Automatic Verification of BPMN Models

Mihal Brumbulli, Emmanuel Gaudin, Ciprian Teodorov

Automatic Verification of BPMN Models

Mihal Brumbulli*, Emmanuel Gaudin*, Ciprian Teodorov**
*PragmaDev, Paris, France
**ENSTA Bretagne, Brest, France

***Abstract*—Models of complex systems and systems of systems are described with NAF (NATO Architecture Framework) or DoDAF (DoD Architecture Framework). Business Process Model Notation (BPMN) is part of NAF and allows to describe the behavior of the different participants in the model. This notation is used by the French Army and its main suppliers to describe the interactions between participants involved in a mission. It is therefore important the models are correct. The VeriMoB project is a research project financed by the DGA (Direction Générale de l'Armement) which aims at developing a tool that will help users to verify their BPMN models. This tool covers three main aspects: static verification, interactive execution, automatic exploration of the possible scenarios. This paper focuses on the automatic exploration of the model with OBP technology coming from ENSTA Bretagne research lab.**

***Index Terms*—BPMN, Model Checking, Verification**

I. Introduction

Business Process Model Notation (BPMN) is a notation standardized by the Object Management Group (OMG) [1]. BPMN is a widely used language to describe interactions between different participants in a complex organization. It describes the possible sequences of tasks executed by several participants to fulfill a specific mission. The BPMN language offers a large set of very expressive operators enabling the specification of complex process models. These models should eventually be able to describe all possible situations and interactions in the organization.

BPMN is used to describe a wide variety of systems such as air traffic control, satellite constellations, or army forces coordination. Inherent to the complexity of such systems, it is paramount to ensure the functional correctness of the models. To ensure the adequacy of the model to the studied mission, it is common practice to gather the different stakeholders in a room and to review the operational scenarios captured by the model. This ensures that the BPMN specification correctly captures the characteristics of the targeted system. During these review sessions the model is typically corrected, and refined until it captures unambiguously all stakeholder requirements. However, while absolutely necessary, these meetings are time consuming and error prone mainly because the model interpretation is performed by humans. To address this problem we propose a tool enabling semantic-driven model manipulation and verification, which ensures the model correctness with respect to BPMN semantics.

The basic principles of the BPMN notation are very simple and straight forward. The model describes a succession of tasks realized one after the other. To represent alternative execution paths, a gateway symbol is used to describe an exclusive, an inclusive, or a parallel choice of path. Still, the gateways can be quite complex and create unexpected behavior when combined with each other. This is especially true when implicit gateways are concerned, because they may be misinterpreted by modelers. A hierarchy of scenario can also be described using the call activities and information between participants is exchanged through messages. Even though the basic concepts are easy to understand there are a number of subtle semantic variations in the standard that makes it tricky to interpret correctly. Moreover, the information on which the paths are selected depends on conditions that are described in natural language. Thus, it is quite probable that the same model might be interpreted differently by two different readers. A tool that can interpret the notation and replay a set of scenarios would be of great help to speed up the verification process and settle down the real meaning of the model.

BPMN model correctness should be ensured along three main axes: syntactic, semantic, and logic. From the syntactic point of view, the model should adhere strictly to the standardized BPMN definition, which guarantees the grammatical correctness and offers a sound basis for its semantic interpretation. From the semantic point of view, the interpretation of the model should be captured formally and unambiguously, which in turn enables semantically guided interactions with the model (step-by-step execution, debugging, profiling, etc). Finally, at the logic level, the model interpreted through the semantics should correctly capture the mission of the system it models. In the context of the VeriMoB project, we address these correctness facets by: (1) performing static syntax checks, (2) using a unique definition of the executable semantics for all model analysis, i.e. interactive model animation, trace simulation, and verification. For verifying the logical correctness of the model we rely on the OBP model-checking framework, which offers a semantics-agnostic LTL verification platform. The main innovation of our approach stems from the direct integration of our industrial-grade BPMN interpreter, capturing the executable semantics, with an off-the-shelf model-checking technology. The operational semantics of the BPMN notation is unambiguously defined, and the integration of the OBP model-checking tool offers an interesting add-on for the logic verification of BPMN models.

This project was funded by the French Army (DGA) in collaboration with Eurocontrol and Airbus DS who provided some real use cases.

This paper will go through the work of the integration of OBP tool within the VeriMoB project and the results that were obtained.

II. Related Work

There is a substantial set of tools supporting BPMN. While all of them provide the ability to edit the different diagrams, some of them provide sharing capabilities dedicated to large organizations, and a few provide execution facilities and simulation features. It appeared the execution facilities that are available are very poor and restrictive. The simulation features are actually referring to statistical simulation that includes capabilities and statistical input information computed automatically.

