Karlsruhe Reports in Informatics 2019,4
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Verifying Workflow Models with Data Values - a Case Study of SMR Spectrum Auctions

Elaheh Ordoni, Jutta Mülle, Klemens Böhm

2019

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Verifying Workflow Models with Data Values – a Case Study of SMR Spectrum Auctions

Elaheh Ordoni, Jutta Mülle, Klemens Böhm

April 2019

Abstract

Industry takes a great interest in verification techniques to improve the reliability of process designs. Providing reliable design in application domains like spectrum auctions is crucial. Spectrum auction revenue is considered as one of the principal sources for governmental income. Hence, analyzing the auction design before applying it can ensure absence of undesirable results of an auction. Those results might even be bad, if they occur with a probability of just higher than zero. Current verification approaches are mainly devoted to verify control flows only, although data values play a significant role in real life applications. Thus, these approaches are not sufficient to support data-centered workflows as spectrum auctions. We address this issue by providing a new data-centered verification approach to analyze Simultaneous Multi-Round (SMR) auction design in BPMN format. We show how to enhance a BPMN model by including important information, namely data values used in the workflow, which the standard BPMN 2.0 does not support. An example of a data value in a SMR auction is the "auctioneer's revenue". To enable the verification of data-centered properties, we have developed a transformation of a data-value enhanced BPMN model to Petri Nets respecting the semantics of certain data value usages. For that, we support dynamic and correlated data values. By employing a model checker and defining data-centered properties in CTL formula, we verify SMR auction models to find undesirable executions for auctioneers. For example, we can precisely detect the worst values of three important measures in auctions: *efficiency*, *revenue*, and *bidder's profit*. With it, we can not only find the undesirable outcomes, but also provide a counter-example to help an auctioneer to improve the auction design.

1 Introduction

Industry takes a great interest in verification techniques to improve the reliability of process designs. Providing reliable design in application domains, like spectrum auctions, is crucial. Spectrum auction revenue is considered as one of the principal sources for governmental income. In Germany and the UK, for instance, it earned 50.8 and 37.5 billion euro in 2000, respectively[15]. Despite of all the experimental analysis, auctions with embarrassing outputs happened in different domains. In the Netherlands UMTS Auction in 2000, for example, the low revenue caused a fiasco in public policy[30]. A decade-long loss of 30 MHz in the U.S. mobile market happened because of a policy to increase bid prices. As a result, the flaws in policies cost around 70 billion dollar[17]. In a Switzerland auction in 2011 one bidder paid 485 million Swiss Francs, and another one paid 360 million Swiss Francs for nearly the same package[7]. In another example mentioned in [1],

about fifty percent of products remained undersold at the end of the auction. Discovering auction flaws in laboratories is difficult, because it needs to check all possible executions. For example, an experimental design in [12] requires the evaluation of more than 13 million possible paths by human subjects. This is completely beyond the capacity of any laboratory. Therefore, the catastrophic results could be the consequence of possible courses of auction which are hidden in experimental evaluations in laboratories.

To this end, many verification methods have been developed to find undesirable system behaviours in a precise and unambiguous manner, see [20] for an overview. However, most of these methods only check workflows without the data aspect, particularly without regarding *data value*s. *Data value*s represent all the possible values of a *data object* in a process model. In some process models *data value*s play a significant role for behavioural analysis. It enables verification of properties which are dependent on *data value*s. For example, to verify the lowest auctioneer's revenue in spectrum auctions, we need *data value*s such as "bidder's budget", "price of products", and etc in the process model. However, current verification approaches do not sufficiently support processes containing specification of *data value*s (see Section 7).

To overcome this issue, we propose a new approach to represent and verify properties dependent on *data value*s with the use case of SMR auctions. To take advantage of current verification methods, first we model the complex auction design with a high-level process modeling language, namely BPMN 2.0 [23]. Then, we enhance the BPMN model in order to represent the usage of *data value*s, and specify its manipulation during the process flow. To do this, we annotate BPMN elements which are dependent on *data value*s by using our Specification Expressions (SE). Then, we apply a new algorithm to map this *enhanced BPMN* model to a mathematical modeling language, namely Petri Nets [28], that allows using a comprehensive set of analysis tools [28]. This mapping consists of two steps. First, we transform the control flow of the BPMN model to Petri Nets using the existing approach of [14]. Second, we enhance the Petri Net to include data values and their usage as specified in the BPMN model. Our approach is analogous to [27]. However, they only address the optional and alternative usage of entire data objects by tasks and not data values. To our best knowledge, involving data values is new for transforming BPMN process models to Petri Nets. Based on the resulting Petri Nets that include the semantics of the usage of data values in the process flow, we define data-aware properties of the process model as CTL formulas [9]. By employing a model checker, we are then able to find, for instance, undesirable outcomes and show the executions yielding these results to an auctioneer.

In this paper we focus on the application domain of Simultaneous Multi-Round (SMR) Auctions as a standard format of spectrum licenses auctions for more than two decades since 1994 [21]. We make the following contributions:

- Specification of the usage of *data value*s of *data object*s in BPMN models,
- Verification of workflows with *data value*s,
- Supporting dynamic and correlated *data value*s in workflow verification,
- Modeling a SMR auction design in BPMN 2.0 including the extensive usage of *data value*s,
- Specification of data-dependent properties in a temporal logic like CTL,
- Transformation of high-level workflows into Petri Nets including the semantics of usage and manipulation of data values,

- Implementation of a practical tool to detect high-risk executions in SMR auctions.

Our evaluation shows that our approach can check properties of SMR auctions which are dependent on *data value*s. For example, we can find the worst values of three important measures [8] of auction designs: *auctioneer's revenue* as the sum of final prices for each product, *bidder's profit* as the sum of left budgets of bidders who won a product, and giving the products to bidders with the highest values, known as *auction's efficiency*. We can also precisely answer questions such as: What is the lowest final price for a particular product. With it, we can not only prove the presence of flows in spectrum auctions, but also their absence. Additionally, in case of undesirable outcome detection, we can provide a counterexample which helps auctioneers to adjust the auction

2 Preliminaries

In this section we introduce preliminaries of this paper. In Subsection 2.1 we describe the basics of BPMN 2.0. In 2.2 we represent the formal description of Petri Nets. The syntax of CTL formulae is described in Subsection 2.3. Subsection **??** presents Simultaneous Multi-Round auction and some related rules.

2.1 BPMN 2.0

The standard Business Process Model and Notation (BPMN) gives the execution semantics and graphical elements to model business processes. It provides a common language so that users can easily read and understand the flow of activities. The elements of BPMN belongs to the following categories:
Activities. They represent the tasks to be performed within a business process. A single activity is a *task* and a composition of activities is a *sub process*.
Events. A process can catch or trigger *event*s while it is running. There are different events based on specific conditions. For example, the *start message event* triggers on receive of a message.
Gateways. They are used to split or merge the sequence of flows. *Exclusive gateway* activates only one of the outgoing sequence flows and *parallel gateway* activates all outgoing branches simultaneously.
Swimlanes. They separate different organization units in a process model. *Pool*s represent the whole unit process and *lane*s show its hierarchical sub divisions.
Data Objects. The most important element to show the *data* in the BPMN model is *data object*, which represents the flow of information through the process. A *data object* can be read or written by an *activity* defining *data need* and *data result*, respectively. The collection of *data need*s and *data result*s for a specific *activity* describe its *input set* and *output set*, respectively. A *data object* as an external input of the whole process is called *data input*, and the data created by the process is called the *data output*.
Artifacts. They help to understand the semantics of processes easier by adding the complementary information. A *text annotation* associates with elements to provide additional information. A *group* describes the set of elements which are logically related.
Figure 1 represents an overview on the BPMN 2.0 elements. BPMN 2.0 is the last published version of this standard. We will use BPMN instead of BPMN 2.0 in the following, if it is unambiguous.

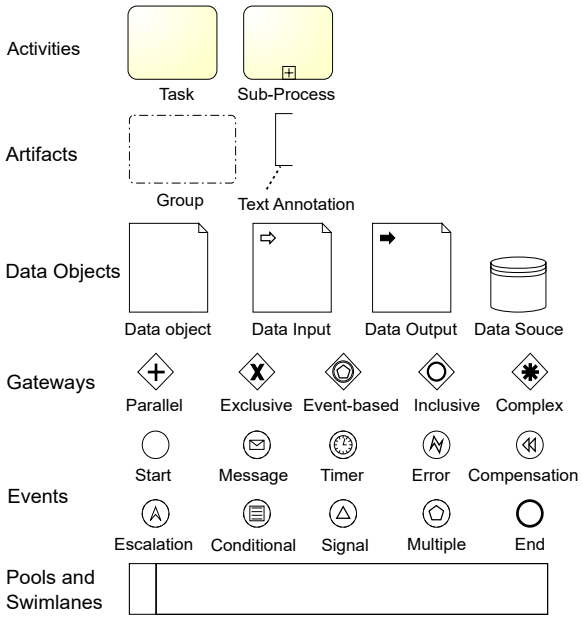


Figure 1: Overview of BPMN

2.2 Petri Nets

A Petri Net [28] is a triple (S, T, W), where

- S is a finite set of *places*
- T is a finite set of *transitions*. $S \cap T = \varnothing$
- $W \subseteq (S \times T) \cup (T \times S)$ is a set of arcs; there is no direct direction from a node to another one with the same type.

