Code Generation for Protocols from CPN models Annotated with Pragmatics

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

Publication date:
2013

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

Code Generation for Protocols from CPN models Annotated with Pragmatics

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

January, 2013

IMM-Technical Report-2013-01

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

DTU Informatics
Department of Informatics and Mathematical Modeling
Technical University of Denmark

Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-Technical Reports: ISSN 1601-2321

Abstract

Model-driven engineering (MDE) provides a foundation for automatically generating software based on models. Models allow software designs to be specified focusing on the problem domain and abstracting from the details of underlying implementation platforms. When applied in the context of formal modelling languages, MDE further has the advantage that models are amenable to model checking which allows key behavioural properties of the software design to be verified. The combination of formally verified models and automated code generation contributes to a high degree of assurance that the resulting software implementation satisfies the properties verified for the model.

Coloured Petri Nets (CPNs) have been widely used to model and verify protocol software, but limited work exists on using CPN models of protocol software as a basis for automated code generation. In this report, we present an approach for generating protocol software from a restricted class of CPN models. The class of CPN models considered aims at being *descriptive* in that the models are intended to be helpful in understanding and conveying the operation of the protocol. At the same time, a descriptive model is close to a verifiable version of the same model and sufficiently detailed to serve as a basis for automated code generation when annotated with *code generation pragmatics*. Pragmatics are syntactical annotations designed to make the CPN models descriptive and to address the problem that models with enough details for generating code from them tend to be verbose and cluttered.

Our code generation approach consists of three main steps, starting from a CPN model that the modeller has annotated with a set of pragmatics that make the protocol structure and the control-flow explicit. The first step is to compute for the CPN model, a set of derived pragmatics that identify control-flow structures and operations, e. g., for sending and receiving packets, and for manipulating the state. In the second step, an *abstract template tree* (ATT) is constructed providing an association between pragmatics and code generation templates. The ATT then directs the code generation in the third step by invoking the code templates associated with each node of the ATT in order to generate code. We illustrate our approach using an example of a unidirectional data framing protocol.

Contents

1 Introduction

Model-driven engineering (MDE) [7] provides a foundation for automatically generating software based on models. Models allow software designs to be specified focusing on the problem domain – abstracting from the details of underlying implementation platforms. If the MDE process starts from modelling languages that have a precise behavioural semantics, we gain the additional advantage that the models are amenable to model checking [1], which allows key behavioural properties of the software design to be verified. The combination of formally verified models and automated code generation increases the confidence that the resulting software implementation is correct with respect to the properties that have been verified for the model.

Coloured Petri Nets (CPNs) [6, 5] have been widely used for modelling and verifying protocol software [9], but limited work exists on using CPN models of protocol software as a basis for automated code generation. The hierarchical structure of CPN models fits well with the protocol software domain and so does the concurrency concept in CPNs. Since CPN models are executable, simulations and state space-based verification can be conducted with the aim of validating the correctness of a protocol model. In this report, we present an approach for generating protocol software from a restricted class of CPN models. The class of CPN models considered aims at being *descriptive* in that the models are intended to be helpful in understanding and conveying the operation of the protocol. At the same time, a descriptive model is close to a verifiable version of the same model and sufficiently detailed to serve as a basis for automated code generation when annotated with *code generation pragmatics*. *Pragmatics* are syntactical annotations that are associated with CPN model elements (e. g., places, transitions, and inscriptions). The primary purpose of the pragmatics is to address the problem that models with enough details for generating code from them tend to be verbose and cluttered. The pragmatics considered fall into three main types: structural, control-flow, and operation pragmatics. Our approach furthermore relies on a set of core pragmatics that are intended to be applicable to all protocols. In addition, our approach is extensible in that it allows the modeller to add new pragmatics if required by the specific protocol under consideration.

Our code generation approach consists of three main steps, starting from a CPN model that the modeller has annotated with a set of pragmatics that makes the protocol structure and the control-flow explicit. The first step is to compute for the CPN model, a set of derived pragmatics that identify common control-flow structures and operations, such as sending and receiving packets, or manipulating states. In the second step, an *abstract template tree* (ATT) is constructed providing an association (binding) between pragmatics and code generation templates. Essentially, every node of the ATT is associated with a code template. In the third step, the ATT is traversed and code is generated by invoking the code templates associated with each node of the ATT.

This report is organised as follows: Section 2 describes a CPN model of a unidirectional framing protocol that is used as a running example throughout this report. Section 3 describes the required structure of the class of CPN models that can be used for code generation. Section 4 gives an overview of the process of generating runnable protocol software using our approach. Section 5 shows how we capture common control-flow structures, and presents an associ-

ated set of pragmatics. Section 6 presents some operation pragmatics. Section 7 describes the abstract template tree data structure that is used as an intermediate representation of the protocol during code generation. Section 8 details how code is generated from abstract template trees. Finally, in Sect. 9, we provide a summary and discuss directions for future work.

The reader is assumed to be familiar with Petri nets and the basic ideas of high-level Petri nets, i. e., the combination of Petri nets with a programming language. CPNs belong to the class of high-level Petri nets and uses the Standard ML (SML) programming language as a basis for defining colour sets (types), declaring variables, and implementing arc and guard expressions.

2 Protocol Example

To present our modelling approach and methodology which supports automated code generation, we consider as an example a unidirectional framing protocol. The overall service provided by this protocol is to send *messages* of arbitrary length from a sender to a receiver by splitting up the message into smaller *packets* sent across a unidirectional channel. The channel is assumed to be reliable and to preserve the order of the transmitted packages (i. e. it is a reliable FIFO channel). The protocol uses a *final bit* in each transmitted packet indicating whether the payload of the packet is the final (last) part of the larger message.

A CPN model is hierarchically organised into a set of *modules* (pages) using *substitution transitions*. Each substitution transition has an associated module which then becomes a *submodule* of the module on which the substitution transition resides. Places connected to a substitution transition are called *socket places* and are associated with *port places* on the submodule of the substitution transition. Any token added/removed to a socket place will be added/removed from an associated port place and vice versa. This implies that associated port and socket places will always have the same marking and this provides the mechanism that allows submodules to exchange tokens with upper level modules.

Our modelling approach for protocols relies on structuring a CPN model into three hierarchical levels: the *protocol system level*, the *principal level*, and the *service level*. In the following we show how the unidirectional framing protocol is modelled with our modelling approach. As we proceed with presenting the CPN model, we also introduce the basic set of core pragmatics that are central to our approach and which the modeller uses as part of the construction of the CPN model. Pragmatics are by convention written in $\langle\langle\ \rangle\rangle$ to distinguish them from text labels (e. g., place and transition names) and SML inscriptions.

2.1 Protocol System Level

Figure 1 shows the top-level module of the CPN model which constitute the protocol system level. The purpose of the protocol system level is to specify the *protocol principals* and the *channels* connecting them. This module has three substitution transitions named Sender, Channel, and Receiver. Substitution transitions are by convention indicated by a double lined borders, and the name of the associated submodule is written in the small rectangular tag below the substitution transition. The two substitution transitions Sender and Receiver represents the two principals of the protocol, and the substitution transition

Channel represents a channel between them. We use the ⟨⟨principal⟩⟩ pragmatic to specify which substitution transitions represent protocol principals, and the ⟨⟨channel⟩⟩ pragmatic to specify substitution transitions representing channels. The channel pragmatic has three associated *properties* specifying that the channel is unidirectional, reliable (i. e., the channel does not loose packets), and that it preserves the order of packets. Our modelling methodology includes a set of channel modules for common channel types and the specific module to be used in the model is selected based on the properties specified for the channel pragmatic. The two socket places SenderChannel and ReceiverChannel connecting principals and channels are implicitly considered *channel places* which means that messages added and removed from these places are assumed to be sent and received, respectively. The ⟨⟨channel⟩⟩ pragmatic is used on places in lower level modules to recognise modelling patterns representing the sending and reception of messages on the channel.

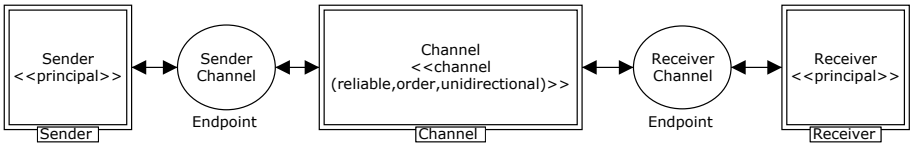


Figure 1: The protocol system level

We require in our modelling methodology that the protocol system module consists of one or more substitution transitions representing principals. A socket place at the protocol system level can be connected to at most one principal substitution transition and at most one channel substitution transition. This requirement is needed since we use the socket places connecting principals and channels to identify which channel or principal a message is intended for.

The concept of a channel represents a means for communication between *endpoints* as determined by the colour set `Endpoint` defined as follows:

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

colset ChannelPackets = list ChannelPacket;

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

The protocol system level and the modelling of channels are parameterised with two colour sets `Packet` and `EndpointId` in that we only assume the existence of these two colour set and do not make any assumptions on how they are realised. A `ChannelPacket` transmitted on the channel has a source (`src`), a destination (`dest`) and a `packet` from the protocol using the channel. An `Endpoint` of the channel consists of the `name` identifying the endpoint, an input buffer (`inb`) and an output buffer (`outb`) for channel packets. Furthermore, a set of protocol independent functions is available for accessing and manipulating endpoints:

```
(* -- initialise an endpoint -- *)
fun InitEndPoint epid =
    {name = eipd, inb = [], outb = []} : Endpoint;

(* -- remove the first packet from the input buffer -- *)
fun RemovePacket ({name,inb,outb} : Endpoint) =
    {name = name, inb = List.tl inb, outb = outb};

(* -- Check if input buffer is non-empty  -- *)
fun PacketAvailable (ep : Endpoint) =
    not (List.null (#inb ep));

(* -- add packet to output buffer -- *)
fun SendPacket (p,ep as
     {name = sepid, inb = inb,outb = outb},depid) =
 {name = sepid,inb = inb,
  outb = outb^^[{src = sepid, dest = depid,packet = p}]};
```

This class of functions also plays a central role in being able to recognize common structural patterns in the CPN models which are captured by the operation pragmatics to be presented in Sect. 6.

