A simulator for high-level Petri nets: An ePNK application

Ekkart Kindler[1] and Mindaugas Laganeckas[2]

[1] DTU Compute
Technical University of Denmark
DK-2800 Lyngby
DENMARK
ekki@dtu.dk
[2] Alumnus of DTU Informatics
(now DTU Compute)
mindaugas.laganeckas@gmail.com

Abstract. The ePNK is a platform for Petri net tools based on the PNML transfer format. One of its important features is its extensibility, which allows developers to plug in new Petri net types and new functions and applications for different kinds of Petri nets.
The basic version of the ePNK provides an editor for high-level Petri nets, but no analysis or simulation functionality. In this paper, we present a simulator for high-level Petri nets, which supports most of the built-in operators of ISO/IEC 15909-2. As an additional feature, this simulator allows the simulation of so-called network algorithms.
In this paper, we briefly show how to use this simulator from the end user's point of view. Moreover, we discuss some of the concepts underlying this simulator and its implementation.

Keywords: ePNK, high-level Petri net, high-level Petri net schema, network algorithm, simulator, PNML, ISO/IEC 15909.

1 Introduction

The ePNK [11, 12] is a platform for Petri net tools based on the PNML transfer format and, in particular, on the underlying PNML core model [6]. One of the important features of the ePNK is its extensibility: it is easy to plug in new Petri net types and new functions and applications. The basic version of the ePNK supports high-level Petri nets: there is a graphical editor which can load, edit, and save high-level nets[3] and check their syntactical correctness. But, there is no real functionality for high-level nets. In particular, there is no simulator for high-level nets yet.

[3] In ISO/IEC 15909-2:2011, high-level nets are called High-level Petri Net Graphs (HLPNGs).

In this paper, we present an extension to the ePNK that allows simulating high-level nets. This simulator was developed as part of a master's project [15]. In addition to simulating normal high-level nets, this simulator allows the simulation of network algorithms [14], which are a special kind of high-level net schemas that can be instantiated with a concrete network on which the network algorithm should run. In this paper, we briefly discuss high-level nets and network algorithms and how to use this simulator from an end user's point of view.

One of the key issues of simulators for high-level Petri nets is the way of computing the firing modes in which a transition is enabled. In this paper, we discuss a data-driven way of computing possible variable bindings, which in particular allows us to extend high-level nets with new operators without changing the binding algorithm itself. In some cases, our simulator automatically finds firing modes of enabled transitions where other tools like CPN Tools [1] do not.

Concerning efficiency, however, our simulator cannot compete with the simulators for high-level nets such as CPN Tools [5, 19], PROD [23], and Maria [16]. The effort for developing these simulators amounts to several person years[4]. By contrast, the simulator presented here was developed by a single student in half a year [15] – including the integration of the Petri net simulation with a 3D visualisation, similar to the one of PNVis [13] which we do not discussed here. Even though our simulator, is not the most efficient simulator, it makes an important contribution: it is the first high-level net simulator that directly simulates HLPNGs as of ISO/IEC 15909-2. For simple nets, it will be a nice additional feature on top of a mere graphical editor for HLPNGs. In the long run, the flexible and extensible architecture of our simulator might prove the stepping stone for developing a more efficient simulator for HLPNGs in the ePNK – developed and extended driven by the needs and requirements of people using the ePNK and its extensions.

The rest of this paper is structured as follows: In Sect. 2, we present high-level nets and network algorithms by the help of an example, and we discuss the idea of how our simulator calculates the enabled firing modes of a transition. In Sect. 3, we discuss this idea and the underlying concepts and mechanisms in more detail; moreover, some of the deviations in the implementation and some other implementation aspects are briefly discussed. Sect. 3.3 discusses the relation to existing simulation algorithms.

Though we briefly discuss the use of the high-level net simulator for the ePNK, this paper does not replace a manual. We refer to the ePNK manual [12] for details: Chap. 3 of the manual discusses the use of the ePNK in general, Sect. 3.5.2 discusses high-level nets, and Sect. 3.6.3 discusses the use of the high-level net simulator specifically.

[4] In reply to a question after the presentation of his paper [19], Mortensen estimated the overall effort for developing the efficient simulator of CPN Tools at about 5 person years.

2 Examples

In this section, we discuss two examples of high-level nets and how they can be simulated with the new ePNK simulator. The first example is a normal high-level net, the second one is a high-level net schema, which actually models a network algorithm.

2.1 Simple high-level net: Prime numbers

Figure 1 shows a simple high-level net which computes all the prime numbers up to some upper limit[5]. The net essentially models the principle of the "Sieve of Eratosthenes" starting out from all the numbers from 2 to some upper bound. In our example, the upper bound is 11, and the multiset of all the numbers is defined as constant N(). Initially, the place numbers contains all these numbers, the type of the place is POS, which is a built-in sort representing all positive numbers.