A *marking* of a Petri Net is a mapping: $M : P \to N$, which assigns a non-negative number of tokens to each *place*. M_0 is the *initial marking*. Each transition contains immediately preceding places called *input places*: $\bullet t = \{s \in S | W(s,t) > 0\}$ and its just succeeding places called *output places*: $t\bullet = \{s \in S | W(t,s) > 0\}$. A transition t fires when all its *input places* have enough number of tokens and it leads to a new state M'. In M', transition t consumes tokens from each of its *input places* and produce tokens in each of its *output places*. The set of all reachable states from the *initial state* M_0 is the state space. It is possible to check if a Petri Net verifies a given requirement ϕ, by the full exploration of its state space. This method is known as model checking [10]. To specify requirements, some powerful formal languages such as CTL [9] are provided.

2.3 Computation Tree Logic

Computation Tree Logic (CTL) is a temporal logic to represent system properties which has efficient model-checking algorithms. CTL distinguishes two kinds of formulae, state formulae and path

formulae. CTL state formulae are formed according to the following syntax:

$$\Phi ::= true \quad | \quad \alpha \quad | \quad \Phi_1 \wedge \Phi_2 \quad | \quad \neg\Phi_1 \quad | \quad E\varphi \quad | \quad A\varphi \tag{1}$$

where α is an atomic formula. It is a state of a Petri Net in our application field. Greek capital letters represent CTL state formulae, and lowercase Greek letters represent CTL path formulae. The path formula is generated by the following grammar:

$$\varphi ::= X\varphi_1 \quad | \quad \varphi_1 \cup \varphi_2 \quad | \quad F\varphi_1 \quad | \quad G\varphi_1 \tag{2}$$

A path operator (A and E) should always occur with a state operator (X, G, F, and U), or vice versa. $A\varphi$: φ has to hold on all the paths. $E\varphi$: at least one path exist where φ holds. $X\varphi$: φ has to hold at the next state. $G\varphi$: φ has to hold on the whole subsequent path. $F\varphi$: φ has to hold somewhere on the subsequent path. $\varphi_1 \cup \varphi_2$: φ_1 has to hold at least until φ_2 holds.

2.4 SMR Spectrum Auction Design

Simultaneous Multi-Round (SMR) auction has been the standard format to assign the spectrum licenses to bidders for more than two decades since 1994 [21]. It consists of several rounds to award the spectrum to bidders with the highest values. At the beginning of a SMR auction, the auctioneer specifies a *reserve price* for each license, which indicates its lowest possible bid. In this auction format, there are several successive rounds which make the opportunity for participants to bid on individual licenses simultaneously. At the end of each round, the auctioneer recognizes the highest bid price for each license, and informs bidders about the winning bids on this round. The highest bid price for each license will be its *reserve price* in the following round. Figure 2 represents the overview of SMR auction process modeled in BPMN. In this auction type there are several activity rules which can impact auction performance [18].
Capacity rule: Before the start of auction, a bidder gets a *capacity point*, which determines the maximum number of licenses which they can win. This rule limits bidders to win lots of products and it avoids monopoly.
Withdrawal rule: Each bidder can withdraw their provisionally winning bids as often as the defined *withdraw points* allows. After a withdrawal, the price of the corresponding license will be the same as the second highest winning price achieved with no winner.
Eligibility rule: Before the auction starts, each bidder has a particular *eligibility point* based on their deposit. It determines how many bids a bidder can have in each round. The bidder's *eligibility point* decrease if their number of bids are lower than their current *eligibility point*. A bidder can not bid any more, if their *eligibility point* is zero.
The auction ends when there is no new bid for any license. Then the bidder with the highest *bid price* for a product, will be its *winner*.

3 Modeling SMR Auction in BPMN 2.0

To verify data-dependent properties in SMR auction, we make the contribution to model it in BPMN. To do this, we have the following assumptions which are specified in the BPMN mode: (1) Each bidder has a certain budget for each individual license, (2) bidders are in demand for a license unless they can not afford it, and (3) winner of a license do not bid for the same license in the next round. The *data object*s and *attribute*s specify the flow of data in this model precisely.

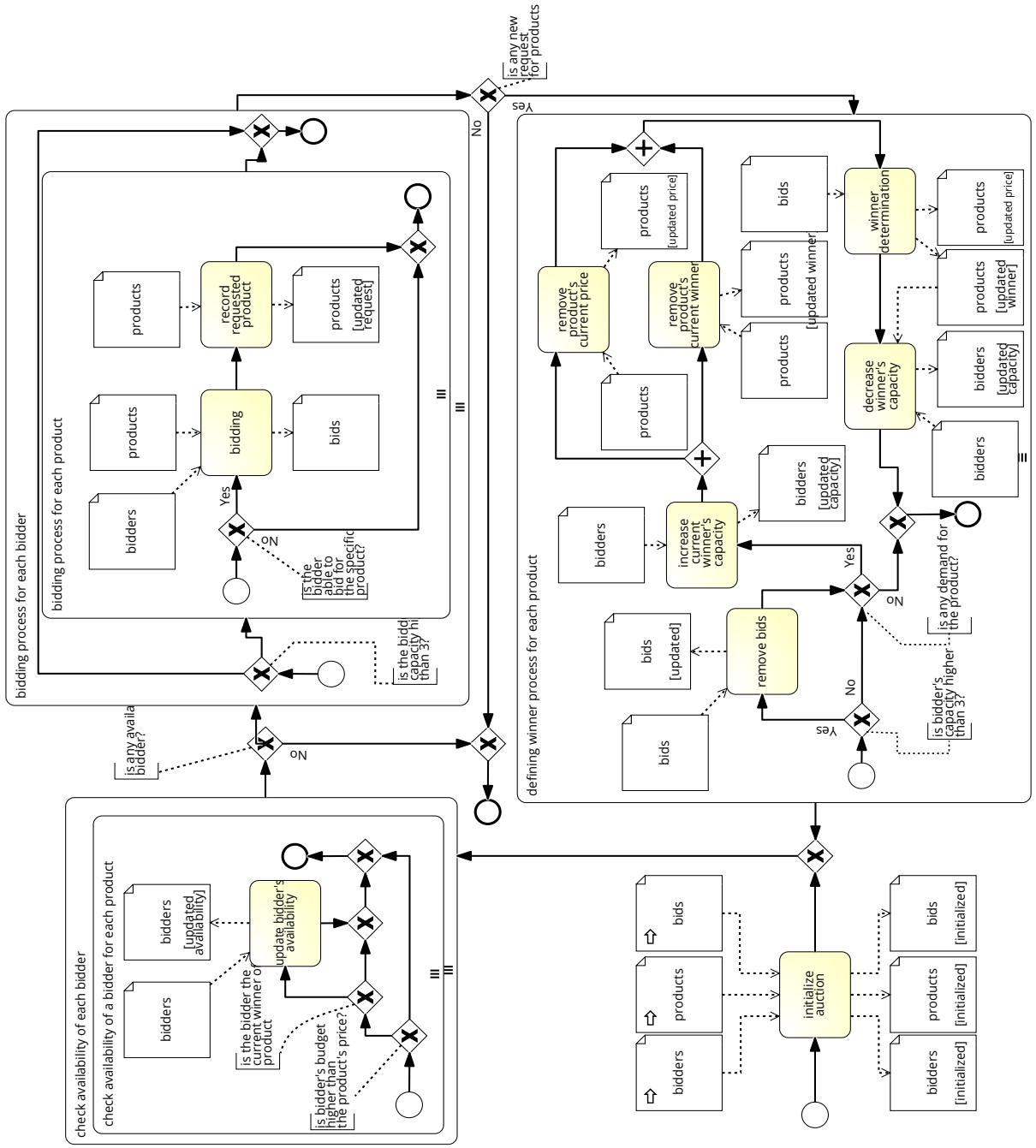


Figure 2: BPMN model for SMR auction

The auction process consists of three phases. First we check whether or not bidders can bid in *Check availability of bidders* phase, by two different *conditional gateways*. One checks if the bidder can afford a certain license, and the second one checks if the bidder is not the current winner of the affordable license. The auction continues to the next phase if these is at least one qualified bidder. All qualified bidders will specify their bid prices in the next phase, *bidding process*. As we have to consider all possible combinations of bid prices in this phase, all the valid bid prices have the equal chance to be selected, even if they have a very low chance in real auctions. To do this, the *bidding task* assigns a random valid bid price to each qualified bidder. With it, we take into account all possible situations in our model to prepare verification of the properties. All the requested licenses will be marked by *record requested products task* so that they are considered in the *winner determination* phase. In the third phase, the *winner determination task* declares the new *reserve price*s and the *winner*s. Based on these results, we update the bidder's capacity points in two tasks: *increase previous winner's capacity* and *decrease winner's capacity*. These three phases iteratively continue until no more qualified bidders exist. Figure 2 represents the overview of SMR auction process modeled in BPMN.

4 Our Verification Approach

To verify *data-value*-dependent properties, we precisely specify the information of interest in our application, see Subsection 4.1. In Subsection 4.2, we formalize the process execution semantics based on involved data values in BPMN by using a transformation algorithm. In Subsection 4.3 we explain how to define properties which we are able to verify by our approach. In Subsection 4.4, we discuss supported features and limitations of our framework.

