Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification

Simonsen, Kent Inge; Kristensen, Lars Michael; Kindler, Ekkart

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification

Kent Inge Fagerland Simonsen [*1,2], Lars Michael Kristensen [†1], and Ekkart Kindler [‡2]

[1]Department of Computing, Mathematics, and Physics, Bergen University College, Norway
[2]DTU Informatics, Technical University of Denmark, Denmark

August, 2014

DTU Compute Technical Report-2014-16

[*]Email: kifs@hib.no, kisi@imm.dtu.dk
[†]Email: lmkr@hib.no
[‡]Email: eki@imm.dtu.dk

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Building 324, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

DTU Compute Technical Reports: ISSN 1601-2321

Abstract

This paper presents the formal definition of Pragmatics Annotated Coloured Petri Nets (PA-CPNs). PA-CPNs represent a class of Coloured Petri Nets (CPNs) that are designed to support automated code generation of protocol software. PA-CPNs restrict the structure of CPN models and allow Petri net elements to be annotated with so-called pragmatics, which are exploited for code generation. The approach and tool for generating code is called PetriCode and has been discussed and evaluated in earlier work already. The contribution of this paper is to give a formal definition for PA-CPNs; in addition, we show how the structural restrictions of PA-CPNs can be exploited for making the verification of the modelled protocols more efficient. This is done by automatically deriving progress measures for the sweep-line method, and by introducing so-called service testers, that can be used to control the part of the state space that is to be explored for verification purposes.

1 Introduction

Although Coloured Petri Nets (CPNs) [2] have been widely used for modelling and verifying network protocols, rather limited research has been conducted into approaches that allow us to automatically generate the implementation of the protocols from the CPN models. And (to the best of our knowledge) there do not exist approaches that at the same time can be used for verification and code generation of network protocol software based on CPN models. In earlier work [5], we have presented an approach and a tool called PetriCode, which allowed us to automatically generate the protocol software from a restricted class of CPNs. One of the objectives of PetriCode was to be able to generate code for different platforms. Another main objective was that the used CPN models could still be applied for verifying the correctness of the network protocols.

The PetriCode approach uses a class of CPNs with a slightly restricted structure. On the one hand, these restrictions help making explicit the structure of the protocol, its principals, channels, and services. On the other hand, these restrictions make it possible to automatically generate code, the protocol software, from the CPNs modelling the protocol. One feature of this class of CPNs are so-called *pragmatics*, which are annotations to certain elements of the CPNs, which indicate the purpose of the respective modelling element and are exploited by the code generator. This way, models from which code can be generated are not cluttered with all kinds of technical information so that the same CPN models can be used for verification and code generation.

The PetriCode approach and tool have been presented, discussed and evaluated in earlier work already [5, 6]. In this paper, we formally define this restricted class of CPNs, which we call *Pragmatic Annotated CPNs* (*PA-CPNs*). In addition, we show that PA-CPNs are still amenable to verification, and that the structural restrictions on that class can actually make the verification more efficient: First, the structure of *PA-CPNs* allows us to automatically add so-called *service testers* to the model of the protocol, which reduce the state space of the model and, therefore, reduce the computation effort needed for verification. Second, the structural restrictions of *PA-CPNs* induce a natural progress measure that can be exploited by a verification technique that is called *sweep-line method* [1, 3], which again makes verification more efficient by reducing the

number of states that need to be stored at the same time in the verification tool. The formal definitions of PA-CPNs are illustrated by a running example, which is a simple framing protocol. By using this example, we also illustrate how the structure of PA-CPNs can exploited for verifying the protocol in a more efficient way.

For the rest of this paper, we assume that the reader is familiar with the basic concepts of Petri nets and high-level Petri nets in general. In Sect. 2, we introduce CPNs by an example and rephrase the standard definitions of CPNs [2]. The example used to explain CPNs in Sect. 2 is already a PA-CPN, but the specific structure mandated by PA-CPNs will first be discussed and formalized in Sect. 3 and Sect. 4: Section 3 covers the definitions concerning the specific pragmatics and restrictions of the different types of PA-CPN modules, and Sect. 4 formalizes one specific aspect, which makes sure that services can be represented by typical constructs for control flow. In Sect. 5, we discuss and formalize the extension of PA-CPNs with so-called service testers, which can be used for more efficiently verifying the model of the protocols. The actual verification by using the sweep-line method is discussed in Sect. 6. At last, in Sect. 7, we sum up the general findings and briefly discuss related work.

2 Protocol Example and Coloured Petri Nets

The definition of PA-CPNs relies on the definition of hierarchical CPNs given in [2]. Below we introduce the basic definitions and notations for hierarchical CPNs and the protocol CPN model that we will use as a running example throughout this paper. We present only the syntactical definition of hierarchical CPNs as PA-CPNs have the same semantics as ordinary hierarchical CPNs for simulation and verification purposes.

2.1 Protocol Example

The CPN model to be used as a running example models a protocol consisting of a *sender* and a *receiver* operating over an unreliable channel which may both re-order and loose messages. The sender sends messages tagged with sequence numbers to the receiver and waits for an acknowledgement for each message to be returned from the receiver before sending the next message. Hence, the protocol operates according to the stop-and-wait principle.

The CPN model of the protocol consists of eight hierarchically organised *modules*. Below we present selected modules of the CPN model used to illustrate the definition and verification techniques in this paper[1]. Figure 1 shows the top-level module consisting of three *substitution transitions* (drawn as double-bordered rectangles) and representing the Sender, the Receiver, and the Channel connecting them. The two *places* SenderChannel and ReceiverChannel model buffering communication endpoints connecting the sender and the receiver to the communication channel. The definition of the *colour set* (type) `Endpoint` determining the kind of tokens that can reside on these two places is provided in Fig. 2. Each of the three substitution transitions has an associated *submodule* indicated by the rectangular tag positioned next to the substitution transition.

