Automatic Structure-based Code Generation from Coloured Petri Nets: A Proof of Concept

L.M. Kristensen[1] and M. Westergaard[2]

[1] Department of Computer Engineering, Bergen University College, Norway
Email: lmkr@hib.no
[2] Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
Email: m.westergaard@tue.nl

Abstract. Automatic code generation based on Coloured Petri Net (CPN) models is challenging because CPNs allow for the construction of abstract models that intermix control flow and data processing, making translation into conventional programming constructs difficult. We introduce Process-Partitioned CPNs (PP-CPNs) which is a subclass of CPNs equipped with an explicit separation of process control flow, message passing, and access to shared and local data. We show how PP-CPNs caters for a four phase structure-based automatic code generation process directed by the control flow of processes. The viability of our approach is demonstrated by applying it to automatically generate an Erlang implementation of the Dynamic MANET On-demand (DYMO) routing protocol specified by the Internet Engineering Task Force (IETF).

1 Introduction

The development of concurrent software systems is complex due to the rich behaviour introduced by concurrency, communication, and non-determinism. Coloured Petri Nets (CPNs) [9] and CPN Tools [2] (and formal modelling in general) have been widely used to address these challenges and construct formal and executable models of system designs with the aim of validating functional and performance properties prior to implementation [7]. Constructing a formal model yields important insight into the system design, and is a very helpful reference artefact when conducting a manual implementation of a software design. Even so, manual implementation is error-prone and time-consuming, making automatic code generation [8, Chap. 21] preferable in order to reduce the risk of introducing errors and to exploit the resources invested in model construction.

Despite the wide use of CPNs and high-level Petri Nets for modelling and design validation, we are aware of relatively few examples where CPNs have been used to automatically obtain an implementation of the final software system. This is in contrast to, e.g., the area of hardware design, where low-level Petri nets have been widely used to synthesise hardware circuits [17]. A *simulation-based approach* to automatic code generation from CPNs has been used in the projects reported in [13] and [10]. Here, the simulation code for the CPN model

generated by CPN Tools is extracted, and after undergoing automatic modifications, e.g., linking the code to external libraries, the generated simulation code is used as the system implementation. A simulation-based approach is also used in [14] to generate Java code from a high-level Petri net. The idea of [14] is to make a class diagram which outlines the classes and method signatures of the program. From this diagram, classes are generated where the method bodies are filled with simulator code. The advantage of a simulation-based approach is that it does not put any additional limitations on the class of models for which code can be generated. Furthermore, the direct use of the simulation code automatically ensures that the implementation is behaviourally equivalent to the underlying model. A main disadvantage is performance. Firstly, the execution speed is affected because each step in the execution of the program involves the computation and execution of enabled transitions (as done by a simulator) in order to determine the next state. Secondly, the approach ties the target platform to that of the simulator which may make the approach impractical for certain application domains due to resource consumption. As an example, the SML/NJ compiler used for the simulator in CPN Tools has a large memory footprint making it ill-suited for the domain of embedded systems. These disadvantages can to some extent be overcome using a *state-based approach*. Here, the state space of the model is used to control the execution of the program and determine the next state. This approach assumes that the state space is finite and small.

The disadvantages of simulation- and state-based approaches to code generation motivate our work on a *structure-based approach*. The key idea is to exploit *structure* in the CPN model, which can be naturally mapped to conventional programming language constructs. This has the advantage that the structure of the CPN model becomes clearly recognisable in the generated code, and that the generated code has a structure closer to code written by a human programmer. Furthermore, the code generated using a structure-based approach contains no simulator scheduler to control the execution, thereby improving performance, and the approach can be made target language independent. Exploiting structure in CPNs for code generation purposes is challenging since CPNs makes it possible to model control flow structures, message passing, and data access more abstractly than supported directly in most programming languages.

To address this, we introduce *Process-Partitioned CPNs* (PP-CPNs) which constitute a subclass of CPNs. PP-CPNs contain additional syntactical information and semantic restrictions that provide an explicit separation of process control flow, message passing, and access to data. This is used in a four phase code generation approach, where the choice of target language is deferred to the last two phases. Figure 1 shows the four phases in our structure-based code generation approach for translating a PP-CPN model into code in a target programming language. The first phase **(1)** translates a PP-CPN into a control flow graph (CFG) for each process subnet, extracting the control flow from the model. Nodes in a CFG represent statements and directed edges represent jumps in the control flow. The CFG may also be subject to static analysis, e.g., dead code elimination. In the second phase **(2)** the CFG is translated into an abstract

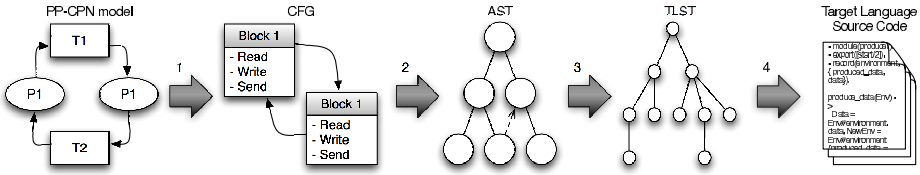

Fig. 1: Structure-based code generation phases.

syntax tree (AST) for a simple intermediate language designed to be abstract enough that it can be translated into most programming languages (see [4] for details), and such that it can capture the assumptions made on the target language in a generic way. The AST can also be used to recognise common control structures such as while loops and if-statements. In the third phase **(3)**, the AST is translated into a syntax tree for the target language (TLST). Finally, in phase four **(4)** the TLST is traversed and the target language source code is produced. The first two phases are target language independent.

