CIGI QUALITA MOSIM 2023
Verification of Control-Command Programs: Counterexample Explanation by Simulation

SORAYA MESLI-KESRAOUI[1], DJAMAL KESRAOUI[1], PASCAL BERRUET[2], FLAVIO OQUENDO[3]

[1] SEGULA Engineering France
Parc Technellys, 165 Rue de la Montagne du Salut, F-56602, Lanester cedex, France
soraya.kesraoui@segula.fr, djamal.kesraoui@segula.fr

[2] Université Bretagne Sud, Lab-STICC
BP 92116, F-56321, Lorient cedex, France
pascal.berruet@univ-ubs.fr

[3] Université Bretagne Sud, IRISA
BP 573, F-56017, Vannes cedex, France
flavio.oquendo@irisa.fr

***Résumé* – Afin de faciliter la compréhension et l'interprétation des contre-exemples, retournés par les Model-Checkers, cet article présente une approche, basée sur l'IDM pour la visualisation et la simulation automatique des contre-exemples Uppaal (traces) directement sur les programmes de contrôle-commande. En effet, les traces Uppaal sont transformées automatiquement en des traces génériques, indépendantes de plateformes. Un métamodèle modélisant ces traces génériques, résultant de l'exécution des programmes de contrôle-commande selon la norme PLCOpen, est également proposé. Les traces génériques sont ensuite transformées en des traces spécifiques, directement simulables par des plateformes de fournisseurs. Notre approche a été validée sur un cas d'étude industriel concret.**

***Abstract* –In order to facilitate the understanding and the interpretation of the counterexamples returned by Model-Checkers, this paper presents an MDE-based approach for the automatic visualization and simulation of the Uppaal counterexamples (traces) directly on the control-command programs. Indeed, Uppaal traces are automatically transformed into generic (platform-independent) traces. A metamodel modeling the generic traces resulting from the execution of the control-command programs, according to the PLCOpen standard, is also proposed. The generic traces are then transformed into specific traces, which can be directly simulated on vendor-platforms. Our approach has been validated on a concrete industrial case study.**

***Mots clés* - Simulation de contre-exemple, Rétro-Annotation, Model checking, Programmes de contrôle-commande, API**
***Keywords* – Counterexample simulation, Back-annotation, Model checking, Control-command programs, PLC.**

1 INTRODUCTION & CONTEXT MOTIVATIONS

Model checking (Baier & Katoen, 2008; Bérard et al., 2013) is an automatic formal method supported by a tool, the model checker. In model checking, the system to be verified is modeled in a formal model (generally as automata). On this formal model, model checking consists of automatically checking specified properties (expressed in temporal logic). The Model-Checker explores in a concrete or symbolic way the entire state space of the formal model and checks for each state the satisfaction of the specified property. If the property is violated, the model checker returns a counterexample illustrating the trace of the error, i.e. the succession of states and transitions that violate the property (Bérard et al., 2013).

In model checking, counterexamples are supposed enabling users to achieve three objectives (Aboussoror, 2013):

1. understand the counterexample and its scenario;
2. understand the error shown in the counterexample;
3. understand the cause of the error.

As the counterexamples returned by the model checkers are in a formal notation, this only enables expert users of these tools to achieve the first objective. However, even for expert users, understanding the counterexamples errors and theirs sources (objectives 2 and 3) requires significant efforts of interpretation. It then becomes clear that system designers who are not familiar with formal notations (which is the general case) are not able to achieve all three objectives (Kaleeswaran et al., 2022). Therefore, the information contained in the counterexamples, as returned by model checkers, is not suitable for the designers and does not satisfactorily support them to correct their system models. The visualization and the simulation of counterexamples on the verified model (Back-annotation) is a crucial step for designers to understand the results of the verification (Guerra et al., 2009; Kaleeswaran et al., 2022). It consists of illustrating the error returned by the model checker in a high-level language, more easily understood by users than formal notations (Hegedüs et al., 2010).

The counterexamples visualization and simulation can concern the static (structural) or the dynamic (behavioral) instances. The static instances visualization allows highlighting the static elements (classes, attributes, etc.) impacted by the error in a high-level language. On the other hand, dynamic instances

visualization consists in simulating the execution sequences that lead to the error in an understood form. It consists in translating execution sequences from formal notation into high-level language. Dynamic instances visualization is more difficult to implement than static visualization because it requires defining the execution trace for both the formal (source) and the high-level (target) language (Hegedüs et al., 2010).
The execution trace model represents the changes of entities over time and provides also an abstract representation of its runtime behavior (Mayerhofer et al., 2012). It can be generic or domain specific. The use of the generic trace model promotes its usability, but increases processing time (Bousse et al., 2015). On the other hand, the domain-specific trace model reduces computational time and complexity (Bousse et al., 2015). However, its design is often tedious.

1.1.1 Motivation

We have proposed in previous work, an approach for checking control-command chains (Mesli-Kesraoui et al., 2016, 2016). In this approach, the control-command programs are translated automatically to networks of timed automata (Kesraoui, 2017). On these automata, a set of safety and liveness properties, expressed in CTL logic (Clarke & Emerson, 1981), were verified using the model checker Uppaal. This tool returns a counterexample explaining the error if the property is not verified. However, the returned counterexamples are in a formal notation, their understanding requires extensive knowledge as well as efforts to understand the meaning of the counterexample by control-command developers.

1.1.2 Contributions

In this paper, we propose a Model Driven Engineering (MDE) approach for the dynamic simulation of counterexamples, returned by model checker, directly on the control-command programs (Back-annotation). Our contributions are resumed bellow.

