Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification

Kent Inge Fagerland Simonsen[1,2], Lars M. Kristensen[1], and Ekkart Kindler[2]

[1] Department of Computing, Bergen University College, Norway
[2] DTU Compute, Technical University of Denmark, Denmark

Abstract. PetriCode is a tool that supports automated generation of protocol software from a restricted class of Coloured Petri Nets (CPNs) called Pragmatics Annotated Coloured Petri Nets (PA-CPNs). PetriCode and PA-CPNs have been designed with five main requirements in mind, which include the same model being used for verification and code generation. The PetriCode approach has been discussed and evaluated in earlier papers already. In this paper, we give a formal definition of PA-CPNs and demonstrate how the specific structure of PA-CPNs can be exploited for verification purposes.

1 Introduction

Coloured Petri Nets (CPNs) [3] and CPN Tools have been widely used for modelling and verifying protocols. Examples include application layer protocols such as IOTP, SIP and WAP, transport layer protocols such as TCP, DCCP and SCTP, and network layer protocols such as DYMO, AODV, and ERDP [2, 8]. Formal modelling and verification have been useful in gaining insight into the operation of the protocols and have resulted in improved protocol specifications. However, earlier work has not fully leveraged the investment in modelling by also taking the step to automated code generation to obtain an implementation of the protocol under consideration. In particular, rather limited research has been conducted into approaches that support automatic generation of protocol implementations from such CPN models. The earlier approaches have either restricted the target platform for code generation to the Standard ML language used by the CPN Tools simulator or have considered a specific target language based on platform-specific additions to the CPN models.

This has motivated us to develop an approach and an accompanying tool called PetriCode to support the automated generation of protocol software from CPN models. Our code generation approach is designed to satisfy five main requirements. Firstly, the approach must support *platform independence*, i.e., it must enable code generation for multiple languages and platforms from the same CPN model. Secondly, the approach must support *integration* of the generated code with third-party code. In particular, it must support *upwards integration*, i.e., the generated code must expose an explicit interface for service invocation, and it must support *downwards integration*, i.e., the ability of the generated code

to invoke and rely on underlying libraries. Thirdly, it must support *verification* in that the code generation capability should not introduce complexity problems for verification of the CPN models. Fourthly, the generated code must be *readable* to enable code review and performance enhancements. Finally, the approach must be *scalable* to industrial-sized protocols.

The foundation of our approach is a slightly restricted subclass of CPNs called *Pragmatic Annotated CPNs* (*PA-CPNs*). The restrictions make explicit the structure of the protocol system, its principals, channels, and services. A key feature of this class of CPNs are so-called *code generation pragmatics*, which are syntactical annotations to certain elements of the PA-CPNs. These pragmatics represent concepts from the domain of communication protocols and protocol software, which are used to indicate the purpose of the respective modelling element. The role of the pragmatics is to extend the CPN modelling language with domain-specific elements and make implicit knowledge of the modeller explicit in the CPN model such that it can be exploited for code generation.

In earlier work [16], we have introduced PA-CPNs informally, and presented the PetriCode tool [17]. In [18], we demonstrated platform independence, integrateability, and readability of the generated code. In [19], we applied the approach for automatically generating an implementation of the industrial-strength WebSocket protocol. This included demonstrating that the generated code was interoperable with other implementations of the WebSocket protocol.

The contribution of this paper compared to our earlier work is threefold. Firstly, motivated by the practical relevance of the net class demonstrated in earlier work, we give a formal definition of PA-CPNs. Secondly, we discuss the process of developing protocol software with our approach from a methodology perspective. Thirdly, we show that PA-CPNs are amenable to verification. Specifically, we show how the structural restrictions of PA-CPNs allow us to add *service testers* to the model of the protocol, which reduce the state space of the model. Furthermore, the structural restrictions of PA-CPNs induce a natural progress measure that can be exploited for verification purposes by the *sweep-line state space exploration method* [4].

The rest of this paper is organised as follows: Section 2 provides the background definitions and notation of CPNs that are used throughout this paper. Section 3 gives the formal definition of PA-CPNs accompanied by an example outlining how PA-CPNs can be used to model a transport protocol. Section 4 discusses the modelling process of PA-CPNs from an application perspective. Section 5 formalises the concepts of tree decomposability of control flow nets which are central in generating code for the protocol services. Section 6 shows how to define progress measures for the sweep-line method based on service and service tester modules of PA-CPNs, and experimentally evaluate their effect on the verification of the transport protocol example. Finally, in Sect. 7, we sum up the conclusions and discuss related work. We assume that the reader is familiar with the basic concepts of Petri nets and high-level Petri nets such as CPNs. This paper is a condensed version of a technical report [20], which contains more motivation and detailed explanations of examples and concepts.

2 Background Definitions on Coloured Petri Nets

The definition of PA-CPNs is based on the standard definition of hierarchical CPNs [3]. Here, we briefly rephrase the definitions of CPNs. Readers familiar with these definitions can skip this section. In this paper, we provide the syntactical definitions of CPNs only, which will be restricted when defining PA-CPNs. Since PA-CPNs are a syntactical restriction of CPNs, we do not need change the semantics of CPNs at all.

A hierarchical CPN consists of a finite set of CPN modules, which we discuss first. Figures 1 and 2 show some CPN modules of our example (which will be used later). The modules of a hierarchical CPNs are related to each other via substitution transitions (shown with a double border) which can have associated submodules, and by linking places connected to the substitution transitions (called socket places) to places (called port places) on the associated submodules.

A *CPN module* consists of a set of *places* P and a set of *transition* T connected by a set of *directed arcs* A connecting either a transition and a place or a place and a transition. A CPN module additionally has a set of *colour sets* (types) Σ containing the types that can be used as colour sets of places and for typing a set of *variables* V which can be used in arc expressions and transition guards. In the formal definition, the colour set of each place (by convention written below a place) is specified by means of a *colour set function* that maps each place to a colour set determining the kind of tokens that may reside on the place. Each directed arc in a CPN module has an associated arc expression used to determine the tokens added and removed by the occurrence of an enabled transition and is specified by an *arc expression function*. The arc expression of each arc may contain variables from the set of variables V. The arc expressions are required to have a type such that the evaluation of an arc expression on an arc connected to a place p results in a multi-set of tokens over the colour set of the place. Transitions may have an associated guard expression specified by means of a *guard function* G which associates a boolean expression with each transition. The initial marking of each place is specified by means of an *initialisation function* I which maps each place into a (possibly empty) multi-set over the colour set of the place.

Definition 1 formally defines a CPN module. In the definition, we use $Type[v]$ to denote the type of a variable v, and we use $EXPR_V$ to denote the set of expressions with free variables contained in a set of variables V. For an expression e containing a set of free variables V, we denote by $e\langle b\rangle$ the result of evaluating e in a binding b that assigns a value to each variable in V. We use $Type[e]$ for an expression e (an arc expression, a guard, or an initial marking) to denote the type of e. For a non-empty set S, we use S_{MS} to denote the type corresponding to the set of all multi-sets over S.