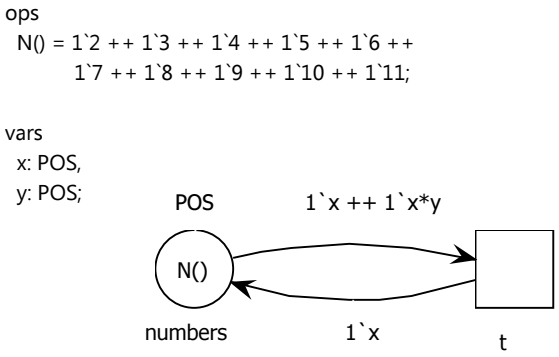

Fig. 1. A high-level net computing prime numbers

The constant N() is actually an operator without parameters, which is the reason it is defined in the operator declaration section – indicated by the keyword ops. The term 1‘2 ++ 1‘3 ++ 1‘4 ++ 1‘5 ++ 1‘6 ++ 1‘7 ++ 1‘8 ++ 1‘9 ++ 1‘10 ++ 1‘11 represents the multiset containing all numbers from 2 to 11 exactly once; the notation resembles the one of CPN Tools[6]. The number in front of the prime symbol (‘) indicates how many times the value after the prime symbol occurs in

[5] This example is included in the set of examples devised for the ePNK 1.0.0, which can be obtained from the ePNK Home Page [3].

[6] Note that ISO/IEC 15909-2:2011 does not define a concrete syntax for terms and sorts of HLPNGs. The ePNK implements its own concrete syntax, which, for simple terms, comes close to CPN Tools. But, the syntax of the ePNK is not identical to the syntax of CPN Tools. For details of the legal syntax of terms and sorts of HLPNGs in the ePNK, we refer to Sect. 3.5.2 of the ePNK manual [12].

the multiset. The second declaration of this net defines the two variables x and y, which both are positive numbers. The single transition t of this net captures the principle of the "Sieve of Eratosthenes": if there is a number x on place numbers, and there is a multiple of x on that place – represented by x*y – then this multiple is removed. Actually, the transition removes both numbers, x and its multiple x*y, but the out-going arc of the transition returns the x. Only when there is no number on place numbers that is a multiple of another number on that place, transition t is not able to fire anymore. Eventually, transition t will have removed all multiples from place numbers – leaving only prime numbers on place numbers.

For simulating a transition in a given marking, the simulator must find an *enabled firing mode* for the transition, which means finding possible values for the variables x and y so that the multiset that results from evaluating the term 1‘x ++ 1‘x*y with these values is contained in the current marking of place numbers. The values bound to the variables are called a *variable binding*. Finding candidates for such variable bindings is simple when the variable x occurs in a term like 1‘x on the top-level of the multiset term; in this case, the only possible candidates for the values of x are the numbers contained in the marking of place numbers. For the other term, 1‘x*y, finding such a variable binding is not so simple any more, but it is still possible: we know that the result of x*y must be a number that is contained in the marking of place numbers. Since we know the possible values of x already and since the multiplication operator, is *invertible in each argument*[7] the possible values of y can be derived too. Exploiting operators that are invertible in each argument – and operators with some other characteristics – all the way down in more complex terms is at the core of our simulation algorithm for finding possible variable bindings. This makes it possible that the simulator works fully automatically in many practical cases. More details are discussed in Sect. 3.1.

Figure 2 shows the ePNK and the simulator for high-level nets running on this example. The blue textual overlay on the top right of the place represents its current marking. The green square shown as an overlay over the transition shows that the transition is enabled in some firing mode. Once the user clicks on that transition, all the different firing modes in which the transition is enabled will be shown, and the user can select one. Of course, the simulator can also simulate automatically – firing transitions at random. In the "Simulation View" on the bottom left, you can see the complete firing sequence from the initial marking up to the current point of the simulation. The user can go back and forth in this sequence by clicking into this sequence or by using the back an forth buttons in the "ePNK: Applications" view (shown at the bottom right of Fig. 2). For details on how to start the simulator and on how to open these views, we refer to Chap. 3.6.3 of the ePNK user manual [12].

[7] If we know the value of x * y and we know the value of either x or y, we can derive the possible value of the other variable too.

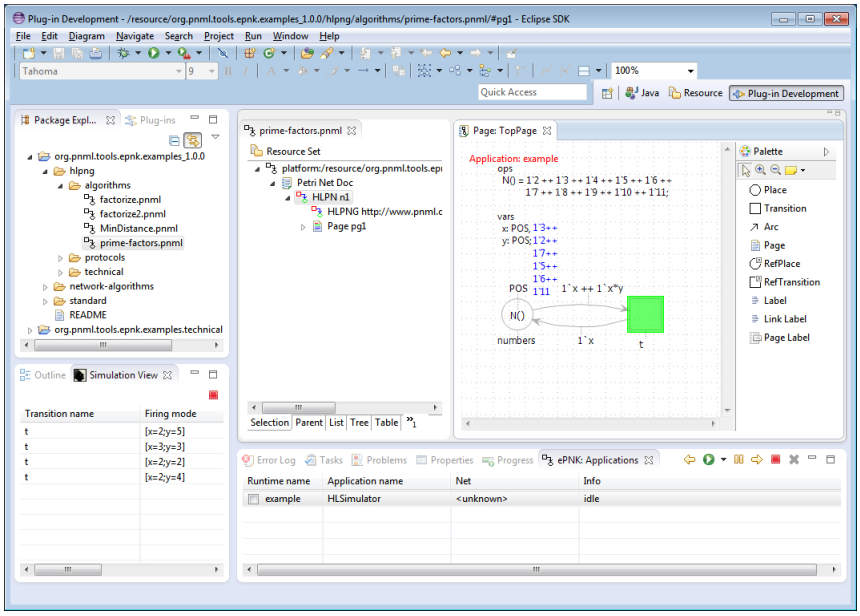

Fig. 2. The simulator running on the prime number example

2.2 Network algorithm: Minimal distance algorithms

Next, we discuss an example of a high-level Petri net schema. Except for syntactic sugar, the example is almost identical to the first publications that used algebraic net schemas for modelling and verifying network algorithms [14, 25, 21]. The example models a simple algorithm that, for a given network of computing agents with some distinguished root agents, computes the minimal distance of each agent from a root agent. The algorithm works in a distributed way, where each agent is part of the computation, and agents communicate via the communication channels of the network only. The Petri net[8] modelling the algorithm is shown in Fig. 3.

