Towards Automatic Code-generation from Process-partitioned Coloured Petri Nets

K.L. Espensen[1], M.K. Kjeldsen[1], L.M. Kristensen[2], and M. Westergaard[1*]

[1] Computer Science Department, Aarhus University, Denmark.
Email: {espensen,keblov,mw}@cs.au.dk
[2] Department of Computer Engineering, Bergen University College, Norway.
Email: lmkr@hib.no

Abstract. Constructing an abstract description in the form of a model can give useful insight into a given system, e.g., to investigate important properties of the system either through simulation or state space analysis, and to use the model as inspiration for subsequent manual implementation. The problem is that a manual implementation may introduce errors in the code not present in the model. Automatic code generation from the model saves resources spent on writing code, and eliminates errors introduced during implementation. This is difficult for coloured Petri nets models as their rich structure translates badly to common programming languages. Approaches either severely restrict the input accepted or generate code that is difficult to extend and modify. In this paper we introduce process-partitioned coloured Petri nets, which is an attempt to restrict the input accepted as little as possible while still allowing automatic inference of the control structure of the model to generate code that can be manually modified afterwards. We illustrate our approach using a simple example and demonstrate the viability of the approach by demonstrating that it can be applied to a model of a real-life system, the Dynamic MANET On-demand (DYMO) routing protocol.

1 Introduction

Software development is a challenging process, and writing a program of substantial size without errors is difficult. A major part of software development is therefore concerned with finding and eliminating errors. Testing is widely used as a technique to detect errors, but the programmer does not know whether the absence of failed test cases means a missing test case or that the software is free of errors. It is especially difficult to write exhaustive test cases for concurrent systems, e.g., for a communication protocol where several process instances are executing at the same time.

Building an abstract representation of the system in the form of a *model* is a way to detect errors early. A model can be used to verify properties of a system, e.g., that a system does not contain deadlocks, or that a communication protocol behaves correct when operating over an unreliable network. A typical

*Supported by the Danish Research Council for Technology and Production.

way of using models in software development is to build a model from a system specification written in plain text. After verifying that the model has the desired properties, it can be used as a basis for an implementation. A problem with this approach is that there may be a mismatch between the specification, the model, and the actual implementation. This is because the translation from one to another is done manually and hence can introduce errors. A way to reduce this problem is to use the model as the specification and automatically generate the implementation from the model. Details abstracted away in the model will of course also lack in the implementation, but eliminating errors in the verified parts of the system leads to more reliable software with fewer errors.

The aim of this paper is to develop a technique to automatically generate code from coloured Petri nets (CPNs or CP-nets) [7]. The code should be readable and intuitive such that the user can read, modify and extend the generated code. We also want the model to be clearly recognizable in the generated code since the people working with the generated code are typically also familiar with the model. The technique should allow different target languages to be used, e.g., C, Java, SML or Erlang [3]. However, the target language should be invisible in the model and the usual inscription language should be used in the model.

To achieve this aim, we use a sub-class of CP-nets called process-partitioned CP-nets (PCP-nets or PCPNs). PCPN models preserve much of the general-purpose strength of CP-nets as we show by constructing a model of the Dynamic MANET On-demand (DYMO) protocol [1]. We have developed a technique that translates from the class of PCP-nets to the Erlang programming language, and have created a prototype of the technique. The prototype is able to generate readable code from the DYMO model, and we validate that the generated code has the same behaviour as the model.

Related Work. There are different approaches to automatically generate code from Petri nets. The chosen strategy has a large impact on the properties of the final code. The approach should preserve the behaviour of the model, but the code generated using one approach might be very efficient while the code generated by using another approach may be very readable and extensible.

In [6] and [13] approaches to automatic code generation is divided into the four categories *simulation-based*, *structure-based*, *state space-based* and *decentralised*. Our approach falls into the structure-based approach.

Simulation-based. The basic idea in simulation-based approaches is to have a central component which controls the flow of the program on the basis of the state of the environment. This is done by a scheduler which from the current state computes which state to proceed to. This process corresponds to finding enabled transitions in CPN models.

A simulation-based approach is used by Philippi to generate Java code from a high-level Petri net in [13]. The idea is to make a class diagram which outlines the classes and method signatures of the program. From this diagram, classes with attribute definitions and methods with empty bodies are generated. The empty bodies are filled with the simulator code made from the formal model.

Simulation-based approaches are also used in the projects described in [12] and [8] where the generated simulator code made from a CPN model (by CPN Tools [2]) is used directly in the final implementation. The simulation kernel is generated from a CPN model and after undergoing automatic modifications, e.g., linking the code to external code libraries, the generated code is used in the final implementations.

One advantage of simulation-based approaches is that code execution follows a simulation of the model very closely, making it easier to establish that the behaviour of the generated code is the same as the behaviour of the model. Naturally, such approaches do not put any limitations on the class of nets to generate code for. The main disadvantage of these approaches is that the generated code is not very natural and often inefficient.

Structure-based. The code generated using a structure-based approach contains no central component to control the execution of the program. Instead the control flow of the program is distributed across the program, e.g., to function calls in functional programming languages. The key idea of these approaches is to recognise *structure* (regular patterns) in the model. Structure is then mapped to well-known programming constructs like sequences, loops, and case constructs. It is not in general possible to recognize such structure in coloured Petri nets mainly because they provide much more opportunities for constructing different control flow structures than common programming languages [13]. Because of this, it is necessary to restrict the class of nets when using a structure-based approach.

A structural approach is found in [6]. In this approach the focus is on identifying processes in a Petri net, i.e., parts of the net that work independent of one other or only have few synchronisation points. Afterwards local variables (i.e., information only used by one process) and communication channels are found.

In [14] the authors translate a class of CP-nets, called coloured workflow nets (CWNs), into BPEL, an XML-based workflow implementation language. CWNs are quite restricted, and mainly focus on the flow of data and not much on data processing, making the approach basically a graphical way to describe control structure instead of a natural way to make CPN models. Furthermore, the BPEL language is not aimed at general application development. [10] improves on this by translating directly to Java by adding a data processing component, but it is very restricted and does not allow the use of general functions in the data processing part. Furthermore, the approach is limited to emitting Java code.

The advantage of using structure-based approaches is that the code obtained is more readable than code obtained with a simulation-based approach. The coding style is more natural and looks more like it is written by a human programmer. The generated code also has a tendency to be more efficient because it does not rely on a central component. The main disadvantage is that the requirements on the modeling language may make the models unnatural.