The concrete implementation of the `Packet` colour set in a protocol model depends on the protocol data units exchanged among the principals in the protocol under consideration. For code generation purposes, the implementation of the `EndpointId` colour set depends on the concrete channel used to realise the communication between the principals. If for instance, the channel is realised using the transport layer of the TCP/IP protocol stack, then the Endpoint colour set will consist of a host (IP address) and a port (a process). Hence, in a TCP/IP context, an endpoint can be implemented as a TCP/IP socket.

2.2 Principal Level

A submodule of a principal substitution transitions in the protocol system module is, in our modelling approach, required to specify the *services* that are provided by the each principal as well as specifying the life-cycle of the principal. This level is in the following referred to as the *principal level*. In addition to specifying constraints on the order of service use, the principal level modules may also model the state to be maintained across invocation of the services. The explicit modelling of the methods that constitute the service in our approach is required in order to generate code that can be integrated into different code contexts.

Figure 2 shows the principal level CPN module for the sender principal. This module is the submodule of the Sender substitution transition in Fig. 1. The module has three substitution transitions annotated with the ⟨⟨service⟩⟩ pragmatic to indicate that they represent services that are to be exposed by the implementation, i. e., be externally visible. In this case, the sender has three services: Open (for opening/initialising communication with the receiver), Send (for sending a message), and Close (for closing/completing the communication with the receiver). The parameters of the ⟨⟨service⟩⟩ pragmatic specify the parameter and return types, and properties of the services. In this case, all

three services provided by the sender principal are synchronous services. We discuss the parameters and return types in more detail when presenting the service level modules.

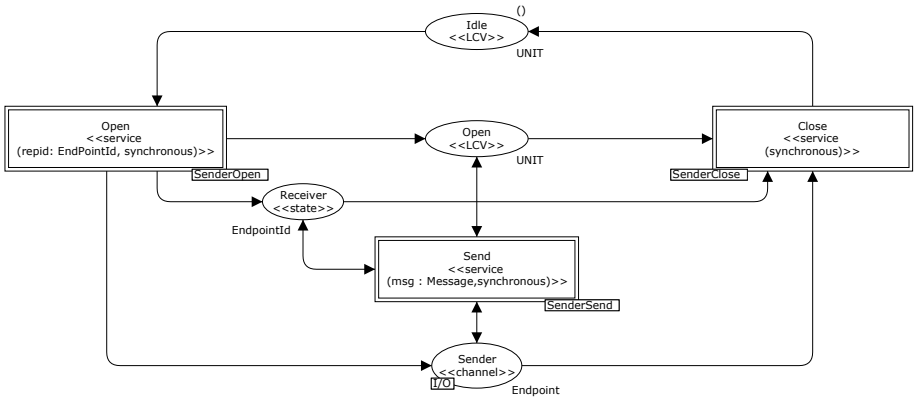


Figure 2: The Sender module

The principal can be in two different states: Idle (no communication initialised) and Open (messages can be transmitted to the receiver). A third implicit state is also possible which is when neither the Idle nor Open places have a token. This state is reached when the client is busy opening, sending or closing. A place modelling a principal life-cycle state is annotated with the ⟨⟨LCV⟩⟩ pragmatic (Life Cycle Variable). The open service can be invoked only when the principal is in state Idle; and once Open, messages can be sent, and the communication can be closed. In the latter case, the sender returns to the Idle state. An additional state variable Receiver is also maintained in the sender for representing the endpoint created by Open, and is required by Send in order to send messages. State variables are indicated using the ⟨⟨state⟩⟩ pragmatic. The port place Sender (bottom) is associated with the SenderChannel socket place in Fig. 1. In the sender module, the place Sender is annotated with the ⟨⟨channel⟩⟩ pragmatic, which is derived from the fact that the associated socket place at the protocol system level is connected to a channel substitution transition.

Figure 3 shows the receiver principal CPN module specifying the three services provided by the receiver principal. The pragmatics used in the receiver principal module are similar to the pragmatics used in the sender principal module. When the receiver is Idle, the Init service can be invoked and the receiver becomes Ready to receive messages. When Ready, the Close service can be used to stop the reception of messages.

The principal level modules do not specify how a wrong use of the services should be handled, e. g., invoking the send service of the sender in a state where the sender is not Ready. The associated error handling is platform dependent, but should, as a minimum, be handled in a uniform manner across all service invocations.

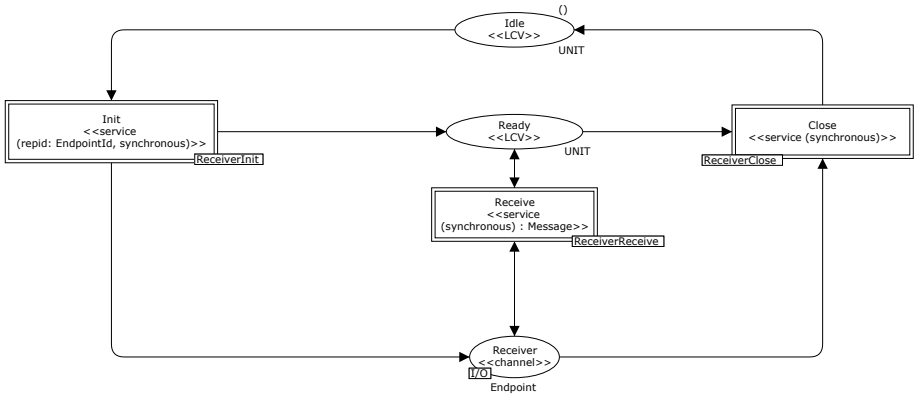


Figure 3: The Receiver module

2.3 Service Level

The submodules of the substitution transitions annotated with ⟨⟨service⟩⟩ on the principal level specify the detailed behaviour of the principals for each of the principal's services. The detailed behaviour is modelled in a control-flow oriented manner using ⟨⟨ID⟩⟩ pragmatics on places to make the control-flow explicit. Modelling the services in a control-flow oriented manner serves two main purposes. The first purpose is to provide for comprehensible models in that the explicit control-flow provides a reading path to the model of the service. This is in contrast to a pure event-oriented approach to modelling (e. g., as discussed in [2]) from which no control-flow is explicit and which consists of modelling a protocol principal using a single place to represent its state and a set of transitions connected to this place which changes the state of the principal depending on packets send and received. The second purpose of modelling in a control-flow oriented manner is to automatically generate code with a structure that resembles what a human programmer would implement. This makes it easier to inspect and maintain automatically generated code, and provides code with better performance since it reflects the intended use of the constructs provided by the target programming language.

2.3.1 Sender Service Level CPN Modules

In the following, we present the models of the three services of the sender.

2.3.2 The SenderOpen Module

The module for sender open is shown in Fig. 4. At this level, the ⟨⟨service⟩⟩ pragmatic is used to indicate the single entry point for the corresponding service primitive. Hence, it is possible to have only one transition annotated with ⟨⟨service⟩⟩. Transitions representing the termination/completion of the service are annotated with the ⟨⟨return⟩⟩ pragmatic. In general, the ⟨⟨return⟩⟩ pragmatic may take parameters representing return values. The parameters for the open service specifies the endpoint of the receiver principal. These parameters are stored in the Receiver state variable and also an endpoint is created on the Sender channel place which the sender will use for sending packets.

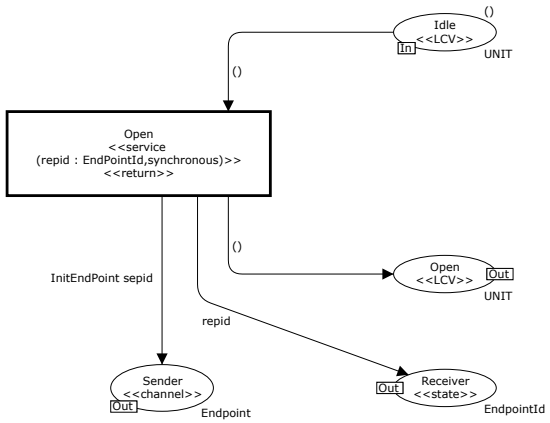


Figure 4: The SenderOpen module

The `InitEndPoint` introduced earlier is used in the sender open module to initialise the endpoint used for communication with the receiver.

2.3.3 The SenderSend module

Figure 5 shows the module for the send service of the sender principal. The message to be sent is represented by the parameter `msg` of the ⟨⟨service⟩⟩ pragmatic.[1] The operation of the send primitive is to partition the message to be sent into a sequence of smaller sub-messages which is placed on Outgoing. The sender then executes a loop in which a packet is sent for each sub-message. Places modelling the control-flow in the send primitive are annotated with an ⟨⟨ID⟩⟩ pragmatic. The modelling of the sender includes some intermediate states (e. g., SendCompleted) which makes the model more verbose, but is used in our approach for recognising control-flow constructs. It is worth noting that, in the modelling of send, we remove the token from Open when sending in order to prevent any further sending or invocation of close while executing the send. This is because the protocol is not designed to handle multiple sends. From a control-flow perspective, the send operation has an overall sequence (starting at transition Start and ending at transition Complete), and a repeat-until loop (starting at place Start and ending in place PacketSent.

The colour sets and functions used in the SenderSend module are as follows:

```
var ep : Endpoint;
var epid : EndpointID;

(* -- partition a string per character -- *)
fun partition msg =
    List.map String.str (String.explode msg);

(* -- packets specific for this protocol -- *)
```

[1] For technical reasons related to CPN Tools and the binding of free variables from large domains, we use `msg` on the arc from Send to Message and `m` on the arc from Message to Partition. Ideally, `msg` should be used in both places.