One of the main features of a Petri net schema is that, the Petri net model itself is completely independent from the actual network of agents on which the algorithm is working – that is why it is called a Petri net schema. In the model from Fig. 3, the set of agents of the network is represented by the sort AGENT, but this is a symbol only – which still needs some interpretation. The multiset constant ROOT represents the set of root nodes, and the constant I represents the set of non-root nodes (or inner nodes).

Moreover, there is an operator N, which takes an AGENT and a natural number as a parameter. This function represents sending a message from one agent

[8] In the example projects deployed for the ePNK 1.0.0, you will find this net in the sub-folder min-distance of folder network-algorithms.

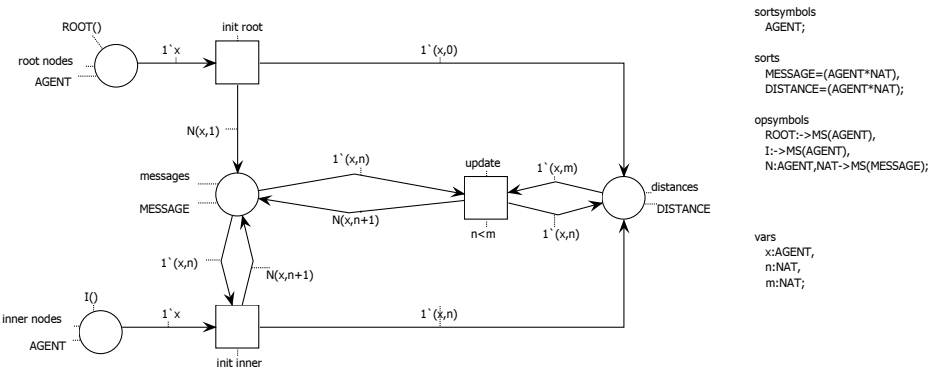

Fig. 3. Network algorithm computing the minimal distance to a root

to all its neighbours in the network – where the message itself is a natural number. A MESSAGE to an agent is represented by a pair, where the first component is the receiver AGENT and the second component is the actual content of the message. If there is a distance computed for some agent already, this is also represented as a pair of an AGENT and a number NAT – for making the difference clear, we call this pair DISTANCE.

Initially, the place root nodes contains all the root nodes of the network; the place inner nodes contains all the inner nodes. The transition init root models the initial step of the root nodes: a root note x sets its own distance to 0, which is represented as a pair (x,0), and sends a message to all its neighbours that they might have distance 1 from a root node – the set of all these messages is represented by N(x,1). Transition init inner models the initial step of the inner nodes: when an inner node x initially receives some message with some distance n, it stores this distance as a potential shortest distance, and sends a message to all its neighbours with distance n+1, which is represented by N(x,n+1). An inner node x can later receive other messages with another distance n; if the other distance n is less than its current distance m, the agent takes distance n as its new distance, and sends a message with distance n+1 to all its neighbours. This is modelled by transition update.

As said before, the Petri net model from Fig. 3 models the minimal distance algorithm for any network. If we want to simulate the algorithm, we need to know on which network it should run. To this end, a very simple network editor is deployed together with the high-level simulator of the ePNK. Figure 4 shows an example of a simple network, which was created with this editor. In this case, it is a network with directed arcs. Note that boxes with rounded edges define different categories for nodes, and each node (represented as a circle) can be associated with one or more categories. In our example, there are two categories: ROOT for root nodes, and I for inner nodes. The colours of the categories and the nodes indicate which node belongs to which category. When the network simulator is started (see ePNK manual for details), the simulator will look for a file with the

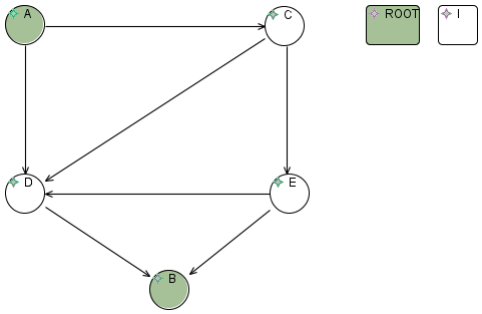

Fig. 4. A network on which the algorithm could work

same name as the PNML document – just with a extension ".networkmodel" or will prompt the user for picking a file with a network. Once the network is selected, the interpretations of the sort AGENT, the constant symbols ROOT and I, as well as the function N (sending a message to all the neighbours of the agent) are defined by this network. For the network of Fig. 4, the set associated with the sort AGENT is { A, B, C, D, E }; the constant ROOT denotes the multiset [A, B], the constant I denotes the multiset [C, D, E]; for x = A and n = 5, the term N(x,n) evaluates to the multiset [(C,5), (D,5)] – representing the message 5 sent to the neighbors C and D of agent A. These interpretations will be used for simulating the net.