State space-based. The idea of state space based approaches is to use the state space of the model to compute the next state. In the state space, we have all

successor states computed for each reachable state which alleviate the overhead of computing the successors each time. Relying on the full state space to be generated is a huge drawback because of the state space explosion problem, and therefore we do not find this method worth pursuing.

Decentralised. The opposite of centralised simulation-based approaches are decentralised approaches. The idea is to implement each place and transition of the net as processes. Here the program does not directly reflect the structure or state of the system. This approach has the advantage that parallelism in the net is preserved, but it also introduces an overhead because of the administration needed, e.g., for locks and message passing.

The rest of this paper is structured as follows: In the next section, we introduce our net class, process-partitioned CP-nets via a simple example. In Sect. 3, we describe our translation algorithm, and in Sect. 4 we describe our experiences with application of our prototype to a model of a real-life protocol made before the definition of the net class. Finally, we sum up our conclusions and provide directions for future work. Part of this work has been published as [4]. The main change in this version is that the presentation has been improved and shortened.

2 Process-partitioned Coloured Petri Nets

We use a simple producer-consumer system as example. The example can be seen in Fig. 1. The system consists of a number of producers that produce data and send it to the consumers (the top part of the model), and a number of consumers consuming the data (the bottom part of the model). Producing data is split up into producing the data and transmitting the data to the consumers. The producers have local Data, which contains the next data value to produce. When a data item is produced, it is transmitted to Produced Data for transmission. Each producer sends its data to a specific consumer, getting the identity of the consumer from the place Next Consumer, and transmitting it onto Buffer. Currently the identity of the receiving consumer is hard-coded to consumer c(1), but we can easily replace this by a load balancer. Consumers receive data from Buffer, which is a simple model of a network, transmit it to the Received Data place, where it will be consumed.

A general coloured Petri net is not limited to regular control flow structures in the same way common programming languages are. For this reason it is not easy to capture the behaviour of a CPN model by using common programming constructs, e.g., sequences, loops, and case-statements. We note that while the model is indeed a CPN model, it is modelled slightly differently from how one would normally go about it. Most notably, we see that we always both consume and produce tokens on places with data in the name and that we explicitly bind the consumer on the Send Data transition. This is because the model is created using the sub-class *process-partitioned coloured Petri nets* (PCPNs or PCP-nets). PCPNs are defined in a way that makes it possible to recognise

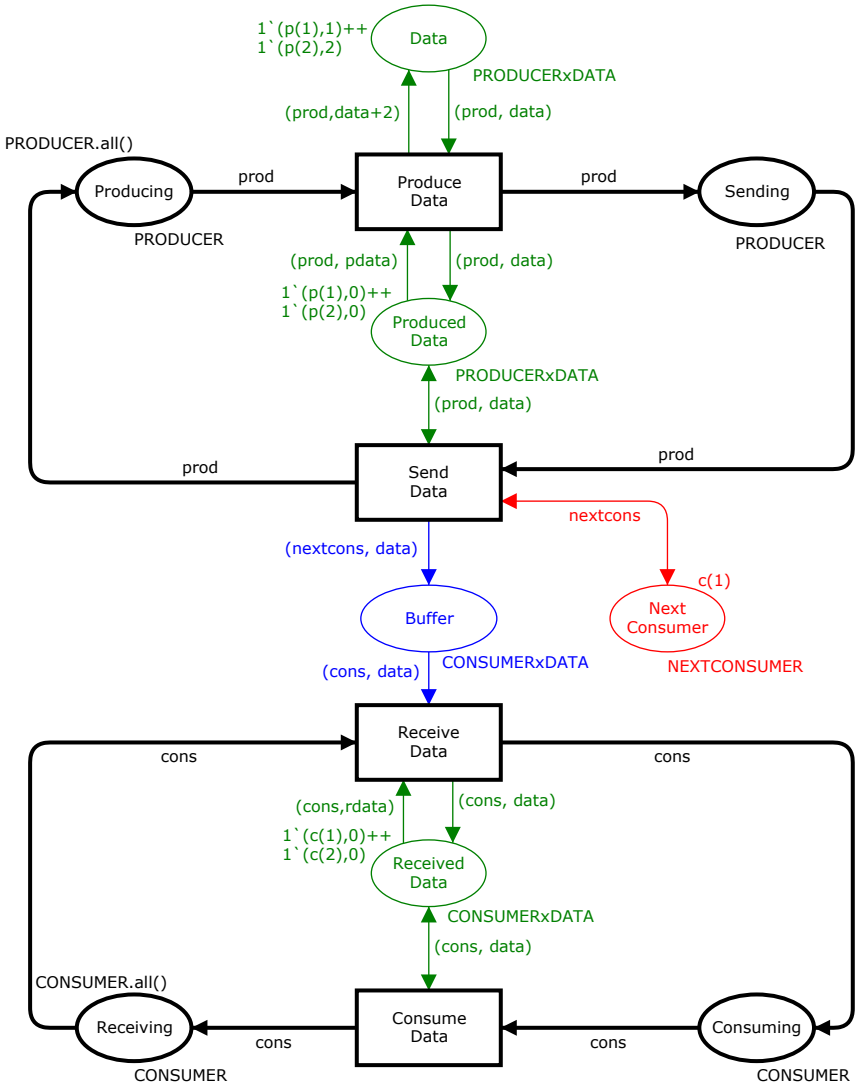

Fig. 1: The producer-consumer CPN model.

control structures and thus to generate code from the model. The definition of PCP-nets is inspired by the definition by Kristensen and Valmari from [9].

A main property of PCP-nets is that they are partitioned into processes that can be executed in parallel without influencing the behaviour of each other except for distinguished synchronisation points. Another important property is

that the control flows of processes are explicit in the net structure, so the state of the model always reflects where the process is in the control flow. Furthermore, access to stored values local to each process partition is also explicit in the model, allowing us to determine the local state of processes.

Here we introduce PCP-nets using the producer-consumer example; for a formal definition, please refer to [4]. The model has two *process partitions*, one modelling producers (top) and one modelling consumers (bottom). Process partitions can be connected by either *buffer* or *shared* places, but are otherwise disjoint. In Fig. 1, the producers and consumers are connected only by the buffer place Buffer. Intuitively, a process partition models the state and actions of one or more *process instances* running the same program code, e.g., producer process partition models two producer process instances running the same program code in the example. Transitions in a PCP-net belong to a unique process partition, e.g., the transition Send Data in Fig. 1 belongs to the producer process partition.

