403+PID_6001+DeviceID"

Entry of a device in *Registry* always has the vendor ID and product ID as the prefix, those ID combine together is unambiguous for each type of device, therefore they are used to find Z1 and Sky mote. Detail about VendorID and ProductID of supported devices could be found in table 3.5

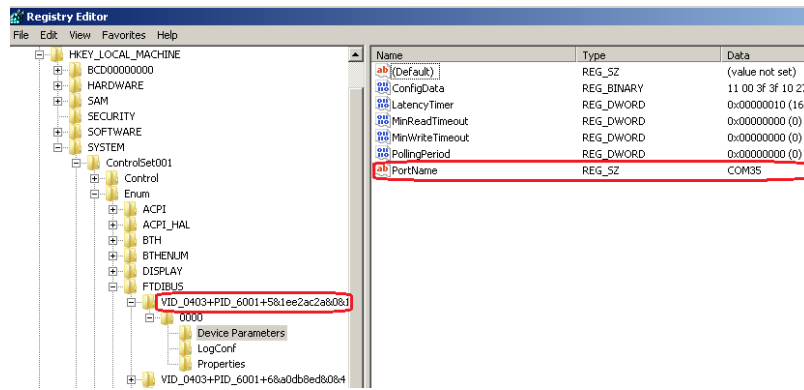


Figure 3.10: Windows Registry: Search for FTDI Device

Fetching the *Registry* results a map of device id and the port number, which has connected to the system. That is, the Windows Registry still keeps these device’s information even if the devices are removed from the system. Therefore this map has to be intersected with the list of available device. After fetching the registry information Finally we gain a list of supported and available devices and their corresponding communication port.

Device	USB controller	Vendor ID(VID)	Product ID(PID)
Zolertia	CP2102 USB-to-serial from SiLabs	10C4	EA60
Tmote Sky	FTDI	0403	6001

Table 3.5: Vendor ID & Product ID of supported Devices [12, 9, 5]

3.6.2 Mac OS

Under Mac OS X the offer only a few information about the connected device. The port is represented as a file under */dev/*, and the information about device is the file name itself. Each device will have a specific prefix in the file name (see table 3.6), therefore these prefixes are used to determine weather the device is supported or not.

Device	Port Prefix
Zolertia	<i>/dev/tty.SLAB_</i>
Tmote Sky	<i>/dev/tty.usbserial-</i>

Table 3.6: Port name prefix for each device

3.6.3 Linux

In Linux the serial port is also represented by a file in */dev/* folder, but the information about each device is stored separately in another folder under */sys/bus/usb/drivers/usb/*. Each subfolder of */sys/bus/usb/drivers/usb/* represents a device, and contains the Vendor ID(in file *idVendor*), Product ID(in file *idProduct*) and device ID(in file *serial*). The device could be now classified base on these the vendor id and product id (see the table 3.5). If the device is supported, the device ID is added to the result together with the port number. So at the end, we will get a map of available port and device ID which would be shown as possible target for deployment.

3.6.4 Deployment Process

The deployment process is started by users. Depend on the selected data, the program will call the validate and convert functions if the user chooses a BPMN2 process to deploy. Otherwise, if the selected data is already in bytecode format of an ukuFlow process, the plugin goes straight to deploy it into the specified sensor node and listen to the feedback from sensor node.

Simultaneously, a timer thread is also started with the deployment. The timer will fire an event after 10 seconds (the default connection time out of the plugin), which causes the deployment process to stop listening to the serial port and release the port. As mentioned in previous chapters, the connection time out could be manually adjusted by using the ukuFlow's preference page. A sequence diagram of the deployment process is shown in the figure 3.11.