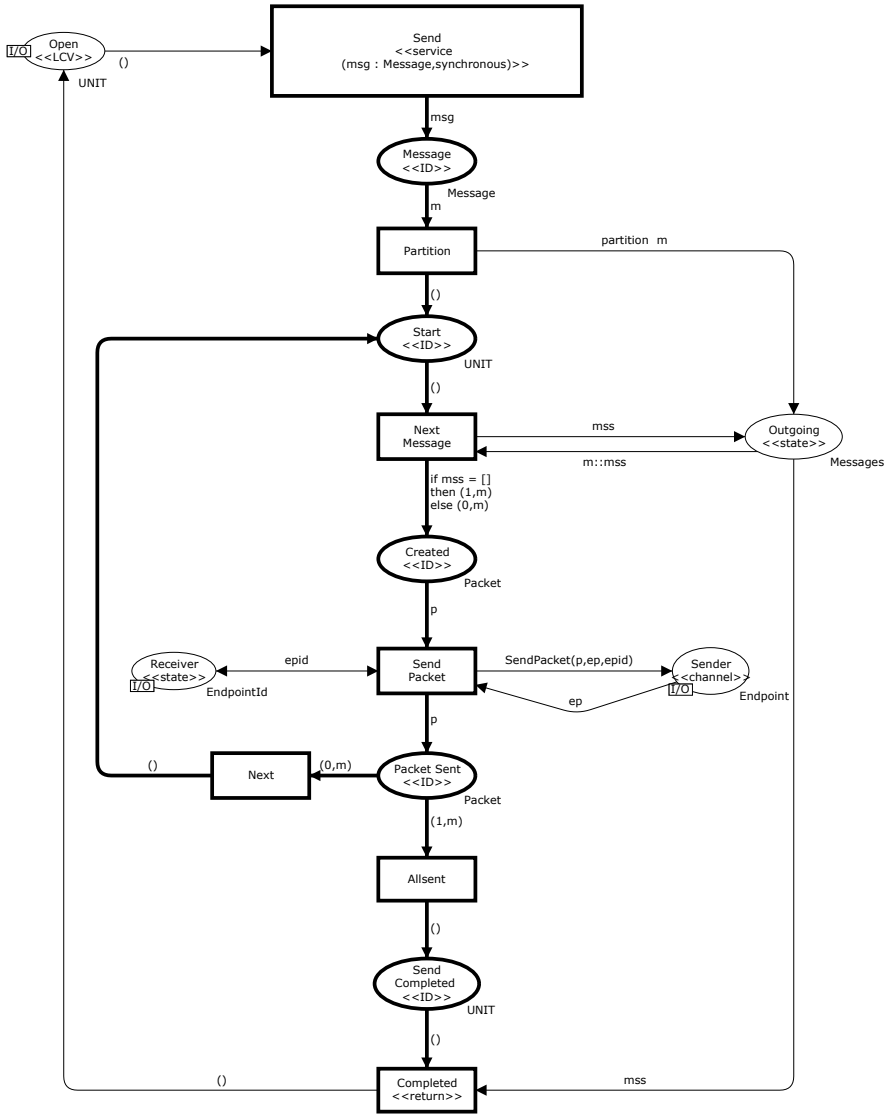


Figure 5: The SenderSend module

```
colset Message  = string;
colset Messages = list Message;

var m : Message;
var mss : Messages;

colset BIT = int with 0..1;

colset Packet = product BIT * Message;
var p : Packet;
```

2.3.4 The SenderClose module

Figure 6 shows the module of the close operation in the sender. The Close operation removes the communication endpoint for the sender.

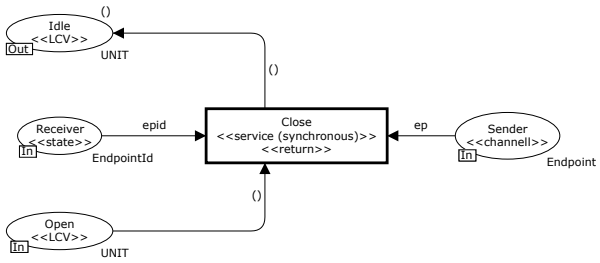


Figure 6: The SenderClose module

2.3.5 Receiver Service Level Modules

In the following, we present the models of the three services of the receiver.

2.3.6 The ReceiverInit Module

The service level module for the receiver init operation is shown in Fig. 7. It creates a communication endpoint on which messages can be received.

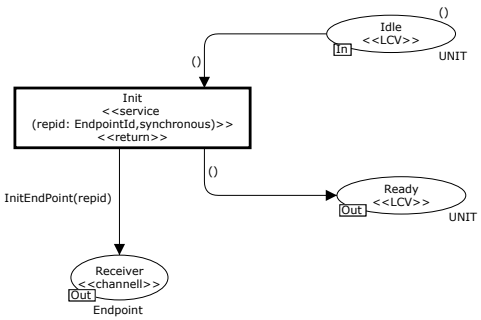


Figure 7: The ReceiverInit module

2.3.7 The ReceiverReceive Module

The service level module for the receiver's receive operation shown in Fig. 8 is dual to the send operation of the sender and the individual parts of the complete message is received in turn. Once the complete message has been received (when a packet with the final bit set arrives) the complete message is returned.

The function `GetMessage` used in this module for getting the submessage (payload) contained in a packet are defined as follows:

```
fun GetMessage ({inb = (p::_),...} : Endpoint) =
    (#2 (#packet p));
```

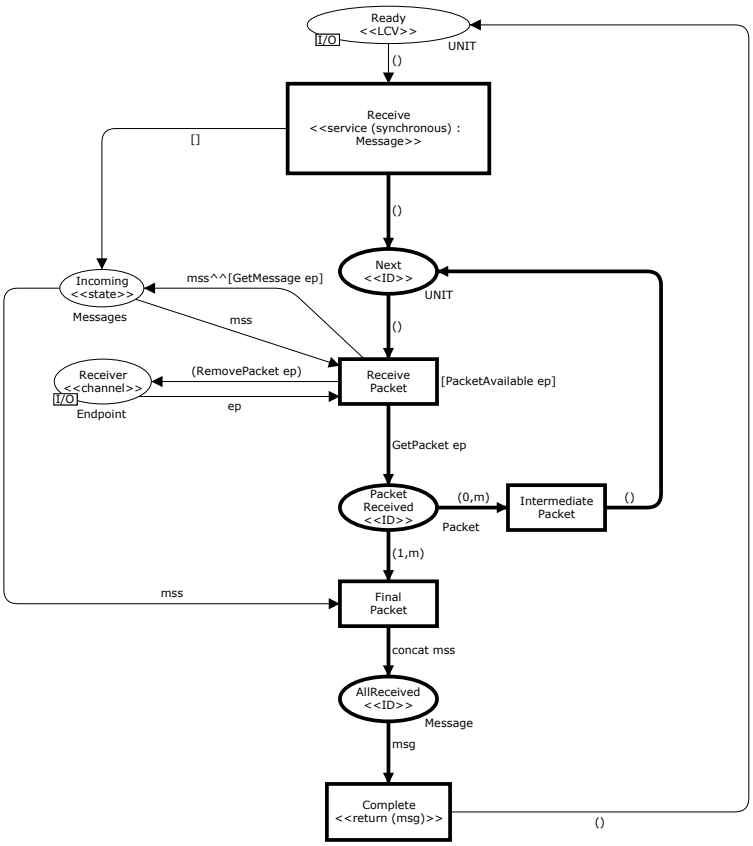


Figure 8: The ReceiverReceive module

2.3.8 The ReceiverClose Module

The module for the close operation of the receiver is shown in Fig. 9 and consists of removing the communication endpoint of the receiver from the channel.

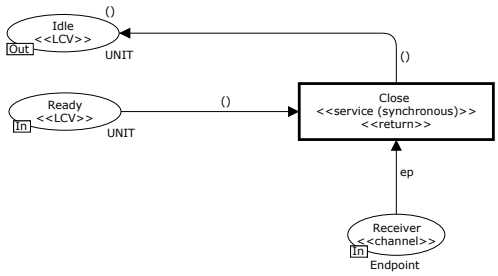


Figure 9: The ReceiverClose module

2.4 The Channel Module

The module for the reliable, unidirectional FIFO channel is shown in Fig. 10. The single transition on this module models the transmission of packets from the output buffer of an endpoint represented on place SenderChannel to the input buffer of an endpoint on place ReceiverChannel. The guard of the transition uses the function `Delivery` to determine whether transmission of a packet from endpoint one (`ep1`) to endpoint two (`ep2`) is possible:

```
fun Delivery (ep1 as ({outb = (p::_),...} : Endpoint),
              ep2 as {name,...} : Endpoint) =
    ((#dest p) = name)
  |   Delivery _ = false;
```

The `Delivery` function expresses the requirement that the first packet `p` in the output buffer of endpoint 1 can be transmitted to the endpoint 2 provided that the destination of the packet matches the name of endpoint 2 (i. e., the packet is to be delivered to endpoint 2).

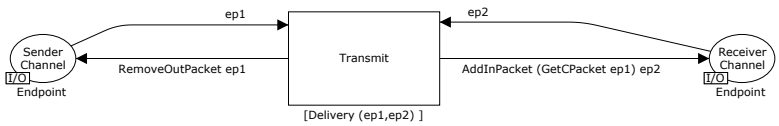


Figure 10: The Channel module

The functions `RemoveOutPacket`, `AddInPacket`, `GetCPacket` are used to remove a packet from an output buffer of an endpoint and add a packet to the input buffer of an endpoint:

```
var ep1,ep2 : Endpoint;

fun RemoveOutPacket {name,inb,outb = p::ps} =
    {name=name,inb=inb,outb=ps};

fun AddInPacket p ({name,inb,outb} : Endpoint) =
    {name=name,inb=inb^^[p],outb=outb};

fun GetCPacket ({outb = (p::_),...} : Endpoint) = p;
```

2.5 Variants and Extensions

Below, we discuss some variants and extensions to the basic modelling approach that has been presented in the previous subsections.

2.5.1 Channels

At the protocol system level (see Fig. 1), we represent channels with a substitution transition and at most one socket place connected to each principal per channel. Tokens on the socket place then represent endpoints for the attached

principal. We could have considered having just a single place representing a channel, and tokens on this place would then represent all endpoints of the protocol system. This, however, complicates the modelling of the principals as it now becomes necessary (in submodules of the principals) to keep track of which endpoints are owned by which principals which in turn clutters up the modelling of the principals.

The long-term vision in our approach is that a set of common channel models is available that can be automatically plugged into the model according to the properties specified for the ⟨⟨channel⟩⟩ pragmatic at the protocol system level.