There are four kinds of places in PCP-nets: *process places*, *local places*, *buffer places* and *shared places*. In Fig. 1, process places are black (Producing, Sending, Receiving, and Consuming) and represent the control flow of processes. Process places have distinguished *process types*, here PRODUCER or CONSUMER, and we impose the restriction that every token from a process type, a *process token*, must reside on exactly one process place. We ensure this by requiring that transitions are always connected to exactly one input and one output process place and that the arc expression must be a variable (allowing a double arc instead of two arcs with the same inscription). We call the variable used the *process variable* of the transition. We require that initially, all process tokens of a given type reside on the same place, corresponding to all processes starting at the same point. For example, initially all producer tokens reside on Producing.

Local places in Fig. 1 are green (Data, Produced Data, and Received Data), and represent variables local to a process. We require that local places have a type that is a product between a process type and a data type. For example, Data has type PRODUCERxDATA, the product of PRODUCER and DATA. We require that if a transition has an arc from a local place, it must also have an arc leading to the place (and vice versa), arc expressions must be pairs where the first component is the process variable of the transition, and each local place must initially have exactly one token for each process token (together ensuring that local places always have exactly one value for each process instance). Finally, the second component of the expression on the arc from a local place must always be a variable only bound on that arc (this ensures that reading a variable from a local place never disables a transition).

Buffer places are blue in Fig. 1 (Buffer) and represent a communication channel between two processes. Like local places, the type of a buffer place must be a product of process type and a data types. Buffer places may contain any number of tokens, but the initial marking is required to be an empty multi-set (corresponding to the communication channel containing no data). Buffer places are allowed to have any number of arcs as long as outgoing arcs have expressions that are pairs of the process variable and an otherwise free data variable (like

for local places), but we impose no special requirements on arcs going into buffer places.

Shared places are red in Fig. 1 (Next Consumer) and represent data shared between multiple processes, corresponding to shared memory. Shared places can have any type that is not a process type (which is why we use NEXTCONSUMER on Next Consumer instead of just CONSUMER). The reason for that is to be able to distinguish shared places from the other kinds of places. We require that a shared place has an initial marking of size one (corresponding to the variable having exactly one value), and we preserve this by always requiring that any transition with an arc from a shared place also has an arc to the shared place.

We require that all arc expressions evaluate to multi-sets of size one to preserve the flow in process partitions. We also require that except for process variables, all variables exist at most once in all expressions on input arcs around one transition. This is to make the enabling calculation simpler in the generated code. It is still possible to make equality tests in the guards, however. We do not allow free variables on output arcs or in guards, as this would correspond to drawing random numbers in programs. Randomness can still be introduced by explicitly calling a random number generator. Variables used in the guard must be bound from local places, i.e., we do not allow input from shared or buffer places, as this would introduce race conditions as we shall discuss later.

3 Translation Algorithm

In this section, we explain the techniques developed for translating a PCPN model into program source code. The producer-consumer system is used to illustrate each phase of the translation. The translation from PCPN models to the target language is divided into five phases. The idea is to move closer and closer to the target language in small steps. Figure 2 illustrates the phases of the translation. The three first phases (top) are independent of the target language, i.e., they make no assumptions about the target language. The first phase consists of decorating the different parts of the PCPN model to allow us to distinguish, e.g., process places and shared places. The second phase translates from the decorated PCPN model into a control flow graph (CFG) for each process partition, extracting the control flow from the model. In the third phase the CFG is translated into an abstract syntax tree (AST) for a simple language designed to be abstract enough that it can be translated into most programming languages. The control flow represented by the CFG is made explicit by, e.g., goto statements in the AST. The last two phases of the translation are shown at the bottom of Fig. 2. These are language dependent, i.e., the phases are specific to a target programming language. We have shown two possible target languages: Erlang and Java. In case of Erlang, the AST is translated into an Erlang syntax tree (EST), mapping the generic concepts of the AST to language specific constructs for the Erlang language. In case of JAVA, the AST is translated into a Java syntax tree (JST) and then into Java source code. The last phase pretty-prints a syntax tree for the concrete target language under consideration.

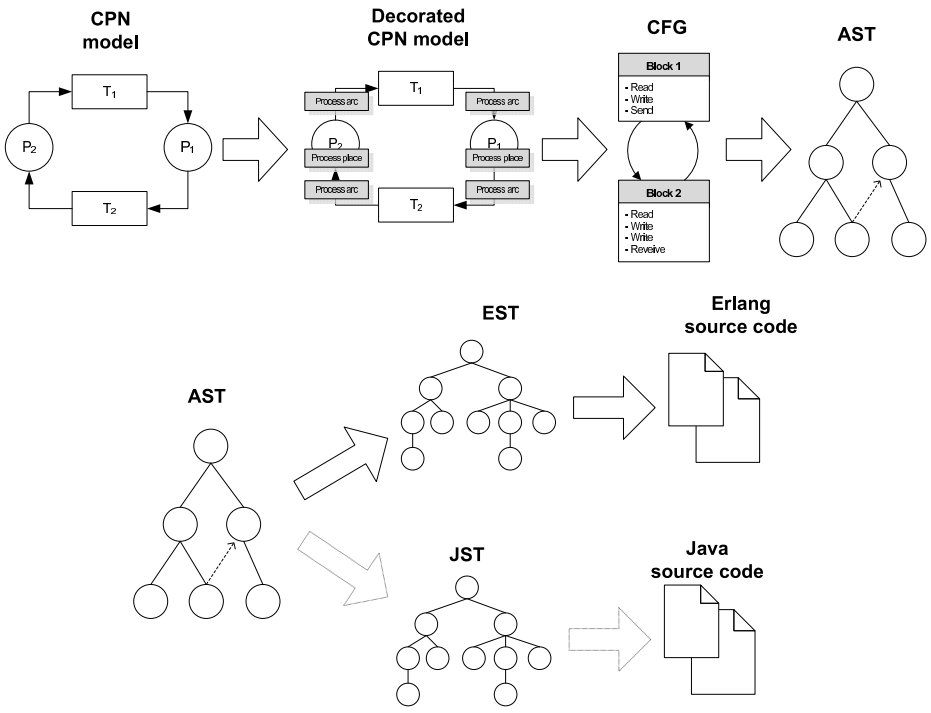

Fig. 2: The first three phases (top) and last two (bottom) of the translation.