4.1 BPMN 2.0 Enhancement with Required Data Values

In many applications, *data value*s play a significant role in process executions. For example, in Figure 2, the gateway with the condition `bidder's capacity points > 0` checks a *capacity rule* in a SMR auction. Here, the *gateway* condition is dependent on the *data value* (*bidder's capacity points*) which also can be changed during the process. In particular, *data value*s as the *precondition*s and *effect*s of process elements change the process execution. However, in the standard BPMN 2.0 there is no specification element to represent this important information. To address it, we provide a *Specification Expression* (SE) to enrich, i.e., annotate BPMN elements which get involved with *data value*s. In particular, we enhance *data object*s, *XOR gateways*, and *task*s
new4as the basic BPMN elements which can be dependent on *data values*.

To do this, we use *annotation*s to consider (1) involved *data value*s and (2) their manipulation by the BPMN element. Table 1 lists the SE features which are important to represent *data value*s in data-centered workflows and sufficient to manipulate *data value*s in our use case. Other functions, e.g., other aggregation functions like minimum, are also possible, but we do not need it in our use case. In the following, we explain the SE features in more details:

EL: Declaration. It represents all the possible *data value*s of a *data object* involved in a process. It consists of two parameters. The second parameter identifies the *attribute*s of a *data object*, their *type*s, and *domain*s. As *type*s, we support *integer*, *boolean*, and *enumeration*s of integer type. The

Expression Label - EL	Format	Associated with
1 Declaration	$[x][attribute : type : values]^+$	data objects
2 Selection	$[attribute : selection\ filter]$	data objects
3 Constant Condition	$[data\ object : attribute : selection\ filter]$ $comparison\ operator\ [constant\ value]$	XOR gateways
4 Dynamic Condition	$[data\ object : attribute : selection\ filter]$ $comparison\ operator\ [data\ object : attribute : selection\ filter]$	XOR gateways
5 Increase++	[increase]	tasks
6 Decrease−−	[decrease]	tasks
7 Reset	[reset]	tasks
8 Random Selection	[random selection]	tasks
9 Find Maximum	[find maximum]	tasks

Table 1: Specification Expression for BPMN elements engaged with data values

first parameter, [x], indicates the first x attributes building the *key*. The key is a single attribute ($x = 1$), or a composition of several *attribute*s ($x > 1$).

Example 1. *Figure 3(a) represents an example of this expression for the data object* product*. The first attribute,* ID *is considered to be the key attribute. It also has a* price *attribute of type integer in a range of [5-20].*

EL: Selection. It indicates the *precondition*s and *effect*s of a BPMN element on *data value*s. Particularly, it specifies a selection of *data value*s corresponding to a *data object's attribute* by a *selection filter*. *Selection filter* consists of two values separated by *dots*: *key attribute value* and *attribute value*. The value is an existing defined value in *EL 1*, or x which means *don't care*.

Example 2. *Figure 3(b) represents a selection of values of the data object* product*. As we have the selection filter* 1.x *for the attribute* price*, it selects all the prices which belong to a product with* $ID = 1$.

EL: Constant Condition. It specifies *XOR gateways* to take deterministic decisions, if their condition is dependent on *data value*s. It consists of two parameters: The first is a set of *data value*s deduced by a *data object*, its *attribute*, and a *selection filter* on it. The second parameter is a constant value and a *comparison operator* $\in \{<, >, =, !=, >=, <=\}$.

Example 3. *Figure 3(c) shows an example to amend a gateway with a precise condition: Is the* capacity points *of* bidder *with* $ID = 1$ *higher than zero?*

EL: Dynamic Condition. It is similar to *SP 3* but both parameters are *data value*s.

Example 4. *Figure 3(d) gives an example of a gateway enhanced by SP 4. It takes a decision on the following condition: Is the* price *of* product *with* $ID = 1$ *higher than the* price *of* product *with* $ID = 2$.

EL: Increase++. It indicates the *task*s increase the value of their *data need*s by 1. Here, we specify the set of *data value*s to be increased with SP1. The manipulated *data value*s define the *data result* of the respective *task*.

Example 5. *In Figure 3(e), the* price *of* product *with* $ID = 1$ *will be increased by 1.*

EL: Decrease−−. It is same as SP 5 but to decrease *data value*s by 1.

Example 6. *Figure 3(f) illustrates the usage of EL 6 to decrease the* price *of* product *with* $ID = 1$. *The last two specifications allow to model the so-called* capacity *rule in a SMR auction.*

EL: Clear. It specifies that the related *task* clears the current value of its *data need*.

Example 7. *In our use case, we want to clear the value of the attribute* price *before assigning a new value. Figure 3(g) represents this situation with clearing the* price *of* product *with* $ID = 1$.

SP: Random Selection. It defines a general function for *task*s to select a random value out of the associated set of *data value*s. Especially, we developed this function to model *bidding* behaviour in our use case. Our goal is to include all the possible SMR executions in our model. Thus, a *task* representing *bidding* can select any *bid* which has a selection probability just higher than zero in reality.

Example 8. *Figure 3(h) indicates an example of this specification. It randomly select a* bid price *for a* product *with* $ID = 1$, *based on the* budget *of* bidder *and the current* price *of* product.

SP: Find Maximum. It can be associated with a *task* to find the maximum value of the set of its associated *data value*s. All the *data value*s should be *integer*. We extend this specification particularly for our use case to find the winner and new prices based on the highest bids.

Example 9. *Figure 3(i) is an example to find the maximum* bid price *of a* product *with* $ID = 1$. *Based on this, it defines the* winner *and the* price *of a* product *with* $ID = 1$

4.2 BPMN2PetriNet Transformation

In order to provide a systematic investigation of scenarios through the whole process, many approaches transform BPMN to Petri Nets. For example, [14] transforms the BPMN control flow to a Petri Net. The approach of [27] additionally allows transforming optional or mandatory *data object*s used in the BPMN model. [19] gives a survey on transformations from business processes to Petri Nets. However, these approaches are not sufficient to transform processes containing specifications of data values like auction processes into Petri Nets for verification.

So, we provide a new approach to transform BPMN models to Petri Nets considering the usage of *data value*s. To do this, we transform the BPMN elements which are independent of *data value*s, with the rules defined in [14]. The approach of [14] indicates the state-of-art for transforming BPMN flow objects into Petri Nets. [27] extends it to handle the usage of entire data objects by tasks as specified in BPMN 2.0, but not data values. Next, we develop new mapping rules to transform BPMN elements with their *data value*s used into a Petri Net representation. As an alternative, one could transform the model into YAWL [29] to consider *data value*s. It would help to support more features like the interruption of the entire net when an event occurs, but the verification of YAWL nets is computationally more expensive compared to Petri Nets [14]. It is an important issue as considering *data value*s leads to even more complex nets which consume more time. To transform the BPMN model into Petri Net, we follow two steps: (1) **Mapping** *data value*s involved in a process to new *place*s of the Petri Net (2) **unfolding** particular subnets for all the *data-value*-dependent elements. These elements are already enhanced by the *annotation* element in the BPMN model.

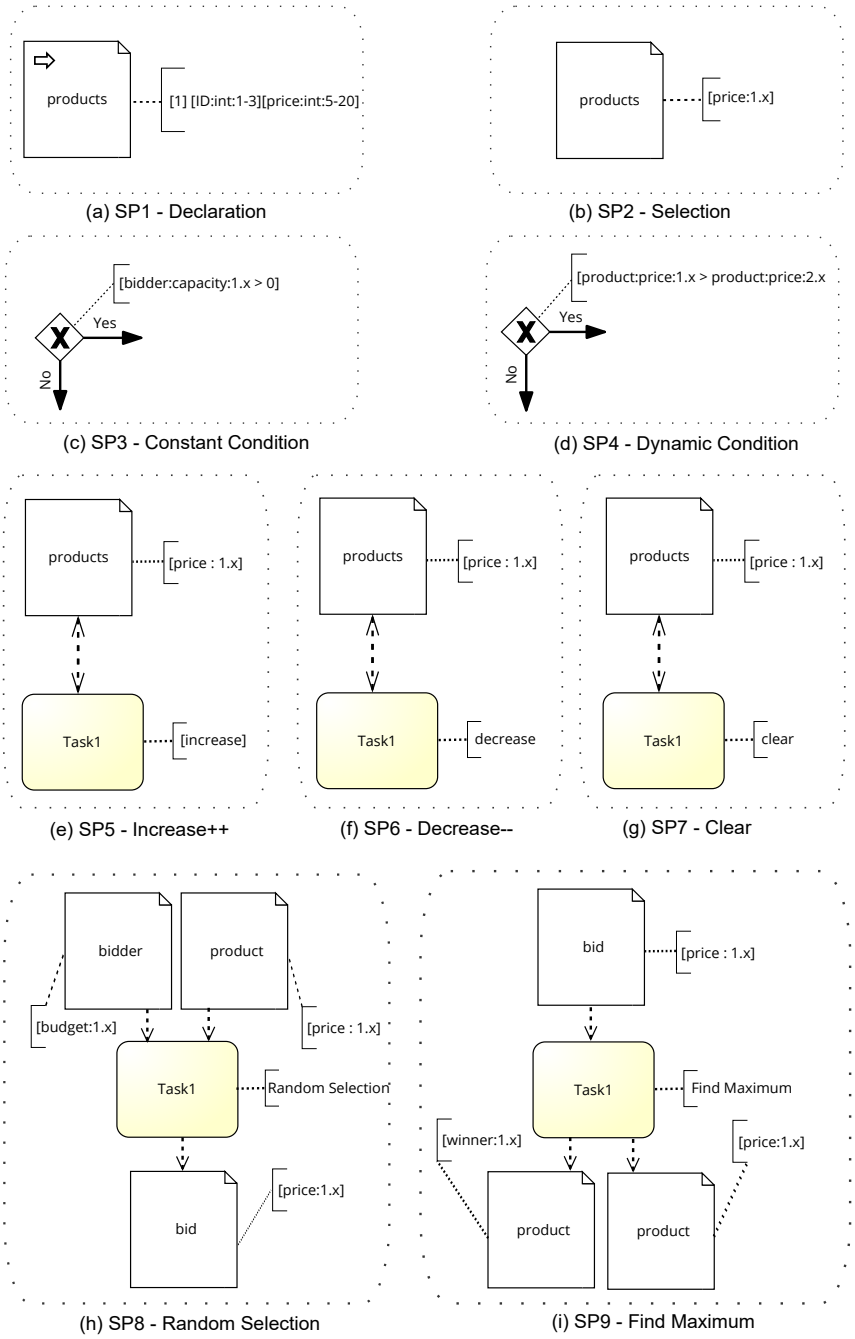


Figure 3: Examples of the BPMN enhancement with specifications introduced in Table 1

