Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 79 (2006) 1444–1455
The Journal of Systems and Software
www.elsevier.com/locate/jss

Automatic code generation from high-level Petri-Nets for model driven systems engineering

Stephan Philippi

University of Koblenz, Department of Computer Science, P.O. Box 201 602, 56016 Koblenz, Germany

Received 12 June 2005; received in revised form 11 November 2005; accepted 17 December 2005
Available online 7 February 2006

Abstract

One of the main concepts of the model driven architecture framework proposed by the OMG is the transformation of platform independent models into either platform dependent ones or into implementations. The latter needs generators which automatically translate models into code of a chosen target programming language. Nowadays most models in systems engineering are created with the UML as standardized set of notations. However, the UML is still largely undefined from a semantical point of view. Automatic code generation from these models is thus only possible if further semantic definitions are provided. Since Petri-Nets are frequently utilized for these purposes, the automatic generation of code from Petri-Nets is an important topic.

Starting from this observation, this article gives an overview of different strategies to generate code from high-level Petri-Nets. One of these strategies is due to its characteristics investigated in more detail. Several improvements of the basic idea are introduced and also the impact of these enhancements in terms of efficient execution of generated code is illustrated with a benchmark.

Keywords: Code generation; Model driven architecture; Modelling; Object-orientation; Petri-Nets; Systems engineering; Visual languages

1. Introduction

The development of almost every kind of technological system is nowadays based on models as abstractions of real-world and/or thought concepts. State of the art in systems modelling are multi-perspective approaches, where different kinds of models represent different views on a system. During the course of systems development such models are enriched with additional information and are thus transformed from more abstract into more concrete ones. In systems engineering the UML (Object Management Group (OMG), 2004) has become the lingua franca for object-oriented modelling in the last years. While object-oriented models traditionally serve as blueprints for manual systems implementation, the *model driven architecture* initiative of the OMG (OMG, 2003) promotes the usage of models throughout the entire development process. Starting from a so-called *platform independent model* (*PIM*), different kinds of transformations lead to *platform specific models* (*PSMs*). Such transformations might automatically generate, for instance, source code for different programming languages from a single PIM. Automatic code generation from object-oriented models is thus an important area of work in the context of model driven architecture.

Since the UML is the standard for object-oriented modelling, code generation from this set of notations seems to be a natural starting point for investigation. However, as the UML is still largely undefined from a semantical point of view, difficulties arise in such a scenario due to the ambiguity of models, which makes them not very well suited neither as a reliable communication basis nor the starting point for automatic code generation. In order to overcome these problems, a plethora of work exists to give (parts of) the UML a formal underpinning. As many of these approaches are based on Petri-Nets (see e.g. Baresi and Pezzè, 2001; Bernardi et al., 2002), automatic code

E-mail address: philippi@uni-koblenz.de

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.12.022

generation from this formalism is a promising lead in the area of model driven architecture.

Code generation from Petri-Nets generally has a long tradition. However, unlike methods for the analysis and simulation of Petri-Nets, code generation is not yet considered a standard feature, and thus not implemented in modelling tools such as Design/CPN (Christensen et al., 1997), Renew (Kummer et al., 2004) and others. An extensive review of existing work in the area of automatic code generation from Petri-Nets is given in (Girault and Valk, 2003). Most of the approaches in this review focus on code generation from (extended) low-level Petri-Nets, e.g. for the generation of controllers (see for instance Uzam and Jones, 1998; Lee et al., 2004). Even though the review also lists approaches for code generation from high-level Petri-Nets, the work in this area is not based on object-oriented principles, and in consequence not applicable to more complex systems. A frequent use of approaches to automatic code generation from Petri-Nets is the validation of requirements in systems engineering. Since models are used in this context as blueprints for traditional systems development, aspects such as readability (and thus extensibility) as well as efficiency of generated code are not important and, therefore, usually not the focus of existing approaches to code generation. As a consequence of (i) models which are not structured in an object-oriented way and (ii) the problems arising from not well readable code for further systems development, some authors state that the final application 'should not be the main goal of code generation [from Petri-Nets]' (Girault and Valk, 2003, p. 434).

In contrast to this view, the motivation for the work presented in this article is to shed some light on Petri-Net based code generation from a different angle. In accordance to the ideas of model driven architecture, models are considered as a central means not only for the capturing and validation of requirements, but for the whole development process in systems engineering. As a direct consequence of this philosophy there are major differences between the goals of the code generation approach to be presented in the remainder of this article and the existing work as reviewed in (Girault and Valk, 2003). First of all, the approach to be introduced is based on a class of object-oriented Petri-Nets in order to be applicable also for complex systems. In addition, the work to be presented is intended to support the use of models throughout the entire systems development process and not only for validation purposes. A fundamental requirement to reach this goal is that automatically generated code closely resembles manually implemented code. In order to make a first step into this direction the subject of discourse in this article is the evaluation of different strategies for automatic code generation from Petri-Nets with the focus on their general applicability as well as the readability, extensibility and efficiency of generated code.

Starting from this perspective, the next section gives an overview of the theoretical possibilities for code generation from (high-level) Petri-Nets. An object-oriented class of Petri-Nets is introduced then together with a running example in section three. Different variants of simulation-based code generation as well as the results of a performance evaluation based on the introduced example are discussed afterwards. Finally, the last section concludes the article with a short summary and an outlook on future work.

2. Code generation from high-level Petri-Nets

Since Petri-Nets are mathematical objects there are different methods for their formal analysis. These methods can also be used as a basis for code generation and are thus a natural starting point for further investigations into this direction. The potential of (i) *structural analysis*, (ii) *simulation* and (iii) *reachability graphs* for automatic code generation from high-level Petri-Nets is thus described next. Afterwards, the applicability of approaches for code generation based on these methods is discussed.