Figure 5 shows the network simulator running the minimal distance algorithm from Fig. 3 on the network from Fig. 4 – after the root node A and the inner node D have taken their initial step. The interaction with the simulator and the way to control the simulation is exactly the same as for normal high-level nets.

In general, for a given network, the sort AGENT is associated with the set of nodes of the network; for every category, the respective constant symbol is associated with the multiset of nodes that are in that category (in our example, these are the root nodes and the inner nodes). For the operator N, N(x,m) denotes a multiset of pairs, where for each outgoing arc from x to y there is a pair (y,m) in the multiset. Note that the network simulator supports a few more operators, which, however, are not discussed here.

3 Computing variable bindings

The ePNK comes with a Petri net type definition for high-level Petri nets, which are called *High-level Petri net Graphs* (*HLPNGs*) according to the International Standard ISO/IEC 15909-2:2011. Implementing HLPNGs requires the implementation of a complete type system for terms: for every term, its associated sort can be computed. This is needed for checking syntactical and type correctness of terms themselves, but also for checking whether the terms for the initial marking and the arc annotations fit the type of the respective place, as well as for checking whether transition conditions have the required type BOOL.

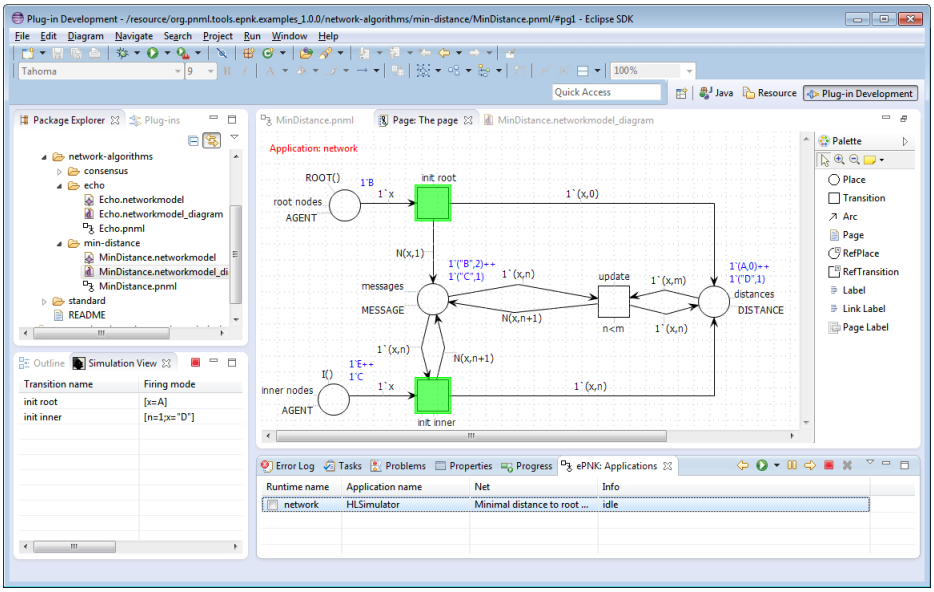

Fig. 5. Network simulator running on the minimal distance algorithm

For implementing a simulator for HLPNGs, we needed to implement the set of possible *values* for each sort of HLPNGs (the sorts domain) – in particular we needed to implement the domain for multiset sorts, which represent the markings of a place. Moreover, we needed to implement an evaluation function for terms: for a given *binding* of values to some *variables*, a term in which only these variables occur is evaluated to a value – using the values of the variables, and the meaning of the operators occurring in the term. We will later use the notation from algebraic specifications [2], were β stands for a binding of variables to values (of the appropriate type), and $\overline{\beta}(e)$ denotes the value to which a term e evaluates under this binding. Since HLPNGs have quite many built-in sorts and operators, and several of the sorts (like multisets) and operators are generic, implementing this evaluation of terms is quite some work; the implementation, however, is straightforward, following through the inductive (bottom up) definition of the evaluation of terms: implementing the domain of values for each basic sort, each generic sort, and implementing the evaluation for each operator of HLPNG.

The challenging part of implementing a simulator for high-level Petri nets, is to automatically compute the *firing modes* in which transitions are enabled. For high-level nets with a Turing complete annotation language, this is not even computable (otherwise, it would be decidable whether a program would return a certain result for some input value). And even in restricted settings it, very often, cannot be computed efficiently (since finding a binding effectively computes the inverse of a function, which is not efficiently possible for one-way functions). Still, in many typical situations – as in our examples from Fig. 1 and 3 – the possible

firing modes can be computed efficiently. Due to the general undecidability, the last resort is prompting the user for possible values for variables. But, this should happen only in rare occasions.

In Sect. 3.1, we discuss some general principles and concepts of how to compute possible firing modes for high-level nets – with the focus on the underlying concepts, but without formalizing them. In Sect. 3.2, we discuss some aspects of the implementation of the simulator for high-level nets in the ePNK, which follows the general idea, but deviates in some details.