- As the visualization of dynamic instances consists in transforming the execution trace, returned by a model checker, into an execution trace of control-command programs model, it is then necessary to define the syntax and the trace model for both model checker and control-command programs. If the abstract syntax of the used model checker (Uppaal) and its trace model have been proposed in (Brandt, 2016), at the level of the control-command programs, only the abstract syntax was defined by PLCOpen (PLCopen, 2009). In this work, we study the execution traces of control-command programs and we define the generic PLCOpen traces (vendor-independent) metamodel. The proposed trace metamodel is based on PLCOpen syntax in order to increase its interoperability (vendor-independent) and its reusability.
- Automatic generation of control program traces from model checker traces. We developed a set of rules translation allowing to parse the model checker traces and producing PLCOpen generic traces. The challenges here are to reduce the abstraction gap and the granularity gap between the two trace models.
- Adaptation of PLCOpen generic traces into vendor-specific traces. The platforms (manufacturers) allowing the editing of control-command programs (like Straton[1]) do not all comply with the PLCOpen standard. Our goal here consists in transforming the generic PLCOpen traces into vendor-specific traces that can be used by a specific platform.
- Automatic simulation of the generated vendor-specific traces. The obtained vendor-specific traces are executed simultaneously with the control-command program to simulate the counterexample and highlight the program error (counterexample), initially returned by the model checker.

1.1.3 Paper structure

The next section presents the background and problem statement of the work carried out (Section 2). It introduces the concepts of Model driven engineering (MDE), control-command programs, the Uppaal counterexamples and discusses the related work on counterexamples visualization. It ends by presenting the problem statement. Our approach for simulating counterexamples, returned by model checker, on control-command programs and its implementation are described in Section 3. Section 4 discusses the validation results of our approach on an industrial case study. Finally, Section 5 concludes and opens to future work.

2 BACKGROUND & PROBLEM STATEMENT

2.1 Model driven engineering (MDE)

MDE is a design approach based on more abstract concerns than conventional programming (Combemale, 2008) in order to separate the designed solution from its implementation. In MDE, the most used concept is the model.
The relationship between the system and the model is called representation. It states that a model is an abstraction of a real system, i.e. the model represents the real system. The model is specified from a metamodel (a syntax) that describes the language of expression (vocabulary) of the model (Combemale, 2008). The model and its metamodel are linked by a compliance relationship meaning that a model must always conform to its metamodel.
The MDE specification introduces also the concept of model transformation, which allows the generation from a source model, conforming to a source metamodel, to a target model conforming to a target metamodel. The main objective of model transformations is to enable the generation of platform-specific models from platform-independent (generic) models and thus promote interoperability (Bézivin & Gerbé, 2001).

2.2 Visualization of counterexamples on control-command programs

2.2.1 Control-command programs

Control-command systems are used to control physical industrial process. The interaction of the control system with the physical process is carried out by observations through sensors and by actions performed via actuators. The sensors transform the physical measurements of the process into electrical signals that can be interpreted by the control-command system. Actuators (motors, transformers, etc.) transform the electrical commands of the control-command system in action orders that allow acting on the physical process by changing its state.
Control-command programs are often deployed on programmable logic controllers (PLCs). They present a cyclical operational functioning (De Smet et al., 2000) over three steps. In the first step, data from the environment (sensors) are read and stored in internal variables. In the second step, the control-command program is executed, and output data is calculated according to the control logic. In the third and final step, these outputs are written, i.e. sent to the actuators (De Smet et al., 2000).

[1]Straton: http ://www.copalp.com/fr/

The design of control-command programs consists of developing programs (or control logic) in one of the five standardized languages defined by IEC 61131-3 (Commission, 2002): Ladder Diagram (LD), Instruction List (IL), Structured Text (ST), Sequential Function Chart (SFC) and Function Block Diagram (FBD). The Figure 1 shows an example of a LD program.

However, the interoperability of these programs between different vendors has not been addressed by IEC 61131-3 (Commission, 2002). To manage this interoperability problem, the PLCOpen committee proposed a common syntax for these programs. This syntax defines all the entities and elements of a control-command program in a platform-independent (generic) way.

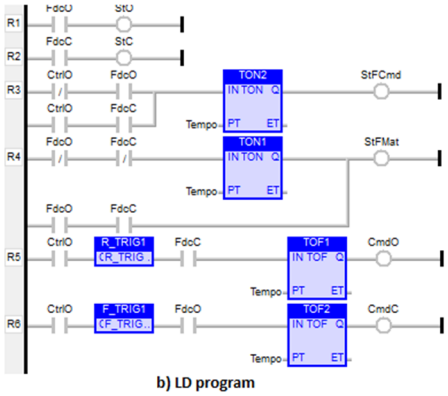

Figure 1. Ladder example program

2.2.2 PLCOpen syntax

According to PLCOpen, a control-command project (Figure 2) is composed of program types (*Types* in Figure 2) and configurations (*Configurations* in Figure 2). Program types are specified as an aggregation of several Program organization units (*Pou* in Figure 2). *Pou* is characterized by interfaces (*variables*) and a code part (*body in* Figure 2) that can be written in one of the five languages of IEC 61131-3, such as FDB and LD for example.

On the other hand, the necessary conditions (*Instance*) for the execution of these programs are made up of configurations. Each configuration consists of a set of variables (*accessVars*, *configVars*, *globalVars*) and a set of resources. A resource is the unit providing processing, storage, and communication required for the execution of programs (*PouInstance*), i.e. resource allows program instances execution as tasks. A task describes the runtime properties for the execution of program instances and can be executed in a periodic or triggered manner.