The main idea with code generation based on the *structural analysis* of Petri-Nets is to search for regular structures in the underlying graph. The patterns to be searched for are those, which can be immediately translated to programming language elements, like sequences, loops and if-constructs. The code resulting from automatic generation based on the detection of such regular graph patterns is almost identical to manually implemented code. In consequence, the advantages of this approach are the readability of the generated code and its efficient execution. However, Petri-Net models are often not constructed from regular structures, i.e. directly translatable graph patterns cannot be found in every model. The deeper reason for this is that Petri-Nets generally provide much more opportunities to construct different control flow structures in comparison to common programming languages. The latter need regular patterns, like correctly nested loops, as a fundamental prerequisite for compilation. This is not the case with Petri-Net models, i.e. there are Petri-Net structures, which cannot be directly represented by programming language constructs in a natural way. The problems arising from Petri-Net models which contain non-regular structures could be easily avoided if the user would be restricted to models which are built from directly translatable graph patterns. Even if such a strategy would result in models from which well readable and efficiently executable code could be generated, the restrictions for the user are not acceptable. Not only would Petri-Nets lose their versatility of use, also their simple construction rule would become much more complicated and therefore more difficult to understand. Furthermore, a model with a regular structure is easily modified in a way which results in a non-regular structure—a single arc may make all the difference. In consequence, code generation solely based on regular structures seems not to be well suited for non-trivial systems.

Code generation based on *simulation* does not suffer from the above problems, since the simulation of a Petri-Net is not restricted to regular structures. The main idea

with this approach is to generate state based code which is controlled by a simulation environment in every method. Even if this approach is most versatile, the generated code is due to its state based nature less readable and in addition also suffers from efficiency problems. The main issue in this context is that the test for activated transitions consumes a lot of resources, i.e. on average too many calculations are needed in order to identify activated transitions. Since these calculations do not immediately contribute to the progress of a model, simulation-based code for models with even a modest number of transitions is not very efficient in comparison to manually implemented code.

In contrast to simulation-based code generation, approaches based on *reachability graphs* do not suffer from the above efficiency problems. The *reachability graph* of a given Petri-Net is the representation of *all* possible states of the underlying model as an automaton. Out of this, all possible follower states are always known, i.e. in comparison to the simulation-based approach, there are almost no unnecessary transition activation checks. In consequence, code generated on the basis of reachability graphs is efficiently executable, but due to its state based nature not as readable as manually written code. However, the main drawback of this approach is the exponential size of the reachability graph. While this may not be a problem for the generation of code from low-level Petri-Net models, the state space for high-level Petri-Net models with their programming language like type system is usually too big for the construction of the reachability graph even for smaller models. In consequence, code generation based on reachability graphs is generally not feasible for high-level Petri-Net models.

As a conclusion, approaches to code generation based on the three standard methods for the analysis of Petri-Nets have very different characteristics as summarized in Table 1. While the generation based on structural analysis results in well readable and efficiently executable code, it is only applicable to models with a restricted structure and thus not versatile enough for practical applications. Code generation based on reachability graphs in principle works on arbitrary net structures and also produces efficiently executable code. However, this approach is also not generally applicable, since the construction of reachability graphs takes exponential efforts. Finally, simulation-based code generation is applicable for every kind of model and does not suffer from the restrictions of the other approaches described above. The drawbacks of the naive approach to code generation by means of simulation are that the resulting code is due to its state based nature less readable than manual implementations and also efficiency is not taken to its best especially for more complex models.

Table 1
Properties of code generation approaches

Generation approach	Applicability	Readability	Efficiency
Structural analysis	−	+	+
Simulation-based	+	0	−
Reachability graph	−	−	+

Since all of the three standard methods for Petri-Net analysis have disadvantages when it comes to code generation, the question raises if the specific problems cannot be overcome? A fundamental prerequisite for the acceptance of any approach to code generation is its practical applicability. In this light, code generation based on structural analysis and reachability graphs are for different reasons not well suited. While approaches to code generation based on structural analysis strongly restrict the use of Petri-Nets as a modelling language, code generation based on reachability graphs is not feasible due to the exponential efforts required for their construction. In comparison to these problems, the disadvantages of simulation-based code generation fall into a different category. While unacceptable restrictions of the user with a fixed set of predefined structures for Petri-Nets as well as the exponential efforts required for the construction of reachability graphs cannot be helped, there are methods to improve the readability and efficiency of code which results from simulation-based generation. The remainder of this article, in consequence, focuses on simulation-based code generation.

3. Object-oriented Pr/T-models

A running example is used throughout the article as a vehicle to demonstrate different methods for the improvement of readability and efficiency of code generated with simulation-based approaches. Since Petri-Nets are due to the absence of proper modularization capabilities often criticized as not being well suited for the modelling of complex systems, the example is given in an object-oriented Petri-Net dialect which overcomes these problems. However, even if the motivation for our work is the automatic generation of code to support approaches for model-driven architecture in an object-oriented context, the code generation principles to be described are generally applicable to other high-level Petri-Nets as well. The reason for this is that we focus on code generation for methods of object-oriented classes as basic building blocks for more complex systems. The principles to be described for these purposes can also be applied, e.g., to the automatic generation of purely procedural code.

As a basis for further discussions so called *Object-Oriented Pr/T-Models*, or *OOPr/T-Models* for short, are briefly introduced in the following. More detailed descriptions of this Petri-Net class may be found in (Philippi, 2000, 2001). With OOPr/T-Models a system is described by means of static, dynamic and functional views (see also Rumbaugh et al., 1991). While the architecture of a system is captured with UML class diagrams (Object Management Group (OMG), 2004), a dynamic model may be assigned to each class of the static view in order to specify concurrency control at the method level. However, since this article focuses on the generation of efficient sequential code as a prerequisite for the generation of efficiently executable con-

current code, dynamic models are not further considered here. Instead, functional models are described in more detail as a means for the specification of the functionality of methods. For these purposes, one-safe Predicate/Transition-Nets (Genrich and Lautenbach, 1981) are extended accordingly. In order to be able to set up functional models for the specification of methods in an object-oriented context, different interface constructs have to be provided for the communication with the environment. To be more concrete, modelling primitives are needed to (i) import attribute and argument values, (ii) communicate through message passing, and (iii) export new attribute values and specify a return value, if any. Therefore, Pr/T-Nets are extended with the following interface elements:

- The *input interface* consists of so called 'preload places', which are inscribed with attribute/argument identifiers or local variable definitions. If a method is activated, these places transparently provide the respective values and further behave like ordinary ones, i.e. preload places specify the object dependent initial marking of any OOPr/T-Net modelled method. Preload places are visually distinguished from traditional places by a boldly painted border.
- To be able to model the calling of a method in OOPr/T-Nets, so called 'message-transitions' are used as *communication interface*. In addition to common transition inscriptions, a message transition specifies which message is to be sent upon activation. As a prerequisite, the identifier of the object the message should be sent to as well as the arguments of the respective method have to be available on the incoming arcs of a message transition. If the method to activate returns a value, this result can be further used on outgoing arcs of the message-transition.
- Similar to the input interface special kinds of places are used for *output interfacing* purposes. To be able to model the export of attribute values, so called 'postsave places' are introduced. A postsave place is inscribed with the attribute to which the token resident on this place should be the new value if the execution of the method ends. Postsave places are given by bold circles like preload places. Both kinds are distinguished in a given model through in- and outgoing arcs. Each preload place has at least one outgoing and may have incoming arcs, whereas postsave places have incoming arcs only.Another element of the output interface is the so called 'exit place', which is given by a double circle. If a token resides on such a place the method execution ends and the value of the token is returned to the caller of the method, if needed.

The semantics of a model built with this notation is defined by a set of rules to derive a single Pr/T-Net from an OOPr/T-Model with its static, dynamic and functional parts. In consequence, it is not only single views that have a formally defined semantics, but the whole model built from these views. The main idea in this context is to use predefined patterns to build an object-oriented runtime system with Pr/T-Nets. This runtime system is then used to integrate the different views of an OOPr/T-Modell at the level of Pr/T-Nets. The elementary structure of a Pr/T-Net constructed this way provides the internal communication infrastructure needed in object-oriented systems for the routing of messages. The underlying routing principles define, for instance, the semantics of inheritance in an object-oriented system. The semantics chosen for OOPr/T-Models with this respect is that which is defined by the Java programming language (Arnold and Gosling, 1996), which is also the target language for code generation throughout this article. A more detailed account on the semantic aspects of OOPr/T-Models is given in (Philippi, 2000).

The use of OOPr/T-Models for systems modelling follows the same principles as every other object-oriented approach for these purposes. The differences in the particular languages used to express specific views on a system, therefore, have no influence on their use within an object-oriented design process. In order to illustrate the use of OOPr/T-Models for the modelling of object-oriented systems in more detail, and at the same time introduce a running example for the discussion of code generation principles in the next section, a model for the calculation of primes is given in the following. The rationale to work with this example throughout the article is that it is well known, not too complex and computationally demanding. Especially the latter will be useful for the demonstration of the efficiency of generated code in comparison to a manually implemented reference system.

The main idea of the calculation of primes using the well known 'Sieve of Eratosthenes' (as usually optimized for implementation) is to perform a sequential search on integers up to a given border. Each number is checked as to whether it can be divided by any element of the list of already calculated primes without a remainder or not. If so, the candidate is not a prime. Otherwise, this number is added to the list of known primes as a basis for the evaluation of further candidates. The object-oriented architecture for this system is given in Fig. 1.

The class diagram consists of two classes, namely 'Sieve' and 'Prime', where the former is intended to produce the candidates which are checked against the prime property by the latter. The result of the calculation, then, is a linked list of 'Prime' instances, each representing a single prime.

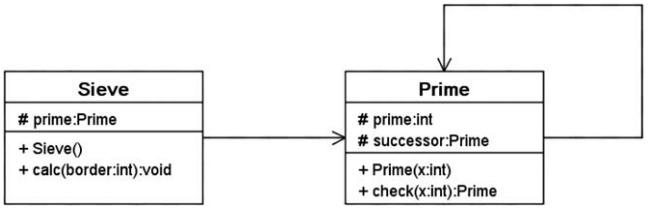

Fig. 1. Object-oriented architecture for the calculation of primes.

Class 'Prime' contains two attributes 'prime' and 'successor', which store the prime an instance of 'Prime' represents as well as a reference to the next known prime in line, if any. The functional model for the constructor of 'Prime' is given in Fig. 2. The argument 'x' passed to the method by its caller is imported into the model by means of a preload place and initializes the 'prime' attribute of the object using the respective postsave place.

Method 'check' of class 'Prime' checks as to whether a passed candidate can be divided by the prime number an instance of 'Prime' represents or not. The functional model for this method is given in Fig. 3. The object dependent initial marking of the model is given by three preload places *P1*, *P2* and *P5*, which import the candidate 'x' passed to the method as well as the current values of the 'prime' and 'successor' attributes. If the candidate 'x' can be divided by 'prime' without a remainder, 'x' is not prime and thus the method ends with exit place *P3*. Otherwise, 'prime' is not a prime factor of 'x'. If the current 'Prime' instance does not hold a reference to a successor, 'x' is a prime itself and an instance of 'Prime' is created for its representation by means of message transition *T4*. A reference to this new object is stored in the 'successor' attribute of the current prime using postsave place *P6* and the method terminates with exit place *P7*. In case a successor already exists, the execution of the method ends with the returning of the reference to the next prime in line by transition *T3*.

Method 'check' of class 'Prime' is invoked by method 'calc' of class 'Sieve'. The functional model of the 'calc' method is given in Fig. 4. Preload place *P1* is inscribed with 'border' and thus imports the corresponding method parameter into the model, which serves as the upper limit for prime calculation. The current value of attribute 'prime' is imported into the model by preload place *P5*—it is a reference to the 'Prime' object representing the first prime '2'. The 'prime' attribute itself is initialized in the constructor of class 'Sieve'. The model of this method is not given here, since it is very similar to the constructor of class 'Prime' in Fig. 2. The third preload place *P2* inscribed with 'int:3' specifies a local variable, which is used as a counter for the generation of candidates to check against the prime property. For each candidate the 'check' method of the first prime is called with message transition *T3*. If a reference to another prime is returned, the prime property of the candidate also needs be checked with the 'Prime' object this reference points to. In case no such reference is returned, either the candidate was not a prime or it has already been appended to the list of known primes by the last call of 'check' (see Fig. 3). The counter is then increased and the next candidate checked. If the counter equals the

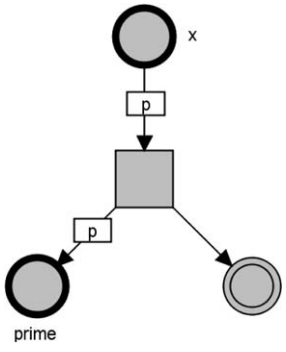

Fig. 2. OOPr/T-Model of the constructor of class 'Prime'.