To validate our automatic code generation approach, we have implemented it in a computer tool where Erlang [3] is used as the target language. Erlang is a concurrency-oriented language developed at Ericsson for reliable and fault-tolerant concurrent software in telecommunication switches. We report on the use of the developed approach and supporting computer tool to automatically obtain an implementation of the Dynamic MANET On-demand (DYMO) protocol [1]. Being able to model an industrial protocol like DYMO demonstrates that PP-CPNs are sufficiently expressive to model systems occurring in practice. The definition of PP-CPNs is inspired by [11], where a process-oriented subclass of CPNs was defined to facilitate partial-order state space reduction.

Outline. Section 2 introduces the basic concepts of PP-CPNs and the associated syntactical and semantical restrictions. Section 3 formally defines syntactical PP-CPNs that statically ensures the restrictions introduced in Sect. 2. Section 4 illustrates the four phases of the translation process using an example. In Sect. 5 we explain how our code generation approach has been used to obtain an implementation of the DYMO protocol [1]. We sum up our conclusions in Sect. 6. This paper is partly based on the thesis [4] (supervised by the authors of this paper), and an early version has appeared in the technical report [6]. The reader is assumed to be familiar with the basic notions of high-level Petri nets, i.e., the combination of Petri nets and a programming language.

Definitions and notation. The formal definition of PP-CPNs use the definitions and notation of CPNs from [9, Chap. 4] as a basis. Let V be a set of variables and $EXPR$ a set of expressions. Then for $v \in V$, $Type(v)$ is the *type* of the variable v, for $e \in EXPR$, $Var(e)$ is the set of *free variables* of e, $Type(e)$ is the *type* of the expression e, and $EXPR_V$ is the set of expressions with free variables contained in V. For a set S, we denote by $\mathbf{N}^S$ the multi-set type over

S, i.e., the set of all multi-sets (bags) over S. We define $\cup$, $\cap$, $-$, $=$, $|\cdot|$, and $\subseteq$ (union, intersection, difference, equality, size, and subset) as normal for multi-sets, and use a notation, where $m = n`a$ is the multi-set with $m(a) = n$ and $m(b) = 0$ for $b \neq a$, and use $++$ for union as an alternative symbol.

Definition 1 (Coloured Petri net). *A **coloured Petri net** CPN is a tuple* $CPN = (P, T, \Sigma, V, C, G, E, I)$ *where:*

1. P *is a finite set of* ***places*** *(by convention drawn as ellipses), and* T *is a finite set of* ***transitions*** *(by convention drawn as rectangles),*
2. Σ *is a set of non-empty* ***colour sets*** *(types), and* V *is a set of* ***typed variables*** *such that* $Type(V) \subseteq \Sigma$,
3. $C : P \rightarrow \Sigma$ *is a* ***colour set function*** *that assigns a colour set to each place,*
4. $G : T \rightarrow EXPR_V$ *is a* ***guard function*** *that assigns a guard (by convention written in square brackets) to each transition* t *such that* $Type(G(t)) = Bool$,
5. $E : P \times T \cup T \times P \rightarrow EXPR_V$ *is an* ***arc expression function*** *with* $Type(E(p,t)) = \mathbf{N}^{C(p)}$ *and* $Type(E(t,p)) = \mathbf{N}^{C(p)}$,
6. $I : P \rightarrow EXPR_\emptyset$ *is an* ***initialisation function*** *that assigns an initialisation expression to each place* p *such that* $Type(I(p)) = \mathbf{N}^{C(p)}$. □

A *binding* of a set of variables V is a function that maps each variable $v \in V$ to an element (value) of $Type(v)$. An expression $e \in EXPR_V$ evaluated in a binding b over V (denoted $e\langle b \rangle$) is the value obtained by replacing all free occurrences of v in e by $b(v)$. A binding of a transition t is a binding over the variables $Var(t)$ of t, and $B(t)$ denotes the set of all bindings for a transition t.

2 Process-Partitioned CPNs and Process Subnets

To introduce and motivate the constructs of PP-CPNs, we use the producer-consumer system shown in Fig. 2. The figure depicts the initial marking of a producer-consumer system with two producers (identified using the colours (values) `P(1)` and `P(2)`) and two consumers (identified using the colours `C(1)` and `C(2)`). Data items produced and consumed are modelled as integer values.

A PP-CPN is a union of *process subnets* each describing the program code executed by one or more processes. The producer-consumer system consists of two process subnets: one process subnet (left) modelling the producer, and one process-subnet (right) modelling the consumer. The transitions of a process subnet model the actions of processes, and the *process places* P_{pr} model control flow locations. The transitions and the process places of a process subnet make the control flow of processes explicit in the model, and from the current marking (token distribution) it is easy to determine where a process is in its control flow. The consumer process subnet (right) has three transitions ReceivedData, ConsumeEven, and ConsumeOdd modelling the actions of consumer processes, and process places ConsumerIdle and ConsumerWaiting modelling control flow locations. ReceivedData is a local place (see below) used by consumers to locally store a received data item before it is consumed. We have modelled the consumption