3.1 Principles and concepts

Before discussing what possible firing modes are and how they can be computed, let us have a brief look at the problem of finding potential variable bindings for which a transition is enabled (i. e. finding possible firing modes of a transition). To this end, we have a look at the example from Fig. 1 again. Let us assume that the marking[9] *m* of place numbers is *1‘2 ++ 1‘3 ++ 1‘5 ++ 1‘6 ++ 1‘7 ++ 1‘11* as shown in Fig. 2. Now we need to find values for variables x and y – a variable binding – such that the term 1‘x ++ 1‘x*y evaluates to a subset (actually a sub-multiset) of the marking *m*. From the structure of the term, we can derive right away that the only way to achieve this is that x is bound to one of the values contained in the multiset *m*, i. e. to *2*, *3*, *5*, *6*, *7*, or *11*. And likewise we know that x*y should evaluate to one of these values. This is actually, were some simulators for high-level nets would stop already; they find possible values for x, but not for y. We checked the example from Fig. 1 in the probably most used tool for high-level Petri nets, CPN Tools [1]: CPN Tools could not automatically fire the transition[10] .

Knowing the possible values for x and the possible values for x*y, however, can be exploited further. If we know a possible value of x, say *3*, and a possible value for x*y, say *6*, we can derive a possible value for y: in this case, it would be *2*. Of course, there could be combinations of result and argument values, for which there is no possible value for y at all. The reason that we were able to derive a value for y in the above example is, that the multiplication operator on positive numbers is *invertible in each argument*: i. e. if we know the value of the result and that of one argument of the operator, we can derive the value of the other argument – or we can conclude that, for the given combination of values, there would be no argument that could produce the result. The latter would, for example be, the case if we chose the possible value for argument x to be *6* and the possible result value for x*y to be *11*.

[9] Note that we use the syntax of terms for representing values since we need to represent values somehow. A value, however, is fundamentally different from a term representing it. In order to emphasize this, we typeset terms that we use for representing a value in italics.

[10] To be fair, let us mention that, with a minor change, CPN Tools would be able to automatically compute the binding again; likewise adding a pattern (see discussion in Sect. 3.3) to CPN Tools´ binding mechanism would make the computation possible again.

This way, operators that are invertible in each argument allow us to derive possible values of other sub-terms, and eventually possible values of some variables in a top-down fashion. Once we find new possible bindings for variables, we can evaluate some other sub-terms bottom up; if, this way, all but one argument of an operator that is invertible in each argument are available, the possible values of the sub-term for the remaining argument can be computed too.

Before we discuss some more details, let us have a brief look at a slightly different situation. Suppose that we have a term x+y, where both variables are of sort POS, and suppose that *4* is a possible value of the term x+y. Then, we can derive possible values of both arguments of the + operator – in this case the two variables x and y: both variable could take values between *1* and *3*. In this case, out of a single possible value for the term, we can derive more than one, but finitely many possible values for both arguments. Note that this is no longer possible if the variables are of type INT: as soon as we allow negative numbers, there would be infinitely many possibilities. We call an operator like the + on positive numbers an operator *with finitely many inverse images*.

Above, we have discussed how to derive the possible bindings for variables that would make a term evaluate to some result value. When actually calculating possible firing modes of a Petri net, there is a minor twist. The arc annotations of a Petri net do not need to evaluate to the value of the marking of the attached place, but to a sub-multiset of this marking. We will discuss this special case on the top-level later in the implementation section. In order to avoid this special case in our conceptual discussion, however, we slightly change the setting now: for the arc, we introduce a fresh and unique variable Z with sort MS(POS), which denotes the multisets over the positive numbers; instead of computing variable bindings that evaluate the term 1‘x ++ 1‘x*y to a sub-multiset of marking *m*, we now compute variable bindings that evaluate 1‘x ++ 1‘x*y ++ Z to *m* – which is equivalent. We call the term 1‘x ++ 1‘x*y ++ Z the extended arc annotation of the respective arc.

In general, a transition t can have more than one incoming arc. In that case, we need to find a variable binding β so that the extended arc annotation e_i of every in-coming arc evaluates to the current marking m_i of the source place of that arc, and the transition condition c needs to evaluate to $true$. We call a variable binding β an *equalizer*[11] for transition t in the current marking, if $\overline{\beta}(c) = true$ and, for each e_i, we have $\overline{\beta}(e_i) = m_i$.

Our examples above gave an idea already on how possible variable bindings can be identified, by identifying possible values for sub-terms, and by evaluating all sub-terms for which the possible values for variable bindings are known. The crucial point now is to characterize the different classes of operators, which allow us to derive possible values for all arguments, when the result value and, possibly, the value of some arguments are known. We distinguish three kinds of

[11] We do not call it *unification*, since unification – though similar in spirit – is typically used for syntactical equality. Our equalizer means semantic equality.

operators: *constructors*[12], operators that are *invertible in each argument*, and operators with *finitely many inverses images*. These are explained below.

Constructors An operator is a *constructor*, if for any given result value, the values of all its arguments can be uniquely determined, so that the operator applied to the argument values gives the result value – or there are no arguments for the operator that would provide the result value at all.

Examples of constructors are: the tuple operator (x,y,z), the list append operator append(L,x) that appends a single element x to a list L, and the multiset operator n'x.

An example of a situation where there would not be any possible argument value for a constructor is the following: there are no argument values for L and x which could make append(L,x) evaluate to the empty list. This, typically, means that there is another constructor for this data type; in the case of lists, this would be the empty list.