Definition 1. *A **Coloured Petri Net Module** (Def. 6.1 in [3]) is a tuple* $CPN_M = (CPN, T_{sub}, P_{port}, PT)$*, such that:*

1. $CPN = (P, T, A, \Sigma, V, C, G, E, I)$ *is a Coloured Petri Net (Def. 4.2 in [3]) where:*

   (a) P *is a finite set of* ***places*** *and* T *is a finite set of* ***transitions*** T *such that* $P \cap T = \emptyset$.
   (b) $A \subseteq (P \times T) \cup (T \times P)$ *is a set of directed* ***arcs***.
   (c) Σ *is a finite set of non-empty* ***colour sets*** *and* V *is a finite set of* ***typed variables*** *such that* $Type[v] \in \Sigma$ *for all variables* $v \in V$.
   (d) $C : P \to \Sigma$ *is a* ***colour set function*** *that assigns a colour set to each place.*
   (e) $E : A \to EXPR_V$ *is an* ***arc expression function*** *that assigns an arc expression to each arc* a *such that* $Type[E(a)] = C(p)_{MS}$, *where* p *is the place connected to the arc* a.
   (f) $G : T \to EXPR_V$ *is a* ***guard function*** *that assigns a guard to each transition* t *such that* $Type[G(t)] = Bool$.
   (g) $I : P \to EXPR_\emptyset$ *is an* ***initialisation function*** *that assigns an initialisation expression to each place* p *such that* $Type[I(p)] = C(p)_{MS}$.
2. $T_{sub} \subseteq T$ *is a set of* ***substitution transitions***.
3. $P_{port} \subseteq P$ *is a set of* ***port places***.
4. $PT : P_{port} \to \{\mathsf{IN}, \mathsf{OUT}, \mathsf{I/O}\}$ *is a* ***port type function*** *that assigns a port type to each port place.*

Socket places are not defined explicitly as part of a module because they are implicitly given via the arcs connected to the substitution transitions. For a substitution transition t, we denote by $ST(t)$ a mapping that maps each socket place p into its type, i.e., $ST(t)(p) = \mathsf{IN}$ if p is an input socket, $ST(t)(p) = \mathsf{OUT}$ if p is an output socket, and $ST(t)(p) = \mathsf{I/O}$ if p is an input/output socket.

The definition of a hierarchical CPN is provided below. A hierarchical CPN consists of a set of disjoint CPN modules, a submodule function assigning a (sub)module to each substitution transition, and a port-socket relation that associates port places in a submodule to the socket places of its upper layer module. The set of socket places for a substitution transition t consists of the places connected to the substitution transition and is denoted by $P_{\mathsf{sock}}(t)$. The definition requires that the module hierarchy (to be defined in Def. 3) is acyclic in order to ensure that there are only a finite number of instances of each module. Furthermore, port and socket places can only be associated with each other, if they have the same colour set and the same initial marking.

Definition 2. *A* ***hierarchical Coloured Petri Net*** *(Def. 6.2 in [3]) is a four-tuple* $CPN_H = (S, SM, PS, FS)$ *where:*

1. S *is a finite set of* ***modules***. *Each module is a* ***Coloured Petri Net Module*** $s = ((P^s, T^s, A^s, \Sigma^s, V^s, C^s, G^s, E^s, I^s), T^s_{\mathsf{sub}}, P^s_{\mathsf{port}}, PT^s)$. *It is required that* $(P^{s_1} \cup T^{s_1}) \cap (P^{s_2} \cup T^{s_2}) = \emptyset$ *for all* $s_1, s_2 \in S$ *with* $s_1 \neq s_2$.
2. $SM : T_{sub} \to S$ *is a* ***submodule function*** *that assigns a* ***submodule*** *to each substitution transition. It is required that the module hierarchy (see Definition 3) is acyclic.*
3. PS *is a* ***port–socket relation function*** *that assigns a* ***port–socket relation*** $PS(t) \subseteq P_{\mathsf{sock}}(t) \times P^{SM(t)}_{\mathsf{port}}$ *to each substitution transition* t. *It is*

required that $ST(t)(p) = PT(p')$, $C(p) = C(p')$, *and* $I(p)\langle\rangle = I(p')\langle\rangle$ *for all* $(p, p') \in PS(t)$ *and all* $t \in T_{sub}$.

4. $FS \subseteq 2^P$ *is a set of non-empty and disjoint* ***fusion sets*** *such that* $C(p) = C(p')$ *and* $I(p)\langle\rangle = I(p')\langle\rangle$ *for all* $p, p' \in fs$ *and all* $fs \in FS$.

The module hierarchy of a hierarchical CPN model is a directed graph with a node for each module and an arc leading from one module to another module if the latter module is a submodule of one of the substitution transitions of the former module. In the definition, T_{sub} denotes the union of all substitution transitions of the hierarchical CPN, and T^s_{sub} denotes all substitution transitions in a module s.

Definition 3. *The* ***module hierarchy*** *for a hierarchical Coloured Petri Net* $CPN_H = (S, SM, PS, FS)$ *is a directed graph* $MH = (N_{MH}, A_{MH})$, *where*

1. $N_{MH} = S$ *is the set of* ***nodes****.*
2. $A_{MH} = \{(s_1, t, s_2) \in N_{MH} \times T_{sub} \times N_{MH} \mid t \in T^{s_1}_{sub} \wedge s_2 = SM(t)\}$ *is the set of* ***arcs****.*

The roots of MH are called ***prime modules****, and the set of all prime modules is denoted* S_{PM}.

3 Pragmatic Annotated CPNs

PA-CPNs mandate a particular structure of the CPN models and allow the CPN elements to be annotated with *pragmatics* used to direct the automated code generation. In the CPN model, pragmatics are shown by annotations enclosed in $\langle\langle\rangle\rangle$. Pragmatics can also have some parameters, which we discuss as they come; but we do not formalize parameters of pragmatics in general here.

A PA-CPN is organised into three levels of modules: the *protocol system level*, the *principal level*, and the *service level* – reflecting the typical structure of protocols. In order to better understand the structure of PA-CPNs, Figs. 1 and 2 show selected modules from each level of a PA-CPN model of the protocol that we use as a running example. The protocol consists of a sender and a receiver principal, with services for sending and receiving data messages, and for sending and receiving acknowledgements. The sender sends each data message, one at a time, with a bounded number of retransmissions awaiting an acknowledgement for each data packet. In addition to the two principals, the protocol system contains unreliable channels for transmitting messages. The complete PA-CPN model of the example protocol is available at [17].

We formally define PA-CPNs as a tuple consisting of a hierarchical CPN: one protocol system module (PSM), sets of principal level modules (PLMs) and service level modules (SLMs) and channel modules (CHMs), and a structural pragmatics mapping (SP) that maps substitution transitions (indicated by double borders) to pragmatics representing the annotations of substitution transitions.

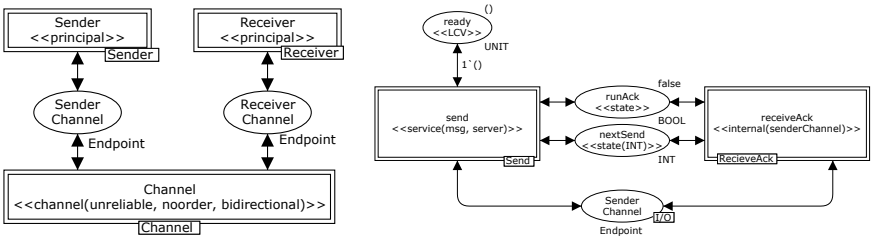

Fig. 1. The top-level CPN system level module (left) and principal level module for the sender principal (right) of the protocol example.