Phase 1: Decorating the PCPN Model. The purpose of this phase is to identify different parts of the PCPN model and decorate them with information such as whether, e.g., places are shared places or process places. This phase uses properties of the PCPN net class to perform the identification. In our prototype, we actually perform this step, but it is likely that one would create an editor that would automatically provide this information, and ensure that the constructed model is indeed a PCPN model. For this reason, we will not give the full details of the decoration, but refer the interested reader to [4]. The main idea is that a user must identify process types, allowing us to find the process partitions. Local places can be identified as places only connected to transitions from a single partition with types that are products of the correct process type (that of the process variables of connected transitions) and another type. Buffer places can be recognized as places connected to multiple process partitions, and finally shared places are identified as places whose type is not a product containing a process type.

Phase 2: Translating the Decorated PCPN Model to a CFG. The main purpose of this phase is to extract the control flow from the decorated PCPN model and make it explicit in a control flow graph (CFG). This phase

also identifies common program constructs, e.g., processes, variables, and access to variables. Furthermore, the phase finds synchronisation points, i.e., messages passing between processes. The CFG we use is a directed graph in which arcs correspond to control flow and nodes corresponds to a sequence of statements to be executed. A CFG is constructed for all process partitions in the model, thus in the producer-consumer system two CFGs are generated: one for the producer process partition and one for the consumer process partition. In Fig. 3, we see the translated CFG for the producer process partition. Transitions are translated into *basic blocks* in the CFG. In the producer process partition, the transition Produce Data is translated into the basic block `produce data`. The contents of basic blocks depends on the places connected to the transition. There is a special `start` basic block for each CFG representing the start point of the process. In Fig. 3 it points to the basic block `produce data` as all tokens initially are on the Producing place, which is input to Produce Data.

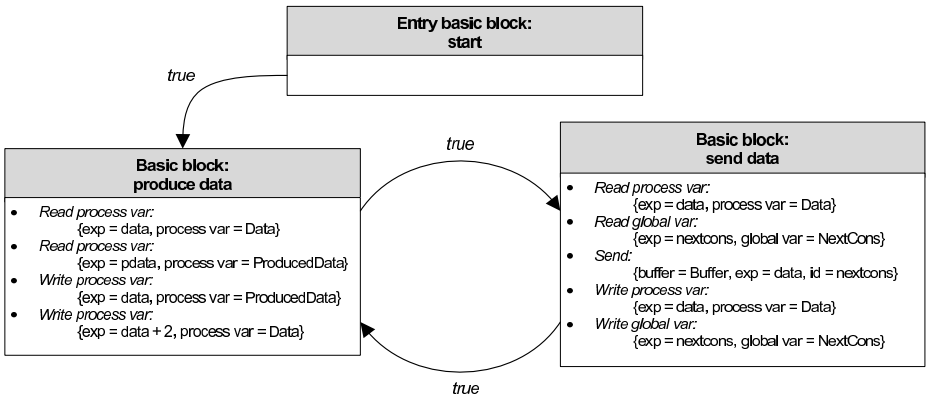

Fig. 3: The CFG of the producer process.

Process places in the PCPN model are not explicitly translated into nodes in the CFG, but are represented as edges between basic blocks. In Fig. 3 we see that the basic block `produce data` has an edge to `send data` signifying that after executing `produce data` control should flow to the basic block `send data`. The edge condition `true` indicates that control flows unconditionally.

Local places are used to store data. In a programming language this corresponds to reading/writing a process local variable. The producer process partition contains the two process variables `Data` and `ProducedData` corresponding to the two local places Data and Produced Data in the PCPN model. The initial values for the variables are extracted from the initial markings of the corresponding places. The initial marking of the local place Produced Data contains 0 for both instances and thus this is the initial value of this variable for both instances. An input arc with an arc expression on the form `(pid, var)` from a local place to a transition corresponds to process instance `pid` reading a vari-

able `var`. The input arc with expression `(prod, data)` from Data to Produce Data is translated to *Read process var* with the expression `data` as shown in Fig. 3. Output arcs corresponds to *Write process var*. `(prod, data)` on the arc from Produce Data to Produced Data is translated to *Write process var* with the expression `data`.

Buffer places are used to send data to a particular process instance. In a programming language this corresponds to sending to and receiving from a shared buffer, thus a buffer place is translated into a *buffer* in the CFG. Input arcs from a buffer place to a transition with arc expression `(pid, var)` correspond to a process `pid` receiving a message which is put into a variable `var`. This kind of input arc can be found in the consumer part of the producer-consumer system on the arc from Buffer to Receive Data (with expression `(cons, data)`) and is translated into *Receive* with the expression `data` meaning that the value of the received data should be read into the variable `data` for later use. Output arcs with expression `(pid, exp)` correspond to sending a value `exp` to a process instance `pid`. The output arc expression `(nextcons, data)` from Send Data to Buffer is translated to *Send* the expression `data` to the receiver process instance `nextcons`.

Shared places in the PCPN model are used to share data between multiple process instances. In a programming language this corresponds to a global variable, e.g., in shared memory. Shared places are translated into *global* variables in the CFG. In the producer-consumer system there is one global variable corresponding to the shared place Next Consumer. Initial markings of shared places are extracted and used as initial values. Input arcs with arc expression `var` from a shared place to a transition corresponds to a process reading a variable with a global scope. In the producer-consumer system, there is an input arc expression `nextcons` to the transition SendData from the shared place Next Consumer. This is translated to a *Read global var*. In a similar fashion, outgoing arcs are translated to *Write global var*.

Phase 3: Translating the CFG to an AST. The main purpose of this phase is to take the control flow given in the structure of the control flow graph (CFG) and translate it into a tree form consisting of nodes representing common programming constructs, e.g., jump statements. Furthermore, read and write expressions contained in the CFG are parsed and translated into trees in order to make them independent of the inscription language.

Figure 4 shows a sub-tree of the AST for the producer-consumer example where only the nodes from the `produce data` block of the `producer` process are shown. When building the AST, a process is created for each CFG process. Figure 4 shows the program contains two processes, namely `producer` and `consumer`. The program node also contains the global variable `NextConsumer`. In the CFG each process contains a number of variables. These variables are translated into process variables and contain an initial expression for each instance of the process. Expressions are parsed using a simplistic SML parser. An AST process also has a number of AST blocks which are created from the basic

blocks of the CFG. The `producer` process contains three such blocks: `produce data`, `send data`, and `start`. The AST contains nodes for reading and writing process and global variables as well as for sending and receiving via buffers. The rightmost node in the `produce data` block is an *unconditional goto statement* containing a pointer to the block it jumps to, in this case the `send data` block. Jump statements are translated from edges between basic blocks in the CFG. An AST can have two types of goto statements, a unconditional jump and a conditional jump with a boolean expression that must evaluate to true for the jump to take place.