A channel corresponding to TCP (bi-directional reliable communication) can be modelled as two unidirectional channels which are both reliable. Hence, they can be modelled using two instances of the channel model in Fig. 10 (one for each direction).

A unidirectional channel with loss, overtaking, and duplication can be modelled as shown in Fig. 11.

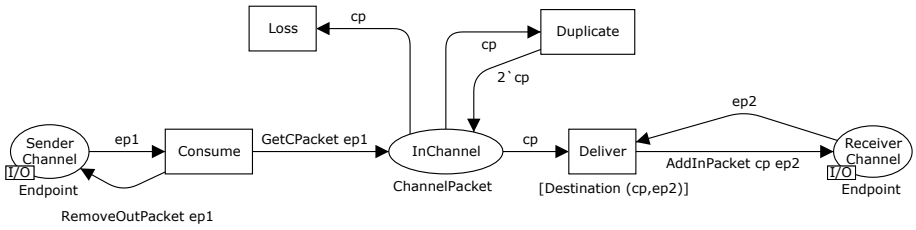


Figure 11: A unidiretioncal channel with loss, duplication, and overtaking

The function `Destination` is used to ensure that packets are delivered only to endpoints corresponding to the destination of the packets, and the function `GetCPacket` returns the first packet in the output buffer of an endpoint.

```
fun Destination (cp:ChannelPacket,
                 ep as {name,...}:Endpoint) =
    ((#dest cp) = name);

fun GetCPacket ({outb = (p::_),...} : Endpoint) = p;
```

A channel corresponding to UDP can be obtained from the channel model in Fig. 11 by again using two instances (one for each direction).

It should be noted that the modelling of the channel models presented in this section are not necessarily the most efficient channel models from the perspective of verification as the transmission of packets introduces many intermediate states. A discussion of channel modelling for efficient verification purposes [3] is outside the scope of this report.

2.5.2 Asynchronous Services

The services provided by the sender and receiver in the example were all synchronous services which means that the caller of the service will be blocked until the service returns. Clearly, a separate thread of execution can be created by a user of the services prior to calling a synchronous service in order to avoid

blocking and making long-running services acceptable. In some situations, it may also be desirable for the protocol implementation to directly support long-running services without having to block the caller (e. g., for receiving messages or keeping connections alive).

Below we outline a concept of asynchronous services. The basic idea of an asynchronous service is that the service returns control to the caller immediately, but execution may continue in the implementation of the service (conceptually in a separate thread). This allows the protocol implementation to have active behaviour even when there is currently no service that is explicitly being executed from the perspective of the users. An asynchronous service is declared using the asynchronous property as part of a ⟨⟨service⟩⟩ pragmatic.

To illustrate the concept of asynchronous services, we discuss the model of an asynchronous version of the receiver where the receive method itself is not responsible for receiving the messages, but merely checks whether a full message has been received. Figure 12 shows the asynchronous variant of the receiver principal. The difference to the synchronous version of the receiver is that the init service is now declared to be asynchronous (as it will have some active behavior for receiving messages), a state place Messages has been introduced where the init service accumulates messages as they are received, and the receiver service no longer access the channel (since the reception of message happens in the asynchronous part of the init service). The receive and close services are still synchronous services.

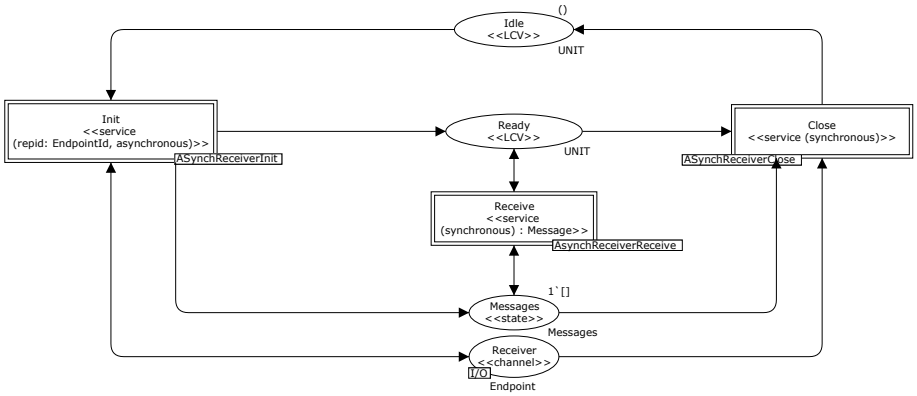


Figure 12: Receiver with an asynchronous init service

Figure 13 shows the init service. The init service returns immediately after the service has been invoked, after which behavior will be executed corresponding to the reception of messages. The reception of messages is essentially the same as for the receive primitive in the synchronous variant (see Fig. 8) with the exception that when a complete message has been received, it is put on place Messages such that it can be accessed in the receive service.

Figure 14 shows the receiver service. It now implements a check (conditional branch) on whether a message is ready on place Message – in which case it will be returned. For simplicity, the empty string (`""`) is returned by the primitive in case no messages are currently available.

Figure 15 shows the close service module of the receiver. It is similar to the

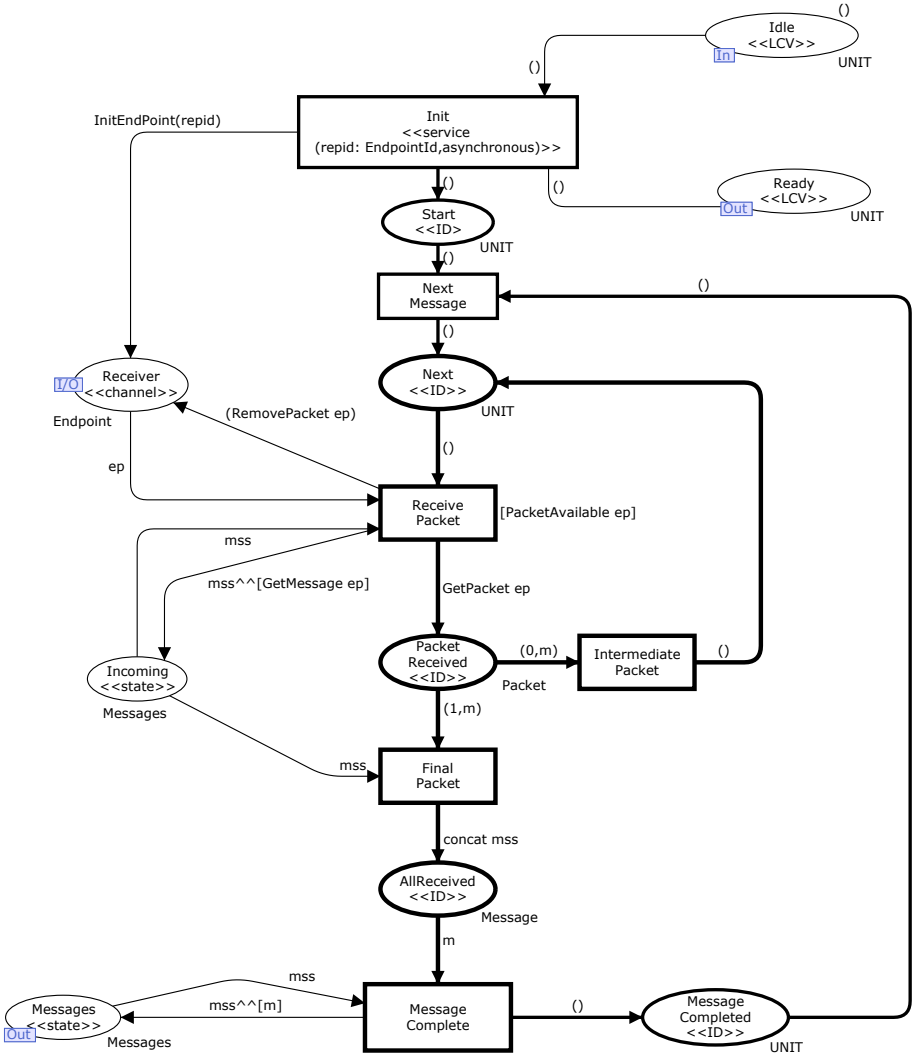


Figure 13: The asynchronous init service

synchronous case expect that it now performs a cleanup of the newly introduced place Messages when called. An extension of the asynchronous receiver would be to add some life-cycle state that would more properly terminate the reception of messages in the init service when the close primitive is invoked.

3 Requirements to the CPN Structure

In Sect. 2, we have discussed a simple unidirectional framing protocol, and we have used a CPN model equipped with some additional pragmatics for formally modelling the protocol. This CPN model follows some modelling rules. These modelling rules serve two different purposes: First, the rules should help protocol designers to come up with clean and comprehensible CPN models. Second,

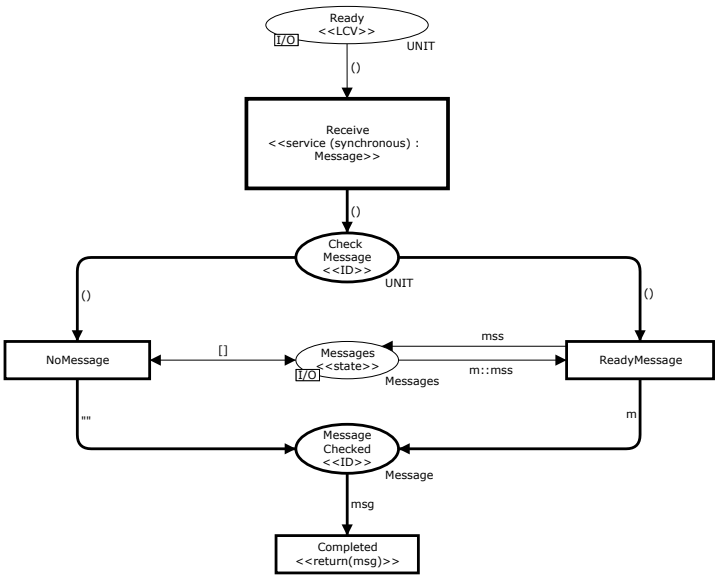


Figure 14: The Receive service module of the asynchronous receiver

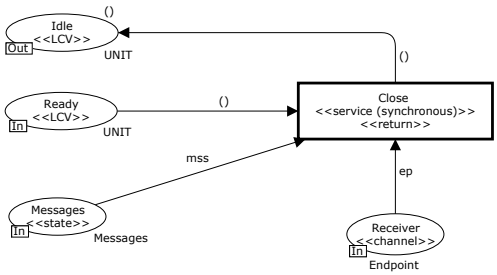


Figure 15: The Close service module of the asynchronous receiver

the CPN models together with the additional pragmatics should contain all information for fully automatically generating the code from these models. This would not be possible for arbitrary CPN models without any additional structure. The example of Sect. 2 exhibits most of these rules already. In this section, we make the rules for CPN models and the way how to add pragmatics to them explicit.