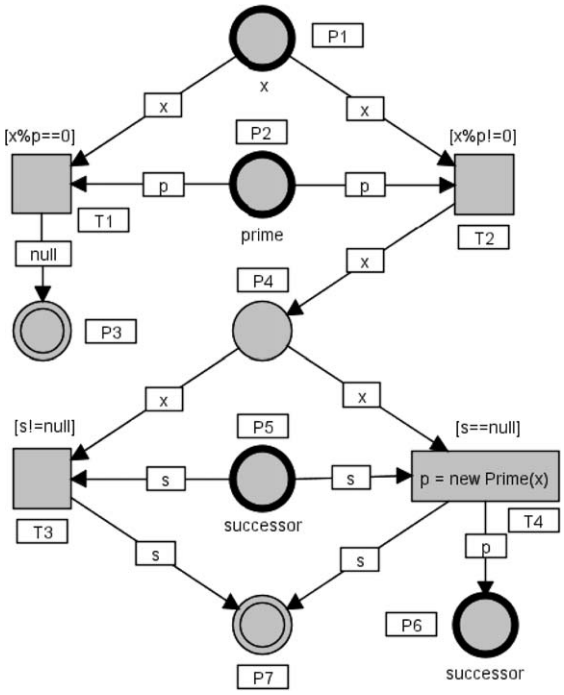

Fig. 3. OOPr/T-Model of method 'check' of class 'Prime'.

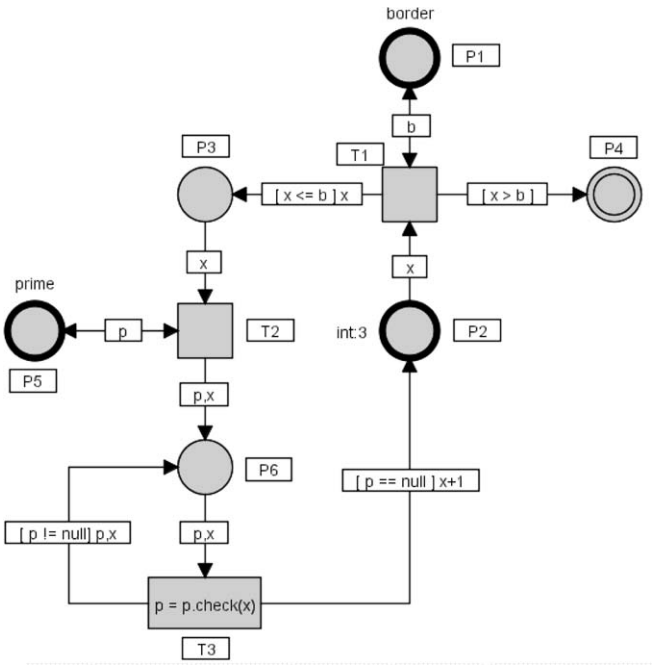

Fig. 4. Functional model for method 'calc' of class 'Sieve'.

calculation 'border', the execution of the method finally ends. In contrast to the model in Fig. 3, guards are annotated in Fig. 4 with arcs instead of transitions. This, in fact, is a syntactical abbreviation for a more complex net structure with an additional place and two guarded transitions with the same semantics.

Starting from OOPr/T-Models as introduced in this section, automatic code generation is discussed in the following.

4. Simulation-based code generation

This section describes different variants of simulation-based code generation with the calculation of primes as running example. Even if this example is given by an OOPr/T-Model, the code generation principles discussed in the following are largely independent of a specific class of high-level Petri-Nets, i.e. they apply to other kinds of Petri-Nets almost without any changes.

As already mentioned above, Java is the target programming language for code generation throughout this article. However, the principles to be described apply to other programming languages alike.

4.1. Naive simulation approach

The first step in code generation for OOPr/T-Models is the straight forward translation of a class diagram into Java constructs. Result of this generation step are class frames which contain attribute definitions as well as method signatures with empty bodies.

In the second step the empty bodies of the methods generated in the first step are filled with the results of the code generation from the respective functional models. The basic concepts of simulation-based code generation are states and the transitions between these states. The state of a Petri-Net is generally given by the current marking of its places, while state transitions occur if Petri-Net transitions are activated. In order to generate code which simulates a given Petri-Net, states and transitions between these states have to be represented with appropriate programming language constructs. For the representation of the state of an object-oriented method specified by a Petri-Net, local variables are to be declared in the target programming language for each place of the functional model. If a place in a Petri-Net, for instance, stores integer values, a local integer variable is needed within the body of the method in question. In case tuples are used on Petri-Net places instead of basic data types, a single variable is to be defined for each element of such a tuple. In order to be able to indicate if a place is currently marked or not, an additional boolean flag is generated for each place of a model, too. Variables generated for preload places are initialized according to the inscriptions in the model, i.e. either argument or attribute values are used for their initialization. In order to store the value to return to the caller of the method, an additional local variable 'result' is declared within the method body. Furthermore a variable 'running' is given, which reflects whether the simulation of the model is still active or if an exit place has already been reached. Fig. 5 illustrates the outcome of the described generation principles for method 'check' of class 'Prime' (Fig. 3).

After the generation of the programming language structures to store the current state of a Petri-Net and their initialization, additional constructs are needed in order to modify such a state in a way which reflects the simulation of the underlying model. In the next generation step, therefore, programming language representatives of the transitions in a Petri-Net model are to be created. The main idea here is to generate a 'while' loop which stops if an exit place is marked. Within this loop each Petri-Net transition is represented by a code block, which is guarded by an 'if' statement. This guard checks whether the current state of the model activates the respective transition or not. If the activation condition evaluates to 'true', the body of the 'if' statement is entered and the actions in terms of state changes associated to the transition are executed. In detail, the place variables with incoming arcs to the transition in question are emptied, in case of a message transition the specified method is called, and finally the variables representing places connected with outgoing arcs are initialized with new values as specified by the arc inscriptions. If a transition places a value onto an exit place, the 'running' variable is set to 'false' in order to terminate the simulation.

```
public class Prime
{
 // Attribute Definitions
    private int prime;
    private Prime successor;

 // Method Definitions
    public Prime check(int x)
    {
     // Place Variable Definitions
        int      P1_value = x;
        boolean  P1_used  = true;
        int      P2_value = prime;
        boolean  P2_used  = true;
        Prime    P3_value = null;
        boolean  P3_used  = false;
        int      P4_value = 0;
        boolean  P4_used  = false;
        Prime    P5_value = successor;
        boolean  P5_used  = true;
        Prime    P6_value = null;
        boolean  P6_used  = false;
        Prime    P7_value = null;
        boolean  P7_used  = false;

     // Simulation Control
        Prime    result   = null;
        boolean running  = true;

        [...]
    }
}
```

Fig. 5. Generation of class and method frames.