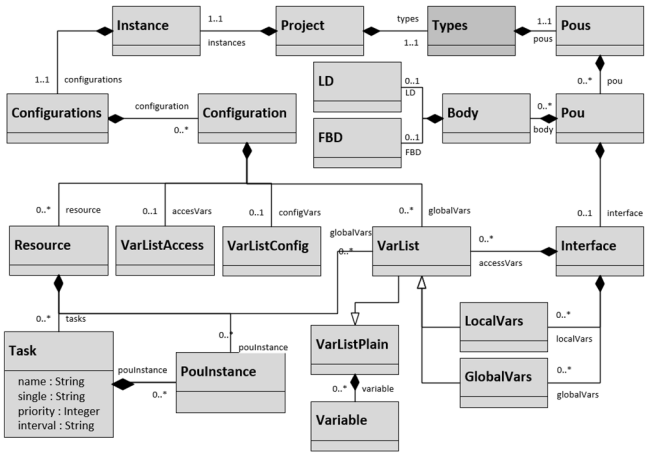

Figure 2. Control-command programs abstract syntax metamodel (excerpt taken from (PLCopen, 2009))

2.2.3 Straton IDE

Straton is an Integrated Development Environment (IDE) for PLC control-command programs. It supports the five standard languages (CFS, FBD, LD, ST, IL) of IEC 61131-3 and offers a virtual machine for running control-command programs on a computer before transferring them to PLCs.

2.2.4 Existing visualization approaches

FBDVerifier (Jee et al., 2010) is a tool for visualizing counterexamples returned by the model checker SVM Cadence on Functional Block Diagrams (FBD) in the form of a chronological diagram. However, this type of visualization does not allow illustrating the causal links between the different blocks. The animation of the original model remains one of the best visualizations allowing the user to achieve different objectives (Loer & Harrison, 2006). This type of visualization was adopted in the MODCHK tool (Pakonen et al., 2018). In fact, this tool allows the visualization of the dynamic traces returned by the model checker NuSMV directly on the FBD. It provides parallel animation of the original FBD and the verified temporal properties to illustrate the source of the error.

Despite the fact that these tools offer interesting solutions for dynamic counterexamples visualization, they remain specific to FBD programs and dependent on the used vendor-platforms.

To be vendor-independent, the dynamic counterexamples visualization requires the execution operations of the IEC 61131-3 programs and a generic trace model resulting from the execution of these programs. To our knowledge, the literature remains fairly poor and presents only a few specifications of the execution operations of some languages of IEC 61131-3 standard (De Smet et al., 2000; Rossi & Schnoebelen, 2000). On the other hand, no trace model modeling the execution traces of control-command programs has been defined in the current state of the art.

2.3 UPPAAL counterexamples visualization

2.3.1 Uppaal Tool

Uppaal is a model checker for real-time systems modeled as a network of communicating timed automata (example illustrated in Figure 2). Each automaton is a state machine that manages physical time through a set of clocks. The automaton evolves either by a delay of time or by the transition guard satisfaction driven by synchronization channels (Alur & Dill, 1990).

The Uppaal model for the LD program of the Figure 1 is illustrated in Figure 3.

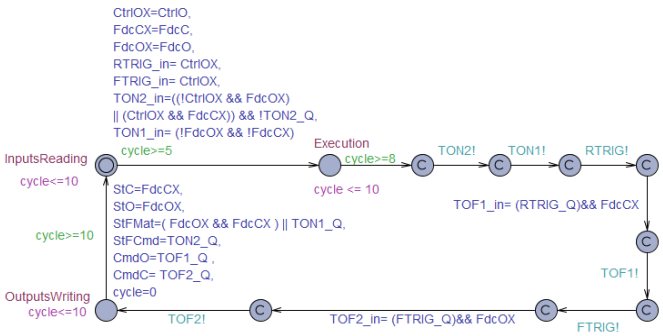

Figure 3. LD automaton for 2-WMV (taken from (Mesli-Kesraoui et al., 2016))

This automaton is composed of three main states (*InputsReading*, *Execution*, *OutputsWriting*) which describe the cyclical execution of programmable logic controllers. In the initial state, the controller is in the state *InputsReading* to read the inputs. Then, it reaches the *Execution* state where different outputs will be calculated. The controller then goes to the

OutputsWriting state to write outputs to its environment. The cycle time is represented by a clock named *cycle.*

2.3.2 Uppaal counterexample

To check for example, that the CmdO and the CmdC variables cannot be at 1 simultaneously in the LD program, the following CTL property is used.

A[] not CmdO ==1 and CmdC==1.

The verification of this property on Uppaal generates a counterexample. This counterexample consists of states and transitions (Figure 4-a).

A metamodel (Figure 4-b) of these traces was proposed in (Brandt, 2016). The Uppaal trace is composed of states (*State*) and transitions (*Abstract-Transition*), related to an automaton (*TemplateInstance*). The state is related to a set of locations and contains all concrete values (*Valuation*) of all variables (example: *FdccO =1*). Transitions can be a simple transition (*EdgeTransition*) or a delay transition. Each transition has a source and a target state.