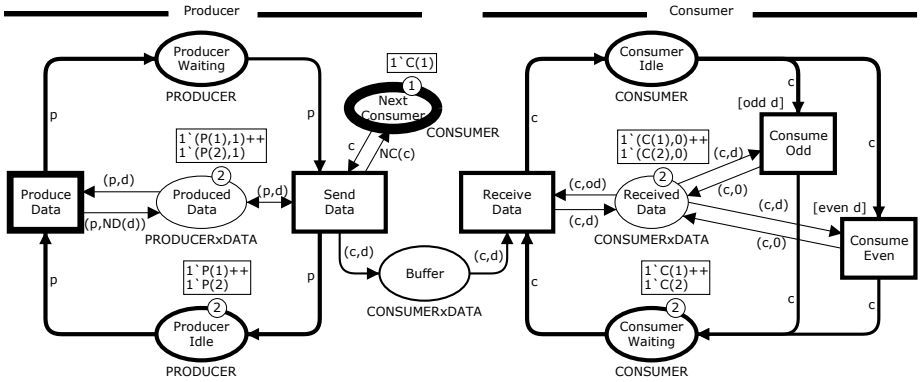

Fig. 2: The producer-consumer PP-CPN model.

of data items using the two transitions ConsumeOdd and ConsumeEven in order to illustrate how our approach handles branches in the control flow of processes.

We introduce a special *process colour set* (type) τ, and a *process variable function* PV. The process variable function provides for each transition a *process variable* of type τ which in an occurrence of the transition identifies the process executing the statement represented by the transition. The variable p of the colour set PRODUCER used on the arcs connecting process places and transitions is a *process variable* which in an occurrence of a binding of a transition identifies the process executing the action modelled by the transition.

A process subnet has a set of *local places* P_{loc} that make explicit data which is local to a process. The place ProducedData is a *local place* (representing data local to a single producer) used to locally store produced data before it is sent to the consumer. The function ND is used to determine the produced data item. A process subnet also consists of a set of *shared places* P_{shr} and a set of *buffer places* P_{buf}. Buffer places and shared places constitute the mechanism provided in PP-CPNs for connecting process subnets and make synchronisation points explicit. Buffer places correspond to communication channels between processes, whereas shared places represent memory shared between processes. The place NextConsumer is a *shared place* representing shared data between the producer processes. The colour of the token on the place NextConsumer identifies the consumer to which a given data item will be sent. The function NC is used to determine the consumer that will receive the next data item. The place Buffer is a *buffer place* connecting the producer process subnet and the consumer process subnet. It models a buffer for the transmission of data items from producers to a specific consumer. We define a set of *process identification functions* $PrId = \{PrId_p\}$ on the set of process-, local-, and buffer places that projects multi-sets of structured types onto the process type, allowing us to project onto the *process identity* of tokens on process, local, and buffer places. We refer to tokens residing on process places as *process tokens*.

Next, we introduce semantical restrictions on the initial marking and arc expressions to make local places behave as local variables, shared places behave as global variables, and buffer places behave as unordered unicast communication channels. These semantical requirements are central to our code generation approach to be presented in Sect. 4.

The first semantical restriction concerns the initial marking. We require that all processes represented by the subnet start in the same location. This translates into the existence of single process place p_I initially containing a token for each process, i.e., $PrId_p(I(p_I)\langle\rangle) = \tau$. All other process places p are required to be initially empty which (together with the above requirement) can be expressed as $\sum_{p \in P_{pr}} PrId_p(I(p)) = \tau$. The other requirements concerning the initial marking is that each local place p initially has a token for each process, i.e., $PrId_p(I(p)) = \tau$; a shared place p holds exactly one token, i.e., $|I(p)\langle\rangle| = 1$; and each buffer place p is initially empty, i.e., $I(p)\langle\rangle = \emptyset$. The requirement on local places reflects that tokens on such a place represent the current value of a local variable (one for each process). The requirement on a shared place reflects that the single token represents the current value of a global variable.

The next requirement concerns the arc expression functions. We require that the occurrence of transitions preserve the control flow of processes, i.e., that the tokens removed from process places (when projected to process identities) by a transition t in a binding b ($\sum_{p \in P_{pr}} PrId_p(E(p,t)\langle b\rangle)$) is equal to the tokens added ($\sum_{p \in P_{pr}} PrId_p(E(t,p)\langle b\rangle)$) when projected to process identities, and that this equals $1`b(PV(t))$, i.e., exactly one process token corresponding to the process variable $PV(t)$ of the transition. For a local place, we require that each transition removes exactly one token from such a place, and that each transition adds exactly one token (or removes/adds zero tokens in case the transition is not connected to the place). Also, we require that the process identities of the tokens added and removed match the binding of the process variable of the transition. For shared places, we only require that each transition removes exactly one token and adds exactly one token (or removes/adds zero tokens in case the transition is not connected to the place). Finally, if a transition t removes tokens from a buffer place p in binding b (i.e., receives an item from the channel represented by p), then a single token is removed, and the process identity of the token removed ($PrId_p(E(p,t)\langle b\rangle)$) matches the value $b(PV(t))$ assigned to the process variable.