The complexity of our transformation algorithm is linear to the number of enhanced elements in BPMN: $O(n)$, where n is the number of annotations in our model. In the following, we explain the transformation steps in detail.

Mapping. A process model contains the information of the domain of a *data object*, i.e., its allowed values, in an annotation *EL Selection*, see Subsection 4.1. The transformation represents each *data value* (*d_value*) by two places in the Petri Net: $p.\neg d\_value$ and $p.d\_value$, for which the sum of its tokens always is 1 ($m(p.\neg d\_value + m(p.d\_value) = 1$).

Example 10. *Figure 4 shows an example of mapping data values to corresponding places. There, all $p.\neg d\_value$ places are marked which means that there is no winner of products in this process state.*

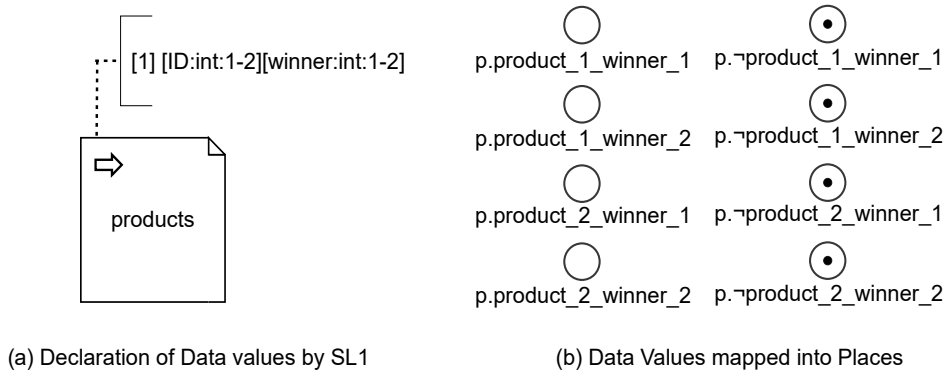
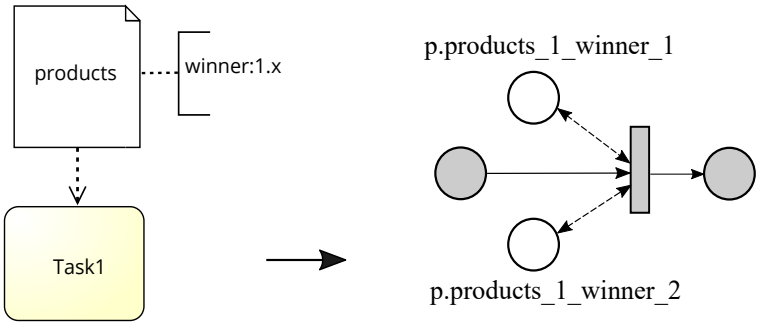


Figure 4: Mapping the Data Values to Petri Net Places

Unfolding. To transform the BPMN elements that involve usage of *data value*s, we extend an already mapped *control flow* Petri Net by adding new subnets. We create these subnets by following specific transformation rules. We define these rules for each SE in Section 4.1. As the subnets are independent, we can apply the unfolding rules in any order. These rules use the new places created in the **mapping** phase. In the following, we explain the unfolding rules for *Selection*, *Dynamic Condition* and *Increase ++*.

Unfolding Rule: Selection. In the already mapped control flow, we transform a task into a transition with one input and one output place. For each *data value* declared in *SP1*, we link its corresponding *p.d_value* to the transition *t1*, with a bi-directional arc. It ensures that the *data value* keeps the same value when firing the task. Figure 5 shows the transformation of *data value*s that are data needs of *Task1*.

Unfolding Rule: Constant Condition. In current approaches like [14], the condition of a *gateway* is not modeled in the Petri Net, although it has an effect on the execution semantics. To address this issue, for each *data value* declared in *SP1*, we create a transition (*t1*) and connect it to its *p.d_value* place with a bidirectional arc. Only one of these transitions can fire at a moment, depending on the current value. We separate the *true*- and *false*-paths with the transitions *t.gw1.y1* and *t.gw1.y2*. Depending on the current value, meeting the condition or not, transitions *t.gw1.y1* or *t.gw1.y2* fire respectively (Figure 6).



(a) Selection of data values by EL1 (b) Unfolding rules for EL1

Figure 5: Reading Data Values by a Task

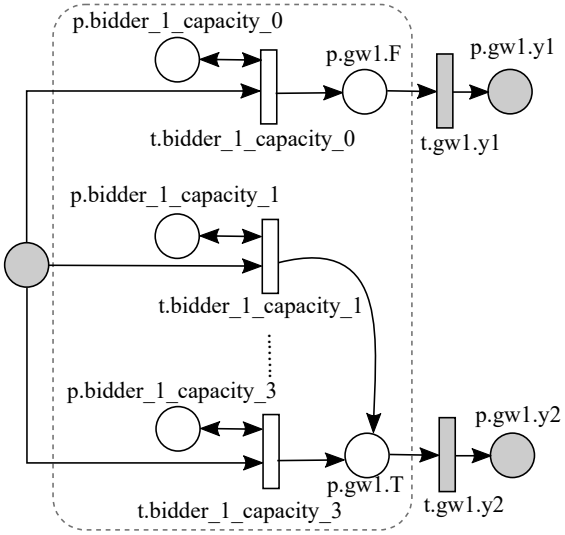


Figure 6: Transformation: Condition Type 1

Unfolding Rule: Dynamic Condition. In this condition both parameters are *data value*s. Therefore, we add their correspondig *p.d_value* places to the subnet. For modeling the condition itself, we create a transition for each combination of *p.d_value* places as represented in Figure 7. Each of the transitions fires if both of their incoming places are marked, i.e., they carry the current value. We take the same steps as for *Constant Condition* to complete the subnet.

Unfolding Rule: Increase++. To unfold activities increasing *data-value*s, we create a separated transition for each certain *data-value* defined for the *data need*s of the *task*. Each of these transitions has a corresponding *p.d_value* place as *incoming place*, and $p.\neg d\_value$ as *outgoing place*. At the same time, we add the *p.d_value* place with $value + 1$ as an *outgoing place*, and $p.\neg d\_value$ place

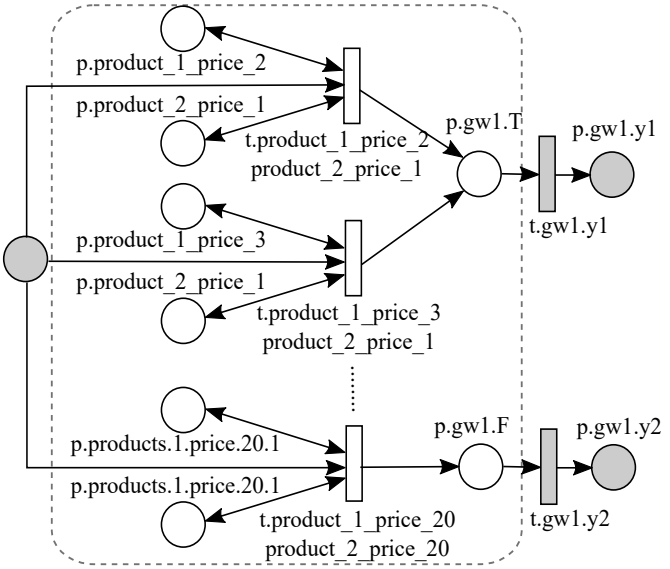


Figure 7: Transformation: Condition Type 2

with $(value + 1)$ as an *incoming place*. We connect this subnet to the subnet which results from the transformation of a task as shown in Figure 8.