Prior to the ending of a method, values residing on postsave places need to be saved to the associated attributes. Fig. 6 illustrates this basic generation pattern with the code generated for transitions *T1* and *T2* of method 'check' of class 'Prime'.

If the model of a method only consists of a single transition connected to an exit place, the above generation pattern may be simplified, since no outermost loop for simulation control is needed. Not only becomes the resulting code much more compact and readable with these optimizations, the generated code in addition is also more efficient, since unnecessary transition activation checks and the surrounding environment for the simulation of models in a method are only generated if needed. In the example this simplification applies to the constructor of class 'Prime' (see Fig. 2).

```
// Main While Loop
   while (running)
   {
    // Code Generated for Transition T1
       if (P1_used && P2_used &&
                   && ((P1_value % P2_value) == 0))
       {
        // Local Transition Variables
           int local_x = P1_value;
           int local_p = P2_value;

        // Empty Input Places
           P1_used = false;
           P2_used = false;

        // Write Values to Output Places
           P3_value = null;
           result   = P3_value;

        // Exit Place ends Method Execution
           running = false;
           break;
       }

    // Code Generated for Transition T2
       if (P1_used && P2_used && !P4_used
                   && (( P1_value % P2_value) != 0))
       {
        // Local Transition Variables
           int local_x = P1_value;
           int local_p = P2_value;

        // Empty Input Places
           P1_used = false;
           P2_used = false;

        // Write Values to Output Places
           P4_value = local_x;
           P4_used  = true;
       }

       [...]

     }

     // Safe Values from Postsave Places to Attributes
     if (P6_used) successor = P6_value;
     return result;
   }

   [...]
```

Fig. 6. Code resulting from basic generation pattern.

With the basic concept for simulation-based code generation illustrated above, arbitrary Petri-Net models can be translated to programming language constructs. However, due to the state based nature of the resulting code its readability is affected since developers are usually not accustomed to think in states, but more in traditional programming language constructs. The main drawback of the basic version of simulation-based code generation is that on average very many transition activation checks have to be performed in order to identify activated ones. In turn, the ratio between activation checks and actual simulation progress of a method is unfavourable.

Starting from the basic idea of the simulation-based approach to automatic code generation, the next sections introduce different enhancements of the basic generation concept in order to produce more efficiently executable and also more readable code.

4.2. Loop detection

In order to minimize unnecessary checks for transition activation, the basic generation approach introduced above needs to be extended with features for the direct translation of specific structural patterns in Petri-Net models into programming language constructs. Loops are the prime candidate for investigations into this direction, since code inside loops is usually executed far more often in comparison to other code. Thus, the direct translation of loops into programming language constructs has at least potentially a great impact on the efficiency of generated code.

The first question to answer for the direct translation of loop structures into programming language constructs is how to identify such structures in a given Petri-Net model in the first place? Generally, cyclic structures in a Petri-Net may be identified by means of invariant analysis (Lautenbach, 1987). Since this analysis technique is based on low-level Place/Transition-Nets, a high-level model needs to be preprocessed prior to invariant analysis with the following steps:

- Arcs inscribed with guards as given for instance in Fig. 4 are a syntactical abbreviation for a net structure which contains an additional place and two additional transitions, where the latter ones are inscribed with the guards of the arcs. Since low-level Petri-Net analysis methods do not take high-level inscriptions into account, it is sensible to transfer as much as possible of this information into the net structure prior to analysis. Out of this, guarded arcs are expanded as illustrated in Fig. 7.
- The input to invariant analysis is the representation of a given Petri-Net model as linear-algebraic matrix, in which places and transitions are the two dimensions and arc weights the matrix elements. Out of this, all arc inscriptions of a given Petri-Net model are replaced

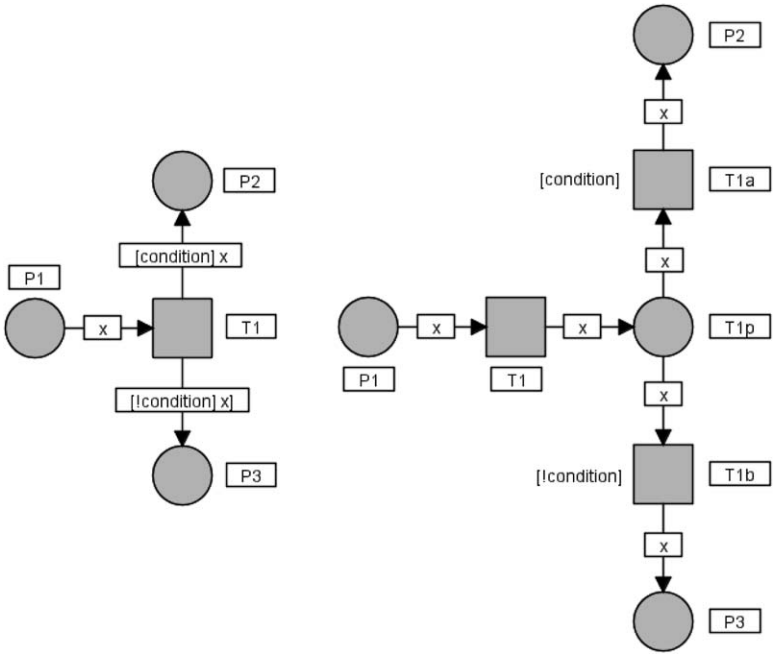

Fig. 7. Expansion of guarded arcs.

by '1' as the default arc weight first. Since information is lost in a matrix if there are two arcs from and also to a specific place from the same transition, such a structure has to be expanded with an additional place/transition pair.