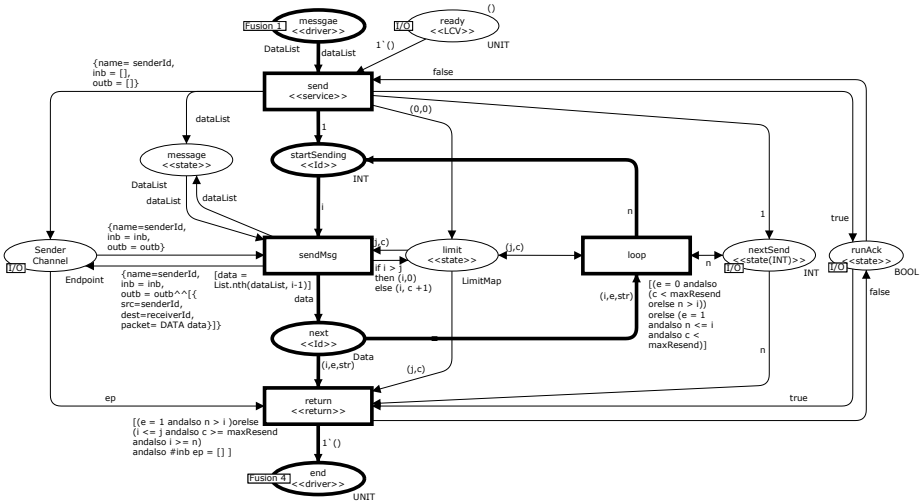

Fig. 2. The send service level module of the protocol example.

Definition 4. *A **Pragmatics Annotated Coloured Petri Net** (PA-CPN) is a tuple $CPN_{PA} = (CPN_H, PSM, PLM, SLM, CHM, SP)$, where:*

1. *$CPN_H = (S, SM, PS, FS)$ is a hierarchical CPN with $PSM \in S$ being a **protocol system module** (Def. 5) and the only prime module of CPN_H.*
2. *$PLM \subseteq S$ is a set of **principal level modules** (Def. 6); $SLM \subseteq S$ is a set of **service level modules** (Def. 7) and $CHM \subseteq S$ is a set of **channel modules** s.t $\{\{PSM\}, PLM, SLM, CHM\}$ constitute a partitioning of S.*
3. *$SP : T_{sub} \rightarrow \{\texttt{principal}, \texttt{service}, \texttt{internal}, \texttt{channel}\}$ is a **structural pragmatics mapping** such that:*
   (a) *Substitution transitions with $\langle\langle$principal$\rangle\rangle$ have an associated principal level module: $\forall t \in T_{sub} : SP(t) = \texttt{principal} \Rightarrow SM(t) \in PLM$.*
   (b) *Substitution transitions with $\langle\langle$service$\rangle\rangle$ or $\langle\langle$internal$\rangle\rangle$ are associated with a service level module:*
   $\forall t \in T_{sub} : SP(t) \in \{\texttt{service}, \texttt{internal}\} \Rightarrow SM(t) \in SLM$.

(c) *Substitution transitions with* $\langle\langle$channel$\rangle\rangle$ *are associated with a channel module:* $\forall t \in T_{sub} : SP(t) = \texttt{channel} \Rightarrow SM(t) \in CHM$.

It should be noted that channel modules do not play a role in the code generation; they constitute a CPN model artifact used to connect the principals for verification purposes. Therefore, we do not impose any specific requirements on the internal structure of channel modules.

Protocol system level. The module shown in Fig. 1(left) comprises the protocol system level of the PA-CPN model of the example. It specifies the two protocol principals in the system and the channel connecting them. The substitution transitions representing principals are specified using the `principal` pragmatic, and the substitution transitions representing channels are specified using the `channel` pragmatic. The PSM module is defined as a tuple consisting of a CPN module and a pragmatic mapping PM that associates a pragmatic to each substitution transition. The requirement on a protocol system module is that all substitution transitions must be substitution transitions that are annotated with either a `principal` or a `channel` pragmatic. Furthermore, two substitution transitions representing principals cannot be directly connected via a place: there must be a substitution transition representing a channel in between. This reflects the fact that principals can communicate via channels only.

Definition 5. *A* ***Protocol System Module*** *of a PA-CPN with a structural pragmatics mapping SP is a tuple* $CPN_{PSM} = (CPN^{PSM}, PM)$*, where:*

1. $CPN^{PSM} = ((P^{PSM}, T^{PSM}, A^{PSM}, \Sigma^{PSM}, V^{PSM}, C^{PSM}, G^{PSM}, E^{PSM}, I^{PSM}), T^{PSM}_{sub}, P^{PSM}_{port}, PT^{PSM})$ *is a CPN module such that all transitions are substitution transitions:* $T^{PSM} = T^{PSM}_{sub}$.
2. $PM : T^{PSM}_{sub} \rightarrow \{\texttt{principal}, \texttt{channel}\}$ *is a* ***pragmatics mapping*** *s.t.:*
   (a) *All substitution transitions are annotated with either a* `principal` *or* `channel` *pragmatic:* $\forall t \in T^{PSM}_{sub} : PM(t) \in \{\texttt{principal}, \texttt{channel}\}$.
   (b) *The pragmatics mapping PM must coincide with the structural pragmatic mapping SP of PA-CPN:* $\forall t \in T^{PSM}_{sub} : PM(t) = SP(t)$.
   (c) *All places are connected to at most one substitution transition with* $\langle\langle$principal$\rangle\rangle$ *and at most one substitution transition with* $\langle\langle$channel$\rangle\rangle$*:* $\forall p \in P^{PSM} : \forall t_1, t_2 \in X(p) : PM(t_1) = PM(t_2) \Rightarrow t_1 = t_2$.

Principal level. On the principal level, there is one module for each principal of the protocol as defined by $\langle\langle$principal$\rangle\rangle$ on the protocol system level. The example protocol has two modules at the principal level corresponding to the sender and the receiver. Figure 1(right) shows the principal level module for the sender. A principal level module is required to model the *services* that the principal is providing, and the *internal states* and *life-cycle* of the principal. For the sender, there are two services as indicated by the `service` and `internal` pragmatics on the substitution transitions send (for sending messages) and receiveAck (for

receiving acknowledgements). Services that can be externally invoked are specified using the `service` pragmatic, whereas services that are to be invoked only internally are specified using the `internal` pragmatic. The non-port places of a principal level module (places drawn without a double border) can be annotated with either a `state` or an `LCV` pragmatic. Places annotated with a `state` pragmatic represent internal states of the principal. In Fig. 1(right), there are two places with $\langle\langle$state$\rangle\rangle$ used to enforce a stop-and-wait pattern in sending data messages and receiving acknowledgements. Places annotated with an `LCV` pragmatic represent the life-cycle of the principal by putting restrictions on the order in which services can be invoked. As an example, the place ready in Fig. 1(right) ensures that only one message at a time is sent using the send service.

Definition 6. *A **Principal Level Module** of a PA-CPN is a tuple* $CPN_{PLM} = (CPN_{PLM}, T_{sub}^{PLM}, P_{port}^{PLM}, PT^{PLM}, PLP)$ *where:*

1. $CPN_{PLM} = ((P^{PLM}, T^{PLM}, A^{PLM}, \Sigma^{PLM}, V^{PLM}, C^{PLM}, G^{PLM}, E^{PLM}, I^{PLM}), T_{sub}^{PLM}, P_{port}^{PLM}, PT^{PLM})$ *is a CPN module with only substitution transitions:* $T^{PLM} = T_{sub}^{PLM}$.
2. $PLP : T_{sub}^{PLM} \cup P^{PLM} \setminus P_{port}^{PLM} \rightarrow \{\texttt{service}, \texttt{internal}, \texttt{state}, \texttt{LCV}\}$ *is a **principal level pragmatics mapping** satisfying:*
   (a) *All non-port places are annotated with either a* `state` *or a* `LCV` *pragmatic:* $\forall p \in P^{PLM} \setminus P_{port}^{PLM} \Rightarrow PLP(p) \in \{\texttt{state}, \texttt{LCV}\}$
   (b) *All substitution transitions are annotated with a* `service` *or* `internal` *pragmatic:* $\forall t \in T_{sub}^{PSM} : PLP(t) \in \{\texttt{service}, \texttt{internal}\}$.