Generally, we distinguish three different levels when modelling a protocol: the *protocol system level*, the *principal level*, and the *service level*. The protocol system level, is the top level, which reflects the overall architecture, of the protocol, the possible communication partners involved, which are called *principals*, and the *channels* between them. Figure 1 showed the protocol system level of our example. For each principal of the protocol, there is one CPN module, which models the *life-cycle* of each principal, i. e. it defines which services of the principal can be invoked at which times. Figures 2 and 3 showed the two modules of our example on the principal level. For each service of a principal, there is a module on the service level, effectively modelling what happens, when

this service is invoked. Figure 5 showed one example of such a service level module.

For the structure of the CPN model, we have the following requirements and conventions that apply to all levels of the model:

1. The CPN models do not contain modules with unassigned port and socket places, i. e., all port places are related to a socket place.
2. Each pair of related port and socket places must be annotated with the same pragmatics. If only the port (or the socket) is annotated with a pragmatic, then a related socket (or port) automatically inherits the pragmatic.

In the next subsections, we present the requirements on the CPN model and its submodules on the three levels in more detail. Some of the requirements relate to pragmatics, which need to be explicitly added to the model by the modeller. Therefore, a set of core pragmatics will be discussed in this section, too.

3.1 Protocol System Level

The CPN model must have exactly one module at the top-level, which represents the *protocol system level*. Since there is only one module on this level, we refer to this module as the *protocol system module*. This module must meet the following requirements:

1. The module may contain only substitution transitions and places (i. e., there are no ordinary transitions).
2. Each substitution transition is annotated with either a $\langle\langle$principal$\rangle\rangle$ pragmatic (referred to as principal transition) or a $\langle\langle$channel$\rangle\rangle$ pragmatic (referred to as channel transition).
3. Any place at the protocol system level is connected to at most one principal transition and at most one channel transition.

We do not go into further detail with the submodules associated with channel substitution transitions. The reason for this is that the internal operation of these modules do not play a direct role in code generation. These modules are present in the CPN model only in order to make the CPN model operational, i. e., for the principals to be able to exchange messages. For code generation purposes, we select platform communication primitives that conform to the properties associated with the channel pragmatics (e. g., properties such as reliable, order, unidirectional). We discuss properties of pragmatics further in Sect. 3.4.

3.2 Principal Level

The principal level is represented by all the submodules of the CPN model associated with a principal substitution transition at the protocol system level. Each module at the principal level has the following requirements:

1. All transitions are substitution transitions and represent services provided by the principal.

2. All transitions are annotated with a $\langle\langle \mathsf{service} \rangle\rangle$ pragmatic, which specifies the parameters of the service together with their types as well as the return type of the service. Moreover, the $\langle\langle \mathsf{service} \rangle\rangle$ pragmatic specifies whether the service is synchronous or asynchronous.

3. All non-port places are annotated with either a $\langle\langle \mathsf{LCV} \rangle\rangle$ pragmatic (representing life-cycle variables) or a $\langle\langle \mathsf{state} \rangle\rangle$ pragmatic (representing global state variables of the principal).

4. At any time in the dynamic behaviour of the protocol (in each reachable marking of the CPN model), each life-cycle and state variable place contains at most one token.

Note that a principal's life-cycle defines in which order the services of a principal could be invoked in order not to violate the protocol. In the actual code for these protocols, however, it is – in most cases – technically impossible to prevent the invocation of services that are not allowed at a certain time in the life-cycle. Therefore, it is left to the implementation, resp. the code generator, how to deal with calls of services that are not expected according to the principal's life-cycle: e. g. the service could throw a runtime exception, it could not terminate at all, or it could return some random result. Anyway, the protocol would be violated; only if the service calls respect the life-cycle of the principals, the generated code would reflect the behaviour of the modelled protocol.

3.3 Service Level

The service level consists of the submodules of the substitution transitions at the principal level. Each module at the service level has the following requirements.

1. In the case of synchronous services: There must be exactly one (initiating) transition with a $\langle\langle \mathsf{service} \rangle\rangle$ pragmatic and exactly one transitions with a $\langle\langle \mathsf{return} \rangle\rangle$ pragmatic. The parameters of the service and return pragmatics are required to match with the parameter and return types specified for the corresponding substitution transition at the principal level.

2. In the case of asynchronous services: There must be exactly one (initiating) transition with a $\langle\langle \mathsf{service} \rangle\rangle$ pragmatic; however, there must not be any transition with a $\langle\langle \mathsf{return} \rangle\rangle$ pragmatic, since the call returns right away – possibly spawning off a longer concurrently running computation. The parameters of the service pragmatic are required to match with the parameters specified for the corresponding substitution transition at the principal level.

3. Each place is either annotated with an $\langle\langle \mathsf{ID} \rangle\rangle$ pragmatic (referred to as a *control flow place*), a $\langle\langle \mathsf{state} \rangle\rangle$ pragmatic (referred to as a *state place*) or it is a port place assigned to a channel socket place at the principal level (referred to as a *channel place*). The net induced by the control flow places of the service level module (i. e. the net that consists of all the control flow places and the arcs and transitions directly attached to control flow places) is called the *control flow net* of the service level module.

4. Each transitions (except the one with a ⟨⟨service⟩⟩ or a ⟨⟨return⟩⟩ pragmatics) has exactly one control flow place in its preset and exactly one control flow place in its postset. The single transition with a ⟨⟨service⟩⟩ pragmatic does not have a control flow place in its preset; if this transition also has a ⟨⟨return⟩⟩ pragmatic, there should be no control flow place in its postset; if the ⟨⟨service⟩⟩ transition is no ⟨⟨return⟩⟩ transition, it must have exactly one control flow place in its postset. A ⟨⟨return⟩⟩ transition that is not a ⟨⟨service⟩⟩ transition must have exactly one control flow place in its preset and no control flow place in its postset.

5. The control flow net of the service level module should be decomposable into blocks of control flow constructs (which is formally characterised in Sect. 5.1).

6. At any time in the dynamic behaviour of the protocol (in any reachable marking of the CPN model), the number of tokens on the control flow places is equal to the number of active concurrent invocations of the service. In the case without parallel invocations of a service, this implies that there is at most one token on the control flow places at any given moment in time.

7. Places of a service level module that have a ⟨⟨state⟩⟩ pragmatic and are not socket places attached to the principle level module, are called *local state places*. We require that at any time (in each reachable marking) there are no token on local state places of a service level module, when there are no tokens on any of its control flow places.

It should be noted that, when specifying the requirements above, we do not allow substitution transitions in service level modules. This is done to make the formulation of the requirement simpler. In practice, it is not a problem to have substitution transitions in service level modules as long as the requirement stated above holds for the service level module obtained where we have replaced each substitution transition with the associated submodule.

3.4 Core Pragmatics

The example protocol described in Sect. 2 is annotated with several pragmatics. In this section, we present a core set of pragmatics that are applicable to all protocols modelled using our approach. These pragmatics are added by the modeller in order to make explicit the protocol and principal levels of a protocol and the control-flow of the services.

A pragmatic is specified in the form of a table giving the pragmatic's key features. These tables have the following fields:

Name is the name of the pragmatic.

Description provides a natural language description of the purpose of the pragmatic.

Name	principal
Description	Denotes a principal agent of the protocol system
Placement	Substitution transitions on the protocol system level
Constraints	The substitution transition can only be connected to channel places
Parameters	none
Type	Structure
Category	Core
Origin	Explicit

Table 1: Description of principal pragmatic.

Placement describes where it is valid to put a pragmatic. This should include the levels where it belongs (protocol, principal or service) and what types of model elements the pragmatic may be associated with (e. g., places and transitions).

Constraints specifies additional constraints to those described under placement.

Type classifies the pragmatic into structure, control-flow, and operation.

Category classifies the pragmatic into core and protocol specific.

Origin is *derived* if the pragmatic may be automatically generated from the net structure, *explicit* if the pragmatic must be added manually by the modeller.

Tables of this form will also be used later when introducing derived and protocol-specific pragmatics.

The ⟨⟨principal⟩⟩ pragmatic (Table 1) specifies that the substitution transition, to which this pragmatic is connected, to be a principal agent or entity in the protocol. The ⟨⟨principal⟩⟩ pragmatic can be used for substitution transitions on the top module of the CPN model (protocol system level).

The ⟨⟨channel⟩⟩ pragmatic, described in Table 2, is associated with substitution transitions on the system protocol level, which represent the communication channels between principals. The ⟨⟨channel⟩⟩ pragmatic has a parameter that defines the channel's characteristics such as reliable, ordered, and unidirectional in Fig. 1. Based on this channel type, the code generator would select the appropriate communication primitives at the given target platform, which meet the characteristics. For analysis and verification purposes, however, the substitution transitions for channels can be associated with a CPN module, which explicitly models the behaviour of the channel (Fig. 10 in our example). By choosing this channel model, the protocol can be verified in different scenarios (e. g. for showing lossless transmission in the presence of unreliable channels or for showing robustness against attacks). The CPN module for the channel will, however, not be used for code generation.

Note that the places that are attached to the channel substitution transition have a ⟨⟨channel⟩⟩ pragmatic attached to them, which, however, is a derived

Name	channel
Description	Denotes a communication channel
Placement	Substitution transitions on the protocol system level and port places related to a socket place of such a substitution transition
Constraints	
Parameters	Channel types
Type	Structure
Category	Core
Origin	Explicit

Table 2: Description of channel pragmatic.