In [2] the authors evaluate the model against the real system through history logs. Even though that is interesting to make sure the system conforms to the process, it does not help to verify the BPMN model is actually correct.

The Business Process Simulator BIMP is an on line simulation tool supported by the University of Tartu and the Estonian Research Council [3]. The model is uploaded and a simulation scenario is defined. As a result the tool will estimate the costs of the system and of each individual process, as well as potential bottlenecks and resource utilization. This will help to optimize the model but not verify it.

In [4] the authors investigate how to make sure a new version of a BPMN model does not suffer from regression through the use of model checking techniques. This process will verify some properties are still valid when the model is modified but that does not verify the model in the first place against the same properties and it does not address the verification of functional properties.

In [5] a list of BPMN simulation tools are evaluated to optimize performance of the model. BIMP is considered simplistic, Bizagi is considered user friendly, BPSim was evaluated the most complete, BonitaSoft was in an early phase of development, Visual Paradigm suffers from its UML origin and has pretty limited possible inputs. Because they are simulation tools to optimize performance the inputs are statistical laws. Their unique goal is to optimize the throughput of the model or the necessary re-sources.

In [6] the BPMN description is transformed to PIF (Process Intermediate Format) which is then transformed to LNT (LOTOS New Technology) to be fed into CADP (Construction and Analysis of Distributed Processes) verification tool from INRIA. Even though the paper does not mention it, it is very unlikely the BPMN semantic is respected through the multiple transformations. In the end the limitation of the CADP tool will end up with a very experimental result.

In [7] the authors present an overview of business process models verification tools. Among the numerous references in the paper two aspects of the model can be verified: the syntactic correctness to prevent improper usage of the modeling elements, the structural correctness to prevent wrong dynamic behavior. For each verification tool the coverage of the different aspects is described but the correctness of the behavior is not addressed.

The investigation of the state of the art around BPMN shows that the VeriMoB project addresses a unique verification capability that no other tool offers. It might be interesting to discuss the vocabulary used in this domain. Usually a simulation is the execution of a model step by step. In BPMN world, simulation is often understood as a statistical set of inputs providing a statistical set of outputs in order to evaluate the necessary resources for a whole system throughput. For that reason we will preferably use the term execution instead of simulation.

III. A Language Agnostic Verification Tool

Verification tools come with their own language and associated semantics, which are different from the one that has been used to model the system to be verified. To verify a model, the usual process is to first translate the model to the verification language. But since the underlying concepts of the languages might not be fully aligned, the translated model is a twisted version of the initial model, sometimes with a lot of additional restrictions because of translation issues. This renders the analysis of the verification results, during diagnosis, very difficult.

Because of the inherent complexity of the modeled systems, the verification tools are very complex. Whether based on exhaustive simulation or on symbolic resolution the complexity leads to set a substantial number of restrictions on the model itself. This could be on data types manipulated in the model or some specific constructs that generate complexity.

In the end the verification tool runs on a twisted model with data or construct restrictions. That substantially lowers the value of the verification results that might be generated.

One of the innovative aspects of the OBP approach is that it relies on an external model execution engine that is compliant with the model semantic. OBP does not know anything about the model; it does not even load the model. It leaves this task to an external editor and execution engine. OBP focuses on the analysis of the execution with respect to logical properties, which themselves are grounded on the semantics defined by the execution engine. In OBP the property to be verified is a logical combination of elementary properties. The logical structure is understood by OBP but the evaluation of the elementary properties is delegated to the language executor. To achieve such integration the connection between OBP and the BPMN executor is based on the API shown in figure 1.

To control the model execution, during a verification run, OBP can:

- retrieve the global state of the model execution;
- set the global state of the model execution;
- collect the possible next executable steps in a given state of the model execution;
- execute one of the possible steps in a given model execution state.

For the evaluation of a temporal logic property, the OBP engine starts by requesting it from the executor. The property written in either LTL or Büchi automata is decomposed in elementary boolean properties to be evaluated by the executor. Based on these interactions the verification engine is capable of proving that the model satisfies the property, using standard