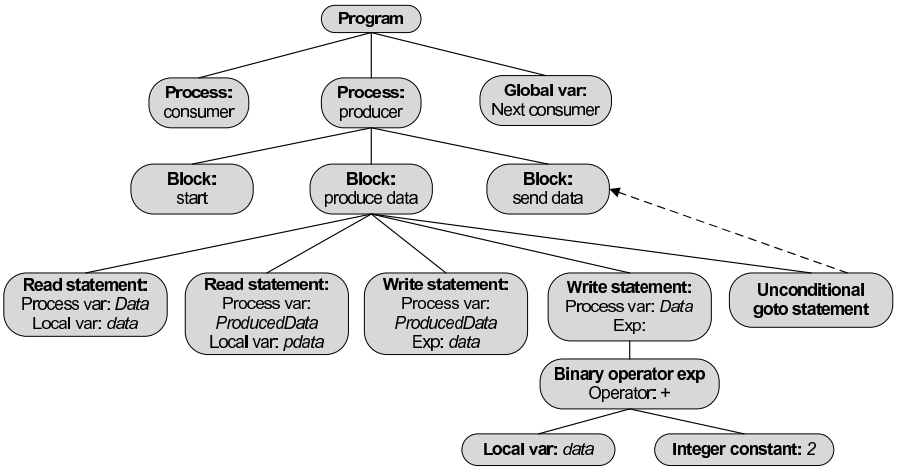

Fig. 4: The AST for the "produce data" part of the producer.

The AST can either be used as is or we can further transform it by extending the abstract syntax with, e.g., loops, if statements, or exceptions. These structured programming paradigms must be inferred from the control flow of the program. For example, we can identify an infinite loop in the producers of producing and transmitting. By doing this at the AST level, any new kind of control structure recognized can be used by all target languages.

Phase 4: Translating the AST to an EST. This phase generates an Erlang syntax tree (EST) based on an AST. The purpose of this phase is to translate the abstract representation of a program into an Erlang program represented as a tree. We have chosen Erlang as implementation because it is a functional language not too far from SML and it has been developed specifically for handling concurrent and critical systems, such as telephone centrals. The control flow, represented by goto statements in the AST, is translated into the functional language paradigm equivalent *function calls*. Since functional languages are stateless the state is passed along with the function calls. Processes are na-

tive in the Erlang language, thus each process in the AST is simply translated into a module. The generated modules are spawned in a special *system* module.

To give an impression of the translation from an AST to an EST we take a look at the producer-consumer system. Figure 5 shows how a part of the producer is represented in the generated EST (top) and the details of `produce_data` (bottom). The `producer` module basically consists of a function, `start` which is exported to the environment to allow the environment to start up processes, a declaration of the environment of the process containing all process variables, and functions for each basic block. Process variables are represented using an environment in the form of a record, so reading and writing translates to record operations. Global variables in the AST are used to share data between processes. There is no native equivalent to global variables in the Erlang language, so instead we construct a module which can be used to spawn processes that acts like global variables. Receiving messages via a buffer is a native construct in the Erlang language. To handle the translation of more advanced control flow constructs, a buffer with extra functionality is needed. For this reason we construct an explicit `buffer` process for each receiver to receive messages. *Goto* statements in the AST are divided into *unconditional* and *conditional* jumps.

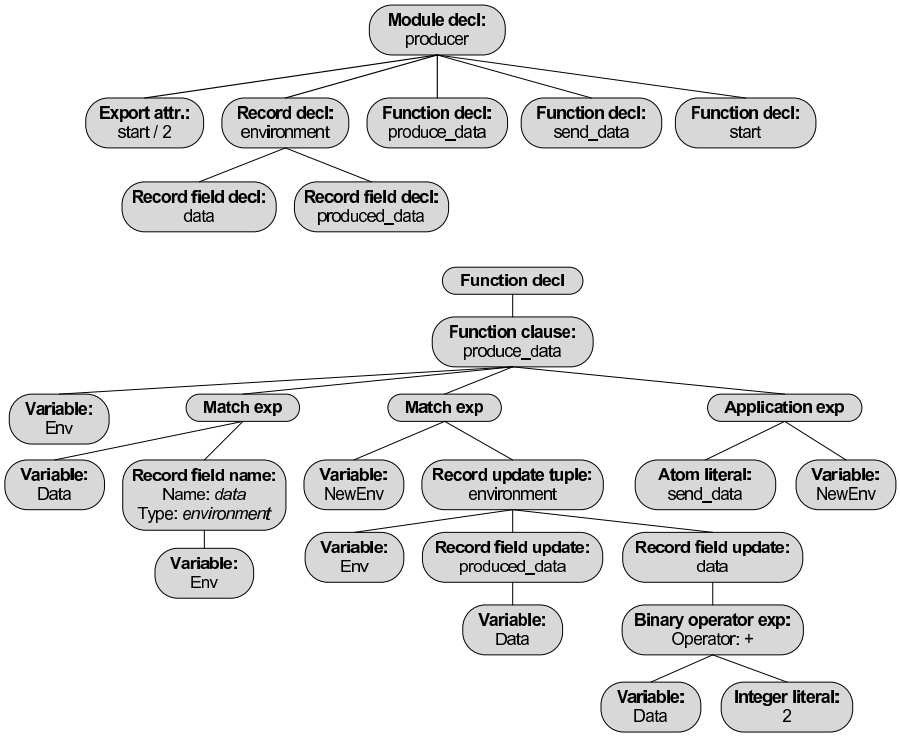

Fig. 5: The producer module (top) and function declaration `produce_data` (bottom) of the EST.

Unconditional goto statements are simply translated into function application expressions in the EST. In the example, we see an application of `send_data` at the far right.

We have a dedicated module to spawn an Erlang process for each process instance. It also spawns `buffer` processes and a process for each global variable. In the producer-consumer system this sums up to: two producers, two consumers, two buffers belonging to the consumers, and a shared instance for `NextConsumer`.

Phase 5: Translating the EST to Erlang Code. The last phase is translating the Erlang syntax tree (EST) into a textual representation. The EST is a concrete representation of Erlang, so the task is to traverse the tree and print a textual representation of each node to a file. A traversal is made for each module declaration because they represent different output files. Listing 1 shows a part of the generated Erlang code for the producer module. In Fig. 5 at the top we see the EST of the module declaration and at the bottom, we see the `produce_data` function declaration. We first declare our module (l. 1) and export the `start` function (l. 2). We then define our environment type (ll. 3–5) and move on to the definition of the `produce_data` function (l. 7–11) starting by reading `Data` from the environment (l. 8) and creating a new environment with updated values of the variables (ll. 9–10). Finally, we hand over control to the `send_data` function with the new environment (l. 11).