Invertible in each argument An operator is *invertible in each argument*, if for any given result value and any given values for all but one argument of the operator, the value of the remaining argument can be uniquely determined so that the operator applied to all argument values gives the result value – or there is no value for the last argument at all for which the operator would provide the result value.

Note that, in order to be precise, we should call this kind of operators "partially" invertible in each argument, since it is okay, if no value for the last argument exists. But, in order not to be too verbose, we drop the "partially" from our terminology.

Examples of operators that are invertible in each argument are: multiset addition ++, addition +, and subtraction - on all kinds of numbers[13]. Actually, also the equality on each sort and the boolean negation are invertible in each argument.

Note that the multiplication * is almost invertible in each argument – the only exception is the case of the result value and one argument value being 0. In that case, the remaining argument could be any number. For positive numbers, however, multiplication * is invertible in each argument – which we exploit in our example.

By definition, every constructor is invertible in each argument – we do not need the argument values at all for determining the value of the remaining argument in a constructor.

Finitely many inverse images An operator has finitely many inverse images if, for any given result value, there are only finitely many possible values for its arguments, for which the operator would evaluate to the given result value (note that there might be no arguments at all).

[12] Note that our notion of constructors resembles the notion of constructors in functional programming. It is different from constructors in object oriented programming languages; in Scala the constructors would be case classes.

[13] With floats, there would be some numerical issues, though, which we do not want to go into here.

Examples of operators with finitely many inverse images are the addition operator + on natural and positive numbers[14], and all the binary boolean operators have finitely many inverse images. Also the multiplication on positive numbers * and the multiset addition ++ has finitely many inverse images – as long as we restrict ourselves to finite multisets.
Again, every constructor has only finitely many inverse images by definition.

If we now have a set of equations that are supposed to be equalized, the characteristics above can be exploited to systematically derive a set of possible values of some sub-terms. To this end, we start with the possible values of the top-level terms, since these are given by the equations. Operators with the above characteristics can be used to compute possible values of sub-terms. At first, i. e. when no values of the other arguments are known, this would be possible only for constructors and operators with finitely many inverse images, but as soon as we have identified possible values of variables, also operators that are invertible in each argument can be exploited to compute possible values of the remaining sub-terms. Note that from the characteristics of the operators above, we can assign also the empty set to a sub-term, indicating that there would be no possible value for this sub-term. When we obtain a value for a sub-term that happens to be a variable, we know the possible value for this variable. Knowing the possible values of a variable, in turn, allows us to compute values of possible sub-terms in a bottom up way, for terms in which only these known variables occur. This way, we can compute possible values of more sub-terms, which – in turn – allow us applying operators that are invertible in each argument for deriving values of other sub-terms. Iterating this as long as new possible values of sub terms can be found will eventually terminate. There are two different cases: if all sub-terms and, in particular, all variables were assigned some set of possible values – including the empty set, we know that these are all the possible values for these variables. If the set of possible values for some sub-term is empty, we would know that there is no possible binding. If the set of possible values for all sub-terms is non-empty, this gives us the possible values for all variables. If some sub-term was not assigned anything, not even the empty set, then the result is inconclusive: in this case, the simulator would need to interact with the end-user to ask for possible values for the non-assigned variables.

Note that, in principle, the above idea works just driven by the already calculated values for the different sub-terms and the rules for deriving new values for sub-terms based on the above categories of operators. The implementation, however, uses some more specific strategies, which are discussed in Sect. 3.2. The idea above does not require the operators to fall into any of the categories above. In that case, however, the possible values of its sub-terms can be determined only if the value of all its variables can be derived from other sub-terms. If this is not possible, user interaction will be necessary.

Actually, the idea above would require to keep track of possible combination of values for the different variables. The reason is that, by the last category

[14] Note that + has infinitely many inverse images when negative numbers are included.

of operators (finitely many inverse images), a sub-term is not assigned a single value, but a set of values. The conceptually easiest way to deal with this is to maintain each combination of possible values for sub-terms separately – each of which would corresponds to a partially computed binding. As soon as several new values need to be added for one sub-term, the existing partial binding would be copied as many times as there are new values for the new sub-term. Since our current implementation works in a different way, we do not discuss the details here. Experiments with different strategies and implementing the above idea in a completely data driven way might be an interesting direction for future work.

Our implementation maintains sets of possible values for all sub-terms, and only in the end tries all combinations of values for the variables, by re-evaluating the terms again. This is needed anyway, as soon as possible values for variables are provided by the end-user since we can never be sure that they are actually possible. Therefore, we need to validate all computed possible values in the end anyway.

3.2 Implementation

In the previous section, we have discussed the general idea of how to compute the possible firing modes of a transition. As discussed already, the actual implementation of our simulator does not work completely data-driven. There are two main differences: First, the top-level terms of arc annotations are handled differently exploiting that an arc annotation needs to evaluate to a subset of the marking directly. Second, the order in which the values of sub-terms and variables are computed, is not implemented in a purely data-driven way; instead, the sub-term for which the next value is calculated is determined by the number of yet un-assigned variables: the sub-term with the least un-assigned variables is chosen. In case, the value for this sub-term cannot be derived, the user is prompted for input.