Finally, we have a requirement for shared and buffer places. The requirement allows us to calculate enabling for transitions (i.e., execute statements of processes) without taking special care of the race conditions that could arise when accessing buffer and shared places (which can also be accessed by other processes). The requirement is that if we have found a binding b of a transition t that satisfies the enabling condition with respect to local and process places (i.e., required tokens are available and the guard $G(t)$ of t is satisfied), then for all shared and buffer places p and colours (tokens) $c \in C(p)$, we can find a new binding b' such that the token removed from p is c, the guard $G(t)$ of t is satisfied, and the tokens removed from all other places p' ($E(p',t)\langle b'\rangle$) is equal to those

removed in the original binding $E(p',t)\langle b\rangle$. The following definition summarises the definition of process subnets based on the description above.

Definition 2 (Process Subnet). *A **process subnet** is a tuple $(CPN, P_{pr}, P_{loc}, P_{shr}, P_{buf}, \tau, PV, PrId)$, where:*

1. $CPN = (P_{pr} \cup P_{loc} \cup P_{shr} \cup P_{buf}, T, \Sigma, V, C, G, E, I)$ *is a CPN cf. Def. 1,*
2. P_{pr} *is a set of **process places**,* P_{loc} *is a set of **local places**,* P_{shr} *is a set of **shared places**, and* P_{buf} *is a set of **buffer places** such that* P_{pr}, P_{loc}, P_{shr}, *and* P_{buf} *are mutually disjoint,*
3. $\tau \in \Sigma$ *is a **process colour set**,*
4. $PV : T \to V$ *is a **process variable function** that assigns a process variable to each transition* t *such that* $Type(PV(t)) = \tau$,
5. $PrId = \{PrId_p : \mathbf{N}^{C(p)} \to \mathbf{N}^{\tau}\}_{p \in P_{pr} \cup P_{loc} \cup P_{buf}}$ *is a set of linear **process identification functions** that maps multi-sets over* $C(p)$ *into multi-sets over* τ *for each place* $p \in P_{pr} \cup P_{loc} \cup P_{buf}$,
6. *The initialisation function* I *additionally satisfies:*
   - 6a. *There exists a process place* $p_I \in P_{pr}$ *such that* $PrId_p(I(p_I)\langle\rangle) = \tau$ *and* $\sum_{p \in P_{pr}} PrId_p(I(p)) = \tau$,
   - 6b. *For all* $p \in P_{loc}$: $PrId_p(I(p)) = \tau$, *for all* $p \in P_{shr}$: $|I(p)\langle\rangle| = 1$, *and for all* $p \in P_{buf}$: $I(p)\langle\rangle = \emptyset$,
7. *The arc expression function* E *additionally satisfies:*
   - 7a. *For all* $t \in T$, $b \in B(t)$: $\sum_{p \in P_{pr}} PrId_p(E(p,t)\langle b\rangle) = \sum_{p \in P_{pr}} PrId_p(E(t,p)\langle b\rangle) = 1\text{`}(b(PV(t)))$,
   - 7b. *For all* $p \in P_{loc}$, $t \in T$, *and* $b \in B(t)$: $PrId_p(E(p,t)\langle b\rangle) = PrId_p(E(t,p)\langle b\rangle) \subseteq 1\text{`}(b(PV(t)))$,
   - 7c. *For all* $p \in P_{shr}$, $t \in T$, *and* $b \in B(t)$: $|E(p,t)\langle b\rangle| = |E(t,p)\langle b\rangle| \leq 1$,
   - 7d. *For all* $p \in P_{buf}$, $t \in T$ *and* $b \in B(t)$: $PrId_p(E(p,t)\langle b\rangle) \subseteq 1\text{`}(b(PV(t)))$,
8. *Shared places and buffer places are neutral with respect to enabling: Let* $t \in T$, $b \in B(t)$ *be such that* $G(t)\langle b\rangle = \mathsf{true}$. *Then for all* $p \in P_{shr} \cup P_{buf}, c \in C(p)$ *there exists a binding* $b' \in B(t)$ *such that:*
   - 8a. $E(p,t)\langle b'\rangle = 1\text{`}c$ *and* $G(t)\langle b'\rangle = \mathsf{true}$
   - 8b. *For all* $p' \in P - \{p\} : E(p,t)\langle b\rangle = E(p,t)\langle b'\rangle$ □

Items (7) and (8) are central to our approach as it allows checking the enabling of a transition in a process subnet by checking a) if the transition is enabled when we ignore all arcs from shared and buffer places, and b) if there are tokens on all incoming buffer places. Hence, enabling becomes monotone, i.e., as soon as both a) and b) hold for a transition $t \in T$ in a binding $b \in B(t)$ with $b(PV(t)) = pid$, it can only stop holding for t and b if a transition $t' \in T$ is executed in a binding $b' \in B(t')$ for which $b'(PV(t')) = pid$, i.e., if another transition is executed for the same process identity. This is shown by applying (7a), (7b) and (7d) to the requirements. Relaxing (8), i.e., allowing variables in arc expressions from shared places to affect the enabling of the transition, would introduce dependency on the value of the token on the shared place, which can be modified by any other process. Allowing values from buffer places to affect

the enabling of transitions would make enabling no longer dependent only on presence but also on the received value. Requirement (7d) is necessary as it is otherwise possible for a transition with another binding of the process variable to consume a value deemed available from a buffer.

A PP-CPN is a union of process subnets only intersecting on buffer and shared places. We do not formally define this union here as it can be obtained by using the above definition of process subnets in Def. 6 of [11].

3 Syntactical Process Subnets

In general, it is undecidable whether requirements (7) and (8) of Def. 2 are satisfied since they depend on all possible bindings for a transition (and the inscription language of CPNs is Turing complete). Hence, for implementation purposes we introduce sufficient syntactical requirements (which can be statically checked) that imply that the semantical requirements in Def. 2 are satisfied.