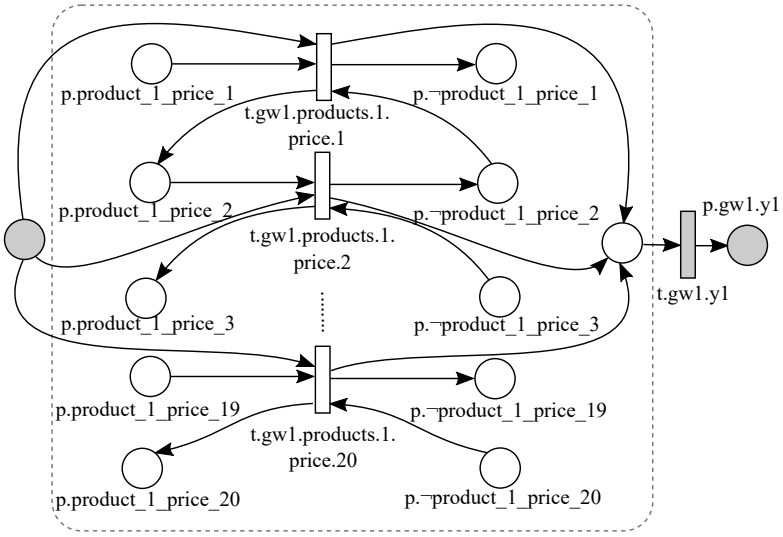


Figure 8: Transformation: Increase Task

Unfolding Rule: Decrease–. The transformation of *task*s that decrease a data value is similar to the unfolding rules of *SP5 - Increase++* but with reversed values, see Figure 9.

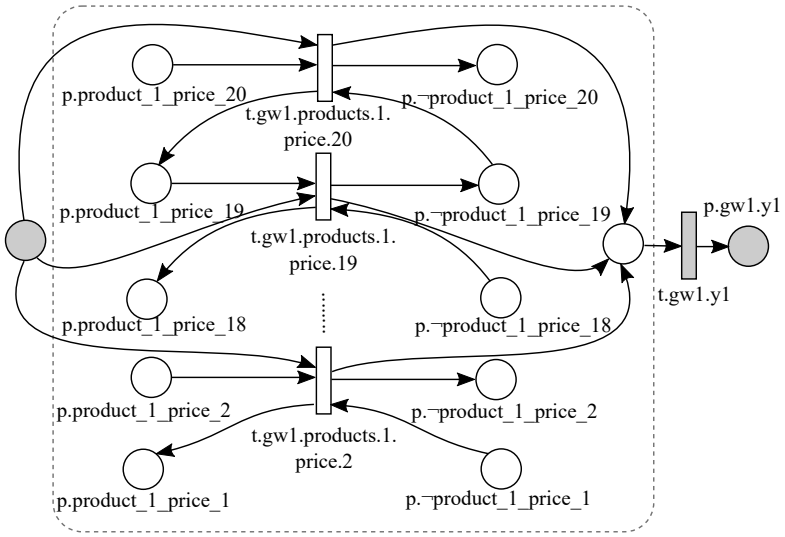


Figure 9: Transformation: Decrease Task

Unfolding Rule: Clear. For each *data value* which is defined as SL, we create two transitions: transition *t1*, which has the place *p.d_value* as its *incoming place*, and transition *t2* which has the *p.¬d_value* place as its *incoming place*. Only one of these transition can fire, as only one of these *incoming place*s could be marked. The outgoing place of both transitions is *p.¬d_value* place, which means the corresponding *attribute* does not have any value. We add the subnet to the normal transformation of task as shown in Figure 10.

Unfolding Rule: Random Selection. For each *data value* specified by SL1 we create a transition. Each *transition* has *p.¬d_value* as *incoming place* and *p.d_value* as *outgoing place*. It means these transitions will compete to use a token and select a *data-value* as non-deterministic choice. In this model, there are outgoing arcs as many as the number of *data-value*s given, which discerns from the normal transformation of an OR-split gateway.

Unfolding Rule: Find Maximum. To find the maximum *bid price* possible in an auction, it is for example possible that two bidders have the same *bid price* for one *product*. So, to find the maximum value, we have the assumption that the *data-value*s must not be unique. To create the subnet, first we create an inner subnet for each single value which has been defined by *SP1*, see Figure 4.2. In the inner subnet, we assign a *transition* to each *data-value* separately. The *p.d_value* place is connected to the transition by the bidirectional arc. All *p.¬d_value* places, which refer to the same value, will be connected to a transition (*t.t1.findMax.bidprice.x.1*) with a bidirectional arc. In case that none of the *data-value*s carries the respective value, this transition fires, i.e., it leads to the execution of the next inner subnet. More detailed, the inner subnets are hierarchically connected to each other, starting from the maximum value. For the sake of convenience in our use case, we write the highest *bid price* and the *winner* directly to the corresponding places as shown in Figure 4.2.

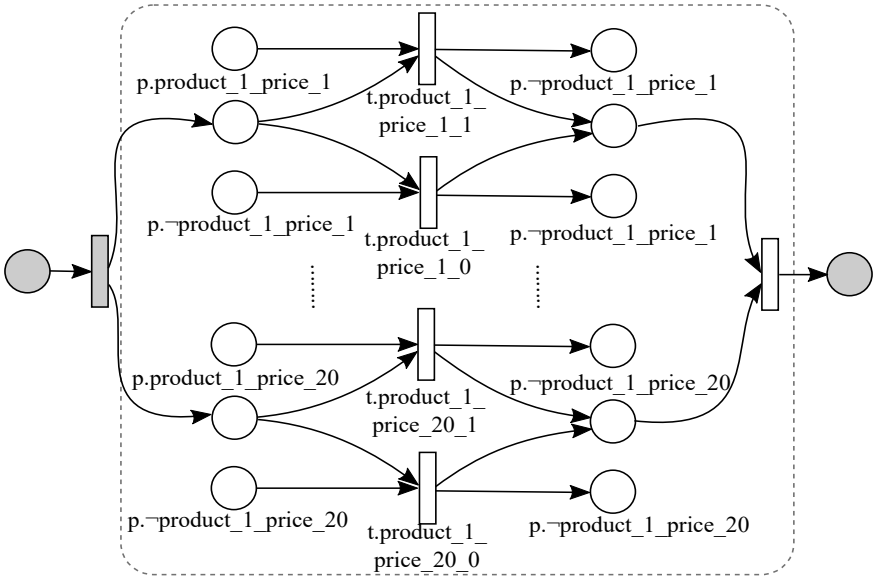


Figure 10: Transformation: Clear Task

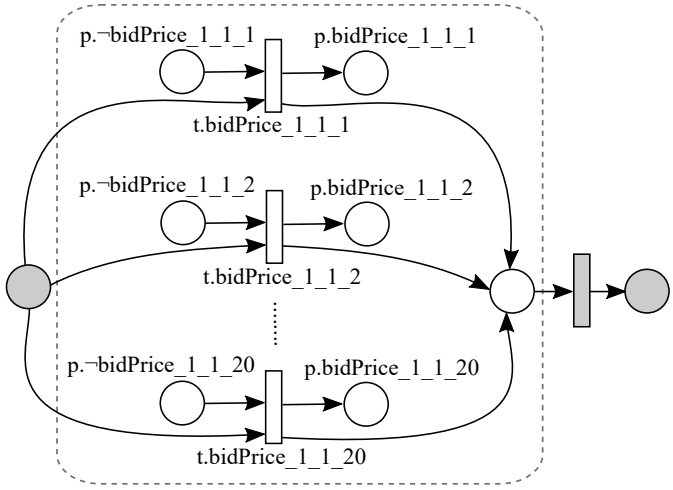


Figure 11: Tasks with Specific Function: Bid Submission

4.3 Definition of Properties

In the following, we explain the data-dependent properties specified in Computation Tree Logic (CTL) [9] that we will verify on our model. In our approach, the atomic formulas in CTL are states of the Petri Net that contain places representing *data value*s used in the process model. Each *data value* defined in the first step with *SP1* is represented by two places in the Petri Net:

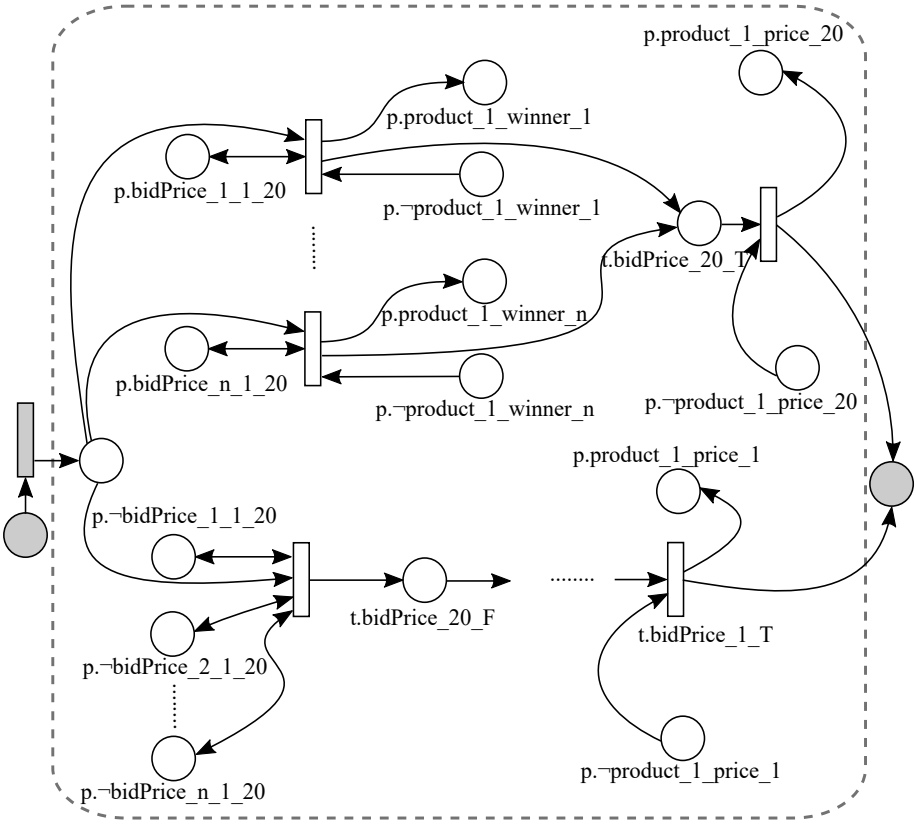


Figure 12: Tasks with Specific Function: Winner Determination

$p.\neg d\_value$ and $p.d\_value$. These places enable verification of properties which are dependent on the *data value*s. The following examples show how to formulate such properties in CTL:

Example 1: The *product* with *ID = 2* always belongs to the bidder with the highest *budget* (bidder with ID = 2).

$$EF(\neg p.product.2.winner.2 \wedge p.end)$$

Example 2: All the *Bidder*s can win at least one *product*.

$$EFAG((p.product.1.winner.1 \vee p.product.2.winner.1 \vee p.product.3.winner.1) \\ \wedge(p.product.1.winner.2 \vee p.product.2.winner.2 \vee p.product.3.winner.2))$$

Example 3: *Bidder*s with *capacity points = 0* can *bid* for any *product*.

$$EF((p.bidder.1.capacity.0 \wedge (p.bid.bidder.1.product.1 \vee p.bid.bidder.1.product.2 \vee \\ p.bid.bidder.1.product.3)) \vee (p.bidder.2.capacity.0 \wedge (p.bid.bidder.2.product.1 \vee$$

$$p.bid.bidder.2.product.2 \vee p.bid.bidder.2.product.3)))$$

Example 4: Is it possible to sell the *product* with *ID = 1* for the *price* of 2.

$$\texttt{EF}\ (p.product.1.price.2 \wedge e.end)$$

By verifying the property of the last example, we will find the lowest *auctioneer's revenue*(R_{lowest}). To do this, first we find the lowest final *price* of *products*, starting with the *reserve price*s as shown in Example 4. For this, Place *e.end* is the result of the transformation of an *end event* in the BPMN model; i.e., it triggers the end of the process. In case the property is not satisfied, we run iteratively the model checker to verify a property with an increased *price* until the system finds a state that fulfills the property.

In the next step, we find the lowest combination of prices (R_{lowest}) in a similar way, starting from the lowest final *price* of *product*s. Assume that the lowest *price* for the *product* with *ID=1, 2, and 3* is 4, 6, and 7 respectively. Then, we have to very the following formula in the first step:

$$\texttt{EF}\ (p.product.1.price.4 \wedge p.product.2.price.6 \wedge p.product.3.price.7 \wedge e.end)$$

In case that the model can not satisfy this formula, we will take other combinations with increased *price*s of *product*s. For this, we have not to look at all *prices* in combinations, but only on those which are in the interval between a maximum and a minimum possible *price*. The maximum *price* is known by the maximum *budget* of *bidder*s for a certain *product*, and the minimum is a result of a verification step. With adequate search strategies combinations to be verified can be further restricted. This is put to future work. By verifying the combination of *price*s, we can find the possible lowest *auctioneer's revenue*.

Besides it, we can also find the minimum *bidder's profit*(P_{lowest}) associated with R_{lowest}. To this end, we check if the *bidder*s with the lowest *budget* can be the *winner*s. Assume *bidder*s with *ID = 1, 2, and 1* have the lowest *budget*s for *product*s with *ID =1,2 and 3*, respectively. The first formula to check then is:

$$\texttt{EF}\ ((p.product.1.price.4 \wedge p.product.2.price.6 \wedge p.product.3.price.7) \wedge (p.product.1.winner.1 \wedge p.product.2.winner.2 \wedge p.product.3.winner.1) \wedge e.end)$$

If the *model checker* can not find a state, i.e., any execution path to fulfill this property, we change the *winner* of products receiving a new formula. Then, we verify the new formula. This is done until the formula is satisfied. To find the lowest *bidder's profit* in each step, we first set the *winner* to the one with the *lowest budget*.

With another measure, we are able to find the lowest market *efficiency* (E_{lowest}) while the *auctioneer's revenue* has its lowest possible value. These measures have been used in auction literature [8] to compare different auction formats. As definition of efficiency we use a measure described in [8]. This definition estimates how *Surplus* with the worst allocation changes compared to it with a random allocation:

$$E_{lowest} = \frac{S_{lowest} - S_{random}}{S_{optimal} - S_{random}} \times 100\ percent$$

Here, the optimal allocation can be computed by giving the *product*s to *bidder*s with the highest *budget* and $S_{lowest} = R_{lowest} + P_{lowest}$.

4.4 Discussion

Features Supported. We developed a new method to verify data-dependent properties in a SMR auction. In this, we not only support data states which is the state of the data contained in a *data object*, but also every single *data value* of it. For this, we support a *data-value* representation with the types *integer*, *boolean*, and *enumeration*. Manipulation of *data value*s is also possible by several functions such as *increase*, *clear*, etc. It allows to verify processes that require rules specified with respect to *data value*s as are spectrum auctions. Further, we have chosen Petri Nets as the target of our transformation, for which efficient static analysis techniques are available. Beside this, we have formalized our properties in the temporal logic CTL, so that we can verify a range of different expressions. Additionally, by employing a model checker, we are able to generate counter-examples that explain unexpected results automatically. It helps model designers to identify the cause of a problem and fix it.

Limitations. Considering every possible state with *data-value*s can result in a huge state space. This problem is well-known as state-space explosion [11]. It can hinder the verification and lead to unacceptable runtimes. Besides, manipulation functions provided in this paper would not be sufficient to cover all processes in different domains, for which we require to provide more general functions like the aggregation ones.

Future Work. To address the above issues, we are planning to design a new approach to reduce the size of state space. This optimization method can apply as a pre-processing step before verification of data-centered workflows. In future work, we also aim to support more general functions for *data-value* manipulations to cover a wider range of data-centered workflows.

5 Implementation

We have implemented a framework to verify data-dependent properties in a SMR auction. Figure 13 visualizes the steps accomplished by our implementation. The framework allows to select an enhanced BPMN model as an input for transforming data-centered BPMN models into Petri Nets. As a use case, we have modeled SMR auctions in BPMN and enhanced it with the Specification Expressions concerning used *data value*s as explained in Section 4.1. After loading the BPMN model into our framework, a transformation of control flow into Petri Nets takes place by applying rules of [14]. We extend the already generated Petri Nets to include representation and manipulation of *data value*s as introduced in Section 4.2. In the next step, the generated Petri Nets are verified automatically against the data-dependent properties expressed as CTL formulas. To do this, we run a CTL model checker in our framework, namely LoLA [26], once or several times for verifying the properties as described in Section 4.3. Since LoLA is quite easy to run in shell scripts, it is quite well manageable for repeated application.
LoLA explores automatically the state space to check a property. For each verified property, our tool provides the execution trace to the user. As the transitions of the trace have meaningful labels according to their function in the BPMN model, it is easy to understand the counter-example and track the execution run.

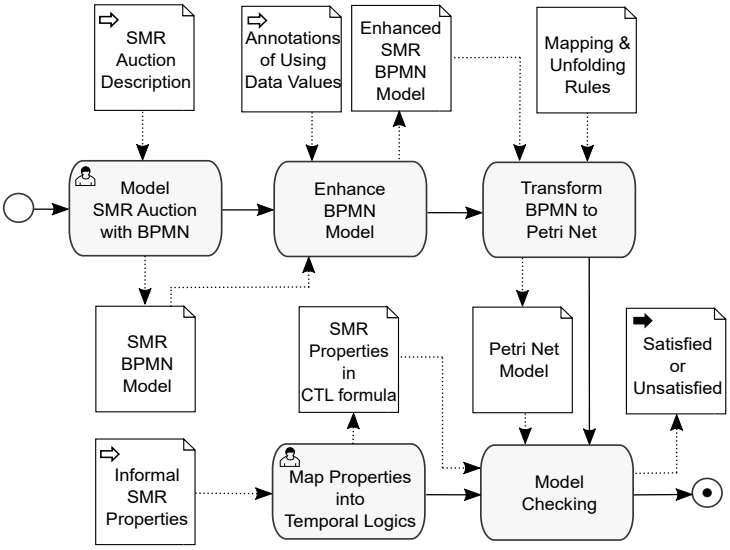


Figure 13: Overview of Our Approach on Verification of Data-Aware Process Models

6 Evaluation

To evaluate our framework we study a SMR auction model including *eligibility rule* and *capacity rule* with two bidders, see Section 2.4 for details. In the first part of our evaluation, we the lowest *auctioneer's revenue*, *bidder's profit* and *efficiency of auction* (for details see Section 4.3) as the essential measures in auction literature [8]. In the second part of our evaluation, we analyze the runtime for transforming enhanced BPMN models to Petri Nets and compare it with the original BPMN models. We also study the runtime for verifying data-dependent properties in a SMR auction.

To do our evaluation, we have considered two different settings of a SMR auction: (1) two *bidder*s compete to win two different *product*s, labeling *Process 1*, and (2) two *bidder*s are in demand for three *product*s, labeling *Process 2*. In both processes, we have assigned a random *budget* to each *bidder* for a certain *product* in a range of $[2-10]$, similar to [8]. Table 2 shows these values for all *bidder*s and *product*s. We also have defined a *reserve price* of 1 for all *product*s. With it, all *bidder*s can afford all the *product*s at the beginning of auctions.