Name	ID
Description	Denotes the control flow path of each service
Placement	A place at the service level
Constraints	none
Parameters	none
Type	Control-flow
Category	Core
Origin	Explicit

Table 3: Description of ID pragmatic.

pragmatics. This will be used in the service level modules in order to identify send and receive operations.

The ⟨⟨ID⟩⟩ pragmatics, described in Table 3, denotes the control flow of the service. ID can only annotate places that reside on the service level. The ⟨⟨ID⟩⟩ pragmatic is central to our code generation approach as it makes the control flow explicit. This will be discussed in detail in Sect. 5.

The Life Cycle Variable (⟨⟨LCV⟩⟩) pragmatics, described in Table 4, denote places that control the life cycle of the protocol by imposing pre- and post-conditions on the invocation of services. In the example of Sect. 2, all these places have the data type `UNIT` which the code generator interprets as boolean set or unset values. Places with pragmatics ⟨⟨LCV⟩⟩ and ⟨⟨state⟩⟩ are used to define the flow of the protocol, i. e. they define the order in which the environment should call the services of the different principals of the protocol. As discussed earlier, it is not always possible for the generated code to prevent external programs from calling services, when this would be illegal according to the protocol's state. In that case, it is up to the implementation to deal with this situation, by either raising an exception or returning error codes in some uniform way. The generated code is guaranteed to work correctly only, when the services are invoked in the order as defined by the life-cycle of the protocol.

The ⟨⟨service⟩⟩ pragmatic, described in Table 5, denotes an interface to the environment such as a method or function. The ⟨⟨service⟩⟩ pragmatic can be used to annotate substitution transitions on the protocol level and transitions on the service level.

The ⟨⟨state⟩⟩ pragmatic is used to denote places that can hold data for a principal or a service. In the example in Fig. 5 there are two state places. One

Name	LCV
Description	Denote places which control the life cycle of the protocol by imposing pre- and post-conditions on services
Placement	Socket places at the principal level
Constraints	
Parameters	none
Type	Control-flow
Category	Core
Origin	Explicit

Table 4: Description of LCV pragmatic.

Name	service
Description	Denotes an interface to the environment of such as a method or function
Placement	Substitution transition on the principal level and transitions on the service level
Constraints	Can exists on only one transition per module at the service level. The arguments must be identical on service level to that on the corresponding substitution transition on the protocol level.
Parameters	The parameters for the service with types and, as the final parameter, the text “synchronous” or “asynchronous” indicating whether this service should be considered synchronous or asynchronous.
Type	Structural at the principal level and control-flow at the service level
Category	Core
Origin	Explicit

Table 5: Description of service pragmatic.

Name	state
Description	Denotes a value holder on the principal or service level
Placement	Places on the principal and service level
Constraints	none
Parameters	none
Type	Structure
Category	Core
Origin	Explicit

Table 6: Description of state pragmatic.

Name	return
Description	Denotes the termination of a synchronous service.
Placement	Transitions on the service level
Constraints	Optionally the value that should be returned
Parameters	The return value
Type	Control-flow
Category	Core
Origin	Explicit

Table 7: Description of return pragmatic.

is the place named Outgoing which holds the list of the remaining outgoing fragments. This state variable is local to the service and is not visible on the principal level. The other state variable in the example is the place named Receiver. This state holds the endpoint for the receiver. This place is a port place referring to the Receiver place on the principal level as seen in Fig. 2. This makes it a principal level state variable which is akin to a field variable in that it is accessible by several services and service instances.

The ⟨⟨return⟩⟩ pragmatic is a service level pragmatic denoting the end of a service. This is similar to the the return keyword in many programming languages. The value that should be returned is determined by inspecting the incoming arc or is given as a parameter. The type of the return value must match the return type specified in the ⟨⟨service⟩⟩ pragmatic for the corresponding service.

4 Overview of Code Generation

In Sect. 2, we presented a descriptive model for a simple communication protocol that conforms to the structural requirements presented in Sect. 3. In this section, we outline how code for implementing each of the principals can be automatically generated from such a CPN model annotated with pragmatics.

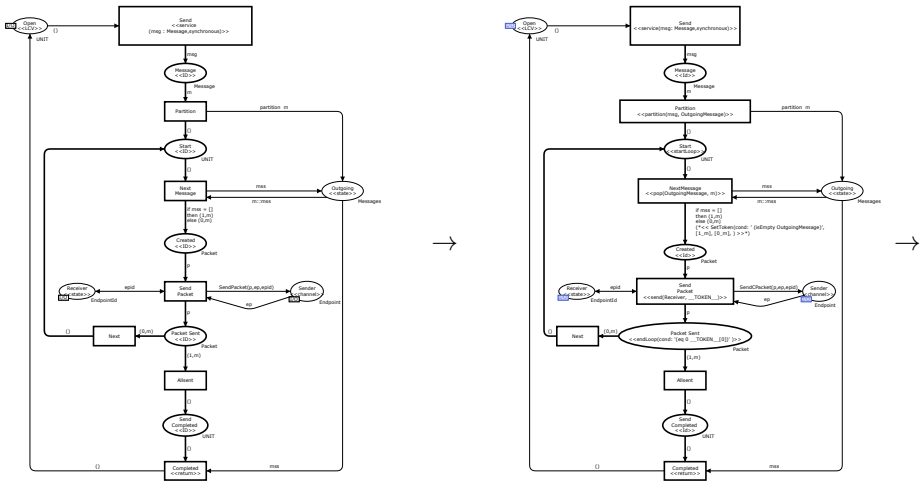
Figure 16 gives an overview of the three main phases of the generation process and the artifacts of the code generation process. On the top-left, the CPN model (in the figure represented by a part of the model only, the SenderSend module from Fig. 5) annotated with the explicit pragmatics is shown. From this model, the code generation starts. In the first phase of the code generation process, the derived pragmatics are added to the CPN model. The result of this

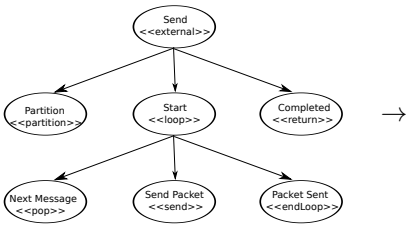
phase is represented on the top-right of Fig. 16 – for the SenderSend module again (the details will be discussed later again and are shown a bit bigger in Fig. 22). In the next phase, an intermediate representation of the CPN model and its pragmatics in the form of an *abstract template tree* (*ATT*) is generated from the CPN model with the derived pragmatics. An excerpt of the ATT is shown at the bottom-left of Fig. 16, which is the part of the ATT corresponding to the SenderSend module. The last phase, actually, generates the code from the ATT. The bottom-right of Fig. 16 shows the code generated for the SenderSend module – in this case, it is code in the Groovy programing language. These three phases are discussed in more detail below.

The first phase in the code generation process is to automatically add further pragmatics to the CPN model based on an analysis of the structure of the CPN model. This analysis results in the addition of pragmatics that makes explicit common control flow structures and pragmatics that represent operations used to access state information and operations used to send and receive messages on the underlying communication channels. The addition of further control flow pragmatics exploits the requirements that the control flow places in the CPN model annotated with the $\langle\langle$ID$\rangle\rangle$ pragmatic conform to a certain block-oriented pattern. The details of how we derive the additional control flow pragmatics will be presented in Sect. 5. The details of how we identify operations pragmatics are provided in Sect. 6.

The second phase is then to transform the CPN model into an ATT which gets its overall structure from the hierarchical structure of the CPN. The root element of the ATT has one sub-ATT for each principal. Each principal has several nodes representing a service, these ATT nodes are bound to the substitution transitions on the top module of each principal which are annotated with the $\langle\langle$service$\rangle\rangle$ pragmatic. The process of obtaining the ATT also extracts the control flow structure from the service level CPN modules and represents it as an ordered tree structure where each node contains information on what pragmatics are bound to the corresponding CPN model elements. Each node of an ATT holds some additional information, which refers to some elements from the CPN model and the associated pragmatics, along with their parameters. The sub-ATT for the send service of the sender principal is shown at the bottom-left of Fig. 16. The ATT structure below the send service node, roughly, reflects the control flow structure of the SenderSend CPN model. The process of constructing the ATT will be presented in detail in Sect. 7

The third and final phase is to transform the ATT into code. This is accomplished by associating the pragmatics of each node in the ATT with code templates and use the templates to generate code. The binding of pragmatics to templates for a certain platform is defined by a *template binding*. The ATT drives the actual code generation (some example code is shown on the bottom-right of Fig 16). Which templates are used depends on the target platform and programming language. In the template, placeholders are filled with attribute values from the respective ATT-node as well as code which is generated by the templates of the sub-nodes – where applicable. The code is assembled using the tree-structure of the ATT. The assembling is done by, from the bottom up, replacing a special `%%yield%%` tag in the template with the contents of a node's children concatenated in order. The principals are considered the units code is generated for. Further details of the code generation process will be given in Sect. 8.






```
class Sender {
  def Idle = true
  def Open

  def SenderSend(){
    if(!Open) throw new Exception...
    def OutgoingMessage = ...
    __TOKEN__ = null//()
    def __LOOP_VAR__ = true
    while(__LOOP_VAR__){
          ...
    }
    return
  }
}
```

Figure 16: Overview of code-generation phases

5 Control Flow Pragmatics and Blocks

Petri nets do not enforce any particular control flow structure. In order to be able to generate code that uses the control flow constructs of typical programming languages, we assume that the control flow net (induced by places of the service level modules that are marked with $\langle\langle \mathsf{ID} \rangle\rangle$) can be decomposed into *control flow blocks*. In this section, we discuss the decomposition of a Petri net into control flow blocks along with constructing an associated block tree. Moreover, we introduce some derived pragmatics that ease the identification of the control flow structure.

5.1 Block Decomposition

To this end, we systematically decompose each service level module into blocks where the containment of the blocks defines the structure of the ATT. Actually, we use only the control flow net of the service level module.