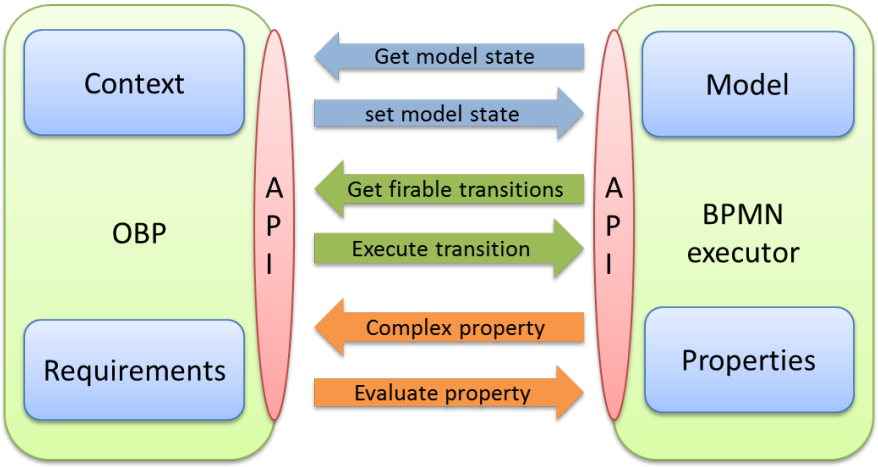

Fig. 1. VeriMoB internal architecture.

model-checking algorithms. If the property is not satisfied, a counter-example is provided.

This approach offers a number of clear advantages, compared to the approaches based on model transformation: it assures the uniqueness of the language semantics between interactive execution and model-checking, which relieves the need of proving semantic equivalences; and the counter-example, produced when the property is not verified by the model, is expressed directly in terms of model concepts, which can be simply replayed by the executor during diagnosis. Furthermore, through the use of LTL and Büchi automata OBP offers a standard property specification language, which can be used either directly or as the transformation target for higher level property specification, such as the Property Sequence Charts [8].

IV. BPMN Executor

Each BPMN element in the model has a state of execution, and depending on its state it may accept actions (i.e., transitions in figure 1). An action can change the state of an element during execution, and it can be either *enable* or *disable*. Since an element can be enabled or disabled several times (i.e., many flows of execution can go through the same element), each element has actually a list of states of execution. The current execution state of a BPMN model is composed of the execution states of all elements in the model. The following execution states are possible for a BPMN element:

- *None*: the element does not accept any action from the user, and it has never been enabled or disabled.
- *Active*: the element is waiting for either an enabling or disabling action from the user.
- *Ready*: an enabling action was issued on the element, but the element cannot be enabled yet because it depends on the state of other elements.
- *Enabled*: an enabling action was previously issued on the element, and all enabling conditions have been fulfilled (i.e., the other elements it depends on are in the required state).
- *Disabled*: a disabling action was issued on the element.

During execution the most recent state of the element is displayed in color: blue for active, orange for ready, green for enabled, and no color for none and disabled.

V. Level of Complexity

At first sight the level of complexity of a BPMN model seems moderate. There are no data types in the model that might create complexity because of the possible values each variable could be assigned to. The scenario alternatives are limited to the gateways. However, the gateways can be quite complex and create unexpected behavior when combined with each other.

A. Loops

Figure 2 shows an example of an infinite loop. The end symbol is never reached by all flows due to the implicit parallel forking gateway in *task 2*. This kind of construct poses a serious issue for model checking, because at every loop iteration a new model state is created by the BPMN executor. As a result, the state space can be infinite, which is something we don't want in model checking.

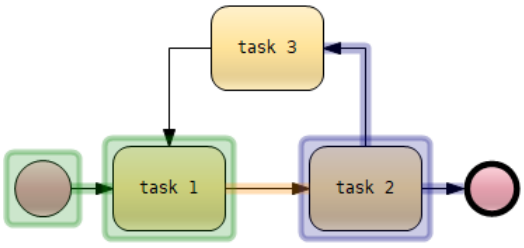

Fig. 2. Infinite loop.

A simple modification of the model can solve the issue. One possible solution in shown in figure 3 via the conditional sequence flow. This makes the loop optional, and therefore creates the possibility to interrupt the infinite loop.

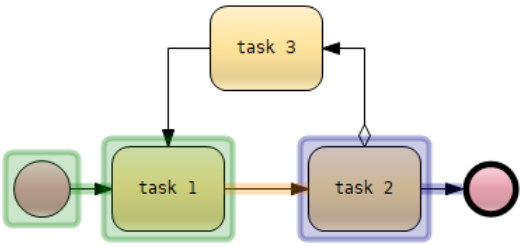

Fig. 3. Conditional infinite loop.

However, this does not solve the problem for model checking, because looping indefinitely remains a possible execution scenario. For this reason we limit loop execution to one iteration. We can do this because there are no data types considered during model checking that can be affected by iteration. The only thing affected by a loop is the coverage of elements that form the looping path, hence a single iteration (and no iteration) will give us full coverage of all possible scenarios of execution.

B. Empty Pool Collaboration

Large models are described in several diagrams written by different modelers. In these situations the pools that are defined by the other modelers are represented by empty pools as shown in figure 4. We call these empty pools black-boxes.