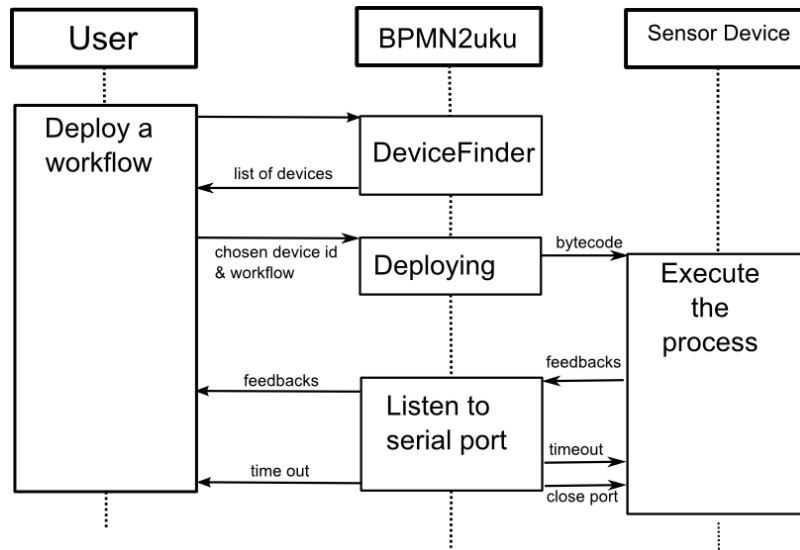


Figure 3.11: Deployment Process in Sequence Diagram

3.7 Used Libraries

3.7.1 Java Compiler Compiler

In an ukuFlow workflow, it is allowed to have scripts and condition expressions. They must be validated, and reasonably produce the feedback (error/warning messages) if user has entered incorrect script/expression. For this purpose, JavaCC is used for validating script and condition expressions. The grammar definition language of JavaCC is like high-level programming language and uses extended Backus-Naur forms. The parser generated by JavaCC is a top-down parser, it means that, left-recursion is not allowed, therefore all left-recursions have to be removed in the grammar definition file. An other important feature is that, the generated code-parser from JavaCC is all in Java [6], which is suitable for the development of an Eclipse Plugin project.²

3.7.2 JDOM 2

JDOM2 is a Java library for accessing, manipulating, and outputting XML data. In this project JDOM2 is used for fetching process data from a BPMN2 process (in XML format). In BPMN2uku, we use JDOM for in for fetching BPMN2 process³

3.7.3 RXTX

RXTX is a Java library for communicating with parallel and serial communication ports[13]. It is developed as an open source project and could work under Windows, Linux and Mac OS. RXTX provides an Java wrapper and several native libraries (for each type of operating system), therefore it could be included in an Eclipse plugin.⁴

3.8 Download

BPMN2uku is an open source project, and can be obtained from <http://code.google.com/p/ukuflow/>. The newest version can be download from the download page of the project. All code and libraries are also available in the "source"

¹ JavaCC is free and provided under BSD license [6]

² JDOM2 is provided under an Apache-style open source license[7]

³ RXTX is provided under the GNU LGPL license[13]

section. There are two folders, one for BPMN2uku (under `git/eclipse-plugin/bpmn2-ukuflow`) and the other for the custom target runtime (under `git/eclipse-plugin/ukuflow-runtime`)

4 Evaluation

4.1 Cross Platform Test

In order to warrant the quality of the development tool, we have tested the plugin under different condition (different version of Eclipse platform and operating systems). The cross platform functionality of BPMN2uku plugin were successfully tested with Eclipse Indigo 3.7 under different operating systems: Windows 7, Ubuntu 12.04, Linux Mint (virtual machine), Mac OS X 10.7 and Mac OS X 10.8. In each test, workflows with and without error were created, which were validated by BPMN2uku plugin. After each validation, bugs were found by checking if all errors were found, correct process were passed. Also some instruction for fixing errors are refined.

Although conversion and Validating (include script validating and process validating) are tested with several BPMN2 processes, but they still need to be tested more with different processes, so that undiscovered bug could be found, as well as instruction for assisting user could be refined

The deployment function is also tested in above mentioned OSs. The tests were using the blink command to test where the sensor nodes act exactly like described in the BPMN2 process. Feedbacks from sensor nodes, which are displayed in ukuFlow console, are also used to evaluated whether the deployed sensor node performs desired activities specified BPMN2 process.

Another tests for management multiple nodes and different types of sensor nodes (Z1 and Tmote Sky) were also performed. It was shown that the plugin was able to detect all connected node and node's identifier were correctly displayed.

4.2 Performance Test

Since a BPMN2 process has not so many elements and the length of the script, condition is also short, performance time for fetching, validating and converting is not critical. However the time needed for getting all available port is about 1 second. That is because RXTX has to try opening all possible ports and see which are available. This process apparently take time and is noticeable while using the plugin.

Size the workflow in bytecode format is apparently much smaller than a original BPMN2 diagram. A comparison of original BPMN2 diagram from BPMN2 modeler and its bytecode is shown in 4.1

Process Information	BPMN2 Dia-gram	Bytecode format
3 elements(include 1 script task) & 2 sequence flows	2,797 bytes	44 bytes
6 elements (include 2 script tasks) & 6 sequence flows (include 2 condition expressions)	6,255 bytes	80 bytes
15 elements(7 script tasks) 18 sequence flows(2 conditions expressions)	15,746 bytes	236 bytes

Table 4.1: Process Size Comparison of BPMN2 and bytecode

5 Conclusions & Future Work

5.1 Conclusions

The goal of this project was that to construct a tool which can convert a BPMN2 diagram to a ukuFlow workflow, as well as deploy and undeploy them on a sensor node. Thus, this tool makes it easier and more comfortable to work with ukuFlow for domain experts, who are not necessarily knowledgeable about low level programming languages such as nesC or C. Other functionalities of BPMN2uku are also implemented so that it can support users to work cross operating systems (Mac OS X, Windows and Linux).