After applying these preprocessing steps to the model of method 'calc' of class 'Sieve' given in Fig. 4, the *T*-invariant analysis reveals that there are two cycles in the model, namely (*T1, T1a, T2, T3, T3b*) and (*T3*, *T3a*) with *T1a*, *T3a* and *T3b* being the transitions automatically inserted into the model as a result of the expansion of guarded arcs. However, a cyclic structure is only a necessary condition for direct translation to a programming language loop. Further conditions for a suitable *T*-invariant are:

- There has to be a pair of guarded transitions in a model whose guards differ only by negation and which evaluate only values from shared input places.
- One of those transitions has to be part of the *T*-invariant while the other needs to be outside.
- In order to conform to standard loop processing in programming languages, situations need to be avoided in which more than a single token is inside a cyclic structure in a model. Out of this, places connected to transitions of a *T*-invariant are not allowed to have incoming arcs from transitions outside of the *T*-invariant. Furthermore, each invariant is allowed to contain at most a single preload place without in- and outgoing arcs to the same transition(s) of a *T*-invariant.

Considering the two *T*-invariants of the model given in Fig. 4 both fulfill the above conditions, i.e. they can be directly translated to programming language constructs. In order to do so, it needs to be checked first if an identified invariant represents a head- or a foot-controlled loop. Since transition *T1* is activated from the start by the preload places, the first invariant is head-controlled and may thus be translated into a 'while' loop. The second invariant is to be translated into a 'do/while' loop, because the loop condition is checked after the activation of *T3*. Fig. 8 gives the result of the code generation for the (*T3*, *T3a*) invariant. After the activation check for *T3* the 'check' method is called as long as no 'null' value is returned. If the loop con-

```
// Main While Loop
   while (running)
   {
    // Code Generated for Transition T3
       if (P6_used)
       {
        // Local Transition Variables
           Prime local_p = P6_value1;
           int   local_x = P6_value2;

        // Empty Input Places
           P6_used   = false;

           do { local_p = local_p.check(local_x); }
           while (local_p != null);

           // Write Values to Output Places
           P2_value = local_x + 1;
           P2_used  = true;
       }

       [...]
```

Fig. 8. Code generated with loop detection.

dition does not hold any longer the output place *P2* is filled with the appropriate value.

The above example illustrates that the identification of suitable patterns leads to the generation of code which better resembles manually written programs in comparison to the naive simulation-based generation concept introduced before. The next section takes the idea of pattern identification further with the direct translation of structures for conditional execution.

4.3. Choice identification

The identification of structures for conditional execution suited for a direct translation into 'if' statements uses some of the basic ideas behind loop detection. A prerequisite for the generation of such a structure is that there exists a pair of guarded transitions whose guards differ only by negation and which exclusively evaluate values from shared input places. In method 'check' of class 'Prime' there are two sub-structures which fulfill these conditions, namely $(T1, T2)$ and $(T3, T4)$. After the identification of suitable structures they are translated into 'if/else' statements as illustrated in Fig. 9 with the result of the generation for transitions *T3* and *T4*. While the outer 'if' clause checks for activation conditions which are the same for both transitions, the inner 'if' checks for differing conditions.

The described translation pattern needs to be modified in case both transitions have different output places. In such a situation an 'if/else' translation is not sufficient, since in addition to guards and shared activation conditions, the occupation of different output places for both transitions needs to be checked. Out of this, an 'if/else/if' construction is needed in these cases instead of 'if/else'.

4.4. Merging of transition sequences

The main idea of the transformations described up to now is the minimization of unnecessary transition activation checks, which at the same time produces also more readable code in comparison to the naive simulation-based pattern. This idea can be further exploited if a model contains transition sequences. A transition sequence is a set of two or more transitions, where all output places of a preceding transition are the only input places to a follower transition. Such a sequence may be merged into a single transition. In consequence, transition activation checks as well as structures formerly needed for the representation of connecting places can be omitted. This principle does not only apply to the transitions originally present in a model, but also to already merged transitions, for instance as a result of choice identification.

Considering for instance transition *T2* of method 'check' of class 'Prime' in Fig. 3. It is clear from the net structure that after the activation of *T2* either *T3* or *T4* is activated. Out of this, the code block representing *T3* and *T4* can be moved into the block representing *T2*. In consequence, place *P4* is not needed any longer, since the integration of the code for transitions *T3* and *T4* into the 'right' branch of *T2* already represents the structural information formerly stored in *P4*. Fig. 10 illustrates the outcome of this merging of transitions for method 'check' of class 'Prime'.

4.5. Removal of simulation environments

If the above steps for the optimization of the simulation-based approach to automatic code generation produces code, in which the structure of the generated commands

```
// Main While Loop
   while (running)
   {
    // Shared Activation Conditions for T3 and T4
       if (P5_used && P4_used)
       {
         if ( P5_value != null )
         {
           // Code for Transition T3
           [...]
         }
         else
         {
           // Code for Transition T4
           [...]
         }
       }
    }

    [...]
```

Fig. 9. Code generated with choice identification.

```
// Main While Loop
   while (running)
   {
    // Shared Activation Conditions for T1 and T2
       if (P1_used && P2_used)
       {
        // Code for Transition T1
           if ( P1_value % P2_value == 0 )
           { [...] }
        // Code for Transition T2
           else
           {
            // Local Transition Variables
               int local_x = P1_value;

            // Empty Input Places
               P1_used = false;
               P2_used = false;

            // Shared Activation Conditions for T3 and T4
               if (P5_used && P4_used)
               {
                // Code for Transition T3
                   if ( P5_value != null )
                   { [...] }
                // Code for Transition T4
                   else
                   { [...] }
               }
           }
    [...]
```

Fig. 10. Code example for merging of transitions.

```
// Code for Transition T1
   if ( P1_value % P2_value == 0 )
   {
    // Local Transition Variables
       int local_x = P1_value;
       int local_p = P2_value;

    // Empty Input Places
       P1_used = false;
       P2_used = false;

    // Write Values to Output Places
       P3_value = null;
       result = P3_value;
   }
// Code for Transition T2
   else
   {
    // Local Transition Variables
       int local_x = P1_value;

    // Empty Input Places
       P1_used = false;
       P2_used = false;

    // Shared Activation Conditions for T3 and T4
       if (P5_used && P4_used) { [...] }
   }

   [...]
```

Fig. 11. Removal of simulation environments.

fully determines the order of execution, further optimizations can be applied. Due to the complete structuring of the generated code, the simulation environment of the naive approach is not needed any longer. In turn, the outermost 'while' loop and the 'running' attribute can be omitted. In case there is an 'if' statement left with only a single branch which wraps a whole method, this statement can be omitted, too. Both optimizations apply to method 'calc' of class 'Sieve' and method 'check' of class 'Prime'.