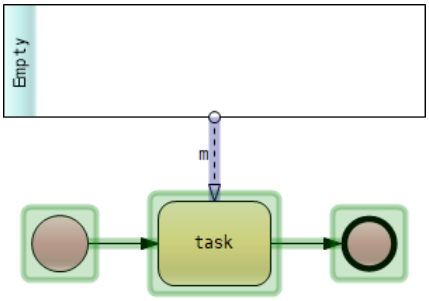

Fig. 4. Collaboration with empty pool.

During execution these black-boxes might be defined or not, and even if they are defined one might not necessarily want to execute them. When a black-box is undefined or not executed its outgoing message flows can be always enabled (i.e., sent), but a sent message may not be received. This is another example of an infinite state space in model checking, because every unreceived message creates a new execution state. There is no solution in this case other than setting a limit for the number on messages sent but not received. In reality this limitation does not play an important role in model checking because the redundant messages do not affect the possible execution scenarios, i.e., scenario coverage does not depend on redundant message flows.

VI. Property Sequence Chart

BPMN is a graphial notation that is used by domain specialist which might not be familiar with modeling notations. When it comes to expressing the properties of the process the mathematical and algebraic notations should be avoided. Since the BPMN executor produces execution traces based on the MSC (Message Sequence Chart) standard [9] representation that is quite easy to read, the use of the PSC (Property Sequence Chart) [10] came as an obvious choice.

The PSC is a simple and expressive formalism that aims at facilitating the non trivial and error prone task of specifying temporal properties in a correct way and without expertise in temporal logic. PSC is a language that extends a subset of UML 2.0 Interaction Sequence Diagrams [11] or the ITU-T Message Sequence Chart. Within the PSC language, a property is seen as a relation on a set of exchanged system messages, with zero or more constraints. PSC may be used to describe both positive scenarios (i.e., the "desired" ones) and negative scenarios (i.e., the "unwanted" ones) for specifying interactions among the components of a system. For positive scenarios, PSC allows to specify both mandatory and provisional behaviours. In other words, it is possible to specify that the execution of the system must or may continue to complete the described interaction. Figure 5 shows the available symbols in PSC diagrams.

Instances are represented as in MSC diagrams. The parallel, alternative and loop operators are represented the same way as the *par*, *alt* and *loop* in-line expressions in MSC diagrams respectively. The relative time constraint has the same representation and semantics as in MSCs. Messages in PSCs have two representations:

- An arrow going from the sender to the receiver as in MSC diagrams;
- A textual representation with the format:

```
<sender-instance-name>.<message-name>.<receiver-
  instance-name>
```

  This representation is used in constraints, explained below.

Unlike messages in MSC diagrams, message arrows in PSC diagram can be of three kinds:

- A regular message, identified by the prefix "e:" for the message text, is a precondition for what follows.
- A required message, identified by the prefix "r:" for the message text, is a message that must occur if the preconditions are met. Required messages must always appear after all regular messages.
- A fail message, identified by the prefix "f:" for the message text, is a message that must not occur if the preconditions are met. Fail messages may appear after the regular messages.

When describing a property, the default ordering is the loose ordering: anything can happen between a message specified in the PSC and the one following it. For cases where a strict ordering is necessary, i.e., when a message in the PSC must be directly followed by the one following, the strict operator can be used either on a message send or a receive. Figure 6 shows an example where the answer message must strictly follow the request message.