Listing 1: Part of the generated code for the producer module

```
1  - module(producer).
2  - export([start/2]).
3  - record(environment, {
4    produced_data,
5    data}).

7  produce_data(Env) ->
8    Data = Env#environment.data,
9    NewEnv = Env#environment {produced_data = Data,
10                             data = Data + 2},
11   send_data(NewEnv).
```

Advanced Control Flow Issues. While covering most of the constructs found in PCP-nets, the producer-consumer model does not contain a branch of the control flow. Here, we describe how branches of control flow are handled in the translation. In the producer-consumer model process tokens residing on a process place are only available to a single transition but the definition allows process tokens to be available to multiple transitions, i.e., a control flow branch. Control flow branches introduce an additional challenge when the target transitions have input arcs from buffer places. These buffer places have to be taken into consideration when choosing the flow of control. Making a function call without looking at the buffer may introduce a deadlock in the program that did not exist in the model. In Fig. 6 we see a part of a PCPN model with one process place Process Place, two transitions T1 and T2, and two buffer places Buffer1 and Buffer2. The process token can be removed by either T1 or T2, and

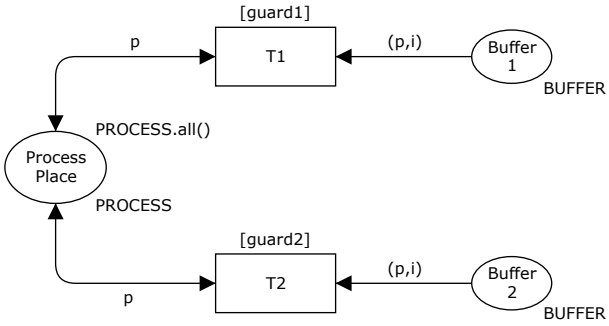

Fig. 6: A process partition with a control flow branch.

are in both cases put back on Process Place. T1 is enabled if guard1 evaluates to `true` and there is a token on Buffer1, and analogously for T2. This means that the generated process can proceed to either T1 or T2 depending not only on the guards but also on presence of tokens on buffer places.

Translation to a CFG. Given the PCPN model shown in Fig. 6, we generate the CFG shown in Fig. 7 (top). It contains an entry basic block `start` which has an edge with condition `guard1` to the basic block `T1`, and an edge with condition `guard2` to the basic block `T2`. The flow of control from `T1` is either to itself or to `T2` depending on the value of `guard1` and `guard2`, and analogously for `T2`. `T1` contains a receive statement from `Buffer1` and `T2` contains a receive statement from `Buffer2`.

Translation to an AST. The CFG is translated to the AST shown in Fig. 7 (bottom). The `Process` node contains two blocks `T1` and `T2`. Taking a look at `T1` it contains a receive statement which has a pointer to `Buffer1` where the incoming messages are stored. The receive statement also contains a local variable `i` into which a message from the buffer is read. `T1` also contains two conditional statements; one holding the condition expression `guard1` and pointing to `T1`, and one holding the condition expression `guard2` and pointing to `T2`. The block `T2` is similar to `T1`.

Translation to Erlang Source Code. The translation becomes more complex when allowing branches. Jumping to the first block were the guard evaluates to `true` could introduce a deadlock in the program if that block contains a receive statement from a buffer that will never have an element added. For instance, in the PCPN model shown in Fig. 6, it could be the case that both guard1 and guard2 evaluates to `true`. Assume that Buffer1 is empty, and that there will never be added a token to it. Assume also that Buffer2 already contains a token. If the program was to jump to the function corresponding to the transition T1 the program would stop on the blocking receive expression. This is not desirable

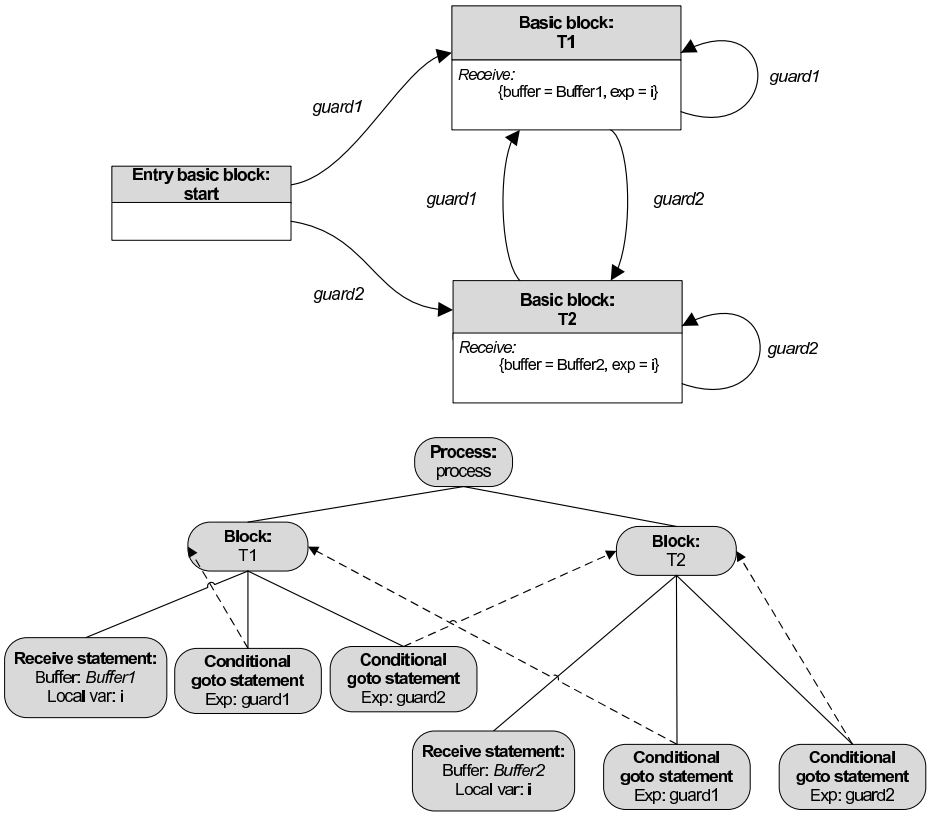

Fig. 7: The CFG (top) and AST (bottom) of the control flow branch.

since T2 is enabled in the PCPN model, thus calling the function corresponding to T2, would not make the program stop.

