BPMN to CIF conversion

Aurelo Makaj

15 September 2022

1 Introduction

The aim of the following document is to introduce the user to the tool implemented for the conversion from a BPMN (Business Process Model and Notation) diagram to a CIF format.

"CIF is a declarative modeling language for the specification of discrete event, timed, and hybrid systems as a collection of synchronizing automata".

The main idea is to give the tools to describe a BPMN diagram in a flexible way, and convert this diagram into another domain (finite-state machine), to allow an automatic verification of the system and constraints, which may not be possible in the first domain (BPMN).

2 Format

The format used to describe the BPMN diagram is the **XML format**. The choice of this format is due to its block-definition and recursive style.
The XML document will start with a `<instance>` tag, which will contains only two sub-blocks:

- `<process>`, which will contain the definition of the BPMN diagram
- `<constraints>` which will contain the definition of the properties and constraints on the diagram

3 Process

One of the main block that cannot be contained in other blocks. It must contain only one sub-block. For example:

```
<process id="ProcessBlockId">
    <sequence>...</sequence>
</process>
```

There are 6 different types of blocks that can be used inside the `<process>` block:

- Task
- Sequence
- And
- XOR
- Loop
- While

Each block is described using a specific tag, which resemble the type of the block. Moreover, each block must have an "id" attribute, for uniquely identifying the block in the diagram. The identifier must contains only letters (lowercase, uppercase), numbers and underscores.

3.1 Task

This is the smallest block, and it cannot contain other blocks. Each task will have associated a set of resources, controllable or uncontrollable. A controllable resource means that the system have the ability to manually allocate that resource to the task. On the other hand, the system has no control over an uncontrollable resources.
The resources will be described through the following attributes:

- **ctrl_res** attribute, for controllable resources.
- **unctrl_res** attribute, for uncontrollable resources.

Each attribute allows a list of names, separated by comma. The name of each resource must contain only letters (lowercase, uppercase), numbers and underscores.
Example:

```
<task id="Loan_Request"
     ctrl_res="Alice, Charlie"
     unctrl_res="Bob"
/>
```

3.2 Sequence

This block contains one or more sub-blocks that will be executed in sequence.

```
<sequence id="SequenceBlockId">
    <task .../>
    <loop id="..." >...</loop>
    ...
</sequence>
```

3.3 And

This block contains one or more sub-blocks that will be all executed at some point. The flow does not proceed until all the block has not been executed. There is no execution order.

```
<and id="AndBlockId">
    <sequence id="...">...</sequence>
    <task id="..."/>
     ...
</and>
```

3.4 XOR

This block contains one or more sub-blocks, but only one of them will be executed, depending on some conditions. Each block is a branch, and each branch defines a particular case. As for the tasks, branches can be controllable or uncontrollable. Moreover, we can specify a default case (controllable or uncontrollable). The following attributes are used:

- **ctrl_branch** attribute, for controllable branches (possibly a list, separated by comma)
- **unctrl_branch** attribute, for uncontrollable branches (possibly a list, separated by comma).
- **default** attribute, for the default branch (not a list)
- **ctrl_default** attribute, with two possible values: "true" if the default branch is controllable, "false" otherwise

The branch name must contain only letters (lowercase, uppercase), numbers and underscores.
The number and the position of the internal sub-blocks must be coherent with the definition of the branches. Each successive sub-block will be associated, in order of definition, first with the controllable branches, and then with the uncontrollable branches. The number of blocks will be the same of the number of branches (minus the default case, if present).

```
<xor
    id="Xor1"
    unctrl_branch="high_amount, medium_amount"
    default="low_amount"
    ctrl_default="false"
>
    <task
        id="Anti_Money_Laundering_Assesment"
        unctrl_res="Evie, Frank"
    />
    <task
        id="Tax_Fraud_Assesment"
        unctrl_res="Charlie, Gary, Hannah"
    />
</xor>
```

The branch "high_amount" will be associated with the first task, while the branch "medium_amount" will be associated to the second and last task. The default branch is not associated to a task.

3.5 Loop

This block may contain one or two blocks. The first block is always executed, and then a guard is checked. If the "repeat" condition is true, the cycle is repeated (and the second block is executed, if specified, before repeating the first one), otherwise if the "exit" condition is true, we exit the loop. Both the "repeat" condition and the "exit" condition can be controllable or uncontrollable. The repeat and exit conditions must contain only letters (lowercase, uppercase), numbers and underscores. The following attributes are available for this block.

- **repeat** attribute, for the "repeat" condition.
- **ctrl_repeat** attribute, either true or false.
- **exit** attribute, for the "exit" condition.
- **ctrl_exit** attribute, either true or false.

```
<loop
    id="Loop1"
    repeat="renegotiate"
    ctrl_repeat="false"
    exit="confirm"
    ctrl_exit="false"
>
   <sequence id="Seq2">... </sequence>
   <task .../>
</loop>
```

3.6 While

This block contains only one sub-block that can be repeated multiple times. The difference with the loop is that the guard is checked before the execution of the sub-block, which means that it may not even be executed one time. The attributes are the same of the Loop block.

```
<while
      id="WhileBlockId"
      repeat="BadQuality"
      ctrl_repeat="false"
      exit="GoodQuality"
      ctrl_exit="false"
>
     <and>...</and>
</while>
```

4 Constraints

We can define constraints on the resources that can be associated to the tasks. The set of constraints will be defined with another XML tag, the `<constraints>` tag. The tags that we can use inside this block are:

- `<property>` tag, for defining custom properties
- `<tcc>` tag, for specifying the constraints

A constraint is defined by a property $\mathcal{P}$ applied to two tasks T_1 and T_2. Then we consider all the possible pairs (r_1, r_2) such that $r_1 \in Auth(T_1)$ and $r_2 \in Auth(T_2)$ and keep only the pairs that respects the property.