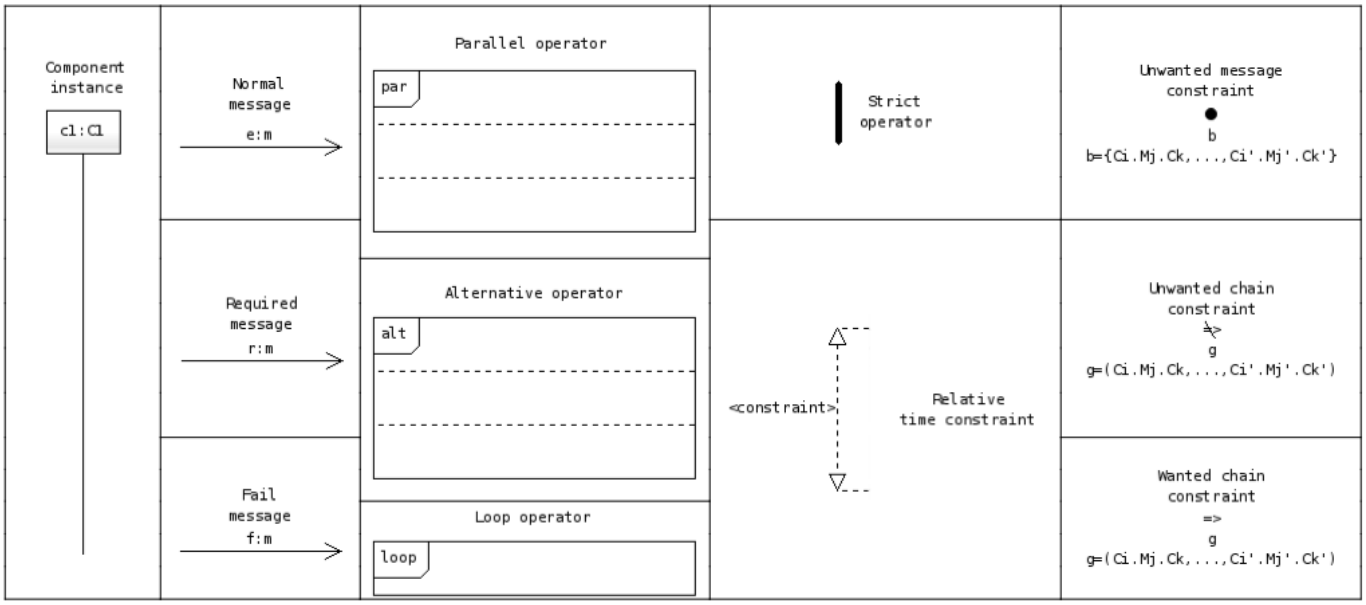

Fig. 5. PSC graphical notation.

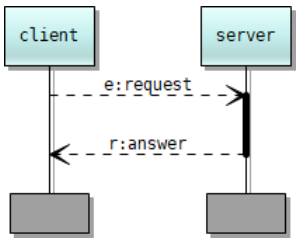

Fig. 6. PSC strict operator example.

To be consistent with the BPMN notation the messages are drawn with a dashed arrow in the MSC.

The PSC diagrams also allow to set constraints on the messages. These constraints are shown as symbols at the beginning or end of message arrows with an associated text. These constraints can have 3 types:

- An unwanted message constraint denotes a set of messages where none should happen before or after the message it is attached too, depending on whether it appears at the beginning or the end of the arrow.
- An unwanted chain constraint denotes a sequence of messages that should not appear as a whole before or after the message it is attached to.
- A wanted message constraint denotes a sequence of messages that must appear as a whole before or after the message it is attached to.

Figure 7 shows a simple PSC example where a request from a client to a server must get an answer as a reply, not another request.

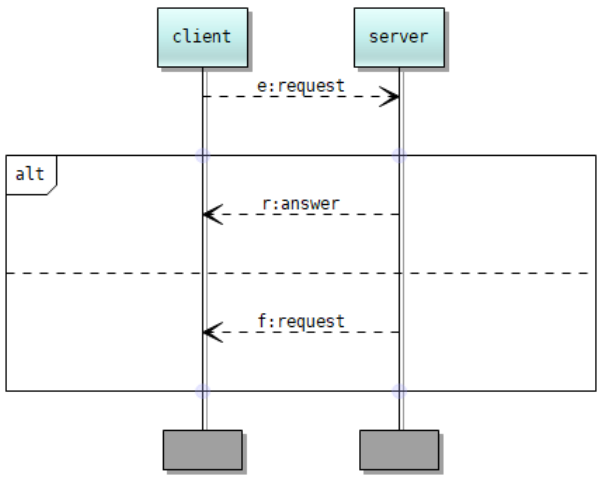

Fig. 7. PSC with alternatives.

VII. Generic Property Specification Language

The Generic Property Specification Language (GPSL) [12] is the language used by OBP for specifying the properties that should be verified during the analysis. Methodologically it is orthogonal from the formalisms used for capturing the operational environment (xGDL scenarios [13]) and for taming the state-space explosion problem during model-checking (state-space decomposition, pruning through state-constraints - TLA [14], etc.).

A. Structural Layer

Structurally a GPSL specification is composed from a set of property definitions, according to the following syntax:

TABLE I
GPSL PROPOSITIONAL OPERATORS

Operation	Syntax	Operation	Syntax
negation	$\neg$ f	exclusion	f $\oplus$ g
disjunction	f $\vee$ g	implication	f $\rightarrow$ g
conjunction	f $\wedge$ g	equivalence	f $\leftrightarrow$ g

TABLE II
GPSL TEMPORAL OPERATORS

Operation	syntax	Operation	syntax
next	$\bigcirc$f	weak until	f W g
eventually	$\Diamond$f	strong release	f R g
always	$\Box$f	weak release	f M g
strong until	f U g		

```
identifier = expression
```

The expression of a property definition can reference a previously defined definition through its name (identifier).