A solution to this problem is to only jump to a function with a receive expression if there is an element in the buffer. Since buffers are local to a process instance, the element will remain in the buffer until removed by the process instance. For this reason, we have introduced an explicit Erlang buffer module with operations for checking if an element is available. An example of this can be seen for our industrial example in Listing 2 in the next section.

4 Using the Method on a Real-life Example

We have used our prototype to generate code for the producer-consumer example, part of which is shown in Listing 1. We have validated this code by manual inspection and by adding logging code verifying that it is possible to reproduce runs of the generated code in the model.

We have also run our prototype on a model of the real-life dynamic on-demand MANET routing protocol (DYMO) [1]. The first versions of our model of this protocol were made before starting this work and the translation effort required to get running code is therefore realistic for other projects as well. Earlier versions of the model have been used to find and fix problems in the protocol specification [5]. The protocol is responsible for establishing routes in a *mobile ad-hoc network* (MANETs), i.e., a network with no preexisting infrastructure. The protocol establishes routes on-demand, i.e., when they are actually needed. The model models the two parts *route discovery* and *route maintenance* of DYMO. Route discovery establishes routes by multicasting *route request* messages. Each intermediate node records a route back to the originator and forwards the message. When the request arrives at the destination, a route is established and used to unicast a *route reply* back to the originator. Route maintenance actively monitors links and uses timeouts to discover loss of connection, and when this happens a *route error* is multicasted to all neighbours.

The CPN model of DYMO has been translated into PCPN from the earlier versions in approximately 20 man hours and consists of 8 modules, 49 places, and 18 transitions, and is thus fairly complex. We have shown part of the route discovery in Fig. 8. The module is responsible for initiating route discovery. We have used the same color conventions as in Fig. 1, i.e., process places are black, local places are green, shared places are red, and buffer places are blue. The module can either create a new route request or cancel the request if we have retried too many times.

Generating code from the DYMO model yields the modules shown in Table 1. We have listed lines of code (L.O.C.) for each module – in total we generated 563

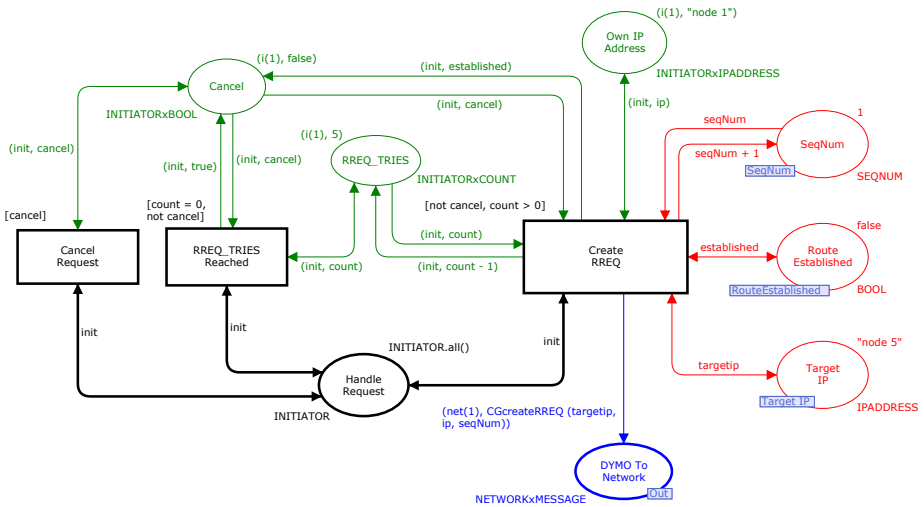

Fig. 8: The Initiator module of the PCPN DYMO model.

Table 1: Generated Modules from the DYMO Protocol.

Module name	L.O.C.	Functions to implement
system.erl	20	0
buffer.erl	36	0
shared.erl	16	0
initiator.erl	116	1
receiver.erl	116	7
processer.erl	111	4
establishchecker.erl	126	0
network.erl	22	0
Total	563	12

lines of code. Since we do not support automatic translation from SML to Erlang, we had to manually implement various Erlang expressions and functions on the basis of the corresponding SML code. These SML expressions are carried along as comments in the generated code. The comments are placed where the expression should be used, thus the structure of the program is preserved. Implementing the functions (12 in total) in Erlang is an easy task, because of the similarity between Erlang and SML. In total, we spent approximately 12 man hours modifying the generated code, including removal of unused extracted values. The time spent could be eliminated by implementing automatic translation from SML to our AST and adding rudimentary liveness calculation for variables.

In Listing 2, we see the generated code for the Create RREQ from the Initiator module. This code is pretty typical for the generated code as it starts by gathering parameters, then performs a calculation, distributes results, and finally passes on control. The code consists of picking the needed values of the environ-

Listing 2: Generated code for the Create RREQ transition of the Initiator module.

```
1  create_rreq(Env) ->
2      Count = Env#environment.rreq_tries,
3      Cancel = Env#environment.cancel,
4      Ip = Env#environment.own_ip_address,
5      is_route_established ! {get, self()}, receive Established -> Established end,
6      target_ip ! {get, self()}, receive Targetip -> Targetip end,
7      seqnum ! {get, self()}, receive Seqnum -> Seqnum end,
8      NewEnv = Env#environment {rreq_tries = Count - 1, cancel = Established},
9      is_route_established ! {set, Established},
10     target_ip ! {set, Targetip},
11     seqnum ! {set, Seqnum + 1},
12     Id1 = 1,
13     Receiver1 = list_to_atom("network_ID" ++ integer_to_list(Id1) ++ "_dymo_to_network"),
14     Receiver1 ! {send, %% CGcreateRREQ (targetip, ip, seqNum)
15                    undefined},
16     Guard_cancel_request_cancel = NewEnv#environment.cancel,
17     Guard_rreq_tries_reached_count = NewEnv#environment.rreq_tries,
18     Guard_rreq_tries_reached_cancel = NewEnv#environment.cancel,
19     Guard_create_rreq_count = NewEnv#environment.rreq_tries,
20     Guard_create_rreq_cancel = NewEnv#environment.cancel,
21     Guard_create_rreq_ip = NewEnv#environment.own_ip_address,
22     if
23       %% cancel
24       undefined -> cancel_request(NewEnv);
25        %% count = 0, not cancel
26       undefined -> rreq_tries_reached(NewEnv);
27        %% not cancel, count > 0
28       undefined -> create_rreq(NewEnv)
29   end.
```