Bidder ID	Product ID = 1	Product ID = 2	Product ID = 3
1	8	9	6
2	5	5	6

Table 2: The reserve price and budget of bidders for different products

For the sake of convenience to model a SMR auction with BPMN, we only have modeled the parts of SMR auction which are related to verifying the properties relevant in our setting. This means that we did not care about parts of a SMR auction which do not have an effect on the

verification of our properties, e.g., bidder's registration to the auction. Table 3 gives an overview on the enhanced Process Models 1 and 2. As we only allow annotating the basic BPMN elements and not, e. g., *iterative sub-processes*, we have to repeat parts of the SMR auction model to represent the increasing of the number of *bidder*s and *product*s. This leads to a slightly higher number of BPMN elements of the model by a constant factor for each iteration. That we handle only three bidders and three products is not a fundamental restriction. For example, for adding a new *bidder* the number of *tasks* and *gateways* of the BPMN model increases by 15 and 27, respectively. Figure 14 shows a part of the enhanced BPMN model for winner determination and capacity rule.

Process	Tasks	Gateways	Annotation	Data Object	Data Association	Data Value
1	50	55	145	9	94	80
2	69	81	206	9	124	119

Table 3: Number of enriched BPMN elements for Processes 1 and 2

We have used these enhanced BPMN models as the input of our tool for transformation into Petri Nets. The generated Petri Nets contain new places resulting of the new representation and manipulation of *data value*s, which allow to verify *data value* dependent properties. By this, we are able to detect the lowest *auctioneer's revenue*.

To do this, first we have detected the lowest possible final *price* of each individual *product* by doing iterative verification, starting with their *reserve price*s as follow:

$$\texttt{EF}\ (p.product.1.price.1 \wedge p.product.2.price.1 \wedge p.product.3.price.1 \wedge e.end)$$

To detect the lowest possible final *price* for all products, we perform the verification for combinations of *price*s as follows:

$$\texttt{EF}\ (p.product.1.price.5 \wedge p.product.2.price.4 \wedge p.product.3.price.6 \wedge e.end)$$

To handle all combinations of *price*s in order to detect the lowest possible *revenue* requires at most to verify 336 respectively 56 properties for Process 1 and 2. However, if we restrict the set of combinations to those *prices* which are bounded by a minimum and maximum value (see Section 4.3), we only have to verify 50 respectively 20 properties for Process 1 and 2. Table 4 and 5 represent the results of the verification runs detecting the values of auction measures of Processes 1 and 2, respectively. In Process 1, the auctioneer will have the lowest *revenue* by assigning *product 1* and *2* to *bidder 1* for the *price* of 5. Having the lowest *auctioneer's revenue*, the *efficiency of the auction* and *bidder's profit* are 100% and 7, respectively. In Process 2, *Product 1* has been sold to *Bidder 2* for the *price* of 5, although both *bidder*s would have more budget to buy a product. With it, the *auctioneer's revenue* is 15 which leads to an *efficiency* of 82.60% in this setting.

Product ID	Winner ID	Final Price	Revenue	Profit	Efficiency
1	1	5	10	7	100%
2	1	5			

Table 4: Verification of properties with 2 bidders and 2 products

Having these results, an auction expert becomes aware of the bad consequences of an auction setting, before performing the auction itself. They can assess the effects of certain modifications

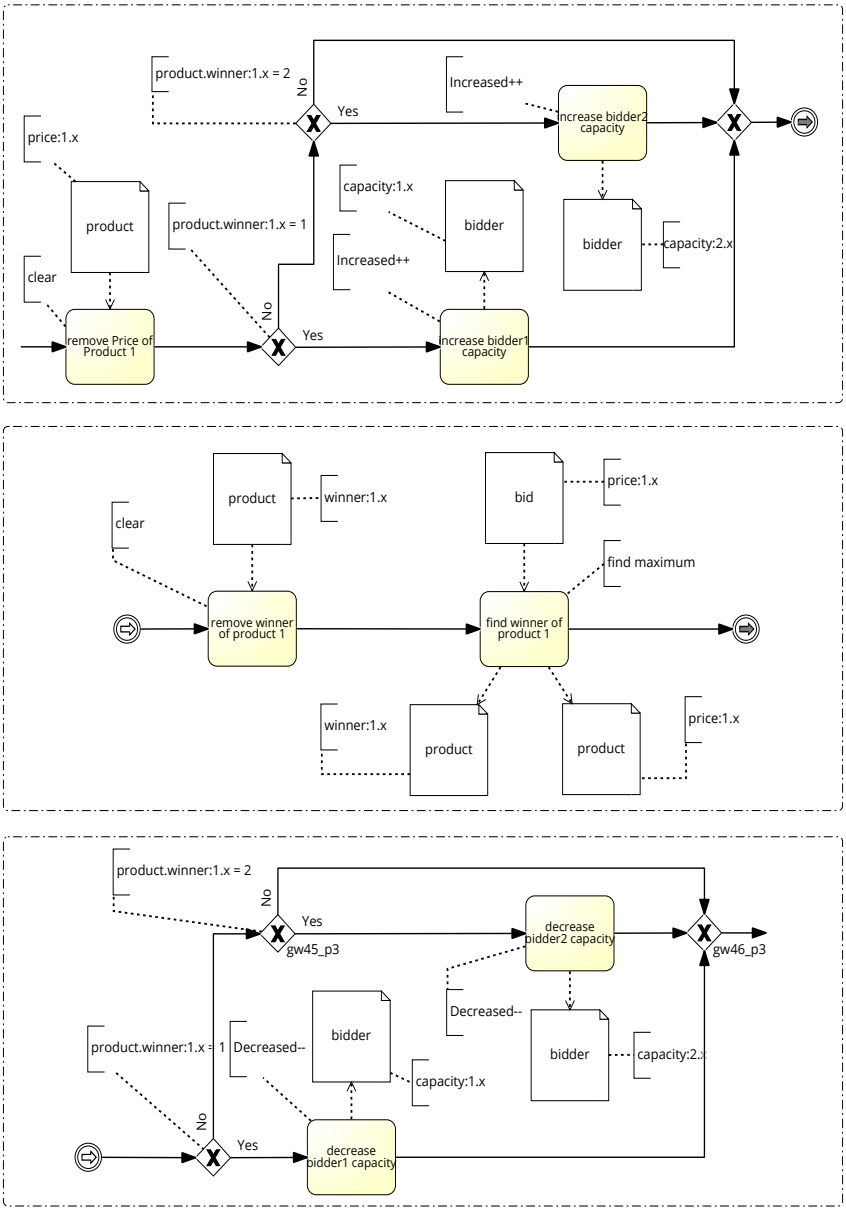


Figure 14: BPMN enhancement of Process 1 for winner determination and capacity rule regarding product 1

of the auction parameters on the final auction result. Such modifications are, e.g., changes of the *reserve price*s or *capacity point*s of bidders. This can be done simply by changing some *data*

Product ID	Winner ID	Final Price	Revenue	Profit	Efficiency
1	1	5			
2	2	4	15	4	82.60%
3	1	6			

Table 5: Verification of properties with 2 bidders and 3 products

*value*s in the BPMN model of the SMR auction, which is quite user-friendly even for non-experts in modeling.

To evaluate the runtime of the entire verification process, we have studied separately the runtime for verification of properties and the transformation of enhanced BPMN into Petri Nets. To this end, we have verified 50 respectively 20 properties for Process 1 and 2. Verification of a single property took less than 4 minutes in the worst case, by a common computer. Table 6 gives an overview of 17 different properties which we have verified with our tool with the results and runtime. We have transformed the entire Processes 1 and 2 into Petri Nets each in less than 1 second. The model checking time relates to the size of the Petri Net which depends also on the usages of data values in the process, i.e., resulting from unfolding of subnets during the transformation step. This may lead to state space explosion for realistic scenarios. To meet this challenge, we plan to work on reducing the size of the Petri Net in order to optimize the model checking, in future work. To do this, we want to apply a relevance-based reduction of the process, see also our previous for optimization of data-flow verifications [22].