To simplify the expression of large formula GPSL uses *"let in"* expression forms to introduce variables. For instance, in the following listing, the variables v_1 through v_n can be used to define subexpressions needed for building the expression $expr_0$.

```
let
    v1 = expr1
    ...
    vn = exprn
in
    expr0
```

B. Atomic Propositions

The main characteristic of GPSL is its independence from the formalism used for model-specification. To achieve this independence relation, the GPSL language delegates the evaluation of the atomic propositions to the verification model semantics (BPMN executor in our case).

Atomic proposition in GPSL are strings of characters enclosed between pairs of '|' (e.g., |*string-of-characters*|). Each atomic proposition can be prefixed with two identifiers specifying the atomic proposition language and the 'verification' module on which it should be interpreted.

C. Propositional Layer

The propositional layer is used to define boolean expression based on *a)* the *true* and *false* literals, *b)* the atomic propositions, and *c)* the previously defined identifiers.

Two expressions f and g can be combined using the propositional operators in Table I.

D. Büchi Automaton Layer

The GPSL language uses Büchi automata for expressing temporal properties. In the language a Büchi automaton is seen as a top level expression that can reference other definitions without being referenceable. Moreover, the *in* clause of the *let* construct is extended with the syntactical term for describing the automaton. In this context an automaton is composed of four parts:

- the set of states,
- the initial state(s),
- the accepting state(s), and
- the guarded transitions.

The set of states is defined by the keyword "states" followed by a comma delimited list of identifiers (state names):

```
states := 'states' identifier (',' identifier)*
```

The initial state(s) are introduced with the keyword "initial" followed by a comma delimited list of state names:

```
initial := 'initial' identifier (',' identifier)*
```

The accepting state(s) are introduced with the keyword "accept" followed by a comma delimited list of state names:

```
accept := 'accept' identifier (',' identifier)*
```

Each transition is specified by the source state, the guard expression, and the target state, where the source and target are state identifiers, and the guard expression is a propositional logic expression composed using literals (*true, false*), atomic propositions (|...|), propositional operators (!, *and*, *or*, ->, etc.), and named variable references:

```
transition := identifier '[' expression ']' identifier
```

The automaton is defined by the following syntax:

```
automaton :=
  states ';'
  initial ';'
  accept ';'
  transition (';' transition)*
```

E. LTL Layer

The Linear Temporal Logic support is provided by syntactically extending the propositional layer with the temporal operators defined in Table II. For verification the LTL expression are transformed to Büchi automata using the ltl3ba tool [15].

VIII. TRANSLATING THE PROPERTIES

The PSC notation is based on the work initially started by the University of l'Aquila [8]. The semantics of the PSC are defined in terms of a transformation to Büchi automata. In our case we have adapted the transformation to generate Büchi automata conforming with the GPSL Büchi automaton layer. As the PSC is a sequence of events, the idea is to build an automaton that will follow up the different steps in the scenario and introduce a state in between the events. In figure 8 the client sends a request to the server that replies with an answer.

An obvious property for such a model would be that, if a request is sent to the server, than an answer should be sent

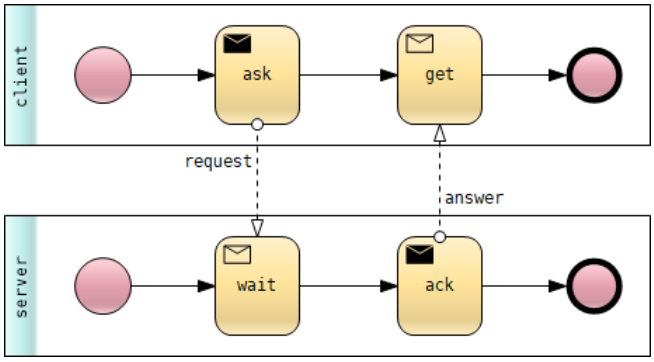

Fig. 8. A simple BPMN client server example.

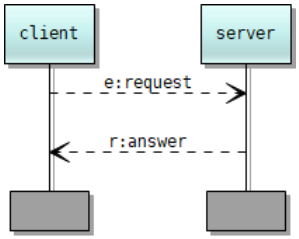

Fig. 9. A simple PSC for the client-server example.

back to the client. Figure 9 shows such a property with the PSC notation, which would be translated in a Büchi automaton as follows:

```
main_property =
  let
    server_answer_client = |+:E:/0:SEM_SYMB_21|,
    client_request_server = |+:E:/0:SEM_SYMB_20|
  in
    states S3, S1, S4;
    initial S1;
    accept S3;
    S1 [not client_request_server] S1;
    S1 [client_request_server] S3;
    S3 [not server_answer_client] S3;
    S3 [server_answer_client] S4;
    S4 [not client_request_server] S1;
    S4 [client_request_server] S3
```

The *server_answer_client* and *client_request_server* are assigned some internal representation of the message sending in the model. Then three states are defined: *S3*, *S1*, and *S4*. *S1* is set to be the initial state and *S3* the accept state. So if the automaton is in the *S3* state at the end of the exploration, that means the property has been satisfied (or violated). The execution starts with *S1* as a starting state. The transition descriptions show that only *client_request_server* can move the automaton from *S1* to *S3*, this means that if the client sends the request message to server, the property automaton will go the *S3* state. Once in state *S3* the condition *server_answer_client* leads to *S4*. This means that if the server sent the answer message to the client, the property automaton will go to the *S4* state. If the second part of the property is verified, the automaton will switch to non accept state.