Listing 3: The `createRREQ` function.

```
1 createRREQ(Target, N, Seqnum) ->
2    #message {src = N, dest = 'LL_MANET_ROUTERS', target_addr = Target, orig_addr = N, orig_seqnum = Seqnum,
3              hop_limit = 5, dist = 1, msg_type = 'RREQ'}.
```

ment (ll. 2–4) and reading variables from shared variables (ll. 5–7), gathering of parameters. Then the code constructs a new environment (l. 8), and sets all shared values (ll. 8–15), calculation and distribution of results. In lines 14–15 we see that the original declaration from the CPN model has been commented out and replaced by `undefined`. We need to manually translate this function, and remove the comment and undefined. We then extract values for use in evaluation of the guard (ll. 16–21), and dispatch based on the result of evaluating the guard (ll. 22–28). We also need to translate the expressions in the switch in lines 22–28, but due to similarity between SML and Erlang, this mostly consists of renaming the variables shown. To illustrate how easy the modifications are, we have shown the manually implemented function in Listing 3.

In order to execute more than one node running the generated DYMO implementation, we use the *distributed Erlang system* which is a mechanism in Erlang allowing a number of Erlang systems to communicate over a network. It consists of a number of independent Erlang runtime systems, each called an *Erlang node*. Each node executes the same generated DYMO code. The processes running the DYMO implementation on different Erlang nodes do not communicate directly with each other. Instead they communicate via a *network simulator* process running on a separate Erlang node. The stub code for the network simulator is generated directly from the `network` process partition of the DYMO PCPN model. The network simulator process implements a simple MANET with a static topology where both unicast and multicast is supported.

To monitor the behaviour of the program, each node prints its own routing table, which can then be manually inspected to verify that the expected routes were established. To make sure that every part of the generated code at some point has been executed, we have executed the generated DYMO code with several different MANET configurations designed to exercise all parts of the code. The generated code established the correct routes in all cases, which provides confidence in the generated code.

5 Conclusion and Future Work

In this paper, we have introduced a sub-class of coloured Petri nets, process-partitioned coloured Petri nets (PCPNs or PCP-nets), which separates control flow from data, and we have shown a structure-based approach to automatically generating (Erlang) code from models created using this sub-class. The approach first extracts a control flow graph from the model, and from the control flow graph infers the higher level control structures and constructs an abstract syntax tree for an abstract language. From the abstract syntax tree, we generate a

syntax tree specific to the target language, translating generic control structures into language specific control structures.

We have validated that the translation and net class works for real-life examples, and have shown that a real life model with 8 pages, 49 places and 18 transitions can be translated to the net class in acceptable time (20 man hours), which indicates that the net class is not too far from general CP-nets, at least for network protocols. The generated code can be edited relatively easily, and even though we do not translate more complex SML functions automatically, the adaptation and setup for testing of the code was done in only 12 man hours, which indicates that the generated code is indeed fairly natural. Using manual inspection and logging, we have validated that traces in the generated code can be reproduced in the model and that the calculated results are correct according to the model.

For future work, we would like to extend the PCPN class to allow using variables from buffer or shared places in guards, which would reduce the complexity of the PCPN DYMO model. This complicates the calculation of guards, as the value may change as other process instances modify/receive values, requiring introduction of a locking mechanism. It would also be interesting to look at dynamic instantiation of processes, which should not be too difficult since we already instantiate processes in our setup process. Furthermore, our current implementation does not allow hierarchical PCP-nets, so they have to be flattened, e.g., using CPN Tools prior to translation. Automatic flattening, or, even better, use of the module concept in CP-nets to further improve the translation would be preferable.

It would also be interesting to improve the control structure recognition, from only using goto and conditional jumps to also include inlining of unconditional unique jumps, recognition of loops, and binary if statements. It would also be interesting to add Java code emission to validate that our approach is indeed output language agnostic. It is also a very interesting option to translate to GCCs GENERIC [11] intermediate language instead, as this would give us direct emission of binary code and allow us to directly link with code written in the plethora of languages supported by GCC.

Tool support can also be improved to, e.g., check that a model is indeed a PCPN model (or to only allow construction of valid PCPN models), and to add a parser to automatically generate AST nodes for all SML functions to eliminate this manual step.

An obvious weak point of the current implementation is that the validation is done in an ad-hoc manner, which increases our confidence in the implementation, but does not deliver the promised certainty that the code is correct.

References

1. I.D. Chakeres and C.E. Perkins. Dynamic MANET On-demand (DYMO) Routing, version 14, June 2008. Internet-Draft. Work in Progress.
2. CPN Tools webpage. `www.cs.au.dk/CPNTools/`.

3. Erlang specification. `www.erlang.org/doc.html`.
4. K.E. Espensen and M.K. Kjendsen. Automatic Code Generation from Process-Partitioned Coloured Petri Net Models. Master's thesis, Dept. of Computer Science, Aarhus University, 2008.
5. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *Proc. of ATPN'08*, volume 5062 of *LNCS*, pages 152–170. Springer-Verlag, 2008.
6. C. Girault and R. Valk. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag, 2003.
7. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
8. L.M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G.E. Gallasch. Model-based Development of a Course of Action Scheduling Tool. *STTT*, 10(1):5–14, 2007.
9. L.M. Kristensen and A. Valmari. Finding Stubborn Sets of Coloured Petri Nets Without Unfolding. In *Proc. of ATPN'98*, volume 1420 of *LNCS*, pages 104–123. Springer-Verlag, 1998.
10. K.B. Lassen and S. Tjell. Translating Colored Control Flow Nets into Readable Java via Annotated Java Workflow Nets. In *Proc. of 8th CPN Workshop*, volume 584 of *DAIMI-PB*, pages 127–146, 2007.
11. J. Merril. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. `gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf`.
12. K.H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In *Proc. of ATPN'00*, volume 1825 of *LNCS*, pages 367–386. Springer, 2000.
13. S. Philippi. Automatic code generation from high-level Petri-Nets for model driven systems engineering. *Journal of Systems and Software*, 79(10):1444–1455, 2006.
14. W.M.P. van der Aalst, J.B. Jørgensen, and K.B. Lassen. Let's Go All the Way: From Requirements Via Colored Workflow Nets to a BPEL Implementation of a New Bank System. In *Proc. of OTM Conferences (1)*, volume 3760 of *LNCS*, pages 22–39. Springer, 2005.