In order to be flexible, the decomposition is defined in terms of decomposition rules, which are shown in Fig. 20. With these rules, the module SenderSend from Figure 5 can be decomposed into a block, which is a sequence, where the first element of that sequence is an atomic block, the second is a loop, which consists of a sequence of two atomic blocks again as shown in Fig. 21.

In this section, we assume that the services are synchronous – i. e. the service call returns only after the service is terminated. We also assume that there is exactly one *start* place and one *end* place; the start place would be the control flow place in the postset of the transition with $\langle\langle \mathsf{service} \rangle\rangle$ pragmatic (in Fig. 5 it is place Message); the end place is the control flow place in the preset of a transition with $\langle\langle \mathsf{return} \rangle\rangle$ pragmatic (in Fig. 5 it is place SendCompleted). We do not deal with the special cases here in which the $\langle\langle \mathsf{service} \rangle\rangle$ transition is also a $\langle\langle \mathsf{return} \rangle\rangle$ transition.

Let us assume that we have the control flow net of the service level model and that we have identified the start and an end place. From there, the ATT is obtained by recursively decomposing the net into *blocks*, where each block is a sub-net of the control flow net with a distinguished start and end place. Mathematically, this decomposition into blocks is defined inductively. Figure 17 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.

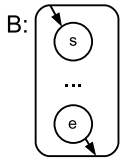


Figure 17: Graphical representation of a block.

Now let us formalise the concept of a block.

Definition 1 (Block, atomic non-returning block)**.**
Let $N = (P, T, F)$ be Petri net, then $B = (N, s, e)$ with $s, e \in P$ is a *block*.

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. it 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. it does not use a token from the end place itself).

An atomic block consists of a single transition, as shown in Figure 20 later. For visualizing blocks with safe entry and safe exit, we introduce an additional graphical notation, which is shown in Fig. 18. The crossed out arc from within the block to the start place indicates that the block itself does not return a token to the start 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 end place (safe exit).

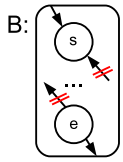


Figure 18: Graphical representation of safe entry and safe exit.

For easing our notation, for a block B_i, we will denote its net by $N_i = (P_i, T_i, F_i)$, its start place by s_i and its end place by e_i. Next we define the general decomposition of a block into sub-blocks.

Definition 2 (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. 19.

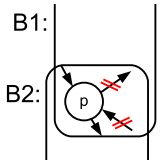


Figure 19: Safe join of two consecutive blocks

Now, the block trees associated with blocks can be inductively defined by using the decompositions into specific constructs. Note that the block trees are ordered labelled trees, i. e. the tree nodes have a label and the order of the child nodes of a node matter. These decomposition rules along with the resulting block trees are graphically represented in Fig. 20.

Definition 3 (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 are 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 blocks $B_1, \ldots, B_n$ are 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 blocks $B_1, \ldots, B_n$ are 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 in order to translate a sequence to programming language constructs, we need to make sure that the control never goes backwards – except for explicit loops. To this end, the above definition requires for two blocks B_i and B_{i+1} of a sequence, that block B_i has a safe exit or block B_{i+1} has a safe entry. The side conditions on safe entry and exit of blocks make this definition a bit more technical. The reason for introducing these side conditions is that nets that actually can be decomposed into blocks do not need to introduce extra control transitions between parts of the net representing different constructs – just for making the net decomposable. Since these extra transitions would make the nets required for code generation quite verbose and unnatural from a Petri net modelling point of view, our blocks can be directly "glued together" at their start and end places – under the given side conditions.

Figure 21 shows the control flow net of the SenderSend module from Fig. 5 with overlays that indicate the decomposition of the control flow net into blocks. The resulting block tree is, roughly, the one shown at the bottom-left of Fig. 16.

Note that, in general, the decomposition rules are not deterministic. There could be more than one tree associated with a given block – the tree is not

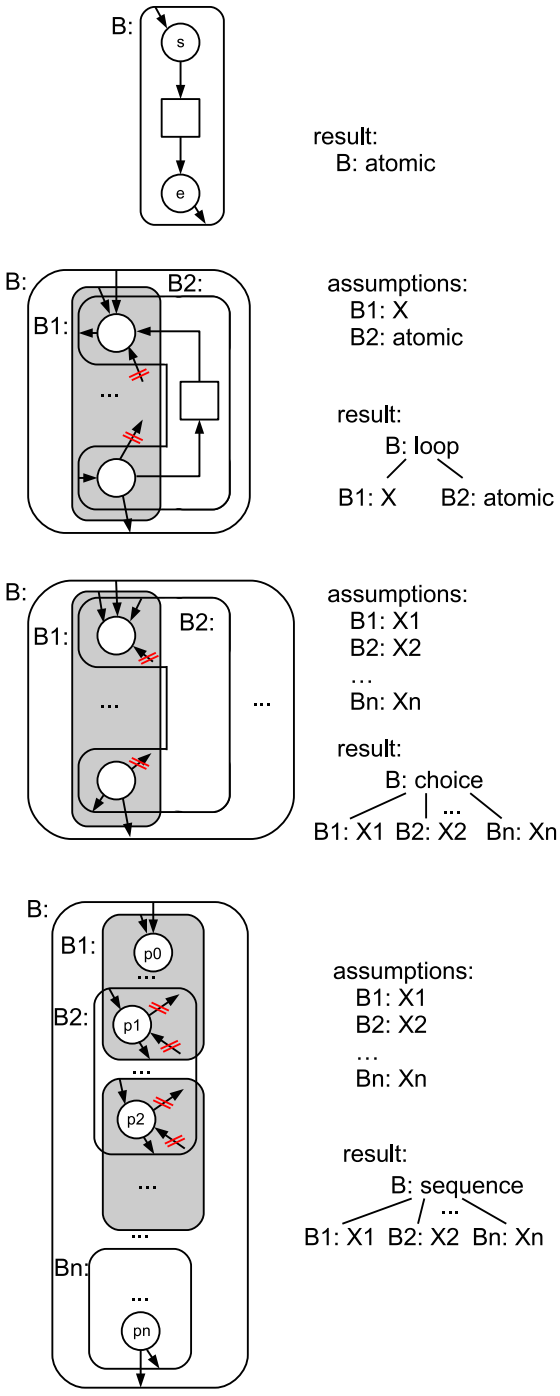


Figure 20: Inductive definition of block trees

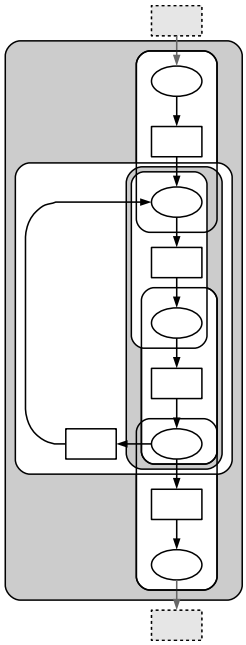


Figure 21: Decomposition of the control flow net of module SenderSend

necessarily unique. For example, a longer sequence of atomic blocks could be decomposed in many different ways in sub-sequences of sub-sequences. In principle, any of these block trees would do for our purposes. The implementation, however, maximises the number of sub-blocks of each sequence, so that there would be only one maximal sequence of blocks. This way, avoiding unnecessary substructures in the block tree generated from these blocks.

There are also nets that do not have a block decomposition at all, and no tree associated with them. Such blocks cannot be transformed into an ATT, and therefore not transformed to code. It is the modellers responsibility to make sure that the control flow net of the service level modules can be decomposed into a block tree. The only exception are service level modules that consist of a single transition only. In this case, the single transition is considered to be both the start and end node of the service and must be annotated with both $\langle\langle$service$\rangle\rangle$ and $\langle\langle$return$\rangle\rangle$ in addition to the corresponding pragmatics for any operations that the service should complete. This is handled as a special case by the ATT and code generation without relying on a decomposition into blocks.

A block tree associated with a control flow net of a service level module, basically, represents the abstract syntax tree of the service level module. Technically, the ATT of the service level modules is constructed in a different way. In order to identify the control flow in the net, the respective start and end places of the choice and loop constructs are associated with a derived pragmatics, which will be be discussed in Sect. 5.2 and 5.3.

Name	branch
Description	Denotes the start of a branch in the control flow
Placement	service level places
Constraints	Only on places with the ID pragmatic in accordance with the block decomposition rules
Parameters	The condition
Type	Control-flow
Category	Core
Origin	Derived

Table 8: Description of branch pragmatic.

Name	merge
Description	Denotes the end of a branch in the control flow
Placement	Service level places
Constraints	Only on places with the ID pragmatic and in accordance with the block decomposition rules
Parameters	none
Type	Control-flow
Category	Core
Origin	Derived

Table 9: Description of merge pragmatic.

5.2 Conditional Pragmatics

The block decomposition rule for a conditional is shown in the third block rule in Fig 20. A conditional begins with a place with a single in-arc and multiple out-arcs. Each of the out arcs lead to some blocks where the end of these blocks has a single outgoing arc to the conditional blocks merge place, of which there may only exist one.

There are two pragmatics for conditionals, firstly the $\langle\langle \mathsf{branch} \rangle\rangle$ pragmatic which represent the branching of the control flow and also the $\langle\langle \mathsf{merge} \rangle\rangle$ pragmatic which represents the place where the branch is merged. These pragmatics are described in Table 8 and Table 9. The $\langle\langle \mathsf{branch} \rangle\rangle$ pragmatic takes one parameter which contains the condition for taking one branch or the other. This condition is given using a language which is described further in Sect. 6. In the ATT, the branch node will become a container-node containing a single container-node for each branch. At the end of each of the branches will be a node with the $\langle\langle \mathsf{merge} \rangle\rangle$ pragmatic.

5.3 Loop Pragmatics

The loop block allows the creation of actions that are to be repeated several times or even an infinite number of times. A loop have a start and an end place connected by a block. The start loop has two incoming arcs, where at least one is part of an atomic block originating at the end of the loop and terminating at the start of the loop. The end of the loop has one incoming arc and two outgoing where one of the outgoing arcs is part of the atomic block ending at the first node of the loop.