Service level. The service level modules specify the detailed behaviour of the individual services and constitute the lowest level modules in a PA-CPN model. In particular, there are no substitution transitions in modules at this level. The module in Fig. 2 is an example of a module at the service level. It models the behaviour of the send service in a control-flow oriented manner. The control-flow path, which defines the control flow of the service, is made explicit via the use of the `Id` pragmatics. The entry point of the service is indicated by annotating a single transition with $\langle\langle$service$\rangle\rangle$, and the exit (termination) point of the service is indicated by annotating a single transition with $\langle\langle$return$\rangle\rangle$. In addition, non-port places can be annotated with a `state` pragmatic to indicate that this place models a local state of the service. The `driver` pragmatic is used by service tester modules (Sect. 6) to facilitate verification. The places with $\langle\langle$Id$\rangle\rangle$ determine a subnet of the module, which we call the *underlying control-flow net*: it is obtained by removing all CPN inscriptions and considering only places with $\langle\langle$Id$\rangle\rangle$ and transitions connected to these places, which in Fig. 2, are indicated by places, transitions, and arcs with thick border. This control-flow net must follow a certain structure so that there is a one-to-one correspondence to control-flow constructs of typical programming languages. This requirement is called *tree decomposability* and is formally defined in Sect. 5.

A service level module is defined as consisting of a CPN module without substitution transitions and with service level pragmatics as described above.

Note that we use the symbol ∃! to indicate that there "exists exactly on element" with the respective property.

Definition 7. *A **Service Level Module** of a PA-CPN is a tuple* $CPN_{SLM} = (CPN_{SLM}, T^{SLM}_{sub}, P^{SLM}_{port}, PT^{SLM}, SLP)$ *where:*

1. $CPN_{SLM} = ((P^{SLM}, T^{SLM}, A^{SLM}, \Sigma^{SLM}, V^{SLM}, C^{SLM}, G^{SLM}, E^{SLM}, I^{SLM}), T^{SLM}_{sub}, P^{SLM}_{port}, PT^{SLM})$ *is a CPN module without substitution transitions:* $T^{SLM}_{sub} = \emptyset$.
2. $SLP : T^{SLM} \cup P^{SLM} \setminus P^{SLM}_{port} \rightarrow \{\texttt{Id}, \texttt{state}, \texttt{service}, \texttt{return}, \texttt{driver}\}$ *is a **service level pragmatic mapping** satisfying:*
   (a) *Each place is either annotated with* `Id`, `state`, `driver` *or is a port place :* $\forall p \in P^{SLM} \setminus P^{SLM}_{port} : SLP(p) \in \{\texttt{Id}, \texttt{state}, \texttt{driver}\}$.
   (b) *There exits exactly one transition with* $\langle\langle$service$\rangle\rangle$ *and exactly one transition with* $\langle\langle$return$\rangle\rangle$*:*
   $\exists! t \in T^{SLM} : SLP(t) = \texttt{service}$ *and* $\exists! t \in T^{SLM} : SLP(t) = \texttt{return}$.
3. *For all* $t \in T^{SLM}$ *and* $p \in P^{SLM}$ *we have:*
   (a) *Transitions consume one token from input places with an* `Id` *pragmatic:* $(p,t) \in A^{SLM} \wedge SLP(p) = \texttt{Id} \Rightarrow |E(p,t)\langle b\rangle| = 1$ *for all bindings* b *of* t.
   (b) *Transitions produce one token on output places with an* `Id` *pragmatic:* $(t,p) \in A^{SLM} \wedge SLP(p) = \texttt{Id} \Rightarrow |E(t,p)\langle b\rangle| = 1$ *for all bindings* b *of* t.
   (c) *Only transitions with* $\langle\langle$service$\rangle\rangle$ *can have input places with* $\langle\langle$driver$\rangle\rangle$*:* $(p,t) \in A^{SLM} \wedge SLP(p) = \texttt{driver} \Rightarrow SLP(t) = \texttt{service}$
   (d) *Only transitions with* $\langle\langle$return$\rangle\rangle$ *can have output places with* $\langle\langle$driver$\rangle\rangle$ *pragmatic:* $(t,p) \in A^{SLM} \wedge SLP(p) = \texttt{driver} \Rightarrow SLP(t) = \texttt{return}$
4. *The underlying control flow net of* CPN_{SLM} *is tree decomposable (Defs. 9,11).*

4 Protocol Modelling Process

In the previous sections, we have formalised the structural restrictions of CPNs and the pragmatics extensions that make them *Pragmatic Annotated CPNs* (*PA-CPNs*); some additional restrictions on the control-flow structure and the service testers will be formalized later in Sect. 5 and 6. Since it is the modellers responsibility to come up with a model meeting these requirements, we briefly discuss the choices underlying the definition of PACPNs and their structural restrictions concerning the modelling process and some methodology for developing protocol software with PA-CPNs here.

The structural requirements of PA-CPNs have been distilled from the experience with earlier CPN models of protocols. The structure and annotations of PA-CPNs are designed to help the modeller come up with a clear model and to give clear guidelines for creating a model that – at the same time – can be used for code generation as well as for verification. As such, the structure of PA-CPNs should be driven by the protocol and its purpose rather than by the artifacts of Petri nets. This is, in particular, reflected by structuring the model in three layers: *protocol system*, *principal*, and *service layer*.

The top layer, the *protocol system layer*, identifies the overall structure of the protocol, which are the *principals* of the protocol and how the principals are connected by *channels* (see Fig. 1 (left) for an example). Each principal and each channel is represented by a substitution transition with a respective annotation, and places connecting the respective principals with channels. The behaviour of each principal is represented by *principal level module*, which identifies the *services* of the respective principal (see Fig. 1 (right) for an example) along with the states of the protocol and its life-cycle. The services are represented by substitution transitions annotated with the `service` pragmatics, the state and the life-cycle of the principal are represented by places with `state` and `LCV` pragmatics. The behaviour of each service is then modelled by a *service level module*, which is associated with the service substitution transitions on the *principal level module* (see Fig. 2 for an example). The service level module has access to the channels that the principal is connected to as well as to the principal's state and life-cycle variables. The most prominent structure (indicated by boldfaced places, transitions, and arcs) of the service level module is the control-flow structure, which is identified by the `Id` pragmatics and which needs to follow very specific rules so that it can be transformed to control-flow constructs of typical programming languages and result in human-readable code. The exact requirements are discussed in Sect. 5.

It should be noted that also the channels (on the protocol system level) need to be associated with PA-CPN modules, which model the exact behaviour of the respective channel. The modules for the channels are not used for code generation, since the generated code will use implementations of channels from the underlying platform (based on the properties required for these channels). But for verifying the protocol with standard CPN mechanisms, we need a CPN module for each channel, which however does not have any further structural restrictions.