We restrict the colour sets of process places to be equal to the process colour set τ, and the colour sets of local places P_{pr} and buffer places P_{bufin} are required to be a cartesian product $\tau \times \sigma$ of the process colour set and some colour set σ. This means that the process identity function on process places becomes the identity function, and for local and buffer places it projects into the first component. As a consequence, we require arc expressions to/from process places of a transition t to have the form $1`PV(t)$.[3] Also, all variables of a transition must be bound via input arcs or in the guard, and all dependencies between variables (except the process variable) must be expressed in the guard. We denote by $InVar(t)$ the set of free variables appearing on input arcs and guards of a transition t. We require input arc expressions from local and input buffer places p to have the form $1`(PV(t), v(t)(p))$ and input arc expression from shared places to have the form $1`v(t)(p)$, where $v : T \to P_{loc} \cup P_{bufin} \cup P_{shr} \to V \cup \{\bot\}$ is a function that assigns a unique non-process variable to be used in the arc expression from p to t. We define $v(t)(p) = \bot$ in case p is not connected to t. Output arc expressions to local places are required to have the form $1`(PV(t), e(t,p))$ where $e(t,p)$ is an expression over free variables from input arc expressions and guards, i.e., $e(t,p) \in EXPR_{InVar(t)}$. For output arc expressions to shared and output buffer places we have the same requirement concerning free variables. Finally, non-process variables in arc expressions from shared places and from input buffer places cannot be referred to in the guard. Our precise requirements are given in the following definition.

Definition 3 (Syntactical Process Subnet). *A **syntactical process subnet** is a tuple $(CPN, P_{pr}, P_{loc}, P_{shr}, P_{buf}, \tau, PV, PrId)$, satisfying:*

1. *$CPN, P_{pr}, P_{loc}, P_{shr}, P_{buf}, \tau, PV, PrId)$ are as defined in items (1)-(6) of Def. 2, $P_{bufin} \subseteq P_{buf}$ denotes the set of input buffer places, and $P_{bufout} \subseteq P_{buf}$ denotes the set of output buffer places,*

[3] For the PP-CPN model in Fig. 2 there is not an explicit 1` (coefficient) in front of arc expressions that evaluates to single token as is convention in CPN Tools.

2. *The colour set function C is defined such that:*
$$C(p) = \begin{cases} \tau & \text{for } p \in P_{pr}, \\ \tau \times \sigma & \text{for } p \in P_{loc} \cup P_{bufin} \text{ and some } \sigma \text{ such that } \tau \times \sigma \in \Sigma, \\ \sigma & \text{for } p \in (P_{bufout} - P_{bufin}) \cup P_{shr} \text{ for some } \sigma \in \Sigma, \end{cases}$$
3. *The process functions $PrId = \{PrId_p : \mathbf{N}^{C(p)} \rightarrow \mathbf{N}^{\tau}\}_{p \in P_{pr} \cup P_{loc} \cup P_{buf}}$ are defined such that that $PrId_p$ is the identity function for $p \in P_{pr}$, and $PrId_p$ projects onto the first component for $p \in P_{loc} \cup P_{buf}$,*
4. *There exists a function $pre : T \rightarrow P_{pr}$ mapping transitions to input process places and a function $v : T \rightarrow P_{loc} \cup P_{bufin} \cup P_{shr} \rightarrow V \cup \{\bot\}$ with $v(t)(p) \neq v(t)(p')$ or $v(t)(p) = v(t)(p') = \bot$ if $p \neq p'$, assigning unique non-process variables to all non-process input places such that:*

$$E(p,t) = \begin{cases} 1`(PV(t)) & \text{for } p = pre(t), \\ 1`(PV(t), v(t)(p)) & \text{for } p \in P_{loc} \cup P_{bufin} \text{ and } v(t)(p) \neq \bot, \\ 1`(v(t)(p)) & \text{for } p \in P_{shr} \text{ and } v(t)(p) \neq \bot, \\ \emptyset & \text{otherwise} \end{cases}$$

5. *There exists a function $succ : T \rightarrow P_{pr}$ mapping transitions to output process places and expressions $e(t,p) \in EXPR_{InVar(t)}$ of correct type such that:*

$$E(t,p) = \begin{cases} 1`(PV(t)) & \text{for } p = succ(t), \\ 1`(PV(t), e(t,p)) & \text{for } p \in P_{loc} \text{ with } v(t)(p) \neq \bot, \\ 1`(e(t,p)) & \text{for } p \in P_{shr} \text{ with } v(t)(p) \neq \bot, \\ e(t,p) & \text{for } p \in P_{bufout}, \\ \emptyset & \text{otherwise} \end{cases}$$

6. *The guard function additionally satisfies that for all $p \in P_{bufin} \cup P_{shr}$ and transitions t: $Var(E(p,t)) \cap (Var(G(t)) - \{PV(t)\}) = \emptyset$.* □

We have that a syntactical process subnet also is a process subnet. Requirements (1)-(6) in Def. 2 are shared via (1) of Def. 3. Items (2) and (3) of Def. 3 comply with the requirement to the process colour set and the process identification function in items (3) and (5) of Def. 2. Items (4) and (5) in Def. 3 implies (7a)-(7d) in Def. 2. Items (4) and (6) in Def. 3 ensures that (8) in Def. 2 holds as all tokens are consumed using distinct variables that are only made dependent in the guard, and as the guard cannot include variables bound on arcs from shared or buffer places, we get the desired result.