[1]The complete CPN model is available via `www.petricode.org/examples/SWProtocol+driver.cpn`

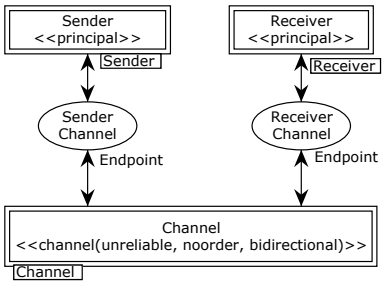

Figure 1: The top-level (prime) CPN module of the protocol model

```
colset Packet  = union DATA : Data + ACK : Ack;
colset EndpointId = INT;

colset ChannelPacket =
 record src : EndpointId * dest : EndpointId
        * packet : Packet;
colset ChannelPackets = list ChannelPacket;

colset Endpoint = record name : EndpointId *
inb : ChannelPackets *
outb : ChannelPackets;
```

Figure 2: Colour set (type) declarations used in Fig. 1

The annotations written in $\langle\langle\ \rangle\rangle$ are the *pragmatics* annotations that we formally introduce in the next section when defining PA-CPNs; they can be ignored for now.

Figure 3 shows the Sender module, which is the submodule associated with the Sender substitution transition in Fig 1 and defines the protocol for the Sender principal. The module has two substitution transitions modelling the main operations of the sender which is the sending of messages (substitution transition send) and the reception of acknowledgements (substitution transition receiveAck). The places ready, runAck, and nextSend are used to model the internal state of the sender. The place ready has an *initial marking* consisting of a token with the colour `()` (unit) which is the single value contained in the predefined colour set `UNIT`. This indicates that initially the sender is ready to perform a send operation. For a place with colour set `UNIT`, we omit (by convention) the specification of the colour set in the graphical representation. The place runAck, which has a boolean colour set, initially contains a token with the value `false` indicating that the sender is not initially in a state where it can receive acknowledgements. The place nextSend is used to keep track of the sequence of the message that the sender is currently sending. The place SenderChannel is a *port place* (indicated by the double border) and is used by the module to exchange tokens with its upper level module, which was shown

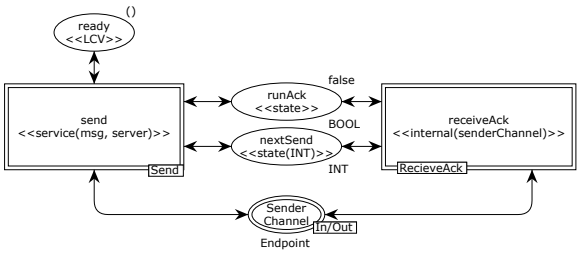

Figure 3: The sender CPN module

in Fig. 1. In this case, SenderChannel is an *input-output port place* as specified by the In/Out tag positioned next to the place. The place is associated with the SenderChannel *socket place* in Fig. 1 which means that any tokens removed (added) from (to) this place in the Sender module will also be reflected in the Protocol module.

Figure 4 shows the Send module which is the submodule of the send substitution transition in Fig. 3. This submodule models the sending of a list of messages from the sender to the receiver. The port places ready, SenderChannel, nextSend, and runAck are associated with the accordingly named socket places in the module shown in Fig. 3. The list of messages to be sent is provided via the place message (top) annotated with the `driver` pragmatic. This place is a *fusion place* as indicated by the rectangular tag positioned next to the place. The name inside the tag specifies the *fusion set* that the place belongs to. A fusion set is a set of places with the property that when tokens are removed (added) to one place in the set, then the token will be removed (added) to all members. Conceptually, all the places of a fusion set are merged into a single compound place. The place end (at the bottom) annotated with a `driver` pragmatic is also a member of a fusion set. These fusion sets are used to connect PA-CPNs to test driver modules to be introduced later; these places are, formally, not part of the service level module or the complete protocol. The places annotated by the `driver` pragmatic are used by the test driver module to control the order and the parameters of the invocation of the services of the protocol during the verification of the protocol (see Sect. 5 and Sect. 6). The code generator ignores these places since, in the actual protocol software, the services of the protocol are invoked externally; the order of invocation of the services and the parameters are determined by the protocol's environment.

The sending of a list of messages starts with the occurrence of the transitions send, which places the list of messages to be sent on place message, puts a token on the place nextSend corresponding to the first sequence number, and a token on place runAck to indicate that acknowledgements can now be received. The place limit is used to put an upper bound on the number of attempts to retransmit a message when the transmission fails. After an occurrence of transition send, transition sendMsg may occur sending a message by putting it in the output buffer modelled by the place SenderChannel. The *guard* used on the transition sendMsg (by convention written in square brackets next to the transition) ensures that the data being sent matches the sequence number of the message currently being sent. If the retransmission limit is reached, the sender will stop as modelled by the transition return putting a token on place end. If

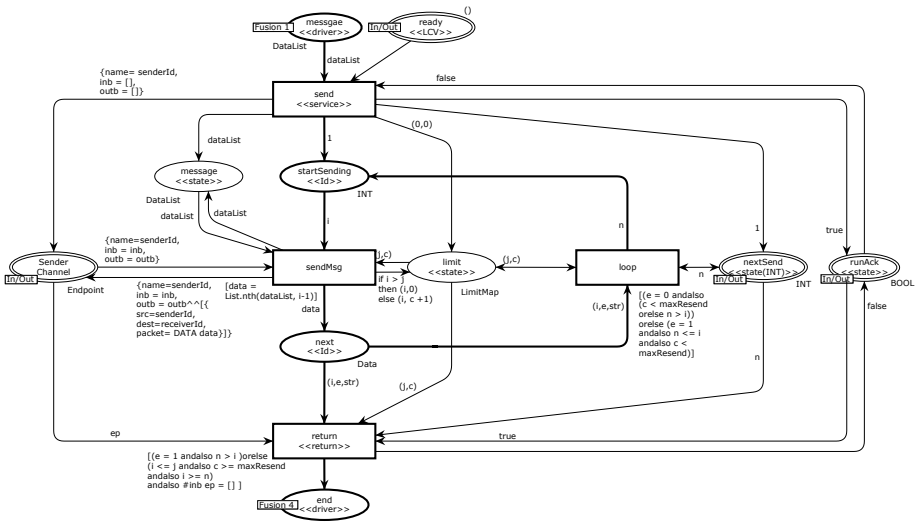

Figure 4: The Send module

```
colset LimitMap = product INT * INT;
colset Data = product  INT * INT * STRING;
colset DataList = list Data;

val maxResend = 2;
var i, j, k, l, c, e, n :INT;
var data : Data;
var dataList : DataList;
var str : STRING;
```

Figure 5: Colour set (type) declarations used in Fig. 4

the retransmission limit is not reached for the current message, the transition loop will put a token back on startSending such that the next message can be sent. The colour set definitions and *variables* used in the inscriptions of Fig. 4 are provided in Fig. 5.