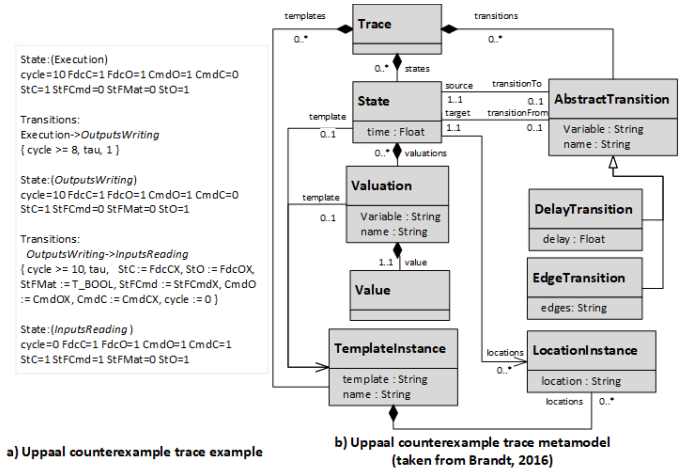

Figure 4. Uppaal trace : a) Uppaal trace example ; b) Uppaal trace metamodel (taken from (Brandt, 2016))

2.3.3 Existing UPPAAL counterexamples visualization approaches

Uppaal counterexample visualization was studied in (Schivo et al., 2017) where an MDE-based approach for Uppaal counterexamples visualization on the MechatronicUML models was proposed. However, the application of this approach to control-command programs remains difficult because it requires the consideration of other parameters such as the cyclical execution of the control-command programs. Indeed, if the Uppaal model checker has been widely used for the verification of control-command programs in the literature (Da Silva et al., 2008; De Vasconcelos Oliveira et al., 2010; Mokadem et al., 2010; Soliman & Frey, 2011), the Uppaal counterexamples simulation on control-command programs has received little research attention.

2.4 Problem statement

The counterexamples returned by Uppaal are in the form of a series of states and transitions. The interpretation of these counterexamples requires simulating all the states and transitions on Uppaal and a great effort to identify the error mining. On the other hand, the projection of the error on the verified control-command program (presented in Figure 1) is not trivial, given the gap in granularity and abstraction between the Uppaal traces and the control-command programs.

The aim of this work is to reduce this gap in order to facilitate the interpretation of the counterexamples directly on the control-command programs. The main questions are:

Q1. Which data will be used when simulating the Uppaal counterexample on control-command programs?

Q2. How to automatize the translation of the returned counterexamples into simulations on control-command programs?

Q3. How to deal with the problem of different targeting vendors platforms of control-command programs when simulating counterexamples?

3 PROPOSED APPROACH

In order to define the data necessary for the simulation (Q1) of the counterexamples directly on the control programs (target data), we carried out some simulations and tests on control-command programs. During these tests, we realized that the simulation of control-command programs consists of manipulating the inputs of a program and comparing the results of its outputs with the expected ones. Therefore, our goal is to turn the model cherckers' traces into a test case, which contains assignments of the program inputs and expectations of program outputs.

To answer the second question (Q2), we have opted for the use of an MDE-based approach, as it provides a well-defined framework for automatic generation. However, implementing this approach required defining all the metamodels and transformation rules necessary for this transformation.

On the other hand, to deal with the problem of different vendor platforms of the control-command programs (Q3), we introduce an intermediate step. This later transforms the traces returned by a model chechers into generic PLCOpen traces. The execution of these generic traces on a given platform requires transforming them into acceptable platform traces. This step is very important, because it allows us to transform the counterexamples into a generic model, independent of all vendor platforms, in order to increase the usability of our solution.

In summary, our proposed approach is MDE-based and allows Uppaal counterexamples simulation directly on control-command programs (Figure 5). It transforms the Uppaal traces into generic PLCOpen traces. These generic traces are then adapted to be directly simulated on a target-specific platform (Straton in our case). For this, we defined the generic trace metamodel for the control-command programs, based on PLCOpen syntax and the vendor-specific trace metamodel for Straton traces.

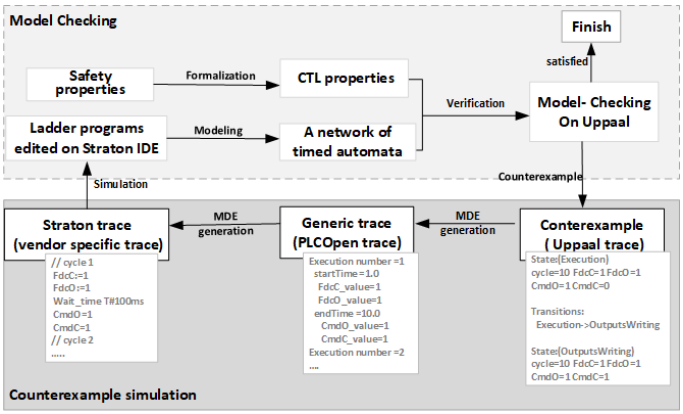

Figure 5. Proposed approach for Uppaal traces simulation on control-command programs

3.1 PLCOpen generic trace

The definition of the generic trace metamodel of control-command programs is based on the identification of the entities that change during the execution time of these programs.

The execution of a control-command program starts with the running of its configuration. The start of the configuration (*start* operation in Figure 6) allows initializing all its variables and the execution of all its resources. Otherwise, the stop of configuration causes the stop of all its resources. Starting a resource causes the initialization of all its variables and enable all its tasks. Otherwise, if the resource is stopped, it disables all its tasks. Depending on the specification, the task can be performed either cyclically or in a trigged manner. In the two cases, the task starts executing at the starting time (*startTime*) and stops execution at the ending time (*endTime*). The task can be interrupted (*interrupt* operation) at any time (*interruptTime*) by another prior task. Once the prior task is finished, the interrupted task is resumed at the *resumeTime*. The task can have several executions. At each execution, the task reads inputs, executes the associated POU and writes outputs. Finally, executing the POU allows running the different programs (FBD, LD, SFC, ST, and IL) and updates values of variables according to the program logic.