4 Translation Process and Phases

We assume that the target language considered has a notion of processes (or threads) and that it allows for message passing between processes. We do not assume direct support for shared memory as it can be implemented using a

separate process and message passing. In addition, we assume that the target language has conditional jumps and a means for storing data local to a process. The Erlang programming language (which we shall concentrate on in this paper as the target language) satisfies these requirements. We assume that the model consists of syntactical process subnets as defined in Sect. 3).

4.1 Phase 1: Translating the PP-CPN Model to a CFG

The main purpose of this phase is to extract the control flow from the PP-CPN model and represent it explicitly in a CFG. A CFG is a directed graph in which arcs correspond to jumps in the control flow and nodes correspond to sequences of statements to be executed. A CFG is constructed for all process subnets in the PP-CPN model. In the producer-consumer system two CFGs are generated: one for the producer process subnet and one for the consumer process subnet. Figure 3 shows the translated CFG for the consumer process subnet. Transitions are translated into *basic blocks*, the nodes in the CFG, yielding three basic blocks for the producer-consumer system and a special basic block, start, that indicates where the process starts.

The content of basic blocks depends on connected non-process places. Basically, arcs from a non-process place correspond to reading a local or shared variable, or receiving a value from a buffer, and arcs to a non-process place correspond to writing and sending. As all arcs from non-process places are of the form 1‘(pid, c) or 1‘c where pid is the process variable and c a variable of the correct type ((4) in Def. 3) and each input arc has a unique variable (also (4) in Def. 3) an input arc is translated to a Read local (for arcs from local places), Read shared (for arcs from shared places), or Receive statements (for arcs from buffer places). Each statement contains the name of the place and a temporary variable to read the value into. The name of the temporary variable corresponds to the name of the variable from the PP-CPN model. In Fig. 3, the basic block ReceiveData contains a statement reading the local variable ReceivedData into the temporary variable od and a statement receiving a value from Buffer into the variable d. Analogously, arcs to non-process places are translated into Write local, Write shared, and Send statements that update variables or transmit values according to the expressions in the PP-CPN model. In Fig. 3, the basic block ReceiveData updates the variable ReceivedData with the value of the variable d.

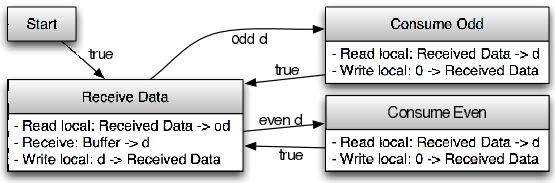

Fig. 3: The CFG of the consumer process.

Process places in the PP-CPN model are represented as arcs between basic blocks. This is possible as we assume that each transition has a unique predecessor and successor process place ((4) and (5) in Def. 3), so we create an arc from t to t' if $succ(t) = pre(t')$. The process place with the initial marking τ (from (6a) in Def. 2) can be considered having an arc from a special transition start adding all the initial process tokens. In Fig. 3, the basic block ConsumeEven has an arc to ReceiveData signifying that after executing ConsumeEven, control should flow to the basic block ReceiveData, and after ReceiveData the program continues to either ConsumeOdd or ConsumeEven. Arcs of a CFG have a condition indicating when control can flow via an arc. The arc condition is extracted from the guard of the destination transition, using true for the absense of a guard. In Fig. 3, the arc from ConsumeEven to ReceiveData has condition true whereas the arc in the opposite direction has arc condition even d. We note that the guards should be evaluated with values of non-process places of the target node, not of the source node, but because of (4) and (6) in Def. 3 (or, equivalently, (8) from Def. 2), stating that guards can only depend on values on local places, this evaluation can be done already at the source node.

4.2 Phase 2: Translating the CFG to an AST

The main purpose of this phase is to translate the CFG into a tree form consisting of nodes representing common programming constructs such as read/write statements and jump statements. We also parse expressions used in the write statements and guards into abstract syntax, making subsequent steps independent of the inscription language used in the PP-CPN model.

Figure 4 shows a sub-tree of the AST for the producer-consumer example where only the nodes from the ReceiveData block of the Consumer have been fully expanded. When building the AST, a process is created for each CFG process. Figure 4 shows that the program contains two processes (Producer and Consumer) and a node for the global variable NextConsumer corresponding to the shared variable NextConsumer. Local variables and buffers are translated into nodes of the processes (processes can only transmit to a single process subnet because of (7) in Def. 2 and the fact that process types are unique to process subnets). In Fig. 4, the buffer place Buffer and the local place ReceivedData are translated into Buffer and Local variable sub-nodes of the Consumer process.