Property	**Result**	**Time**
EF $(p.product.1.price.8 \wedge p.product.2.price.8 \wedge p.product.3.price.6 \wedge e.end)$	YES	2:18
EF $(p.product.1.price.8 \wedge p.product.2.price.8 \wedge p.product.3.price.5 \wedge e.end)$	YES	2:49
EF $(p.product.1.price.8 \wedge p.product.2.price.8 \wedge p.product.3.price.4 \wedge e.end)$	YES	3:34
EF $(p.product.1.price.8 \wedge p.product.2.price.8 \wedge p.product.3.price.3 \wedge e.end)$	NO	2:48
EF $(p.product.1.price.8 \wedge p.product.2.price.7 \wedge p.product.3.price.7 \wedge e.end)$	NO	2:52
EF $(p.product.1.price.8 \wedge p.product.2.price.7 \wedge p.product.3.price.6 \wedge e.end)$	YES	2:29
EF $(p.product.1.price.8 \wedge p.product.2.price.7 \wedge p.product.3.price.5 \wedge e.end)$	YES	3:36
EF $(p.product.1.price.8 \wedge p.product.2.price.7 \wedge p.product.3.price.4 \wedge e.end)$	YES	3:38
EF $(p.product.1.price.8 \wedge p.product.2.price.7 \wedge p.product.3.price.3 \wedge e.end)$	NO	2:56
EF $(p.product.1.price.8 \wedge p.product.2.price.6 \wedge p.product.3.price.7 \wedge e.end)$	NO	2:26
EF $(p.product.1.price.8 \wedge p.product.2.price.6 \wedge p.product.3.price.6 \wedge e.end)$	YES	2:22
EF $(p.product.1.price.8 \wedge p.product.2.price.6 \wedge p.product.3.price.5 \wedge e.end)$	YES	3:00
EF $(p.product.1.price.8 \wedge p.product.2.price.6 \wedge p.product.3.price.4 \wedge e.end)$	YES	3:38
EF $(p.product.1.price.8 \wedge p.product.2.price.6 \wedge p.product.3.price.3 \wedge e.end)$	NO	3:38
EF $(p.product.1.price.8 \wedge p.product.2.price.5 \wedge p.product.3.price.7 \wedge e.end)$	NO	3:01
EF $(p.product.1.price.8 \wedge p.product.2.price.5 \wedge p.product.3.price.6 \wedge e.end)$	YES	2:29
EF $(p.product.1.price.8 \wedge p.product.2.price.5 \wedge p.product.3.price.4 \wedge e.end)$	NO	0:42

Table 6: Verification times of properties

Our evaluation shows that our framework can transform a BPMN model of the SMR auction with more than 400 elements into Petri Nets in less than 1 second. It enables to verify data-dependent properties in a data-aware process model by employing a model checker.

Process	Enhancement	Places	Transitions	Transformation Time
1	yes	669	694	0.3 sec
1	no	181	182	0.1 sec
2	yes	998	1031	0.4 sec
2	no	260	254	0.1 sec

Table 7: Overview of Petri Nets generated for Processes 1 and 2

7 Related Works

There is a growing stream of research to address workflow verification considering the data perspective, see [2] for an overview. [4] is one of the first Petri Net based approaches to detect and resolve modeling errors which occur in presence of data. They transformed BPMN 1.2 to Petri Nets by mapping options for data objects. Authors in this approach only considered the states of *data object*s, and not their values. Furthermore, they only deal with *equality* of states for mapping data-dependent *gateway*s. Another approach to detect data anomalies based on Petri Nets is represented in [27]. They unfolded the execution semantics of BPMN process models regarding data and formalized data-flow errors in a set of anti-patterns. In this, authors only considered the data-flow errors and do not provide a method for verification of data-dependent properties. [5] formalized data-aware compliance rules by Linear Temporal Logic with Past Operators (PLTL) and employed a model checker to verify properties. In case of detecting an error, they apply Temporal Logic Querying (TLQ) to explain violation. However, they only support data states in their process models and data-aware compliance rules, although *data value*s play a significant rule for compliance checking. So, it is not sufficient to use this approach for verification of properties based on *data value*s. Another approach to deal with conditions on numeric data is represented in [16]. They verify temporal properties of a business process by employing bounded model checking in answer set programming. The article addresses the verification of properties with *data value*s, however the challenge of *data value* manipulation still remains.
[6], [3], and [25] introduce a method based on Logic Programming. In [25] authors propose a framework based on Constraint Logic Programming (CLP) for representing and reasoning on process models by modeling them as a state transition system. They model representation and manipulation of *data value*s based on arithmetic constraints. However, in case they detect an error in the process model, they can not provide a counterexample to fix it. Additionally, they do not have any restriction on their workflow and data which makes the verification undecidable in general. [6] extends the result of [3] by introducing three new predicates (*final*, *hasSuccessor*, and *findPath*) based on the *next* predicate to detect more flows on data-aware business process models. They translate each business process element (task, event, gateway) into a logic predicate and control the transitions between them by *sequence flow*s. Although they represent *data value*s in the process model, they do not allow to use them in their predefined queries. Another drawback is that they can not provide counterexamples which are based on intermediate states.
Further approaches exist using so-called artifact systems for verification of business processes. In [24], authors propose a restricted class of artifact systems and LTL-FO properties to make the satisfaction decidable. In that paper, artifacts carry values of an attribute which can be updated by external services. These services work with an underlying database. However, the results show their approach fails in presence of data dependency (integrity constraints on the database). They do not support arithmetic operations which play a significant rule in some real life applications such

as spectrum auctions. [13] alleviate these problems by extending the artifact model. However, they impose syntactic restrictions on process models and properties which limit its applicability.

8 Conclusions

In this paper, we have developed a new method to verify data-dependent properties in a SMR auction. This approach takes representation and manipulation of *data value*s into account. To do this, we have provided Specification Expressions to enhance a BPMN model with annotations of used *data value*s. To take the advantages of an existing model checker, we have transformed the enhanced BPMN model into Petri Nets by applying new mapping and unfolding rules. Then, we have verified data-dependent properties of an SMR auction expressed as CTL formulas to detect high-risk executions. Our implementation shows that our tool can verify data-dependent properties to calculate the worst possible values for important measures of a SMR auction.

References

[1] Lawrence M. Ausubel, Peter Cramton, and Paul Milgrom. The clock-proxy auction: A practical combinatorial auction design. *Stanford Institute for Economic Policy Research*, 03(34):1–27, 2006.

[2] Raffaele Dell Aversana. Verification of Data Aware Business Process Models : A Methodological Survey of Research Results and Challenges. pages 393–397, 2015.

[3] Raffaele Dell Aversana. Data Aware Business Process Models : A Framework for the Analysis and Verification of Properties. pages 75–82, 2016.

[4] Ahmed Awad, Gero Decker, and Niels Lohmann. Diagnosing and repairing data anomalies in process models. *Lecture Notes in Business Information Processing*, 43 LNBIP(i):5–16, 2010.

[5] Ahmed Awad, Matthias Weidlich, and Mathias Weske. Specification, verification and explanation of violation for data aware compliance rules. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5900 LNCS:500–515, 2009.

[6] Raffaele Dell Aversana B. A Decision Framework for Understanding Data-Aware Business Process Models. 2018.

[7] Martin Bichler, Jacob Goeree, Stefan Mayer, and Pasha Shabalin. Spectrum auction design: Simple auctions for complex sales. *Telecommunications Policy*, 38(7):613–622, 2014.

[8] Christoph Brunner, Jacob K. Goeree, Charles A. Holt, and John O. Ledyard. An experimental test of flexible combinatorial spectrum auction formats. *American Economic Journal: Microeconomics*, 2(1):39–57, 2010.

[9] E M Clarke. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. 8(2):244–263, 1986.

[10] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[11] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[12] Peter Cramton. Spectrum auctions. *Handbook of Telecommunications Economics*, pages 605–639, 2002.

[13] Elio Damaggio. Artifact Systems with Data Dependencies and Arithmetic. (May 2014), 2011.

[14] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, 2008.

[15] Dirk Engelmann and Veronika Grimm. Bidding behaviour in multi-unit auctions - An experimental investigation. *Economic Journal*, 119(537):855–882, 2009.

[16] Laura Giordano, Alberto Martelli, and Matteo Spiotta. Business process verification with constraint temporal answer set programming. 13(July):641–655, 2013.

[17] Thomas W Hazlett, Roberto E Muñoz, and Diego B Avanzini. What Really Matters in Spectrum Allocation Design. *Northwestern Journal of Technology and Intellectual Property*, 10(3):93–124, 2012.

[18] Anthony M. Kwasnica and Katerina Sherstyuk. Multi-Unit Auctions. Working Papers 201301, University of Hawaii at Manoa, Department of Economics, January 2013.

[19] Niels Lohmann, Eric Verbeek, and Remco Dijkman. Petri net transformations for business processes - A survey. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5460 LNCS:46–63, 2009.

[20] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media, 2012.

[21] Paul Milgrom. *Putting Auction Theory to Work*. Cambridge University Press, Cambridge, 2004.

[22] Jutta Mülle, Christine Tex, and Klemens Böhm. A Practical Data-Flow Verification Scheme for Business Processes. *Information Systems*, 81:136–151, March 2019.

[23] Object Management Group. Business Process Model and Notation, V2.0. http://www.omg.org/spec/BPMN/2.0/PDF, 2011.

[24] Fabio Patrizi and U C San Diego. Automatic Verification of Data-Centric Business Processes. 2009.

[25] Maurizio Proietti and Fabrizio Smith. Reasoning on data-aware business processes with constraint logic. *CEUR Workshop Proceedings*, 1293:60–75, 2014.

[26] Karsten Schmidt. LoLA A Low Level Analyser. *Application and Theory of Petri Nets 2000*, 1825:465–474–474, 2000.

[27] Silvia Von Stackelberg, Susanne Putze, Jutta Mülle, and Klemens Böhm. Detecting Data-Flow Errors in BPMN 2 . 0. *Open Journal of Information Systems (OJIS)*, 1(2):1–19, 2014.

[28] W. M. P. van der Aalst. The Application of Petri Nets To Workflow Management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998.

[29] Wil MP Van Der Aalst and Arthur HM Ter Hofstede. Yawl: yet another workflow language. *Information systems*, 30(4):245–275, 2005.

[30] Elmar Wolfstetter. *The Swiss UMTS Spectrum Auction Flop*. Humboldt-Universität zu Berlin, Wirtschaftswissenschaftliche Fakultät, 2001.