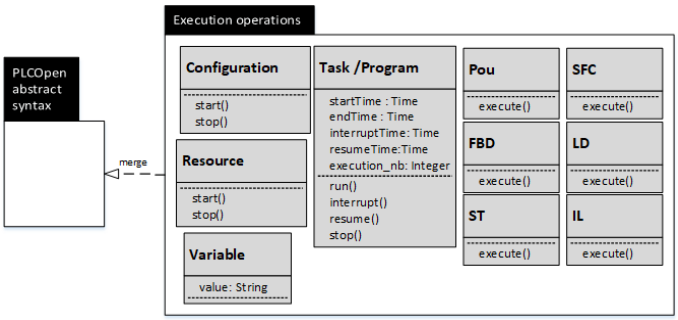

Figure 6. The execution operation of control-command programs.

Figure 7 illustrates an example of a generic PLCOpen trace. In this example, the *TracedTask* was executed twice (two cycles). At the beginning (*StartTime*) of the first cycle, the input *FdcC* variable was at 1. At the end of this cycle (*endTime*), the output variable *CmdO* has the value of 0. In the second cycle, the *FdcC* remained at 1, the execution of this cycle allows the *CmdO* variable to be set to 1.

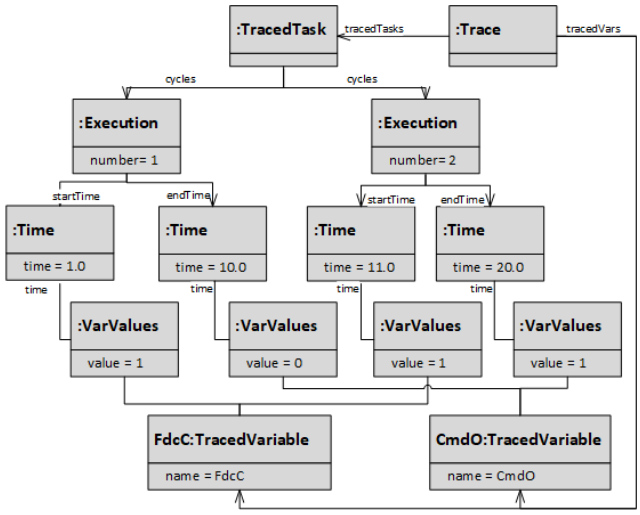

Figure 7. Example of a generic trace model

As explained above, PLCOpen trace results from the execution of different tasks and changes in the values of variables. At each time, each variable is characterized by its value and task is characterized by its starting, interrupting, resuming and ending time. To capture PLCOpen traces, we introduced the *value* property in the *Variable* class that stores the different concrete values of the variables during the execution. In the same way, the *startTime*, *interruptTime*, *resumeTime*, and the *endTime* properties in the *Task* class allow capturing respectively the task start, interruption, resume and end time.

The proposed generic PLCOpen traces metamodel is illustrated in Figure 8. Trace is related to a resource and results from:

1) the execution of different tasks (*TracedTask* in Figure 8) and;
2) changes in the values of variables (*TracedVariable*).

These two classes have an association (*originalObject*) with the original *Task* and *Variable* classes defined in the abstract syntax PLCOpen. The different valuations of each variable are stored in the attribute *value* of the class *VarValue*. Each *TracedTask* contains all its executions (*Execution*) or its cycles. The *number* property in the *Task* class indicates the execution number of the task. Each execution is characterized by 4-time values (*Time*) representing the *startTime*, the *interruptTime*, the *resumeTime* and the *endTime*. The association *varValues* allow storing different values of variable at each of the 4 task times.

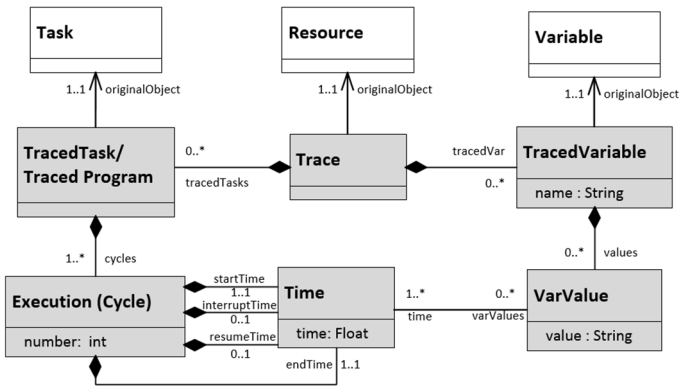

Figure 8. A generic PLCOpen trace metamodel

3.2 Uppaal trace to PLCOpen trace

We developed transformation rules to derive Uppaal traces into generic PLCOpen traces. This transformation takes both the Uppaal trace metamodel (Figure 4-b) and the PLCOpen abstract syntax metamodel (Figure 2) in inputs and produces a model that conforms to the generic PLCOpen traces metamodel (Figure 8). The Uppaal trace metamodel is used to deduce the trace error in the counterexample, on the other hand, the PLCOpen metamodel is used to identify different variable types (input/ output). We developed the following rules, illustrated in Figure 9.