Above, we have presented the example CPN model that will be used as a running example throughout this paper, and we have informally introduced the hierarchical constructs of CPNs in the form of modules, substitution transitions, port and socket places, and fusion places.

2.2 Formal Definitions of Hierarchical CPNs

In this subsection, we formally define hierarchical CPNs as the later formal definition of PA-CPNs will be based on the formal definition of hierarchical CPNs. Definition 2.1 provides the formal definition of CPN modules. 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 $\mathit{Type}[e]$ for an expression e (an arc expression, a guard, or an initial marking) to denote the 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 2.1. A **Coloured Petri Net Module** (Def. 6.1 in [2]) is a tuple $CPN_M = (CPN, T_{\mathsf{sub}}, P_{\mathsf{port}}, PT)$, such that:

1. $CPN = (P, T, A, \Sigma, V, C, G, E, I)$ is a Coloured Petri Net (Def. Y in [2]) 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 $\mathit{Type}[v] \in \Sigma$ 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) $E : A \to \mathit{EXPR}_V$ is an **arc expression function** that assigns an arc expression to each arc a such that $\mathit{Type}[E(a)] = C(p)_{MS}$, where p is the place connected to the arc a.
   (f) $G : T \to \mathit{EXPR}_V$ is a **guard function** that assigns a guard to each transition t such that $\mathit{Type}[G(t)] = \mathit{Bool}$.
   (g) $I : P \to \mathit{EXPR}_\emptyset$ is an **initialisation function** that assigns an initialisation expression to each place p such that $\mathit{Type}[I(p)] = C(p)_{MS}$.
2. $T_{\mathsf{sub}} \subseteq T$ is a set of **substitution transitions**.
3. $P_{\mathsf{port}} \subseteq P$ is a set of **port places**.
4. $PT : P_{\mathsf{port}} \to \{\mathsf{IN}, \mathsf{OUT}, \mathsf{I/O}\}$ 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 transition. 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 are the place 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. 2.3) 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 2.2. A **hierarchical Coloured Petri Net** (Def. 6.2 in [2]) 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. $SM : T_{\mathsf{sub}} \to S$ is a **submodule function** that assigns a **submodule** to each substitution transition. It is required that the module hierarchy (see Definition 2.3) 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 $t \in T_{\mathsf{sub}}$.
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 2.3. 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. $A_{MH} = \{(s_1, t, s_2) \in N_{MH} \times T_{\mathsf{sub}} \times N_{MH} \mid t \in T^{s_1}_{\mathsf{sub}} \wedge s_2 = SM(t)\}$ 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 mandates a particular structure of the CPN models and enables the CPN elements to be annotated with *pragmatics* used to direct the automated code generation. In a PA-CPN, the modules of the CPN model are required to be organised into three levels referred to as the *protocol system level*, the *principal level*, and the *service level*. In a PA-CPN, it is required that there exists exactly one prime module. This prime module represents the protocol system level. The Protocol module shown in Fig. 1 comprises the protocol system level of the PA-CPN model of the framing protocol; it specifies the protocol principals

in the system and the channels 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. In the CPN model, pragmatics are shown by annotations enclosed in guillemets. On the principal level, there is one module for each principal of the protocol as defined on the protocol system level. The framing protocol has two modules at the principal level corresponding to the sender and the receiver. Figure 3 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, which are indicated by the `service` pragmatics: send and receiveAck. Substitution transition representing 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 service level modules model the behaviour of the individual services. The module in Fig. 4 is an example of a module at the service level modelling the send service provided by the sender principal.

We formally define PA-CPNs as a tuple consisting of a hierarchical CPN, a protocol system module (PSM), a set of principal level modules (PLMs), a set of service level modules (SLMs), a set of channel modules (CHMs), and a structural pragmatics mapping (SP) that maps substitution transitions into structural pragmatics and capturing the annotation of the substitution transitions. It should be noted that since channel modules do not play a role in the code generation but are only a CPN model artifact used to connect the principals for simulation purposes, we do not impose any specific requirements to the internal structure of channel level modules.

Definition 3.1. 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.
2. $PSM \in S$ is a **protocol system module** (see Def. 3.2) and the only prime module of CPN_H.
3. $PLM \subseteq S$ is a set of **principal level modules** (see Def. 3.3).
4. $SLM \subseteq S$ is a set of **service level modules** (see Def. 3.4).
5. $CHM \subseteq S$ is a set of **channel modules**.
6. PSM, PLM, SLM, CHM constitute a partition of S.
7. $SP : T_{sub} \rightarrow \{\texttt{principal}, \texttt{service}, \texttt{internal}, \texttt{channel}\}$ is a **structural pragmatics mapping** such that:
   (a) Substitution transitions annotated with a `principal` pragmatic have an associated principal level module:
   $\forall t \in T_{sub} : SP(t) = \texttt{principal} \Rightarrow SM(t) \in PLM$
   (b) Substitution transitions annotated with a `service` or `internal` pragmatic are associated with a service level module:
   $\forall t \in T_{sub} : SP(t) = \texttt{service} \wedge SP(t) = \texttt{internal} \Rightarrow SM(t) \in SLM$

(c) Substitution transitions annotated with a `channel` pragmatic are associated with a channel module:
$\forall t \in T_{sub} : SP(t) = \texttt{channel} \Rightarrow SM(t) \in CHM$

□

The protocol system module (PSM) models the principals of the protocol and the channels that connects them. 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 to the module is that all substitution transitions must be substitution transitions and annotated with either a `principal` or a `channel` pragmatic. Furthermore, two substitution transitions representing principals cannot be connected only by a place, there must be a substitution transition representing a channel in between. This reflects the fact that it is possible for principals to communicate via channels only[2].