Any model that meets the requirements of PA-CPNs can be used for code generation as well as for verification – irrespective of the way it was produced. The typical modelling process of protocols with PA-CPN starts at the top-level by identifying the principals of the protocol and how they are connected by channels. Then, the services of each principal are identified on the principal level, and then each service is modelled. So the general modelling direction is top-down. Of course, additional services and even additional principals could be added later, when need should be.

5 Tree Decomposability of Control Flow Nets

As discussed earlier, the control-flow structure of a service level module, called the underlying control-flow net, must correspond one-to-one to control-flow constructs of programming languages. The main purpose of this requirement is to generate readable code. In this section, we formally define the *underlying control flow net* of a service level module and its one-to-one correspondence to control-flow constructs. This is achieved by inductively decomposing the control-flow net

into a tree of sub-blocks, each of which corresponds to a control-flow construct: atomic step, sequence, choice and loop.

Figure 3 shows the *underlying control flow net* of the service level module from Fig. 2. All places and transitions in the rounded rectangle (representing the block border) are part of the block; an arrow from the block border to a place indicates the entry place; an arrow from a place to the block border indicates the exit place. The control flow net in Fig. 3 can be decomposed in a loop block, which in turn consists of an atomic block.

First, we define *blocks*: these are Petri nets with a fixed entry and exit place.

Definition 8. *Let $N = (P, T, A)$ be a Petri net and $s, e \in P$. Then $B = (P, T, A, s, e)$ is called a **block** with **entry** s and **exit** e. The block is **atomic**, if $P = \{s, e\}$, $s \neq e$, $|T| = 1$ and for $t \in T$, we have ${}^\bullet t = \{s\}$ and $t^\bullet = \{e\}$. The block has a **safe entry**, if $s \neq e$ and ${}^\bullet s = \emptyset$. The block has a **safe exit**, if $s \neq e$ and $e^\bullet = \emptyset$.*

For easing the following definitions, we introduce an additional notation: For a block B_i, we refer to its constituents by $B_i = (P_i, T_i, A_i, s_i, e_i)$ without explicitly naming them every time. The block that is underlying a service level module is determined by all the places with $\langle\langle$Id$\rangle\rangle$ pragmatics and the transitions in their pre- and postsets. The unique transition with $\langle\langle$service$\rangle\rangle$ defines the entry place, and the unique transition with $\langle\langle$return$\rangle\rangle$ defines the exit place of this block; note that for technical reasons, these two transitions are not part of the block. Therefore, these transitions are shown by dashed lines in Fig. 3. Formally, the control flow net underlying a service level module is defined as follows.

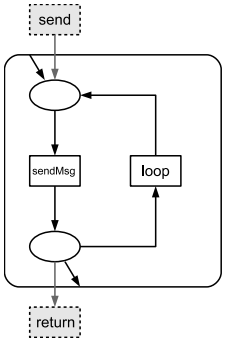

Fig. 3. Decomposition of the service level module in Fig. 2

Definition 9. *Let CPN_{SLM} be a service level module as defined in Def. 7. Let $P = \{p \in P^{SLM} \setminus P^{SLM}_{port} | SLP(p) = Id\}$, let $T = T^{SLM} \cap {}^\bullet P \cap P^\bullet$, and let $A = A^{SLM} \cap ((T \times P) \cup (P \times T))\}$; moreover, let $s \in P$ be the unique place such that there exists a transition $t \in T = T^{SLM}$ with $(t, s) \in A^{SLM}$ and $SLP(t) =$ `service`, and let $e \in P$ be the unique place e such that there exists a transition $t \in T = T^{SLM}$ with $(e, t) \in A^{SLM}$ and $SLP(t) =$ `return`. Then, $N = (P, T, A, s, e)$ is the **underlying control flow net** of CPN_{SLM}.*

The control flow of the code that is being generated is obtained by decomposing the underlying control flow net of a service level module into sub-blocks representing the control-flow constructs. We define the decomposition in a very general way at first, which does not yet restrict the possible control-flow constructs. The decomposition into blocks, just makes sure that all parts of the block are covered by sub-blocks and that they overlap on entry and exit places

only. In a second step, the decomposition is restricted in such a way that the decomposition captures certain control flow constructs (Def. 11).

Definition 10. *Let $B = (N, s, e)$ be a block with net $N = (P, T, F)$. A set of blocks $B_1, \ldots, B_n$ is a* ***decomposition*** *of B if the following conditions hold:*

1. *The sub-blocks contain only elements from B, i. e. for each $i \in \{1, \ldots, n\}$, we have $P_i \subseteq P$, $T_i \subseteq T$, and $F_i \subseteq F \cap ((P_i \times T_i) \cup (T_i \times P_i))$.*
2. *The sub-blocks contain all elements of B, i. e. $P = \bigcup_{i=1}^{n} P_i$, $T = \bigcup_{i=1}^{n} T_i$, and $F = \bigcup_{i=1}^{n} F_i$.*
3. *The inner structure of all sub-blocks are disjoint, i. e. for each $i, j \in \{1, \ldots, n\}$ with $i \neq j$, we have $T_i \cap T_j = \emptyset$ and $P_i \cap P_j = \{s_i, e_i\} \cap \{s_j, e_j\}$.*

As the final step, we define when a decomposition of a block reflects some control flow construct. The definition does not only define decomposability into control flow constructs; it also defines a tree structure which reflects the control-flow structure of the block; the type of each node reflects the construct. The definition is illustrated in Fig. 4. The top left part of Fig. 4 shows the inductive definition of a loop construct: The assumptions are that two blocks $B1$ and $B2$ are identified already. $B1$ is any kind of block (represented by X) with a safe entry place s and a safe exit place e; $B2$ is an atomic block with entry place e and exit place s. Thus, block $B1$ represents the loop body, and block $B2$ the iteration. Then, the union of both blocks and entry place s and exit place e, form a block B, which is a loop consisting of the loop body $B1$ and the atomic block $B2$ for the iteration. The definitions of choices and sequences are similar.

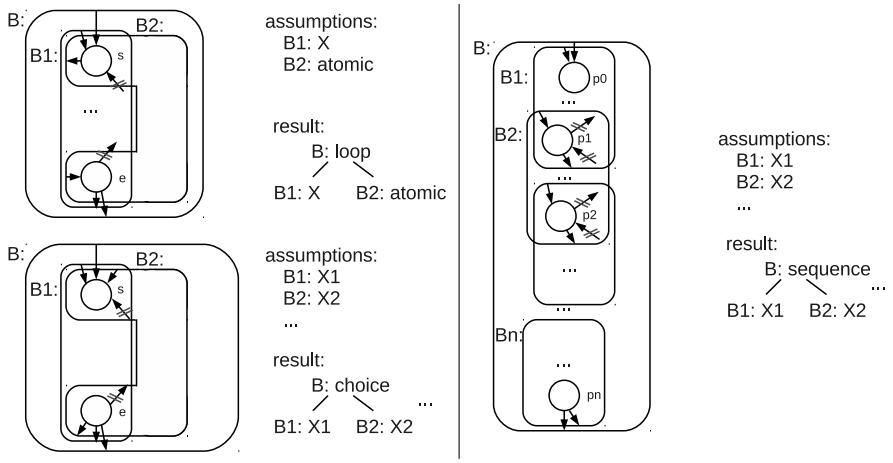

Fig. 4. Inductive definition of block trees

Definition 11 below formally defines block tree as illustrated in Fig. 4.