1. **Trace2Trace.** The Uppaal trace project is translated into a PLCOpen trace project. This later consists of a set of *TracedVariable* and *TracedTask*. *TracedVariable* are generated by the rule *Variable2TracedVariable*. *TracedTask* gathers all the cycle generated by the rule *EdgeTransition2Execution*.
2. **EdgeTransition2Execution**. In the Uppaal automaton (Figure 3), all the states and transitions between the *InputsReading* state and the *OutputsWriting* state

constitutes one program execution or cycle (Figure 8). For this, we translate from Uppaal trace each *EdgeTransition* starting by the *InputsReading* state to an *Execution* instance in the generic PLCOpen trace. Each *Execution* instance contains both the start and the end time (*Time)* instance, generated by the *State2Time* rule (example in Figure 9).

3. **State2Time**. Each *InputsReading* and *OutputsWriting* states are translated to a *Time* Instance (example in Figure 9). The time attribute for these instance correspond to the cycle value in the state instance, as illustrated in Figure 9.
4. **Variable2TracedVariable.** From the PLCOpen abstract syntax, all inputs and outputs variables are translated to a *TracedVariable* in PLCOpen generic trace. Input *TracedVariables* are related to *startTime* of a cycle and the outputs traced variables to the *endTime*.
5. **Valuation2VarValue**. Each *Valuation* (Figure 4) from the Uppaal trace is transformed into *VarValue* and related to its *TracedVariable*, generated by the above rule (example *CmdO* variable in Figure 9).

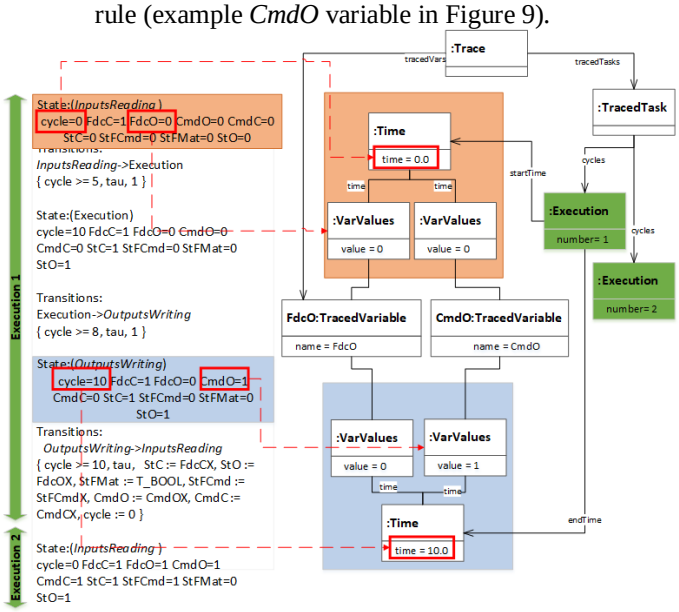

Figure 9. Tranlation rules illustration

The obtained model is generic and therefore requires adaptation to a target vendor-platform for its execution and its simulation. In the following, we present the adaptation of this model to a specific trace model for the *Straton* platform.

3.3 PLCOpen trace to Straton trace

Several control-command programs IDE (like Straton, Tia portal, etc.) allow the exploitation and the animation of specific trace sequences directly on their editors. These specific traces are similar on the different IDEs. They essentially comprise simulation steps and in each step, assignments of variables or checks on the values of variables are carried out.

For this work, we have used the Straton IDE. After studying examples of Straton trace (Figure 11), we proposed the metamodel presented in Figure 10. Straton trace consists of several steps (*Step* in Figure 10) that are of three types of expressions: assignment, evaluation, and wait (Figure 10). *AssignmentExpression* allows values to be assigned to variables i.e. variable initialization (lines 2, 3, 4 in Figure 11). *EvaluationExpression* is used to check variable values (lines 10-15, in Figure 11). Finally, *WaitExpression* allows the test execution to be suspended for the specified time (lines 7 in Figure 11).

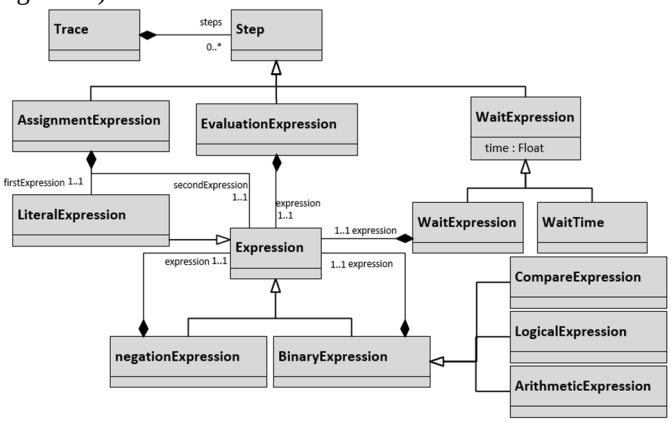

Figure 10. Straton Trace metamodel

To generate Straton traces from PLCOpen traces, we developed the following rules.

1. **Trace2Trace.** The generic PLCOpen trace project is transformed into a Straton trace. This later gathers all the steps generated from *Execution* instances with the following rules.
2. **StartTime2AssignmentExpression.** All the *TracedVariables* related to the *startTime* of a cycle (inputs), illustrated by the orange box in Figure 9, have been transformed into a set of assignment expressions (*AssignmentExpression* for Straton trace). For example *FdcO:=0* in Figure 11.
3. **EndTime2EvaluationExpression**. *TracedVariables* related to the *endTime* of a cycle (outputs), illustrated by the blue box in Figure 9, have been transformed into evaluations expression (*EvaluationExpression* for Straton trace). For example *CmdO=1* in Figure 11.
4. **Time2WaitExpression**. To manage the execution time of the cycle, a *WaitExpression* was also generated from the *endTime-startTime* of a time attribute.