Definition 3.2. A **Protocol System Module** of a PA-CPN with a structural pragmatics mapping SP_{PA} is a tuple $CPN_{PSM} = (CPN^{PSM}, PM)$, where:

1. $CPN^{PSM} =$
$((P^{PSM}, T^{PLM}, A^{PSM}, \Sigma^{PSM}, V^{PSM}, C^{PSM}, G^{PSM}, E^{PSM}, I^{PSM}),$
$T^{PSM}_{sub}, P^{PSM}_{port}, PT^{PSM})$ is a CPN module (see Def. 2.1).
2. All transitions are substitution transitions $T^{PSM} = T^{PSM}_{sub}$.
3. $PM : T^{PSM}_{sub} \rightarrow \{\texttt{principal}, \texttt{channel}\}$ is a **pragmatics mapping** satisfying:
   (a) All substitution transitions are annotated with either `principal` or `channel` pragmatic: $\forall t \in T^{PSM}_{sub} : PM(t) \in \{\texttt{principal}, \texttt{channel}\}$.
   (b) The pragmatics mapping must coincide with the structural pragmatic mapping of PA-CPN: $\forall t \in T^{PSM}_{sub} : PM(t) = SP(t)$.
   (c) All places are connected to at most one substitution transition annotated with principal and at most one annotated with channel:
   $\forall p \in P^{PSM} : \forall t_1, t_2 \in X(p) : PM(t_1) = PM(t_2) \Rightarrow t_1 = t_2$.

□

A principal level module specifies the services provided by a principal and is defined as a tuple consisting of a CPN module and a principal level pragmatic mapping PLP. Each service is represented by a substitution transition which can be annotated with either a service or internal pragmatic depending on whether the service is visible externally or not. The non-port places of a principal level model can be annotated with either a `state` or an `LCV` pragmatic. Places annotated with a `state` pragmatic represent internal states of the principal, whereas 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. 3 ensures that only one message at a time is sent using the send service.

[2]In the definition, we use $X(p)$ to denote the set of transitions connected to a place p.

Definition 3.3. A **Principal Level Module** of a PA-CPN with a structural pragmatics mapping SP_{PA} is a tuple $CPN_{PLM} = (CPN_{PLM}, T^{PLM}_{sub}, P^{PLM}_{port}, PT^{PLM}, PLP)$ where:

1. $CPN_{PLM} = (P^{PLM}, T^{PLM}, A^{PLM}, \Sigma^{PLM}, V^{PLM}, C^{PLM}, G^{PLM}, E^{PLM}, I^{PLM})$ is a CPN module (see Def. 2.1).
2. All transitions are substitution transitions: $T^{PLM} = T^{PLM}_{sub}$
3. $PLP : T^{PSM}_{sub} \cup P^{PLM} \setminus P^{PLM}_{port} \rightarrow \{\texttt{service}, \texttt{internal}, \texttt{state}, \texttt{LCV}\}$ is a **principal level pragmatics mapping** satisfying:
   (a) All non-port places are annotated with either a `state` or a `LCV` pragmatic: $\forall p \in P^{PLM} \setminus P^{PLM}_{port} \Rightarrow PLP(p) \in \{\texttt{state}, \texttt{LCV}\}$
   (b) All substitution transitions are annotated with a `service` or `internal` pragmatic: $\forall t \in T^{PSM}_{sub} : PLP(t) \in \{\texttt{service}, \texttt{internal}\}$.

□

It should be noted that we do not associate any pragmatics with the port place of the module as it follows from the definition of the protocol system module that a port place in a principal level module can only be associated with a socket place connected to a channel substitution transition.

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 Send module in Fig. 4 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 a `service` pragmatic, and the exit (termination) point of the service is indicated by annotating a single transition with a `return` pragmatic. In addition, a non-port place can be annotated with a state pragmatic to indicate that this place models a local state of the service. The `driver` is used by service tester modules to facilitate verification by reducing the state space of the protocol model. The places annotated with an `Id` pragmatic determine a subset of the module, which we call the *underlying control-flow net*; and it is required that this net is block decomposable (which will be defined later in Sect. 4) in order to support a natural translation into programming language control flow structures. In Fig. 4, the underlying control flow net is highlighted via the places, transitions, and arcs having a thick border. A service level module is defined as consisting of a CPN module without substitution transitions and with a service level pragmatic mapping that associates pragmatics to the model elements as described above.

In the definition below we use the notation $\exists! x \in X : p(x)$ to denote that there exists exactly one element x in a set X satisfying a predicate p – i. e. the element x is uniquely characterized by property $p(x)$.

Definition 3.4. A **Service Level Module** is a tuple $CPN_{SLM} = (CPN_{SLM}, T^{SLM}_{sub}, P^{SLM}_{port}, PT^{SLM}, SLP, SLT)$ where:

1. $CPN_{SLM} = (P^{SLM}, T^{SLM}, A^{SLM}, \Sigma^{SLM}, V^{SLM}, C^{SLM}, G^{SLM}, E^{SLM}, I^{SLM})$ is a CPN module (see Def. 2.1)
2. There are no substitution transitions: $T^{SLM}_{sub} = \emptyset$.
3. $SLP : T^{PLM} \cup P^{PLM} \setminus P^{PLM}_{port} \to \{\texttt{Id}, \texttt{state}, \texttt{service}, \texttt{return}, \texttt{driver}\}$ is a **service level pragmatic mapping** satisfying:
   (a) Each place is either annotated with `Id`, `state`, `driver` or is a port place : $\forall p \in P^{SLM} \setminus P^{SLM}_{port} : SLP(p) = \texttt{Id} \vee SLP(p) = \texttt{state} \vee SLP(p) = \texttt{driver}$
   (b) There exits exactly one transition annotated with `service`: $\exists! t \in T^{SLM} : SLP(t) = \texttt{service}$
   (c) There exits exactly one transition annotated with `return`: $\exists! t \in T^{SLM} : SLP(t) = \texttt{return}$
4. For all $t \in T^{SLM}$ and $p \in P^{SLM}$ we have:
   (a) Transitions consume one token from input places annotated 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 annotated 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 annotated with a `service` pragmatic can have input places annotated with a `driver` pragmatic: $(p,t) \in A^{SLM} \wedge SLP(p) = \texttt{driver} \Rightarrow SLP(t) = \texttt{service}$
   (d) Only transitions annotated with a `return` pragmatic can have output places annotated with a `driver` pragmatic: $(t,p) \in A^{SLM} \wedge SLP(p) = \texttt{driver} \Rightarrow SLP(t) = \texttt{return}$
5. The underlying control flow block of CPN_{SLM} (Def. 4.2) is tree decomposable (Def. 4.4).