Let us discuss the way our simulator handles the top-level terms in a little more detail. The simulator assumes that the arc annotations have the structure n1‘m1 ++ n2‘m2 ++ ... ++ nk‘mk, where the ni are terms of sort POS and mi are terms of the multiset sort over the places sort[15]. For each of the sub-terms ni‘mi, the possible values for ni and mi are determined by the value of the current marking of that place. For each value v contained in the multiset, v is a possible value for sub-term mi; and all the values from 0 up to the number of times v occurs in the multiset are possible values for ni. This is (almost) what an application of the operators ++ as an operator with finitely many inverses and the operator ‘ as a constructor would give as a bindings for the term n1‘m1 ++ n2‘m2 ++ ... ++ nk‘mk ++ Z. But, since it exploits the structure directly, we deemed that doing this directly would be more efficient – in particular, this seems to be what other tools do as well (see Sect. 3.3). Since the possible values for the different terms ni‘mi are now determined independently of each other, there

[15] If the term does not have this structure, the value of its variables must be derived from the other terms – or user input will be required.

might be some overlap between the different terms in using the same available tokens; therefore, we need to do a final evaluation in the end whether the bound values do really match.

In order to make the implementation more flexible and easier to extend, the simulator has a registry to which the evaluation for each operator can be registered. Likewise the computation of the inverse for the remaining argument of an argument can be registered with the simulator. This way, it is straight-forward to implement the remaining operators of ISO/IEC 15909-2, if need should be, and also to implement the inverse for more operators.

As mentioned above, the binding algorithm is not yet implemented in a fully data-driven way. This is left for future experiments, which would probably require much fine-tuning in order to obtain the necessary performance.

3.3 Discussion

Above, we have distilled the basic idea and concepts for computing the enabled firing modes of a transition, which is at the core of all simulators for high-level Petri nets that do not unfold the high-level net to a low-level net. High-level nets and simulators for high-level Petri nets have been around for quite a while now [4, 9, 20, 22, 8, 10] and different kinds of simulators have been implemented for high-level Petri nets, with similar ideas underlying the computation of enabled firing modes. We do not claim that our ideas are specifically original or that our simulator is more efficient than some existing ones – actually it is less efficient. But, to the best of our knowledge, our simulator is the first supporting HLPNGs as of ISO/IEC 15909-2 directly. In addition, our simulator is able to automatically simulate some nets, which cannot be simulated automatically with two of the most powerful simulators CPN Tools [1], and probably[16] Maria [17].

Another contribution of this paper is distilling the essence of the algorithm that computes the firing modes of a transition in such a way that it is completely data driven. The concepts used are close to the ones described in the algorithm underlying CPN Tools [5], PROD [23, 24] and Maria [16]. But our concepts and our implementation is sightly more general – and therefore less efficient. One of our main concerns was that, in most cases, the end-user should be presented with all possible firing modes, from which he can select. And in case the firing mode cannot be computed, it should be possible for the end-user to provide some suggestions for bindings for some variables, based on which a full binding is computed automatically– in the GUI, this will be indicated by grey overlays instead of green ones.

Let us briefly relate the concepts used in CPN Tools and Maria to our concepts. In both tools, much effort was put into the efficient implementation of the binding algorithm and, in particular, in efficiently updating possible bindings

[16] We checked the example of Fig. 1 in CPN Tools, and it could not compute a firing mode. We do not have a running version of Maria, but from the analysis of the binding algorithm [16], Maria would not be able to find a binding in this case.

after a has transition fired. CPN Tools algorithm [5, 19], basically, use two techniques to automatically compute the bindings: patterns are very similar to what we call constructors; moreover, CPN Tools uses dividables, which, in particular, are used to split up multisets. And patterns could also be used to deal with inverse functions. The dividables are similar to our implementation of splitting up the multiset addition ++. Haagh and Hansen [5] briefly mention the possibility of invertible operators, but this is not followed up on – and is probably not implemented. Haagh and Hansen also exploit the equations in the transition condition for deriving values of some variables. In our conceptual approach, it is not necessary to do this explicitly since the boolean operators have finitely many inverse images, and the equality operator is invertible in each argument: with this information it is possible to deduce the value of the variable without introducing this special case – except for efficiency.

In Maria, Mäkelä [16] used three notions, which are constructors, unary invertible operators, and splitting of arcs. Our notion of constructors is identical with his, and splitting of arcs is very similar to the dividable arcs of Haag and Hansen and our implementation for top-level annotations. Mäkelä explicitly exploits invertible operators, but only in the case of unary operators – in the special case of unary operators, our notion of being invertible in each argument is identical to being invertible. But our algorithm can deal with operators with more than one argument, which is slightly more general – as shown by our example.

In his PhD thesis [18], Mäkelä points out that many other simulators and state space generators actually do not directly compute a firing mode, but rather unfold the high-level net. CPN Tools, PROD – a kind of precursor of Maria – and Maria are some of the few notable exceptions. With our simulator, we add another one. And the conceptual presentation should give some room for future experimentation with different kinds of implementations of simulators for HLPNGs based on the ePNK.

4 Conclusion

In this paper, we have presented a simulator application for the ePNK, which allows simulating high-level Petri nets. To the best of our knowledge, this is the first simulator directly supporting HLPNGs of ISO/IEC 15909-2. The ePNK, the simulator and more details on how to install, how to use the ePNK and the simulator as well as some example nets can be found on the ePNK home page [3].