Definition 11. *The* ***block trees*** *associated with a block are inductively defined:*

Atomic *If B is an atomic block, then the tree with the single node* ***B:atomic*** *is a* ***block tree*** *associated with B.*

Loop *If B is a block and B_1 and B_2 is a decomposition of B, and for some X, $B_1 : X$ is a block tree associated with B_1, and $B_2 :$ atomic is a block tree associated with B_2, and if B_1 has a safe entry and a safe exit s.t $s_1 = s$, $e_1 = e$, $s_2 = e$, $e_2 = s$, then the tree with top node* ***B:loop*** *and the sequence of sub-trees $B_1 : X$ and $B_2 :$ atomic is a* ***block tree*** *associated with B.*

Choice *If B is a block and for some n with $n \geq 2$ the set of blocks $B_1, \ldots, B_n$ is a decomposition of B, and have a safe entry and a safe exit, and $B_1 : X_1, \ldots, B_n : X_n$ for some $X_1, \ldots, X_n$ are block trees associated with $B_1, \ldots, B_n$, and if for all $i \in \{1, \ldots, n\}$: $s_i = s$ and $e_i = e$, then the tree with top node* ***B:choice*** *with the sequence of sub-trees $B_i : X_i$ is a* ***block tree*** *associated with B.*

Sequence *If B is a block and for some n with $n \geq 2$ the set of blocks $B_1, \ldots, B_n$ is a decomposition of B, and, for some $X_1, \ldots, X_n$, the trees $B_1 : X_1, \ldots, B_n : X_n$ are block trees associated with $B_1, \ldots, B_n$, and if there exist different places $p_0, \ldots, p_n \in P$ such that $s = p_0$, $e = p_n$, and for each $i \in \{0, \ldots, n-1\}$ we have $s_i = p_i$, $e_i = p_{i+1}$, and B_i has a safe exist or B_{i+1} has a safe entry, then the tree with top node* ***B:sequence*** *and the sequence of sub-trees $B_i : X_i$ is a* ***block tree*** *associated with B.*

A net for which such a tree exists is said to be ***tree decomposable****.*

Note that in order to simplify the definition of tree decomposability, the tree decomposition of a block is not necessarily unique according to our definition. For example, a longer sequence of atomic blocks could be decomposed in different ways. In the PetriCode tool, such ambiguities are resolved by making sequences as large as possible. Note also that for two consecutive constructs in a sequence, it should not be possible to go back from the second to the first; therefore, the above definition requires that consecutive blocks have a safe entry or a safe exit. And there are some similar requirements for loops and choices.

6 Service Testers and Sweep-Line Verification

The service level modules constitute the active part of a PA-CPN model. The execution of an individual service provided by a principal starts at the transition with a $\langle\langle$service$\rangle\rangle$ pragmatic. The transitions annotated with a service pragmatic typically has a number of parameters which need to be bound to values in order for the transition to occur. An example of this is the Send service transition in Fig. 2 which has the variable dataList as a parameter. This means that there are often an infinite number of bindings for a service transition.

To control the execution of a PA-CPN model in verification by means of state space exploration, we introduce the concept of *service tester modules* which can be used to guide the verification process and represent a user of the services provided by the principal modules. An advantage of service testers is that they

contribute to reducing the state space during verification and enable progress measures for the sweep-line method [4] to be automatically computed.

The service tester modules are connected to the rest of the PA-CPN model through fusion sets, and the service tester modules invoke the service provided by the principal by putting tokens on fusion places and the service tester receives any results from the invoked services via tokens on these places.

Fusion sets and fusion places are standard constructs of hierarchical CPNs (see Def. 2). A fusion set consists of a set of fusion places such that removing (adding) tokens from (to) a fusion place is reflected on the markings of all members of the fusion set. In addition to the fusion places, `Id` pragmatics are used to make the control flow of the service tester explicit in a similar manner as for service level modules.

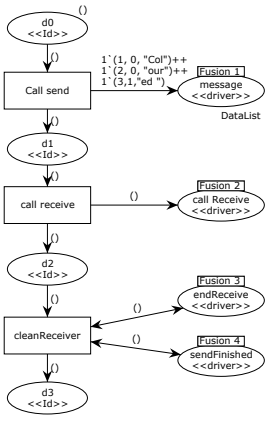

Fig. 5. Service tester module

Figure 5 shows an example of a service tester module for the PA-CPN model introduced in Sect. 3. The service tester drives the execution of a CPN model through fusion places. A service tester module can have many places with `Id` pragmatics; but only one of them may contain a token initially (place d0 in Fig. 5). The service tester first invokes the send service in Fig. 2 by putting a token in the fusion place message. Next, the service tester invokes the receive service in the receiver principal. Service tester modules are formalised below.

Definition 12. *A **Service Tester Module** is a tuple* $CPN_{STM} = (CPN_{STM}, T^{STM}_{sub}, P^{STM}_{port}, PT^{STM}, TPM)$ *where:*

1. $CPN_{STM} = ((P^{STM}, T^{STM}, A^{STM}, \Sigma^{STM}, V^{STM}, C^{STM}, G^{STM}, E^{STM}, I^{STM}), T^{STM}_{sub}, P^{STM}_{port}, PT^{STM})$ *is a CPN module with no substitution transitions:* $T^{STM}_{sub} = \emptyset$.
2. $TPM : P^{STM} \to \{\texttt{Id}, \texttt{driver}, \texttt{LCV}\}$ *is a **tester pragmatic mapping**.*
3. $\exists! p \in I$*:* $|I^{STM}(p)\langle\rangle| = 1$*, and for all* $t \in T^{STM}$ *and* $p \in P^{STM}$ *we have:*
   (a) *Transitions consume one token from input places with an* `Id` *pragmatic:* $(p,t) \in A^{STM} \wedge TPM(p) = \texttt{Id} \Rightarrow |E(p,t)\langle b\rangle| = 1$ *for all bindings* b *of* t.
   (b) *Transitions produce one token on output places with an* `Id` *pragmatic:* $(t,p) \in A^{STM} \wedge TPM(p) = \texttt{Id} \Rightarrow |E(t,p)\langle b\rangle| = 1$ *for all bindings* b *of* t.
4. *Transitions and places with an* `LCV` *pragmatic must be connected with a double arc:* $\forall p \in P^{STM}, t \in T^{STM} : TPM(p) = \texttt{LCV} \Rightarrow ((t,p) \in A^{STM} \Leftrightarrow (p,t) \in A^{STM})$
5. *The underlying control flow block of* CPN_{STM} *is tree decomposable (Defs. 9,11).*

Service tester modules are connected to a PA-CPN by means of fusion places in order to control the execution of the services. We therefore define a PA-CPN

equipped with service tester modules as a hierarchical CPN consisting of a set of modules that constitute a PA-CPN according to Def. 4 and a set of service tester modules which are all prime modules. We also require that fusion places are connecting the service level modules and the service tester module so that they correspond to the invocation of services and collecting of a results from an executed service. As with PA-CPNs, the modeller must construct the service tester modules such that they satisfy the formal requirements. Due to space limitations we omit the formal definition of PA-CPNs with service testers which can be found as Def. 5.2 in [20].

The set of service tester modules determine the state space of the PA-CPN model under analysis. The service tester modules may specify a more or less strict execution order on the services being invoked. It is therefore possible to use the service tester modules to control the size of the state space of the PA-CPN model being verified. Below we show that in addition to the use of service testers, the structural requirements imposed by PA-CPNs can be exploited by the sweep-line method [4] to further reduce the peak memory usage during verification.