□

4 Block Decomposition of Control Flow Nets

In this section, we define formally when the control flow of a service level module is decomposable into blocks. Figure 6 shows the underlying control flow net of the service level module for the send operation of the sender from Fig. 4, which is a loop construct, basically.

In order to formally define the block decomposition, we need to define blocks first: these are Petri nets with a fixed entry and exit place. Then, we define the underlying control flow net of a service module. At last, we define when a block is decomposable into blocks, which represent the control flow constructs.

Figure 7 shows a graphical representation of a block in general, where the start and end place of the block are graphically represented by arcs from resp. to the border of the block.

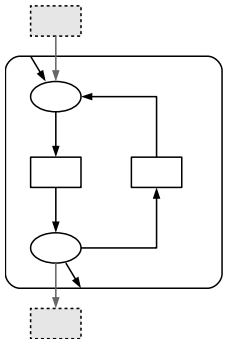

Figure 6: Decomposition of the control flow net of module SenderSend

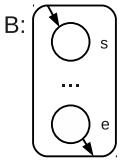

Figure 7: Graphical representation of a block.

Definition 4.1 (Block, atomic non-returning block)**.**
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 ${}^\bullet s = \emptyset$ (i. e. the block will not return a token to the start place itself). The block has a *safe exit*, if $s \neq e$ and $e^\bullet = \emptyset$ (i. e. the block does not use a token from the end place itself).

An atomic block consists of a single transition, as shown in Fig. 10 later. For visualizing blocks with safe entry and safe exit, we introduce an additional graphical notation, which is shown in Fig. 8. The crossed out arc from within the block to the start place indicates that the block itself does not return a token to the entry place (safe entry); the crossed out arc from the end place to the interior of the block indicates that the block itself does not remove a token from its exit place (safe exit).

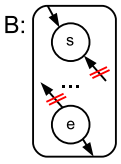

Figure 8: Graphical representation of safe entry and safe exit.

For easing the following definitions, we introduce an additional notation: For a block B_i, we refer to its constituents by $B_i = (P_i, T_i, A_u, s_i, e_i)$ without explicitly naming them every time.

The block that is underlying a service level model is basically obtained by all the places that are annotated with the `Id` pragmatics and the transitions in their pre- and postsets. The unique transition with the `service` pragmatics defines the entry place, and the unique transition with the `return` 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. 6

Definition 4.2. [Underlying control flow net of SLM] Let CPN_{SLM} be a service level module as defined in Def. 3.4. 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 $SLP(t) = \texttt{service}$, 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 $SLP(t) = \texttt{return}$.

Then, we call $N = (P, T, A, s, e)$ the **underlying control flow net** of CPN_{SLM}.

The control flow of the generated code will be obtained by decomposing the underlying control flow net of a service level module into sub-blocks, which represent the control flow constructs. Note that we define the decomposition in a very general way at first, which does not yet restrict the possible control 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. 4.4).

Definition 4.3 (Decomposition of a block)**.**
Let $B = (N, s, e)$ be a block with net $N = (P, T, F)$.

A set of blocks $B_1, \ldots, B_n$ is called a decomposition of block B, if the following conditions are met:

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 $P_i \cap P_j = \{s_i, e_i\} \cap \{s_j, e_j\}$.

Note that, in some cases, two consecutive blocks should be safe, which means that either the exit of the preceding block is safe, or the entry of the succeeding block is safe or both. We represent this graphically as shown in Fig. 9.

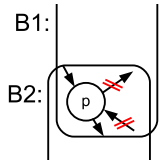

Figure 9: Safe join of two consecutive blocks

At last, we define when a decomposition of a block reflects some control flow construct. Note that this definition does not only define decomposability into control constructs; it also defines a tree structure which reflects the control structure of the block. The formal definition is illustrated in Fig. 10.

Definition 4.4 (Tree decompositions of a block)**.**

The *block trees* associated with a block are inductively defined:

- If B is an atomic block, then the tree with the single node $B : \mathit{atomic}$ is a *block tree* associated with B.
- 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 $B_2 : \mathit{atomic}$ is a block tree associated with B_2, and if B_1 has a safe entry and a safe exit and $s_1 = s$, $e_1 = e$, $s_2 = e$, and $e_2 = s$, then the tree with top node $B : \mathit{loop}$ and the sequence of sub-trees $B_1 : X$ and $B_2 : \mathit{atomic}$ is a *block tree* associated with B.
- 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 every $i \in \{1, \ldots, n\}$ we have $s_i = s$ and $e_i = e$, then the tree with top node $B : \mathit{choice}$ with the sequence of sub-trees $B_i : X_i$ is a *block tree* associated with B.
- 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 $B : \mathit{sequence}$ and the sequence of sub-trees $B_i : X_i$ is a *block tree* associated with B.

Note that the tree decomposition of a block is not necessarily unique. For example a longer sequence of atomic blocks could be decomposed into different junks. The reason is that sequences can have arbitrary length according to our definition, which makes the definitions much more elegant and allows us to have long sequences in a single sequence construct. The tool actually resolves this ambiguity by making blocks with a sequence as large as possible.

Fig. 4 is an example of an SLM. Its underlying control flow net was shown in Fig. 6. This block is decomposed in a loop, which in turn consists of an atomic block. The service transition itself as well as the return transition are actually not part of the underlying control flow net.