In our presentation, we distilled some characteristics of operators of high-level nets which can be exploited to compute the enabled firing mode in a completely data driven way. Though this idea is not completely new, some concepts are slightly more general, which allows our simulator to automatically simulate some nets which cannot be simulated by other high-level Petri net simulators.

The simulator was developed as a master's project at the Technical University of Denmark [15] with the focus on flexibility and extensibility. We hope that this provides the stepping stone for a stepwise improvement of the simulator,

which eventually leads to an efficient simulator for HLPNGs [7] as an application for the ePNK.

Acknowledgements

Michael Westergaard made very detailed comments on an earlier version of this paper. His comments together with the comments from an anonymous reviewer, hopefully, helped us improving the paper and clarifying its contribution. We would like to thank both of them for their efforts.

References

1. CPN Tools home page. http://cpntools.org/, February 2013.
2. Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications 1, Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
3. ePNK home page. http://www2.imm.dtu.dk/~ekki/projects/ePNK/, February 2013.
4. Hartmann J. Genrich and Kurt Lautenbach. System modelling with high-level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.
5. Torben Bisgaard Haagh and Tommy Rudmose Hansen. Optimising a coloured Petri net simulator. Master's thesis, University of Aarhus, Department of Computer Science, December 1994.
6. L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Treves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. In K. Jensen, editor, 10^{th} *Workshop on Coloured Petri Nets (CPN 09)*, pages 101–120, October 2009.
7. ISO/IEC. Systems and software engineering – High-level Petri nets – Part 2: Transfer format, International Standard ISO/IEC 15909-2:2011, February 2011.
8. K. Jensen and G. Rozenberg (Eds.). *High-level Petri Nets, Theory and Application*. Springer-Verlag, 1991.
9. Kurt Jensen. Coloured Petri nets and invariant methods. *Theoretical Computer Science*, 14:317–336, 1981.
10. Kurt Jensen. *Coloured Petri Nets, Volume 1: Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.
11. Ekkart Kindler. The ePNK: An extensible Petri net tool for PNML. In *Applications and Theory of Petri Nets - 32^{nd} International Conference, Proceedings*, volume 6709 of *LNCS*, pages 318–327. Springer, 2011.
12. Ekkart Kindler. The ePNK: A generic PNML tool - users' and developers' guide for version 1.0.0. Technical Report IMM-Technical Report-2012-14, DTU Informatics, Kgs. Lyngby, Denmark, December 2012. URL http://www2.imm.dtu.dk/~ekki/projects/ePNK/PDF/ePNK-manual-1.0.0.pdf.
13. Ekkart Kindler and Csaba Páles. 3D-visualization of Petri net models: Concept and realization. In J. Cortadella and W. Reisig, editors, *Application and Theory of Petri Nets 2004, 25^{th} International Conference*, volume 3099 of *LNCS*, pages 464–473. Springer, June 2004.
14. Ekkart Kindler and Wolfgang Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:16–31, December 1996.

15. Mindaugas Laganeckas. A simulator for high-level Petri nets: Model-based design and implementation. Master's thesis, IMM-M.Sc.-2012-101, DTU Informatics, Technical University of Denmark, September 2012.
16. Marko Mäkelä. Optimising enabling test and unfoldings of algebraic system nets. In J.-M. Colom and M. Koutny, editors, *Application and Theory of Petri Nets 2001, 22^{rd} International Conference*, volume 2075 of *LNCS*, pages 283–302. Springer, June 2001.
17. Marko Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In J. Esparza and C. Lakos, editors, *Application and Theory of Petri Nets 2002, 23^{rd} International Conference*, volume 2360 of *LNCS*, pages 434–444. Springer, June 2002.
18. Marko Mäkelä. *Efficient Computer-Aided Verification of Parallel and Distributed Software Systems*. PhD thesis, HUT.TCS-A81, Helsinki University of Technology, Laboratory for Theoretical Computer Science, November 2003.
19. Kjeld H. Mortensen. Efficient data-structures and algorithms for a coloured Petri nets simulator. In Kurt Jensen, editor, *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, volume DAIMI PB - 554, pages 57–74. Datalogisk institut, Aarhus Universitet, August 2001.
20. Wolfgang Reisig. Petri nets with individual tokens. *Theoretical Computer Science*, 41:185–213, 1985.
21. Wolfgang Reisig. *Elements of Distributed Algorithms — Modeling and Analysis with Petri Nets*. Springer, 1998.
22. Wolfgang Reisig and Jacques Vautherin. An algebraic approach to high level Petri nets. In *Proceedings of the VIII European Workshop on Application and Theory of Petri Nets*, 1987.
23. Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. PROD reference manual. Series B, Technical Reports 13, Helsinki University of Technology, Digital Systems Laboratory, August 1995.
24. Kimmo Varpaaniemi, Keijo Heljanko, and Johan Lilius. prod 3.2 — an advanced tool for efficient reachability analysis. In Orna Grumberg, editor, *Computer Aided Verification: 9th International Conference, CAV'97, Haifa, Israel, June 22–25, 1997, Proceedings*, volume 1254 of *LNCS*, pages 472–475. Springer-Verlag, 1997.
25. M. Weber, R. Walter, H. Völzer, T. Vesper, W. Reisig, S. Peuker, E. Kindler, J. Freiheit, and J. Desel. DAWN: Petrinetzmodelle zur Verifikation Verteilter Algorithmen. Informatik-Bericht 88, Humboldt-Universität zu Berlin, December 1997.