IX. Example

A. Exploring the State-Space

Once the integration between the BPMN executor and OBP was done, it was possible to explore all the possible execution paths. A first, since there was no property defined yet, the only result that could come out of such an exploration was the number of possible *configurations* or possible states. This basic information turned out to be informative of the model complexity. If the number of possible execution paths is very large compared to the model complexity, it probably means that there is a mis-construct in the model that is creating such complexity. A typical situation we had to deal with was the *implicit gateways*. Figure 10 shows an excerpt of one of our use cases coming from the army (the CAS - Close Air Support model).

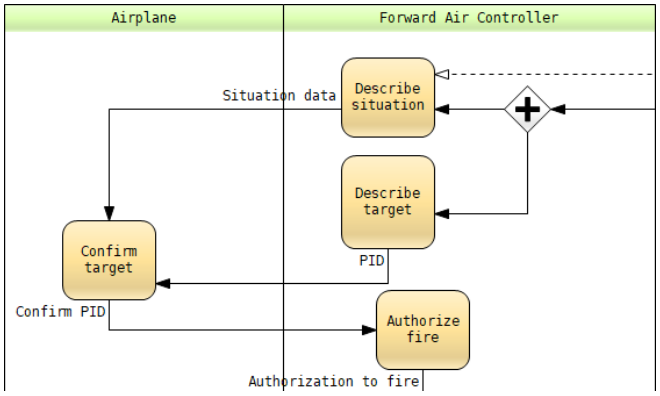

Fig. 10. Example of an exclusive implicit merge in the CAS model.

The forking gateway is an explicit parallel one, however the merging one is an implicit exclusive gateway in the "Confirm target" task. This means the two flows of execution generated by the forking gateway will not wait for each other when merging back. A simple exploration of such an example ended up with more than 9,000 configurations! As such this information demonstrates an mis-construct. After fixing the model with a proper merging parallel gateway as shown in figure 11, a new exploration led to only 38 configurations.

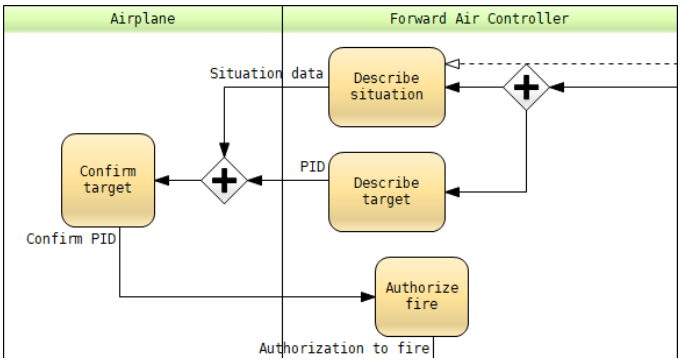

Fig. 11. Corrected CAS model with an explicit parallel merge.

A simple exploration can therefore generate a complexity index of the model. If the index is too high it is quite likely there is a mis-construct in the model.

B. Property Verification

The mis-construct in the CAS model can be better identified via property verification. For this we defined the following GPSL property:

```
// S -> "Describe situation" was enabled
S = |=:E:CAS:MegaId-331F39A752DD5FDC|

// T -> "Describe target" was enabled
T = |=:E:CAS:MegaId-331F39BE52DD603B|

// P -> "Authorize fire" becomes active
P = |+:A:CAS:MegaId-331F3B2A52DD67E0|

// Property:  P if both S and T
main_property = <>P -> (!P U S) && (!P U T)
```

In a nutshell, the property says that "Authorize fire" should be possible only if both "Describe situation" and "Describe target" have been enabled (or done executing). Clearly such property will not be satisfied when verified against the original CAS model in figure 10, and OBP will generate the counter-example that violates the property. The counter-example generated by OBP can be replayed by the BPMN executor, and the result is shown in figure 12.