5 Service Testers

The service level modules constitute the active part of a PA-CPN model, and the execution of the individual service provided by a principal starts at the transition annotated with a `service` pragmatic. The transitions annotated with a service pragmatic typically has a number of parameters which need to be

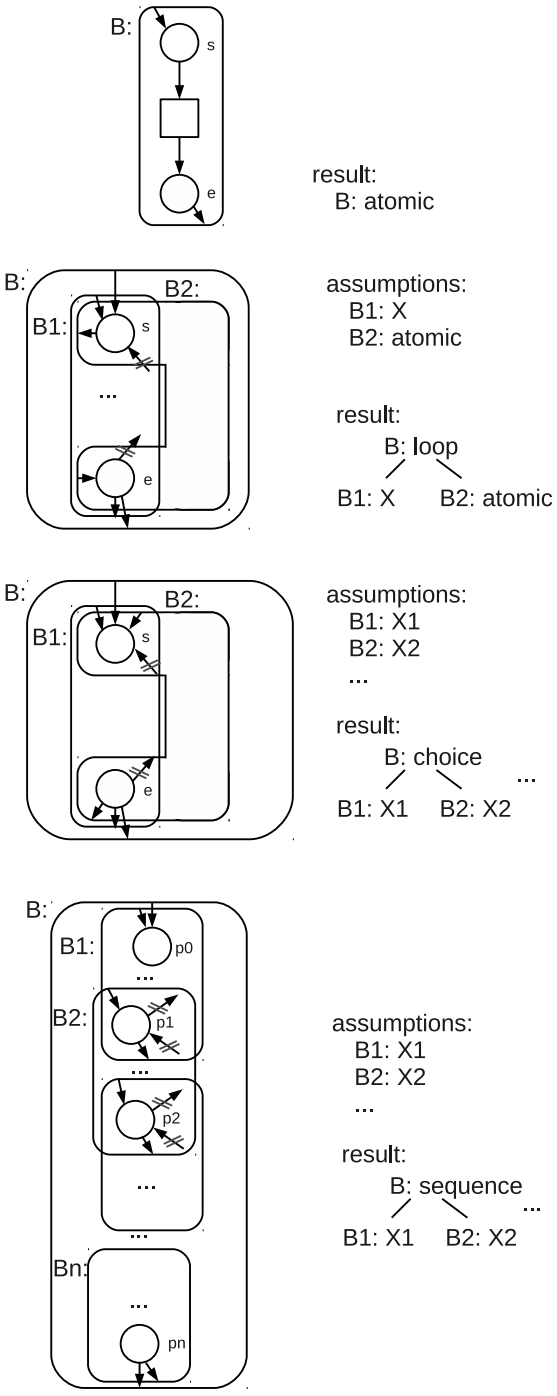

Figure 10: Inductive definition of block trees

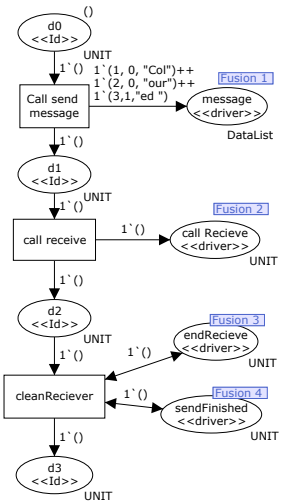

Figure 11: The Service tester

bound to values in order for the transition to occur. An example of this is the Send service transition in Fig. 4 which has dataList is 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 when conducting validation by means of simulation and verification by means of state space exploration, we introduce the concept of *service tester modules* which can be used to guide the validation and verification process and represent a user of the service provided by the principal modules. An added advantage of service testers is that they can further contribute to reducing the state space during verification and progress measures can be automatically computed and used in conjunction with the sweep-line method for state-space exploration as will be explained in Sect. 6.

The service tester modules will be connected to the rest of the PA-CPN model through fusion places, and the service tester modules invokes the service provided by the principal by putting tokens on these places. Similarly, the service tester also receives any results from the invoked services via tokens on these fusion places. The fusion places used to connect service level modules and service tester modules are the only way fusion places are used on top of the concepts of PA-CPNs. In addition to the fusion places, a service tester module has an explicit control flow path similar to service level modules and `Id` pragmatics are used to make this explicit.

Figure 11 shows a server 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 is allowed to have only a single `Id` place that initially contains a token. In the case of Fig. 11, this is the place d0. The test driver first invokes the send service in Fig. 4 by putting a token in the fusion place message. Next, the service tester invokes the receive service in the receiver principal.

Below we formalise the control flow part of a service tester. In the definition, we use $I(p)\langle\rangle$ to denote the result of evaluating the initial marking expression $I(p)$ of a place p.

Definition 5.1. A **Service Tester Module** is a tuple $CPN_{STM} = (CPN_{STM}, T^{STM}_{sub}, P^{SLM}_{port}, PT^{SLM}, TPM)$ where:

1. $CPN_{SLM} = (P^{SLM}, T^{SLM}, A^{SLM}, \Sigma^{SLM}, V^{SLM}, C^{SLM}, G^{SLM}, E^{SLM}, I^{SLM})$ is a CPN module (see Def. 2.1)
2. There are no substitution transitions: $T^{STM}_{sub} = \emptyset$.
3. $TPM : P^{STM} \rightarrow \{\texttt{Id}, \texttt{driver}, \texttt{LCV}\}$ is a **service tester pragmatic mapping**.
4. $\exists! p \in I$ such that $|I^{STM}(p)\langle\rangle| = 1$.
5. For all $t \in T^{SLM}$ and $p \in P^{SLM}$ we have:
   (a) Transitions consume one token from input places annotated with an `Id` pragmatic: $(p,t) \in A^{SLM} \wedge TPM(p) = \texttt{Id} \Rightarrow |E(p,t)\langle b\rangle| = 1$ for all bindings b of t.
   (b) Transitions produce one token on output places annotated with an `Id` pragmatic: $(t,p) \in A^{SLM} \wedge TPM(p) = \texttt{Id} \Rightarrow |E(t,p)\langle b\rangle| = 1$ for all bindings b of t.
6. Transitions and places annotated with a `LCV` pragmatic must be connected with a double arc:
   $\forall p \in P^{STM}, t \in T^{SLM} : TPM(p) = \texttt{LCV} \Rightarrow ((t,p) \in A \Leftrightarrow (p,t) \in A)$
7. The underlying control flow block of CPN_{STM} (Def. 4.2) is tree decomposable (Def. 4.4)

□

As explained above, the idea is that a set of service tester modules can be connected to a PA-CPN by means of fusion places in order to control the execution of the services. Formally, 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. 3.1 and a set of service tester modules which all constitute prime modules. Furthermore, we 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.