The outcome of the generation process for the latter is illustrated in Fig. 11. In comparison to the code in Fig. 10, the result of the above optimization steps is better readable and also more efficiently executable.

4.6. Exit transition optimization

The code generated so far does not fully take into account that the execution of a method ends if an activated transition is connected to an exit place. This knowledge can be further utilized to minimize the code generated for such transitions. In detail, the code for exit transitions does not need to contain commands which empty input places and also may immediately end the execution of the method by means of a 'return' statement. As a consequence of the latter, also the 'return' variable introduced with the naive simulation approach is not needed any longer. Fig. 12 illustrates the outcome of this optimization step for transition *T1* in method 'check' of class 'Prime'. In comparison to Fig. 11, the code generated for *T1* is further reduced and also the method ends immediately.

```
// Code for Transition T1
   if ( P1_value % P2_value == 0 )
   {
    // Local Transition Variables
       int local_x = P1_value;
       int local_p = P2_value;

    // Write Values to Output Places
       P3_value = null;
       return (P3_value);
   }
```

Fig. 12. Code generated for *T1* in method 'check' of class 'Prime'.

4.7. Variable substitution

What can be learned from the code in Fig. 12 is that there are declarations of variables local to transition *T1* which are actually not needed at all. Furthermore, the use of the local variable representing place *P3* of the model for the temporary storage of the value to return to the caller of the method can be omitted. In order to get rid of these unnecessary constructs, the code generated for a (possibly merged) transition has to be analyzed with respect to the use of locally declared variables and possible substitutions. This way, the code in Fig. 12 can be reduced to a single line which then simply returns 'null'. In addition to the substitution and omission of variables local to transitions, also the declaration and use of variables representing places of a model can be reduced. Fig. 13 gives the outcome of this processing step for method 'check' of class 'Prime', which largely resembles a manual implementation style.

While it is obvious that the enhancements to the naive simulation-based approach to automatic code generation lead to a more readable and also a more efficient outcome, it is not clear which optimization step contributes in which way to a better efficiency. Also it is not clear as to whether the code generation concepts are suited to match the efficiency of manual implementations. In order to illustrate the contributions of the various optimization steps

```
public Prime check(int x)
{
  if ( x % prime == 0 )
  {
    return (null);
  }
  else
  {
    if ( successor != null )
    {
      return (successor);
    }
    else
    {
      successor = new Prime(x);
      return (null);
    }
  }
}
```

Fig. 13. Code generated for method 'check' of class 'Prime'.

Table 2
Benchmarking results for prime calculation up to 1 million

Approach	Execution time (s)	Factor
Naive simulation	266.29	2.74
Loop detection	182.38	1.88
Choice identification	166.49	1.72
Merging of transition sequences	160.86	1.66
Removal of simulation env.	147.78	1.52
Exit transition optimization	143.92	1.48
Variable substitution	97.17	1
Reference implementation	97.09	1

described above to the overall efficiency of automatically generated systems, the next section focuses on the evaluation of the presented approach.

5. Evaluation of automatic code generation

The calculation of primes used as a running example throughout this article is further used as a basis for the evaluation of the efficiency of simulation-based code generation and the above described enhancements. In detail, a manual reference implementation of the system is evaluated against different versions generated according to the above descriptions. Table 2 gives the absolute execution times as well as the relative efficiency with respect to the manual reference implementation for the calculation of primes up to 1 million with the different versions of the system.[1]

As expected, the code produced with the basic simulation concept was least efficient and thus proved the general characteristics already summarized in Table 1. Loop detection drastically improved performance due to the minimization of transition activation checks for code, which is executed far more often in comparison to code outside of loop structures. Choice identification and the merging of transitions further improved efficiency, but only by smaller margins. In contrast, the removal of simulation environments and 'if' statements enclosing the code for a whole method considerably contributed to the overall performance. While exit transition optimizations contribute not that much, variable substitution finally has the second largest impact on performance. With the utilization of compiler strategies, storage space requirements for method initialization are minimized and also statements not at all contributing to the progress of a method are omitted. As a result, the overall performance of the code generated according to the above description matches the efficiency of the manual reference implementation and is at least partially as readable.

Even if the result of the benchmark proves that automatically generated code from Petri-Net models may be as efficient as manual implementations, this is likely not the case for every model. If for some reason one or more of the different improvements to the basic simulation-based approach to code generation are not applicable, the above benchmark results roughly indicate the performance penalty to be expected. In order to be able to identify additional patterns for the direct translation to programming language primitives, further investigations with more complex case studies are needed. However, in contrast to the structural analysis based approach to code generation, the concepts introduced in this article also work if models are not strictly constructed from predefined structural patterns. The presented approach, in fact, is a hybrid, since the simulation-based concepts for code generation are extended with ideas from the structural analysis based approach. This way, the advantages of both are combined, i.e. the outcome of this combination is generally applicable to arbitrary kinds of models like the simulation-based approach, but produces more efficient and readable code with the help of concepts integrated from the structural analysis based generation.

An interesting train of thought is the additional integration of concepts from reachability graph based code generation into the presented approach. In case one or more of the above improvements do not apply to a given model, the efficiency of generated code could possibly be further enhanced this way. The main idea is that if the code generated for a transition is extended with statements which locally predict the next activated transition, the amount of unnecessary transition activation checks can be further minimized. The order of these checks can be individually arranged for each transition, i.e. the reachability of transitions from the currently activated one determines the order of local transition activation checks. This way, the global activation checks by means of 'if' guarded code blocks, as for instance given in Fig. 6, are exchanged with a 'switch' statement, which *immediately* gives control to the code of the next activated transition. Fig. 14 illustrates

```
// Main While Loop
   while (running)
   {
     switch(nextTransition)
     {
       // Code Generated for Transition T1
       case 1:
       {
         // Operational Code for T1
            [...]

         // Local Activation Prediction
            if (P1_used && !P2_used)
                nextTransition = 2;
            if (P1_used && P3_used)
                nextTransition = 3;

            [...]

            break;
       }

       [...]
```

Fig. 14. Local prediction of transition activation.