The sweep-line method addresses the state explosion problem by exploiting a notion of *progress* exhibited by many systems to store subsets of the state space in memory during state space exploration. To apply the sweep-line method, a *progress measure* must be provided for the model as formalised below where $\mathcal{S}$ denotes the set of all states (markings), $\rightarrow^*$ the reachability relation on the markings of the CPN model, and $\mathcal{R}(M_0)$ denotes the states reachable from the initial marking M_0.

Definition 13. *A* ***progress measure*** *is a tuple* $\mathcal{P} = (O, \sqsubseteq, \psi)$ *such that* O *is a set of* ***progress values****,* $\sqsubseteq$ *is a total order on* O*, and* $\psi : \mathcal{S} \rightarrow O$ *is a* ***progress mapping****.* $\mathcal{P}$ *is* ***monotonic*** *if* $\forall s, s' \in \mathcal{R}(M_0) : s \rightarrow^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$*. Otherwise,* P *is* ***non-monotonic****.*

The subsets of states that need to be stored at the same time are determined via a *progress value* assigned to each state, and the method explores the states in a least-progress-first order. The sweep-line method explores states with a given progress value before progressing to the states with a higher progress value. When the method proceeds to consider states with a higher progress value, it deletes the states with a lower progress value from memory. If it turns out that the system regresses (a non-monotonic progress measure), then the method will mark states at the end of *regress edges* as *persistent* (i. e., store them permanently in memory) in order to ensure termination. In the presence of regression, the sweep-line method may visit the same state multiple times (for details, see [4]).

The structure imposed on CPNs by PA-CPNs and services testers means that PA-CPN models have several potential sources of progress. The control-flow in the service modules is one source of progress as there is a natural progression from the entry point of the service towards the exit point of the service. The life-cycle of a principal is another potential source of progress as there will often be an overall intended order in which the services provided are to be invoked,

and this will be reflected in the life-cycle variables of the principal. Finally, the service testers are also a source of progress as a service tester will inherently progress from the start of the test towards the end of the test. For our example protocol, the progress mapping can be defined as a vector of place-wise measures using the number of tokens on some of its places. This is written below where we omitted the parts of the model that we did not show in this paper and used $s(p)$ to denote the marking of a place p in the state s:

$$\psi(s) = (|s(\mathsf{d0})|, |s(\mathsf{d1})|, |s(\mathsf{d2}))|, |s(\mathsf{d3})|, |s(\mathsf{startSnd})| + |s(\mathsf{next})|, |s(\mathsf{end})|) \quad (1)$$

Two such vectors can be compared lexicographically, meaning the order of the different entries represents their significance. The first four entries represent the progress in the service tester (Fig. 5). The next two entries represent the progress within the send service (Fig.2). Note that since the places startSending (abbreviated as startSnd in (1) and (2)) and next are on a loop, tokens can flow back from place next to place startSending. The end place is actually the respective driver place from the tester, which propagates the progress between the service and tester. Therefore, the tokens on both places within this loop are counted the same (added up in the same entry of the vector). An alternative progress measure is shown below (omitting the parts of the model that we did not show in this paper):

$$\psi(s) = (|s(\mathsf{d0})|, |s(\mathsf{d1})|, |s(\mathsf{d2}))|, |s(\mathsf{d3})|, |s(\mathsf{startSnd})|, |s(\mathsf{next})|, |s(\mathsf{end})|) \quad (2)$$

The difference between (1) and (2) is how loops are handled. In the progress measure (2), the places on loops are appended to the vector as if the loop was not there. In the present example this is shown by having replaced the + operator in (1) between startSending and next with a comma in (2).

We generalise the above idea by defining progress measures on top of the tree decomposition of the blocks underlying the corresponding service tester module or the service level module. We define a simple progress measure and a complex one. The simple one is monotonic and adds up the number of all tokens within a top-level loop; the complex one is not monotonic, but takes progress within a loop into account. Since both definitions are very similar, we define only the complex progress measures formally here (the simple one can be found in [20]).

Definition 14. *Let BT be a block tree for a CPN module. The sequence of* ***complex progress measure entries*** *is defined inductively over the block tree BT of the CPN module:*

Atomic *If BT is B : atomic, then complex progress sequence consist of* $|s|, |e|$ *where s is the entry place of the block B and e is the exit place.*

Sequence *If BT is B : sequence with subblocks* $B_1, \ldots B_n$, *and* $e_i^1, \ldots, e_i^{k_i}$ *are the complex progress sequences for* B_i, *then* $e_1^1, \ldots, e_1^{k_1-1}, e_2^1, \ldots, e_2^{k_2-1}, \ldots, e_n^1, \ldots, e_n^{k_n}$ *is the complex sequence for BT.*

Choice *If BT is B : choice with subblocks* $B_1, \ldots B_n$, *and* $e_i^1, \ldots, e_1^{k_i}$ *are the complex progress sequence for each block* B_i, *then the sequence* $e_1^1, \ldots, e_1^{k_1-1}, e_2^2, \ldots, e_2^{k_2-1}, \ldots, e_n^2, \ldots, e_n^{k_n}$ *is the complex progress sequence for BT*

Loop *If BT is B : loop with places with sub-block B_1 and B_2 with the complex progress sequence $e^1, \ldots, e^n$ for B_1, then $e^1, \ldots, e^n$ is the complex progress sequence for BT.*

A progress measure for the complete system can be built from the progress sequences (either the simple or the complex one) for the tester and service modules by concatenating the sequences. The concatenation would first choose the sequences for the service testers and then the sequences for all the service level modules. Note that if there is a driver place of a service tester attached to the service, this driver place would also be added to the progress measure sequence of the service level module at the end (as for the `end` place for the `send` service).

Table 1 shows some experimental performance results on the protocol example for different configurations (number of transmitted messages) and channel characteristics (lossy/non-lossy) using the sweep-line method with the simple and the complex progress measure. In the experiments, we consider exploration of the complete state space. This is done since the sweep-line method (unless combined with other reduction techniques) in the worst-case needs to explore all states in order to model check a property. One example of this is checking that in all terminal states, the protocol has correctly delivered all packets. Since the simple progress measure is monotonic, the number of explored states using that measure is identical to the number reachable states of the respective example, which for clarity are indicated in the first column again. Since the complex progress measure is not monotonic, some states might be visited (explored) multiple times. Therefore, the number of explored states is higher than the reachable states of the respective example. The ratio columns give the ratio in percent between the peak number of states stored (with the respective progress measure) and the number of reachable states. It can be seen that the runtime as well the peak memory use are better when using the complex progress measure. The complex measure provides better performance due to the fact that the send service has a loop as the top-level control-flow construct. It can be seen that the peak memory use with the complex progress measure is reduced to between 40 and 77%.