For using the BPMN2uku plugin to convert and deploy a workflow to a wireless sensor network, the user should know about the syntax of script as well as condition expression for ukuFlow. It is also required that users have experience on creating a BPMN2 process. With the validation mechanism and error reporter, the plugin will further assist the users in discovering errors in the workflow

5.2 Future Work

Several test cases have been evaluated using this software, but still, it should be tested more for discovering bug as well as feedbacks from users for further enhancements. Currently we have seen some point that could be improved. As mentioned above, the plugin is still not able obtain information of sensor devices under Mac OS, I believe that, it could be improved to works on Mac OS like other OS. Or all scripts in a workflow must be entered by user in text form, and it may be complicated, some script format could be illustrated with graphical representation, this UI feature could be also developed in the future.

During the development stage of this work, The BPMN2 Modeler plugin is still in *incubation* phase, therefore it sometime led to errors and bugs. We were able to detect and report a number of bugs, as well as to implement and submit solutions that made their way to the final code of the main BPMN2 Modeler plugin.

As mentioned in chapter 2, not all elements of BPMN2 are allowed in ukuFlow processes. The other elements are therefore superfluous and could confuse the user. For this purpose, a custom BPMN2 Modeler target runtime for ukuFlow is currently being developed which hides all irrelevant BPMN2 elements. Moreover, to my knowledge, a so called "target runtime" could also be used to define new elements or new properties for the existing elements of BPMN2. Therefore, a more specific target runtime for ukuFlow could be developed in the future that helps domain experts even more in their development of ukuFlow processes

List of Figures

1.1	Gap between BPMN2 process and ukuFlow Engine	4
2.1	Event Based Gateway	10
3.1	Overview of BPMN2uku plugin	13
3.2	Console while Listening to the Serialport	14
3.3	ukuFlow's Preference Page	14
3.4	XML file: Process Information & Diagram Information	15
3.5	Workflow Enhancement for Mixed Gateway	17
3.6	Class Diagram of UkuExpression	18
3.7	Example of parsing an UkuExpression	18
3.8	Conversion from BPMN2 to .uku64	21
3.9	Class Structure of DeviceManager & Associated Classes	21
3.10	Windows Registry: Search for FTDI Device	22
3.11	Deployment Process in Sequence Diagram	23

Bibliography

- [1] N. Burri, R. Schuler, and R. Wattenhofer, “Yeti: A tinyos plug-in for eclipse,” *... of the ACM Workshop on Real- ...*, 2006. [Online]. Available: <http://www.disco.ethz.ch/publications/realwsn2006.pdf>
- [2] Z. Z. Datasheet, “Z1 Datasheet,” pp. 1–20. [Online]. Available: http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf
- [3] A. Forster and G. Engels, “Verification of business process quality constraints based on visual process patterns,” *... Aspects of Software ...*, 2007. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4239964
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.
- [5] S. I. Hip, U. S. B. To, and U. B. Ridge, “Uart b,” pp. 1–18, 2010. [Online]. Available: <https://www.silabs.com/Support%20Documents/TechnicalDocs/cp2102.pdf>
- [6] JavaCC Features. 29.11.2011, 12:39. [Online]. Available: <http://javacc.java.net/doc/features.html>
- [7] Jdom.org, “Jdom2.” [Online]. Available: <http://www.jdom.org/index.html>
- [8] A. D. Nicola, M. Missikoff, M. Proietti, and F. Smith, “An open platform for business process modeling and verification,” *Database and Expert ...*, pp. 76–90, 2010. [Online]. Available: <http://www.springerlink.com/index/7104062777N64W50.pdf>
- [9] T. Note, “Technical Note TN _ 100 USB Vendor ID / Product ID Guidelines,” vol. 44, no. 0, pp. 0–11, 2011.
- [10] O. M. G. D. Number and A. S. Files, “Business Process Model and Notation (BPMN),” no. January, 2011.
- [11] RGuerrero, “ukuFlow.” [Online]. Available: <http://www.dvs.tu-darmstadt.de/research/ukuflow/>
- [12] L. Power, W. Sensor, S. V. Supervisor, and I. Humidity, “Key Features Table of Contents,” pp. 1–28, 2006.
- [13] Rxtx.qbang.org, “Rxtx.” [Online]. Available: http://rxtx.qbang.org/wiki/index.php/Main_Page
- [14] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann, “Scoping in wireless sensor networks: A position paper,” *Proceedings of the 2nd ...*, 2004. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1028521>