[1] For benchmarking purposes a 2.4 GHz Intel Pentium IV machine running Windows XP and Java 1.5 was used.

the structure of code automatically generated based on these ideas.

Even if the local prediction of the next activated transition obviously minimizes transition activation checks, the performance of the generated code is generally *not* improved. Extensive tests with this generation concept revealed that the overhead introduced with the 'switch' statement only leads to a more efficient execution of generated code, if the underlying model contains at least 20 transitions. Since models describing methods in an object-oriented context are only rarely that complex, the integration of reachability graph based concepts to automatic code generation is not beneficial from a performance point of view. Furthermore, also the readability of generated code deteriorates, due to the additional code needed for each transition for the local prediction of the next activation.

6. Conclusions and further perspectives

Due to the model driven architecture initiative of the OMG, automatic code generation is nowadays a topic of major interest. The main theme of this article is code generation from high-level Petri-Nets, which are often used for the semantic definition of UML notations. As a result of the characterization of different strategies for code generation from Petri-Nets, a hybrid approach was presented which combines the advantages of simulation-based generation principles and structural analysis. The hybrid approach in consequence is generally applicable, i.e. the modeller is not restricted to build models from predefined patterns, and also the generated code is efficiently executable as well as, to a certain extent, well readable. The potential of the approach has been illustrated with the results of a benchmark, in which the generated code performed as efficient as a manual reference implementation.

The presented work is part of an ongoing project for the development of an experimental CASE tool (Philippi, 2002). In this context, several prototypical code generators have been built over the last years (George, 1999; von Hutten, 2000; Wojke, 2002). In order to be able to evaluate the approach described in this article with more complex case studies as sketched e.g. in (Philippi, 2001; von Hutten and Philippi, 2001), future work includes the redesign of the CASE tool as well as the underlying code generation engine (Philippi et al., 2005). Further topics of interest are the mapping of UML notations to the object-oriented class of Petri-Nets introduced in this article as well as the interactive display of model structures which cannot be translated to efficiently executable code. With such a feature the developer could be hinted to potential problems and decide if modifications in specific parts of a model are worth the effort from a performance point of view. Since this article covered automatic generation of efficiently executable sequential code, a natural extension of the presented approach is the generation of efficiently executable concurrent systems.

Acknowledgement

The author would like to thank Thomas George, Pascal von Hutten and Philipp Wojke for their contributions to the NEPTUN project.

References

Arnold, K., Gosling, J., 1996. The Java Programming Language. Addison-Wesley.

Baresi, L., Pezzè, M., 2001. Improving UML with Petri nets. In: Electronic Notes in Theoretical Computer Science, vol. 44. Elsevier.

Bernardi, S., Donatelli, S., Merseguer, J., 2002. From UML sequence diagrams and statecharts to analysable Petri net models. In: Proceedings of the 3rd International Workshop on Software and Performance. ACM Press, pp. 35–45.

Christensen, S., Jorgensen, J.B., Kristensen, L.M., 1997. Design/CPN—A computer tool for coloured Petri nets. In: TACAS'97—Tools and Algorithms for the Construction and Analysis of SystemsLNCS, vol. 1217. Springer, Berlin.

Genrich, H.J., Lautenbach, K., 1981. System modelling with high-level Petri nets. Theoretical Computer Science 13 (1), 109–136.

George, T., 1999. OOPr/T-modeller: a Tool for the modelling of object-oriented systems based on UML and Petri-Nets. Diploma Thesis, University of Koblenz (in German).

Girault, C., Valk, R., 2003. Petri-Nets for Systems Engineering. Springer, Berlin.

Kummer, O., Wienberg, F., Duvigneau, M., 2004. Renew—User Guide, Release 2.0. www.renew.de.

Lautenbach, K., 1987. Linear algebraic techniques for place/transition nets. In: Petri Nets, Central Models and their Properties. LNCS, vol. 254. Springer, Berlin.

Lee, G., Zandong, H., Lee, J., 2004. Automatic generation of ladder diagram with control Petri Nets. Journal of Intelligent Manufacturing 15 (2).

Object Management Group (OMG), 2003. MDA guide version 1.0.1. www.omg.org/mda.

Object Management Group (OMG), 2004. 'UML specification 2.0'. www.omg.org/uml.

Philippi, S., 2000. Seamless object-oriented software development on a formal base. In: Workshop on Software Engineering and Petri-Nets, 2000, Aarhus, Denmark.

Philippi, S., 2001. Visual programming of concurrent object-oriented systems. Journal of Visual Languages and Computing 12 (2).

Philippi, S., 2002. A CASE—tool for the development of concurrent object-oriented systems based on Petri-nets. Petri-Net Newsletter 62, 9–22.

Philippi, S., Pinl, A., Rausch, G., 2005. A first view on a generalised modelling toolkit for graph-based languages. In: Proceedings of the 12th Workshop on Algorithms and Tools for Petri-Nets. Humboldt-University, Berlin, Germany.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., 1991. Object-oriented Modeling and Design. Prentice Hall International.

Uzam, M., Jones, A.H., 1998. Discrete event control system design using automation Petri nets and their ladder diagram implementation. International Journal of Advanced Manufacturing Systems 14 (10), 716–728 (Special Issue on Petri Nets Applications in Manufacturing Systems).

von Hutten, P., 2000. Modelling a ray tracing system with OOPr/T-models. Diploma thesis, University of Koblenz (in German).

von Hutten, P., Philippi, S., 2001. Modeling a concurrent ray-tracing algorithm using object-oriented Petri-nets. Technical Report 01-2001, University of Koblenz.

Wojke, P., 2002. Generation of efficient Java code from visual models in the NEPTUN CASE Tool. Diploma thesis, University of Koblenz (in German).