Table 1. Verification using simple and complex progress measure

Config		Simple PM				Complex PM			
	Reachable	Explored	Peak	Ratio	Time	Explored	Peak	Ratio	Time
1:noloss	156	156	77	49.3	<1 s	165	63	40.3	<1 s
1:lossy	186	186	99	53.2	<1 s	196	78	41.9	<1 s
3:noloss	2,222	2,222	2,014	90.6	<1 s	2790	1,582	71.2	<1 s
3:lossy	2,928	2,928	2,700	92.2	<1 s	4037	2,187	75.7	<1 s
7:noloss	117,584	117,584	115,373	98.1	216 s	143,531	86,636	73.6	32 s
7:lossy	160,620	160,620	158,888	98.1	532 s	263,608	124,661	77.6	80 s

7 Conclusions and Related Work

In this paper, we focused on the formal definition of PA-CPNs and how the structure of PA-CPNs can be exploited for more efficient verification. The PA-CPN net class has been motivated by the objective of developing a code generation approach to protocol software which allows the same model to be used for both code generation and verification – and which satisfies five main requirements: *platform independence*, *code integration*, *verifiability*, *readability*, and *scalability*. The development of PA-CPNs has been driven by practical experiments in order to empirically validate that the approach satisfied the five requirement in practice. The experimental results have been reported in earlier papers [16, 17, 19] using an implementation of the approach in the PetriCode tool.

The requirements of *platform independence*, *code integration*, *readability*, and *scalability* are relevant for use in practice: Platform independence ensures that the approach is not locked to a particular target programming language. Code integration ensures that the generated code can be integrated with existing other parts of the software. *Readability* of the generated code is important for developing trust in the approach, and for further maintaining the protocol software in the future. *Scalability* is important for being able to apply the approach to industrial strength protocols. Concerning *verifiability*, it often is the case that one model is used for verification, and then the protocol is implemented manually from that or generated from another model. This imposes extra work and decreases the confidence in that the actual software meets the requirements verified on the model. Therefore, we required the same model being used for code generation and for verification.

As stated in the introduction, CPNs have been primarily used for modelling and verifying protocols in the past. Still, related approaches for CPNs – and more generally for high-level Petri Nets (HLPNs) – have been developed. Below, we relate our work to other approaches using HPLNs for code generation by discussing them in the context of the five requirements that have driven the development of PA-CPNs.

Kaim [6] contains a generic discussion of aspects related to generating code from low-level and high-level Petri net models with the purpose of executing it outside the simulation environment where they are created. Kaim discusses both centralised and parallel approaches to interpretation of Petri net models. A main aspect of the parallel approach is a structural analysis of the model in order to identify subnets that can be mapped to different processes. In the PetriCode approach, the structural pragmatics provided by the modeller and the structural restrictions of PA-CPNs provide similar information. Kaim does not consider the issues of code integration and the readability of the generated code.

The approach presented by Philippi [14] is a hybrid of simulation-based and structural analysis approaches to code generation for HLPNs. The motivation for the hybrid approach is to produce more readable code than a pure simulation approach would because fewer checks are needed in the code. Philippi targets the Java platform only and is therefore not platform independent in its basic form. The generated code can be integrated into third party code in that the

API of the generated code is defined by UML class diagrams. The paper [14] does not discuss the scaling to large applications. Lassen et al. [11] aim to generate readable code by creating code with constructs that are similar to what human programmers would have created. Since the approach of Lassen is based on Java annotations of CPN models, the approach is tailored to the Java programming language and does not provide a generic infrastructure that supports code generation for different platforms.

Reinke [15] studies, in the context of the functional programming language Haskell, how to use language embedding for mapping constructs from HLPNs into Haskell code. The focus of Reinke is on generating code for a HLPN simulator. The work of Reinke is not aimed at providing a general mechanism for generating readable code and on integrating the code into a larger application. Kummer et al. [10] are concerned with the execution of reference nets in the context of the Renew tool which is based on the Java platform. Reference nets as supported by Renew are known to be verifiable [12] but the approach is specifically tailored to the Java platform. The work does not focus on integration at the code level but other means are providing for integrating the code into larger applications [1].

Mortensen's approach [13] is a simulation based approach based on extracting the generated simulation code from CPN Tools. As such the work of Mortensen is aimed at making an SML implementation of the modelled system and not on conducting verification of the models or to target multiple platforms. Furthermore, being a simulation based approach, the goal from the outset is not to generate code that is intended for humans to read. The use of a simulation-based approach also means that there is a considerable performance overhead due to the many enabling checks in the code. The approach of Kristensen et al. [7] is similar to the approach in [13]. PP-CPNs are used in [9] as the basis for code generation targeting the Erlang language but the approach is not designed to address readability of the generated code. Furthermore, the approach is tailored to the Erlang platform and may not be easily adapted to other platforms even though PP-CPNs and the intermediary representation of control-flow graphs are independent of the target language. Jørgensen et al. [5] propose an approach for generating BPEL code. The approach is targeted at BPEL and does not create code for other languages or aims to address verifyability, code integration, readability and scalability.

It follows from the discussion above that PA-CPNs and the PetriCode approach complement existing related approaches to code generation for high-level Petri Nets. Furthermore, none of the approach discussed above specifically address the domain of protocol software. This paper can be viewed as completing the development of the PA-CPN net class by giving a formal definition and hence establishing the formal foundation of our approach.

References

1. T. Betz et al. Integrating web services in Petri net-based agent applications. In *Proc. of PNSE'13*, pages 97–116, 2013.

2. J. Billington, G.E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 210–290. Springer, 2004.
3. K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
4. K. Jensen, L.M. Kristensen, and T. Mailund. The Sweep-line State Space Exploration Method. *Theoretical Computer Science*, 429:169–179, 2012.
5. 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 *In Proc. of CoopIS'05*, volume 3760 of *LNCS*, pages 22–39. Springer, 2005.
6. W. El Kaim and F. Kordon. Code generation. In C. Girault and R. Valk, editors, *Petri Nets for System Engineering*, chapter 21, pages 433–470. Springer, 2003.
7. L. M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G. E. Gallasch. Model-based Development of a Course of Action Scheduling Tool. *International Journal on Software Tools for Technology Transfer*, 10:5–14, 2008.
8. L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *ToPNoc VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.
9. L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.
10. O. Kummer et al. An Extensible Editor and Simulation Engine for Petri Nets: Renew. In *Proc. of ICATPN '04*, volume 3099 of *LNCS*, pages 484–493. Springer, 2004.
11. 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*, 2007.
12. M. Mascheroni, T. Wagner, and L. Wüstenberg. Verifying Reference Nets by Means of Hypernets: A Plugin for Renew. In *Proc. of the PNSE'19*, Berichte des Fachbereichs Informatik, pages 39–54. Universität Hamburg, 2010.
13. K. H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 367–386, 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. C. Reinke. Haskell-coloured petri nets. In *Int. Workshop on Implementation of Functional Languages*, volume 1868 of *LNCS*, pages 165–180, 1999.
16. K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *LNCS*, pages 227–242. Springer, 2013.
17. K.I.F. Simonsen. PetriCode: A Tool for Template-based Code Generation from CPN Models. In *Proc. of WS-FMDS 2013*, volume 8368 of *LNCS*, pages 151–163, 2013. Project website: `http://www.petricode.org`.
18. K.I.F. Simonsen. An Evaluation of Automated Code Generation with the PetriCode Approach. In *In Proc. of PNSE '14*, volume 1160 of *CEUR Workshop Proceedings*, pages 295–312. CEUR-WS.org, 2014.
19. K.I.F Simonsen and L.M. Kristensen. Implementing the WebSocket Protocol based on Formal Modelling and Automated Code Generation. In *Proc. of IFIP DAIS'2014*, volume 8460 of *LNCS*, pages 104–118. Springer, 2014.
20. K.I.F. Simonsen, L.M. Kristensen, and E. Kindler. Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification. Technical Report 16, DTU Compute, `http://goo.gl/9j6lDz`, 2014.