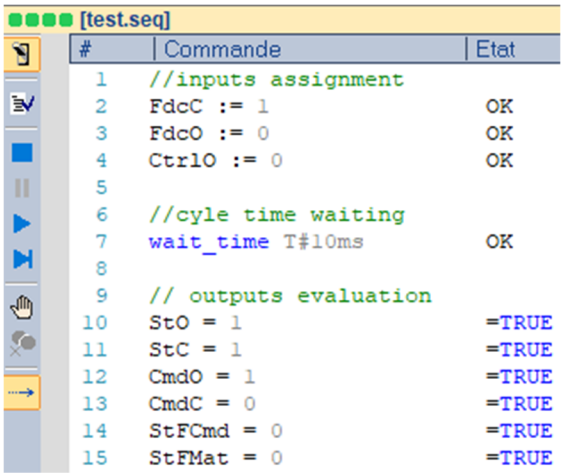

Figure 11. Sample Straton Trace Model

Another translation module has been developed to translate the obtained Straton trace model from the XML notation to the textual notation to be executed directly by Straton.

3.4 Automatic simulation of the generated trace on Straton

Straton executes the generated trace as a test case. First, Straton executes assignment expressions in order to initialize control-command variables (Figure 11). When executing the wait expression, the test is suspended for the time specified in the

wait expression. After this, Straton compares evaluation expressions with the values obtained after the real execution of the control-command program. Then, it generates a report on each step ("Etat" column in Figure 11).

4 CASE STUDY

4.1 Presentation

To evaluate our approach, we have validated it with a concrete case study: a 2-way motorized valve (*2WMV*) component. This physical component acts as a barrier. Indeed, when it is opened, it allows passing fluid. Nevertheless, if it is closed, fluid is blocked at its ends (Figure 12).

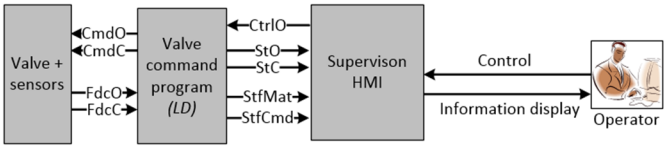

Figure 12. Two-way motorized valve

The control-command program is written in LD under the Straton software (Figure 1). The program runs in a cyclic manner. In each cycle, it reads the inputs (*CtrlO*, *FdcO*, *FdcC*) received respectively from the supervision interface, opens switch limit, and closes switch limit sensors (Figure 12). Then, it calculates the outputs. At the end of the cycle, it updates the outputs (*CmdO*, *CmdC*, *StO*, *StC*, *StFMat*, *StFCmd*). The Uppaal model for the LD program of the valve is illustrated in Figure 3. The detailed presentation of our case study can be found in (Mesli-Kesraoui et al., 2016).

4.2 Evaluation method

In order to evaluate the performance of our approach and its ability to simulate Uppaal counterexamples on control-command programs, we applied it to a set of three (3) Uppaal counterexamples resulting from the formal verification of the properties of the case study.

For this evaluation, two measures were considered. The first measure (M1) calculates the automatic generation time of the Straton traces from the Uppaal counterexamples. The purpose of this measure is to study the ability of our approach to transform different sizes of Uppaal traces. The second metric (M2) aims to calculate the number of simulation steps required to simulate the counterexample in the two tools: Uppaal and Straton. The number of simulation steps is an important complexity indication. Indeed, if the number of simulation steps is high, then understanding the counterexample is difficult. This metric measures the usefulness of our approach to facilitate the interpretation of counterexamples when verifying control-command programs.

4.3 Results & discussion

The experiment was carried out on a computer with an I7 microprocessor with a frequency equal to 2.6 GHz and 8 GB of memory. The results are presented in TABLE I.

The results show that transformations take less than 0.3 seconds for a counterexample containing 54 states and 53 transitions (EC3).

On the other hand, the results show that the number of simulation steps (M2) was divided by 24 between the Uppaal counterexample and Straton. On Uppaal, the user needs to run 24 simulation steps to execute a single program cycle. Conversely, on the generated Straton trace, a cycle is carried out in one simulation step. These results show that our approach significantly reduces the simulation effort.

TABLE I. Experimentation results

Counterexample (CE)		**CE1**	**CE2**	**CE3**
Size	state	15	32	54
	Transition	14	31	53
	Total	**29**	**63**	**107**
Uppaal tarce→ PLCOpen generic trace (second)		0.248	0.245	0.258
PLCOpen generic trace → Straton trace (second)		0.027	0.004	0.013
Total time (M1) (second)		**0.275**	**0.249**	**0.271**
Simulation step in Uppaal (M2)		**27**	**55**	**107**
Simulation step in Straton (M2)		**1**	**2**	**4**

5 CONCLUSION

In this work, we presented a novel solution for the automatic simulation of Uppaal counterexamples on control-command programs (Back-annotation).

In these works, we have proposed a PLCOpen generic trace metamodel to specify the execution traces of command programs in a generic way. Unlike the related work, the generic trace metamodel will promote its usability and nteroperability. Indeed, the generic PLCOpen traces metamodel can be used for the visualization of other model checkers' counterexamples on other control-programs vendor-platforms. One have to transform the counterexamples of these model checkers into generic traces, and then to transform these generic traces into specific traces for the target platforms.