Each basic block is translated into a subtree of the AST, and the contents of basic blocks are translated into statements that correspond to the statements from the basic blocks, except that expressions are parsed into trees. For example, the Write local expression of the basic block ReceiveData in Fig. 3 is translated into Write local in the ReceiveData block in the AST in Fig. 4. The actual value to write (in the CFG just represented as the string d) is parsed into an expression consisting of the variable d in the AST. Arcs in the CFG are translated into conditional or unconditional jump statements in the AST. The conditions are also parsed, as can be seen in the conditional jump from ReceiveData to ConsumeOdd. The jump destinations are expressed using pointers rather than names

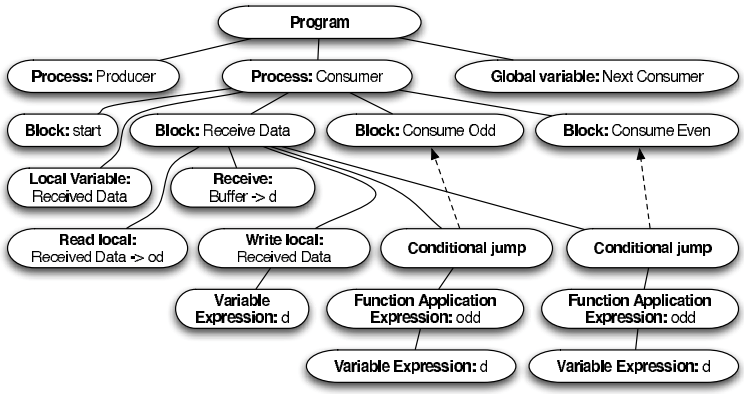

Fig. 4: The AST for the Receive Data block of the consumer.

because variables used in the conditions are in the scope of the destination, and to keep the flow of control explicit in the AST.

4.3 Phases 3 and 4: Translating the AST via TLST to Target Code

Phase 3 generates a syntax tree for a concrete target language based on an AST and phase 4 produces the actual target language code. Figure 5 shows part of the TLST produced for Erlang from the AST in Fig. 4. We will not go into details about the constructs, but only discuss some of the higher level concepts to give an impression of how the AST can be mapped into a TLST. We map each process into a module in Erlang, which is the primitive for code separation and processes. In addition to the processes directly represented in the AST in Fig. 4, we have added three other modules: system, buffer, and shared. The system module is responsible for setting up the system, instantiating processes, and making sure that processes have access to shared variables and buffers. The buffer module is added if a model contains a buffer place, and implements buffers using Erlang channels in order to be able to check if a buffer has any values available. The shared module uses processes to implement shared variables on top of the functional language Erlang. We represent jumps by function application, and generate a function for each block in the AST. Additionally, we introduce an environment record for each process to keep track of local variables and buffers. The environment is created by the start function and is modified and passed on by each function. Examples of Erlang code are not contained in this paper due to space limitations. The reader is referred to [4, 6] for examples.

5 Application to the DYMO Protocol

We have implemented our approach in a prototype in Java using the Access/CPN framework [16]. In addition to evaluate the prototype on smaller examples (such

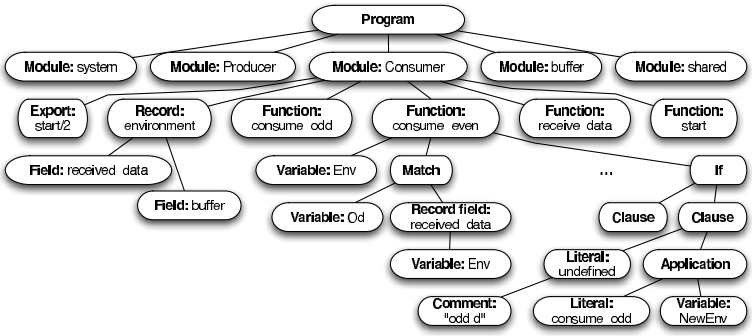

Fig. 5: Partial TLST for generated Erlang target code.

as the producer-consumer system), we have applied it to a PP-CPN model of the Dynamic On-demand MANET routing protocol (DYMO) [1]. The protocol is developed by the Internet Engineering Task Force and is intended for establishing routes in a *mobile ad-hoc network* (MANET). The protocol establishes routes on-demand, i.e., when they are actually needed. The model specifies the *route discovery* and *route maintenance* procedures of DYMO. Route discovery establishes routes by forwarding and multi-casting of *route request* messages. Route maintenance monitors links and uses timeouts to discover loss of connection, which causes *route errors* to be multi-casted to all neighbours.

The CPN model of the DYMO protocol [5] was constructed before starting our work on code generation, and were used to identify and resolve problems in the protocol specification. The complete PP-CPN model of DYMO consists of 8 modules, 49 places, and 18 transitions, and is thus fairly complex. Figure 6 shows an example module of the PP-CPN model for procedures initiating route discovery. The module can create a new route request (RREQ) or cancel the request when the retransmission limit is reached. A detailed description of the DYMO CPN model can be found in [5].

Generating Erlang code from the DYMO model yields the modules listed in Table 1. We have listed lines of code (LOC) for each module – in total we generate 563 lines of code. Since we do not support automatic translation of sequential SML to Erlang, we have to manually implement various Erlang expressions and functions on the basis of the corresponding SML code. Implementing the functions (12 in total) in Erlang is a fairly easy task, and in total we only spent approximately 12 person-hours on this, including removal of unused extracted values. This part could be handled automatically by an off-the-shelf SML parser and code generator, but we have considered it outside the scope of this paper.