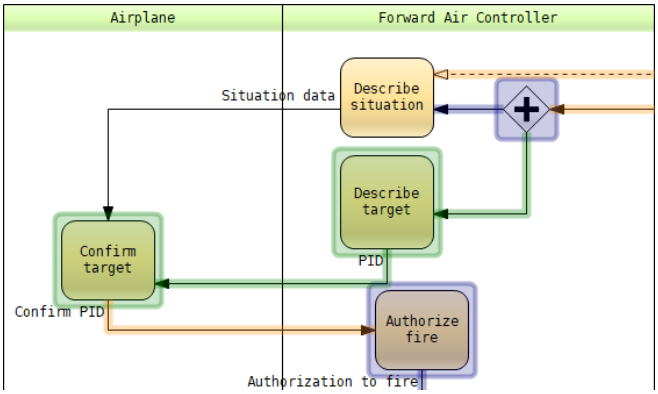

Fig. 12. Counter-example generated from violated property.

The figure shows that "Describe situation" has not been enabled even though "Authorize fire" is active, hence confirming once again the presence of a mis-construct in the model. Obviously, the same property is satisfied when verified against the corrected model in figure 11. In this case no counter-example is generated, and OBP will report satisfaction of the property.

X. Outline

VeriMoB project was aiming at verifying BPMN models. For that matter a set of tools have been developed including a syntactic analyzer, a dynamic execution engine, a tracing mechanism, and a property verification engine based on OBP.

Even though the BPMN semantic looks fairly simple at first sight, it actually contains a very complex set of variations. After analyzing the different industrial use cases involved in the VeriMoB project, it turned out most of the models were dynamically not correct.

This paper went through some BPMN constructs that we encountered in the use cases, and explained how we did address the common problem of potential state-space explosion in model-checking. We then switched focus to property verification by introducing PSC as a graphical alternative to GPSL and the Büchi automaton for describing properties to be verified by OBP. Even though the PSC is the easiest of the three for describing properties, it is more limited due to its notation, most notably the fact that it supports only message flows. We intend to explore ways for addressing PSC limitations in the future for bringing the it closer to the expressiveness of GPSL and Büchi automaton. Finally, the benefit of BPMN model verification was illustrated with an example extracted from one of the industrial use cases provided in the VeriMoB project.

References

[1] OMG, "OMG Business Process Model and Notation (BPMN). Version 2.0.2," Object Management Group, OMG Standard, 2013, https://www.omg.org/spec/BPMN/.

[2] O. Allani and S. A. Ghannouchi, "Verification of BPMN 2.0 Process Models: An Event Log-based Approach," *Procedia Computer Science*, vol. 100, pp. 1064 – 1070, 2016.

[3] BIMP, "BIMP - The Business Process Simulator," http://bimp.cs.ut.ee/, 2019.

[4] J. C. P. Aguilar, K. Hasebe, M. Mazzara, and K. Kato, "Model Checking of BPMN Models for Reconfigurable Workflows," *CoRR*, vol. abs/1607.00478, 2016.

[5] A. P. Freitas and J. L. Pereira, "Process simulation support in BPM tools: The case of BPMN," http://hdl.handle.net/1822/39192, 2015.

[6] A. Krishna, P. Poizat, and G. Salaün, "VBPMN: Automated Verification of BPMN Processes," in *13th International Conference on integrated Formal Methods (iFM 2017)*, 2017.

[7] A. Suchenia, P. Wisniewski, and A. Ligeza, "Overview of Verification Tools for Business Process Models," in *Communication Papers of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017*, 2017, pp. 295–302.

[8] M. Autili, P. Inverardi, and P. Pelliccione, "Graphical scenarios for specifying temporal properties: an automated approach," *Automated Software Engineering*, vol. 14, no. 3, pp. 293–340, 2007.

[9] ITU-T, "Message Sequence Chart," International Telecommunication Union – Telecommunication Standardization Sector, ITU-T Recommendation Z.120, 2011, https://www.itu.int/rec/T-REC-Z.120-201102-I/en.

[10] E. Gaudin and E. Brunel, "Property Verification with MSC," in *SDL 2013: Model-Driven Dependability Engineering*, F. Khendek, M. Toeroe, A. Gherbi, and R. Reed, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 19–35.

[11] OMG, "OMG Unified Modeling Language (UML). Version 2.5.1," Object Management Group, OMG Standard, 2017, http://www.omg.org/spec/UML/2.5.

[12] OBP2, "OBP2," http://www.obpcdl.org/, 2019.

[13] L. Le Roux and C. Teodorov, "Partially bounded context-aware verification," in *Software Engineering and Formal Methods*, P. C. Ölveczky and G. Salaün, Eds. Cham: Springer International Publishing, 2019, pp. 532–548.

[14] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[15] T. Babiak, M. Křetínský, V. Řehák, and J. Strejček, "Ltl to büchi automata translation: Fast and more deterministic," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Flanagan and B. König, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 95–109.