For example, let T_1 and T_2 be two tasks, and Alice, Bob and Thomas be resources such that $Auth(T_1) = \{\text{Alice, Bob}\}$ and $Auth(T_2) = \{\text{Bob, Thomas}\}$. Let $\mathcal{P}$ be the **EQUAL** property. All the possible pairs are (Alice, Bob), (Alice, Thomas), (Bob, Bob), (Bob, Thomas), but only the pair (Bob, Bob) satisfies the property.

Beside the typical properties EQUAL and the negation NOT EQUAL, we allow user-defined properties.

4.1 Property definition

The simplest way to define custom properties is to write all the possible pairs of resources that we know respect the property. The `<property>` tag is used for such task, inside which we will write the pairs of our desired property.

For example, let's say that the property that we want to define is the RELATIVE property, which tell us that the resources are relatives with each other. We will write something like this:

```
<property id="RELATIVE">(Evie, Charlie)</property>
```

Each `<property>` tag must specify an **"id"** attribute, which will be the name of the property used in the constraints. Inside the tag we can write a comma

separated list of pairs (or tuples).
In the evaluation of the property, we will always consider the **symmetric pair too**: if we have specified that Evie and Charlie are relatives, we will consider also Charlie and Evie pair.
If we have three (or more) brothers for example, it can became boring to write all the pairs. So, we can write a single tuple with all the resources, and it will be expanded to a set with all the possible pairs of elements. For example, the following property:

```
<property id="RELATIVE">(Alice, Bob, Alex, Mark)</property>
```

will produce the following pairs

(Alice, Bob), (Bob, Alice),
(Alice, Alex), (Alex, Alice),
(Alice, Mark), (Mark, Alice),
(Bob, Alex), (Alex, Bob),
(Bob, Mark), (Mark, Bob),
(Alex, Mark), (Mark, Alex)

4.2 Constraint definition

To define a constraint, we use the `<tcc>` tag and inside of it we define our rule, using the following format:

first_task_id [NOT] PROPERTY_NAME second_task_id

For example:

```
<tcc>
    AntiMoneyLaunderingAsmnt !RELATIVE AsmntNotification
</tcc>
```

Essentially, the property works as an infix operator for the two tasks.
In this case, we want that the resource that will be assigned to the task "Anti-MoneyLaunderingAsmnt" cannot be equal to the resource that will be assigned to the task "AsmntNotification".

The built-in property is the **EQUAL** property (or simply the equal symbol =). For each property, even the custom ones, you can use the NOT keyword (or simply the exclamation mark !) to specify the negation of the property.

5 Software

All the software has been written in Python 3. The list of the external packages used are:

- **ply**: an implementation of Lex-Yacc parsing tools (Link)

The entry point is the **convert.py** file, which will use the following files:

- **parser_structure.py**: contains all the functions for parsing the XML document. It uses the following files:
  - **parser_properties.py** for parsing the properties.
  - **parse_constraints.py** for parsing the constraints.
- **writer.py** file, which contains all the functions for converting the XML into the CIF format

5.1 Workflow

A rough workflow of the software:

- Read the xml file
- Parse the syntax of the process block and all the sub-blocks
- Collect all the events
- Parse the properties and compute all the pairs that respects the property
- Parse the constraints
- Write a file containing all the events
- Write a file for each block, containing the automaton in the CIF format
- Write a file for each constraint, containing two automata.
- Write a tooldef script, containing the instruction for mergin all the files and applying the Synthesis Supervisor
- If specified, execute the tooldef script

5.2 Command Line Exectuion

The program can be launched only from command line, with the following parameters:

- **--input** [OPTIONAL]: path to the file containing the XML structure (the default path is the the directory from which the script is launched, and looks for a **structure.xml** file).
- **--sup-synth** [OPTIONAL]: tells the script to apply supervisor synthesis
- **--tooldef** [OPTIONAL, REQUIRED if --sup-synth is specified]: path to the tooldef script

As output we obtain a directory, named as the id attribute of the `<process>` block. This directory contains three sub-directories:

- **plant**: contains an **events.cif** file, listing all the possible events, and then a .cif file for each user-defined block.
- **requirements**: contains all the .cif files describing the automata of the requirements
- **supervisor**: contains a **synthesis.tooldef** file, with the instructions for applying the Supervisor Synthesis. If the synthesis has success, it will contain also the result of it.

Command example:

```
$ python3 convert.py --sup-synth --tooldef /home/pippo/Documents/eclipse-escet-v0.6/bin/tooldef
```