Definition 5.2. A **Pragmatics Annotated Coloured Petri Net with Service Testers** is tuple $CPN_{PAT} = (CPN_H, PSM, PLM, SLM, CHM, SP, STM)$, where:

1. $CPN_H = (S, SM, PS, FS)$ is a hierarchical CPN.
2. $CPN_{PAT} = (CPN_H, PSM, PLM, SLM, CHM, SP)$ is a PA-CPN
3. $STM \in S$ is a set of service tester modules all of which are prime modules.
4. The following conditions hold for all fusion sets $fs \in FS$:

(a) Places in a fusion set are either all annotated with a `driver` pragmatic or all annotated with a `LCV` pragmatic.

(b) A fusion set containing places with `driver` pragmatics can only contain places from a single service layer module and a single service tester module.

(c) A fusion set containing places with `LCV` pragmatics can only contain places related via a port-socket relationship or places belonging to service tester modules.

(d) If $p \in fs$ belongs to a service level module and has an output arc to a transition with a `service` pragmatic, then all places $p_t \in fs$ belonging to a service tester module STM have only input arcs from transitions in STM.

(e) If $p \in fs$ belongs to a service level module and has an input arc to a transition with a `return` pragmatic, then all places $p_t \in fs$ belonging to a service tester module STM can have output arcs to transitions in STM only.

□

6 Verification

State space exploration is the main verification method for CPNs and is based on the idea of explicitly enumerating the set of reachable states of the CPN model. Generally, this approach is limited by the available memory since the states need to be stored while the state space is generated. A large collection of techniques have been developed in order to alleviate this inherent complexity problem. In this section, we show how the sweep-line method [1, 3] can be used to alleviate the state explosion problem when conducting verification of PA-CPNs with service testers.

6.1 The Sweep-Line Method

The basic idea of the sweep-line method is to exploit a notion of *progress* exhibited by many systems. Exploiting progress makes it possible to explore all reachable states while storing only small subsets of the state space in memory at a time. This way, much larger state spaces can be investigated since never all states need to be stored at the same time. The additional structure imposed on CPNs by PA-CPNs and services testers means that PA-CPN models have several potential sources of progress that can be exploited by the sweep-line method. 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 by a principal is 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.

The subsets of states stored 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. The basic idea is to optimistically assume that the system does not *regress*, and hence states with a lower progress value will not be visited again and do not need to be kept in memory. If it turns out that the system regresses, 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 some states multiple times. The fact that the sweep-line method deletes states means that verification of properties needs to be conducted on-the-fly during the 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) and $\rightarrow^*$ the reachability relation of the CPN model:

Definition 6.1 (Progress Measure)**.** 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 $\forall s, s' \in \mathcal{R}(\iota) : s \rightarrow^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$. Otherwise, P is **non-monotonic**. □

The sweep-line method does not mandate any origin of the progress measure and in many cases the progress measure is provided by the modeller based upon knowledge about the modelled systems. For PA-CPNs, however, a reasonable progress measure can be derived from the model automatically as we will show in the next section.

6.2 Progress Measures for Sweep-line Verification Methods

For our example protocol above, the progress measure could be a vector of measures using the number of tokens on some of its places (omitting the parts of the model that we did not show in this paper):

$$(|d0|, |d1|, |d2|, |d3|, |startSending| + |next|, |end|, \ldots)$$

The order on two such vectors would 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. 11). The next two entries represent the progress within the send service (Fig.4); note that since the places *startSending* 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):

$$(|d0|, |d1|, |d2|, |d3|, |startSending|, |next|, |end|, \ldots)$$

The difference between the two are based on how loops are handled. In this progress the places on loops are append to the vector as if the loop was not there. In the present example this is shown by havin replaced the + operator between *startSending* and *next* with a comma.

We used the service tester as the first and, therefore, most significant measures since these are indicating the progress within the test. Then we measure the progress within the service. In some cases, it might make sense to take life-cycle variables into account for measuring the progress. But, this depends very much on the protocol and whether the life-cycle variables monotonically increase in the course of the protocol. In our example, the life-cycle variable *ready* of the sender module does not indicate any progress; therefore, it is not part of the progress measure that we have shown above.

Generally, coming up with a good progress measure requires some experience and a good understanding of the protocol. For the test drivers and the services modules, however, some reasonable progress measures can be derived automatically by exploiting the block structure of the respective modules (a sequence for the tester and a loop for the service, in our example). We formalize this below, basically generalizing the idea from the above example.

The progress measure is defined on top of the tree decomposition of the blocks underlying the corresponding service tester model or the service module. Technically, the tree decomposition of the blocks was formally defined for service level modules only. It is straight forward to adjust this definition to service tester modules, but we do not formalize that here. In the following, we assume that we have the tree decomposition of the respective module. Then we define a simple progress measure and a complex one. The simple one, would just add up the number of all tokens in loops, not looking into their detailed structure; the complex one would also take the progress within loops into account.

Definition 6.2 (Progress measures)**.** Let BT be a block tree for a CPN module.