Our proposed approach was evaluated on an industrial case study. Experimentation showed that the use of our approach enables traces to be generated in a very short time (less than 0.3 seconds) and reduce simulation steps. This solution helps system designers to understand counterexamples and make it possible for them to use formal techniques in an industrial context.

Further experiments with different user profiles (experts and non-experts in formal verification) will be required to assess the contribution of our solution in the interpretation of counterexamples.

6 REFERENCES

Aboussoror, E. A. (2013). *Méthodes de diagnostic avancées dans la validation formelle des modèles* [PhD Thesis]. Université de Toulouse III-Paul Sabatier.

Alur, R., & Dill, D. (1990). Automata for modeling real-time systems. *Automata, Languages and Programming: 17th International Colloquium Warwick University, England, July 16–20, 1990 Proceedings 17*, 322–335.

Baier, C., & Katoen, J.-P. (2008). *Principles of model checking*. MIT press.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., & Schnoebelen, P. (2013). *Systems and software verification: Model-checking techniques and tools*. Springer Science & Business Media.

Bézivin, J., & Gerbé, O. (2001). Towards a precise definition of the OMG/MDA framework. *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 273–280.

Bousse, E., Mayerhofer, T., Combemale, B., & Baudry, B. (2015). A generative approach to define rich domain-specific trace metamodels. *Modelling Foundations and Applications: 11th European Conference, ECMFA 2015, Held as Part of STAF 2015, LAquila, Italy, July 20-24, 2015. Proceedings 11*, 45–61.

Brandt, J. (2016). Understanding attacks: Modeling the outcome of attack tree analysis. *25th Twente Student Conference on It*, *25*.

Clarke, E. M., & Emerson, E. A. (1981). *Design and synthesis of synchronization skeletons using branching time temporal logic*.

Combemale, B. (2008). Meta modeling Approach for Model Simulation and Verification: Application To Process Engineering. *French Phd Thesis,(IRIT, Enseeiht)*.

Commission, I. E. (2002). Programmable controllers-part 3: Programming languages. *IEC 61131-3 (Ed. 2.0)*.

Da Silva, L. D., de Assis Barbosa, L. P., Gorgônio, K., Perkusich, A., & Lima, A. M. N. (2008). On the automatic generation of timed automata models from function block diagrams for safety instrumented systems. *2008 34th Annual Conference of IEEE Industrial Electronics*, 291–296.

De Smet, O., Couffin, S., Rossi, O., Canet, G., Lesage, J. J., Schnoebelen, P., Papini, H., de Fabrications, C., & Paris, C. (2000). Safe programming of PLC using formal verification methods. *Computer Science*, *7*, 8.

De Vasconcelos Oliveira, K., Perkusich, A., Lima, A. M. N., Gorgônio, K., & da Silva, L. D. (2010). Standard-based formal validation of programmable logic controller programs. *2010 IEEE International Conference on Industrial Technology*, 1655–1660.

Guerra, E., de Lara, J., Malizia, A., & Díaz, P. (2009). Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information and Software Technology*, *51*(4), 769–784.

Hegedüs, Á., Bergmann, G., Ráth, I., & Varró, D. (2010). Back-annotation of simulation traces with change-driven model transformations. *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, 145–155.

Jee, E., Jeon, S., Cha, S., Koh, K., Yoo, J., Park, G., & Seong, P. (2010). FBDVerifier: Interactive and visual analysis of counter-example in formal verification of function block diagram. *Journal of Research and Practice in Information Technology*, *42*(3), 171–188.

Kaleeswaran, A. P., Nordmann, A., Vogel, T., & Grunske, L. (2022). A systematic literature review on counterexample explanation. *Information and Software Technology*, *145*, 106800.

Kesraoui, S. M. (2017). *Intégration des techniques de vérification formelle dans une approche de conception des systèmes de contrôle-commande*.

Loer, K., & Harrison, M. D. (2006). An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation. *Automated Software Engineering*, *13*, 469–496.

Mayerhofer, T., Langer, P., & Kappel, G. (2012). A runtime model for fUML. *Proceedings of the 7th Workshop on Models@ Run. Time*, 53–58.

Mesli-Kesraoui, S., Bignon, A., Kesraoui, D., Toguyeni, A., Oquendo, F., & Berruet, P. (2016). Vérification formelle de chaines de contrôle-commande d'éléments de conception standardisés. *Proceedings of the 11th International Conference on Modeling, Optimization & Simulation (MOSIM 2016)*.

Mesli-Kesraoui, S., Toguyeni, A., Bignon, A., Oquendo, F., Kesraoui, D., & Berruet, P. (2016). Formal and joint verification of control programs and supervision interfaces for socio-technical systems components. *IFAC-PapersOnLine*, *49*(19), 426–431.

Mokadem, H. B., Berard, B., Gourcuff, V., De Smet, O., & Roussel, J.-M. (2010). Verification of a timed multitask system with UPPAAL. *IEEE Transactions on Automation Science and Engineering*, *7*(4), 921–932.

Pakonen, A., Buzhinsky, I., & Vyatkin, V. (2018). Counterexample visualization and explanation for function block diagrams. *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, 747–753.

PLCopen. (2009). *PLCOpen XML 2.01*. https://plcopen.org/

Rossi, O., & Schnoebelen, P. (2000). Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs. *Proc. 4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM'2000), Dortmund, Germany*, 177–182.

Soliman, D., & Frey, G. (2011). Verification and validation of safety applications based on PLCopen safety function blocks. *Control Engineering Practice*, *19*(9), 929–946.