In order to execute more than one node running the generated DYMO protocol implementation, we use a *distributed Erlang system* which is a mechanism in Erlang allowing a number of independent Erlang run-time systems (nodes) to communicate over a network. Each node executes the generated DYMO code. The processes running the DYMO implementation on different Erlang nodes do

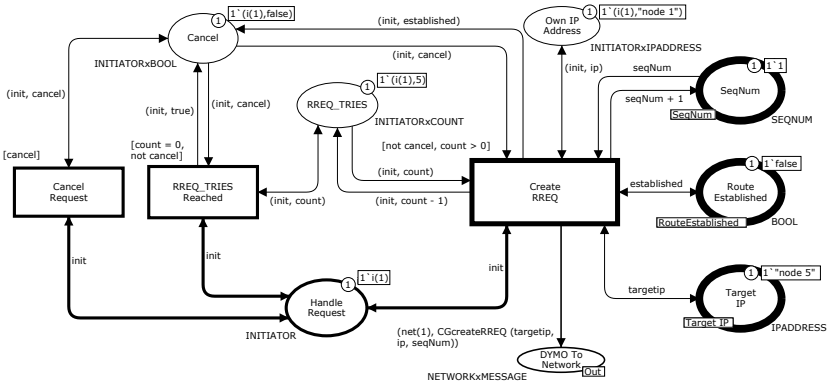

Fig. 6: The Initiator module of the PP-CPN DYMO model.

Table 1: Generated Erlang modules for the DYMO Protocol.

Module name	LOC	Sequential functions
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

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 subnet of the DYMO PP-CPN model. The network simulator process implements a simple MANET where both unicast and multicast is supported.

To monitor the behaviour of the program, each node prints its own routing table, which can be inspected to verify that the expected routes were established. To ensure that all parts of the generated code have been executed, we have tested the generated DYMO implementation 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 and a proof-of-concept of our code generation approach.

6 Conclusions and Future Work

We have introduced the PP-CPN sub-class of CPNs, which forms the basis for a structure-based approach to automatically generate (Erlang) code from CPN models. The approach first extracts a control flow graph from the model, and

from the control flow graph constructs an abstract syntax tree for an intermediate 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. Furthermore, we have validated that our approach applies to real-life examples by applying it to a PP-CPN model of the DYMO protocol consisting of 8 modules, 49 places, and 18 transitions. Using manual inspection and logging, we have validated that traces in the generated code can be reproduced in the model and that the calculated routes are correct.

A structural approach to code generation in high-level Petri nets is also applied in [8]. The focus of [8] is on identifying processes in a Petri net, i.e., parts of the model that work independently of each other or only have few synchronisation points. Afterwards local variables (i.e., information only used by one process) and communication channels are found. In comparison, we provide this information explicitly in the form of the PP-CPN model. In [15], a class of CP-nets is translated into BPEL (Business Process Execution Language) which is an XML-based workflow implementation language. In contrast to our work, [15] focus on the flow of data and not on data processing and the BPEL language is not aimed at general application development. [12] 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. Also, the approach [12] is limited to producing Java code, whereas our approach is target language independent.

One area of future work concerns extending the PP-CPN subclass. One direction is to allow using variables from buffer or shared places in guards. This complicates the calculation of guards, as the value may change as other process instances modify/receive values, requiring introduction of a locking mechanism for shared places. We can easily allow dependence on values from buffer places as long as all branches from any process place consume the same number of tokens from all buffer places, as we can just read all values and dispatch accordingly, allowing us to receive and dispatch a value from a buffer in a single step as opposed to the two steps required now. Allowing dependencies on input arcs, at least on arcs from local places, would make the allowed PP-CPN models more natural. It would also be interesting to look at dynamic instantiation of processes, which is not overly difficult since we already instantiate processes in our generated code. This could either be done using a language extension to PP-CPNs or simply allowing creation of new tokens on process places.

Currently, we do not perform static analysis in phase 1 in our prototype, making the generated code more verbose than needed and neither do we perform control structure recognition, which also makes the generated code a bit unnatural. It would be interesting to see how the generated code would be affected by actually conducting these steps. We have considered code generation for Erlang, but all steps until the generation of the target language dependent syntax tree are target language independent, and it would be interesting to also experiment with other target languages. A limitation of the current implementation is that the validation is done in an ad-hoc manner. Future work also

includes formally proving correctness of the translation is correct by, e.g., formally defining our abstract language as represented by ASTs and proving that the generated abstract code is behaviourally equivalent to the PP-CPN model.

Acknowledgements. The authors acknowledges the work of K.L. Espensen and M.K. Kjeldsen on which this paper is partly based.

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. `http://www.cs.au.dk/CPNTools/`.
3. The Erlang programming language. `http://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. `http://www.hib.no/ansatte/lmkr/ppcpn-thesis.pdf`.
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. K.L. Espensen, M.K. Kjeldsen, L.M. Kristensen, and M. Westergaard. Towards Automatic Code Generation from Process-Partitioned Coloured Petri Nets. In *Proc. of 10th CPN Workshop*, pages 41–60. Aarhus University, 2009.
7. Examples of Industrial Use of CP-nets. `www.cs.au.dk/CPnets/intro/example/indu.html`.
8. C. Girault and R. Valk. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag, 2003.
9. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
10. L.M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G.E. Gallasch. Model-based Development of COAST. *STTT*, 10(1):5–14, 2007.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. M. Westergaard and L.M. Kristensen. The Access/CPN Framework: A Tool for Interacting With the CPN Tools Simulator. In *Proc. of ATPN 2009*, volume 5606 of *LNCS*, pages 313–322. Springer-Verlag, 2009.

17. A. Yakovlev, L. Gomes, and L. Lavagno. *Hardware Design and Petri Nets*. Kluwer Academic Publishers, 2000.