The sequence of *simple progress* measures entries for BT is defined inductively over the block tree BT of the CPN module:

- If BT is $B : atomic$, then simple progress sequence consist of $|s|, |e|$ where s is the entry place of the block B and e is the exit place.
- If BT is $B : sequence$ with sub blocks $B_1, \dots B_n$, and $e_i^1, \dots, e_i^{k_i}$ are the simple progress sequences for B_i, then $e_1^1, \dots, e_1^{k_1-1}, e_2^1, \dots, e_2^{k_2-1}, \dots, e_n^1, \dots, e_n^{k_n}$ is the simple progress sequence for BT.
- If BT is B : *choice* with sub blocks $B_1, \dots B_n$, and $e_i^1, \dots, e_1^{k_i}$ are the simple progress sequence for each block B_i, then the sequence $e_1^1, \dots, e_1^{k_1-1}, e_2^2, \dots, e_2^{k_2-1}, \dots, e_n^2, \dots, e_n^{k_n}$ is the simple progress sequence for BT
- If BT is $B : loop$ with places $p_1, \dots p_n$, then either the single entry $|p_1| + \dots + |p_n|$ is the simple progress sequence for BT.

The sequence of *complex progress* measures entries is defined inductively over the block tree BT of the CPN module:

- 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.

- If BT is B : *sequence* with sub blocks $B_1, \ldots B_n$, 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 simple sequence for BT.
- If BT is B : *choice* with sub blocks $B_1, \ldots B_n$, 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
- 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 also the complex progress sequence for BT.

Now, the progress measures for the complete system can be built from the progress sequences (either the simple or the complex ones) 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 in our example).

Note that for each single service either the simple or the complex measure could be chosen. It very much depends on the nature and the depth of the loop which of these choices help reducing the effort for exploring the state space. The choice for each service level module would be left to the modeller. In principle, it is even possible to combine the complex measure and the simple measure within a single service level module – starting with the complex measure for the outer loop constructs and then switching to the simple measure at some nesting level. But, defining these combined measures for a single service level module would result in a very technical definition. Therefore, we do not formally define combined complex and simple measures for a single service here.

Anyway, as for all heuristics, these measures serve as a good guess only; it might still be possible to be improved by choosing the simple or the complex progress measures for the different modules or by by adding some entries concerning the the live-cycle variables. Such manual manipulations would also be left to the modeller.

6.3 Verification Results

In order to demonstrate the feasibility of verification using the sweep-line method and the progress measures defined above, we present the verification of a simple end state property assuring that the protocol will terminate in a consistent state. The property checks that all the modules are ended in all final states. This was checked manually and by automatically checking all end states with the simple predicate *P1* shown below. The predicate says that the four places d3, endFinalAtomic,endRecAck and endRec are all marked with a unit token. This means that the service tester and all the services are terminated.

- **P1:** d3 = [()] andalso endFinalAtomic = [()] andalso endRecAck = [()] andalso endRec = [()]

Configuration	Visited states	Peak stored	Num unique end states	P1	time
1 msg, non-lossy	156	77	2	yes	0.034905s
1 msg, lossy	186	99	2	yes	0.029867s
3 msg, non-lossy	2222	2014	4	yes	0.399659s
3 msg, lossy	2928	2700	4	yes	0.643134s
7 msg, non-lossy	117584	115373	8	yes	216.197694s
7 msg, lossy	160620	158388	8	yes	532.674399s

Table 1: Verification results using the simple progress measure

Configuration	Visited states	Peak stored	Num unique end states	P1	time
1 msg, non-lossy	165	63	2	yes	0.029353s
1 msg, lossy	196	78	2	yes	0.035034s
3 msg, non-lossy	2790	1582	4	yes	0.489735s
3 msg, lossy	4037	2187	4	yes	0.855804s
7 msg, non-lossy	143531	86636	8	yes	32.384360s
7 msg, lossy	263608	124661	8	yes	80.835973s

Table 2: Verification results using the complex progress measure

We ran the verification using two different progress measures. The results of the verification are shown in Table 1 using the simple progress measure and Table 2 using the complex progress measure. We used two configuration parameters, the number of messages to be sent and whether the channel is lossy. We see that the predicate holds for all configurations. Also, we see that the number of states grows fairly fast with the number of messages. We see that the verification using the simple progress measure consistently visits fewer states (counting duplicate encounters twice) than the complex progress measure. This is because the states surrounding and inside loops are added, which means they count as a single progress measure point by the simple progress measure, while they are all included as if the loop was a sequence using the complex progress measure. The peak number of states stored at the same time, however, is consistently lower for the complex progress measure. This means that the peak memory consumption, is lower using this metric. Furthermore, for the larger state spaces, the time the verification takes is also significantly lower using the complex progress measure. This means that the complex measure is probably preferable with large state spaces when the model includes loops. It is likely that a tailored progress measure would out-perform both the complex and simple progress measure.

7 Conclusion and Related Work

In this paper we have presented a formal definition of Pragmatics Annotated Coloured Petri Nets (PA-CPNs), the net class that forms the basis for our code generation technique. Furthermore we have shown that the structure of PA-CPNs can be exploited to automatically derive some suitable progress measures for the sweep-line method.

PA-CPNs are not the first formally defined sub-class of CPNs for code generation: also Process-Partitioned CPNs (PP-CPNs) [4, 7] were defined for making code generation possible. One important advantage of PA-CPNs over PP-CPNs

is that they clearly display the available services of a principal in the PLMs. With PA-CPNs, we are also able to use the PLMs to reduce the number of states in memory at any one time during state-space generation by taking into account the `LCV` places, even though we could not exploit that in the example discussed in this paper.

PA-CPNs have been introduced mainly with code generation in mind (with different objectives as discussed in [5]). In this paper, we have formally defined PA-CPNs, and it turned out that the objective of code generation does not spoil the possibility of verifying the respective models; on the contrary, the additional structure can even be exploited for improving verification in combination with the sweep-line method.

References

[1] Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. A sweep-line method for state space exploration. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS, Proceedings*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.

[2] K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[3] K. Jensen, L.M. Kristensen, and T. Mailund. The sweep-line state space exploration method. *Theoretical Computer Science*, 429:169–179, 2012.

[4] 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.

[5] 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.

[6] K.I.F. Simonsen. PetriCode: A Tool for Template-based Code Generation from CPN Models. In *To appear in Proc. of WS-FMDS 2013*, 2013.

[7] M. Westergaard. Verifying parallel algorithms and programs using coloured petri nets. In *TOPNOC VI*, volume 7400 of *LNCS*, pages 146–168. Springer, 2012.