Name	startLoop
Description	Denotes the start of a loop
Placement	Service level places
Constraints	Only on places with the ID pragmatic and in accordance with the block decomposition rules
Parameters	none
Type	Control-flow
Category	Core
Origin	Derived

Table 10: Description of startLoop pragmatic.

Name	endLoop
Description	Denotes the end of a loop.
Placement	Service level places
Constraints	Only on places with the ID pragmatic and in accordance with the block decomposition rules
Parameters	the condition for continuing to loop
Type	Control-flow
Category	Core
Origin	Derived

Table 11: Description of endLoop pragmatic.

The loop construct is accompanied by the ⟨⟨startLoop⟩⟩ and ⟨⟨endLoop⟩⟩ pragmatics which, analogously to the pragmatics for conditionals, mark the beginning and the end of the loop respectively. The loop pragmatics are derived based on the structure of the CPN model. The pragmatics can, however, be seen in Fig. 22. In the future, it may be useful to add other loop constructs such as for-loops and iterators. The pragmatics are described in Table 10 and Table 11. The ⟨⟨endLoop⟩⟩ pragmatic takes one parameter which contains the condition for continuing in the loop. This condition is given using a language which is described further in Sect. 6.

In Fig. 22, a loop is present starting at the place Start and ending at the place Packet Sent. In the ATT, the start node will become a container-node containing a set of the subsequent nodes on the control flow path up to and including Packet Sent, which will also be annotated with the ⟨⟨endLoop⟩⟩ pragmatic. The ⟨⟨endLoop⟩⟩ pragmatic must be on the last node contained in the loop.

6 Operation Pragmatics

In addition to the general structural core pragmatics and the pragmatics governing the control flow of services, another category of pragmatics is needed to denote operations to be executed at various points in the control flow.

It is impossible to anticipate every possible operation that will be needed for any protocol. Hence, we propose a set of operation pragmatics that are useful in general and allow users to define their own pragmatics as well in order to include sufficient flexibility in our approach. As with pragmatics in general, the

operation pragmatics can be divided into several subcategories depending on their function and generality.

One category of operation pragmatics is derived from typical structural patterns related to accessing state variables and channels. A second category of operation pragmatics is used to facilitate a mapping of the ML expressions and functions used in the arc and guard inscriptions of the CPN model onto the platform for which code is being generated. The latter means that the translation of ML expression in the inscriptions will be based on a mapping determined by pragmatics instead of explicitly parsing and translating ML expressions into the target programming language of the platform for which code is being generated. The general idea is that most of these pragmatics could, at least in principle, be automatically be derived from the CPN models. The option of adding these pragmatics manually leaves the possibility to add operations directly where deriving them automatically would not be worth the effort.

A fully annotated version of the send service in Fig. 5 is shown in Fig. 22. Many additional operation pragmatics are present here that are not present in Fig. 5. In the following, we discuss three of these additional pragmatics, ⟨⟨send⟩⟩, ⟨⟨pop⟩⟩, ⟨⟨partition⟩⟩ in more detail. It is expected that most or all of these extra annotations will be automatically derived so that the modeller will not actually have to add them to the model by hand except in rare cases.

An example of an operation which is useful for most or all network protocols is sending data over the network. The ⟨⟨send⟩⟩ pragmatic is used to send a message over the adjacent network channel. ⟨⟨send⟩⟩ is a derived pragmatic which can be automatically derived from the CPN model by identifying a specific pattern in the model. The ⟨⟨send⟩⟩ pragmatic is present on the transition SendPacket in the SenderSend module in Fig. 22. The pragmatic takes two parameters. The first is an identifier of the endpoint the message should be sent to. The second parameter is the message that is to be sent. In this case this is the `__TOKEN__` value, which is the value of whatever data is in the token coming from the control-flow input place. Analogously, we have a ⟨⟨receive⟩⟩ pragmatic (not shown in the figure, but used in the receive service) that similarly takes two arguments, Receiver which is assumed to contain the endpoint for the receiver and the variable the received message should be put to.

Another example of a pragmatic that may be of general use is the ⟨⟨pop⟩⟩ pragmatic on the Next Message transition. Here the pop does basically the same thing as the arcs to and from the Outgoing place.

An example of a protocol-specific pragmatic is the ⟨⟨partition⟩⟩ pragmatic used on the Partition transition in Fig. 22. It denotes the `partition` function used on the arc to Outgoing, which, for this particular protocol, specifies how the messages are partitioned into sub-messages. In this case, the pragmatic will be defined by the modeller which need to accompany the pragmatic with a code template describing how partition is to be realised on the platform for which code is being generated. The protocol-specific pragmatics enables the code generation to rely on modeller-provided pragmatics and templates to handle net structure and patterns that are not directly supported and can be recognised by the code generation.

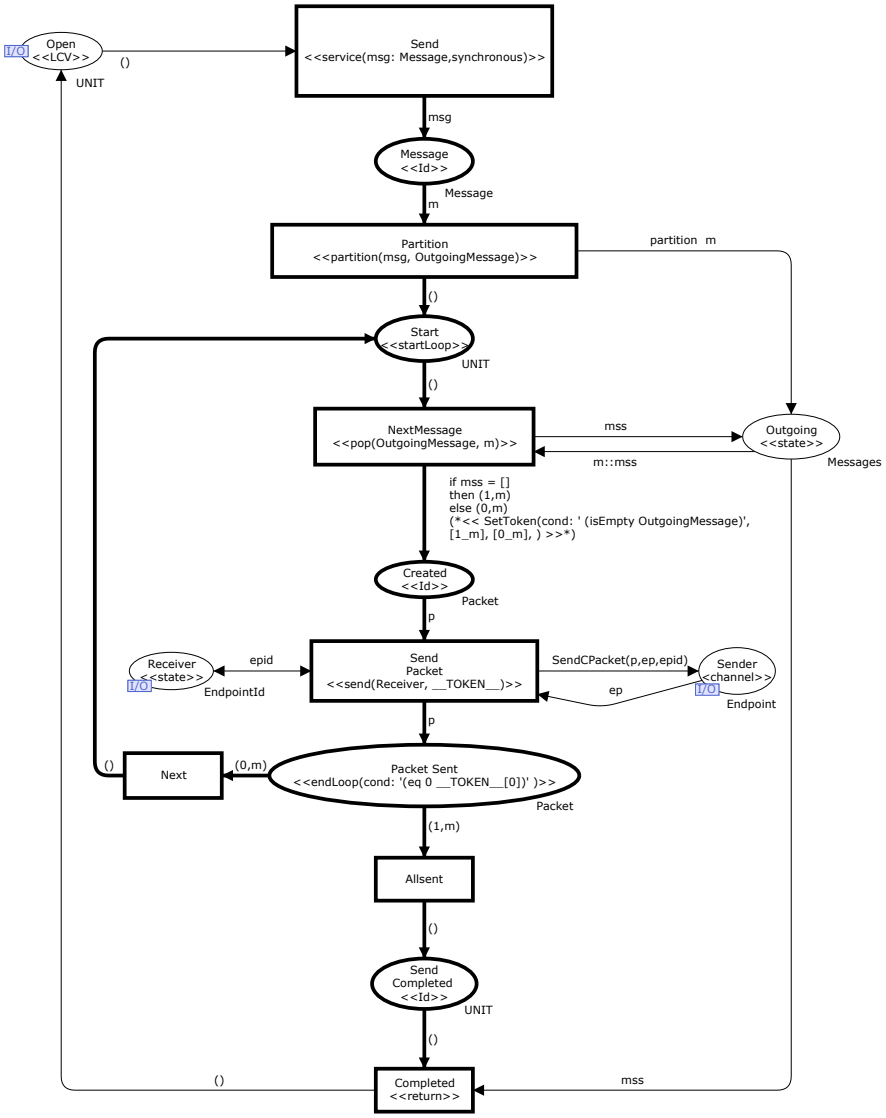


Figure 22: The SenderSend module with all derived pragmatics

6.1 The Condition Language

The observant readers will have noticed that two of the pragmatics in Fig. 22 have parameters that have some more structure, representing the expressions for the loop or expression. These are present in the cond parameter of the ⟨⟨endLoop⟩⟩ pragmatic on the place Packet Sent and in the cond parameter of ⟨⟨setToken⟩⟩ pragmatic which resides on the inscription of the arc between the transition NextMessage and the place Created. In this section the language used to write these two expressions will be briefly described.

The simplest expressions in the language are ones like the one in the cond parameter of the ⟨⟨endLoop⟩⟩ pragmatic on the place Packet Sent which is also

Operator	Description
eq	Check for equality between the two parameters
neq	Check for inequality between the two parameters
t	evaluates to true
isEmpty	Checks whether the collection given as a parameter is empty

Table 12: The current list of operators for the conditional language

shown below:

```
(eq 1 __TOKEN__[0])
```

This expression simply checks for the equality of the two values `1` and `__TOKEN__[0]`. The keyword `eq` denotes that this expression checks for equality. The `__TOKEN__[0]` is currently just passed through. This means that the implementation platform must understand this to take the first element of the collection that is held in the `__TOKEN__` variable. To make this more general is a part of future work

The language syntax is inspired by Lisp. Currently, only boolean expressions are available, and all expressions follow the same syntactical structure: `(operator operands)`. The available operators are shown in Table 12. The set of operators will be extended as needed as we develop our approach further.

7 Abstract Template Trees

An Abstract Template Tree (ATT) is an ordered tree of nodes that contain information to generate code using templates. The ATT is inspired by the abstract syntax trees that is frequently used by compilers as an intermediate representation when generating executable code from programming language code.

The two major types of nodes in the ATT are leaf (operation) and container nodes. A leaf node contains pragmatics for one or more sequential operations such as sending or removing an element from a list. A leaf node, as the name suggest, does not have any children. A container node, however, contains an ordered list of child nodes. The types of container nodes at the service level mimics the blocks defined in Section 5 and have the same constraints.

An excerpt of the ATT generated for the example is shown in Fig. 23. The first level below the root of the ATT is expected to contain only Principals, which are considered block nodes. Each principal contains some service nodes. Services are also block nodes in that they contain their function. If we look at the SenderSend service under the PrincipalSender principal we see that it contains three leaf nodes and one block node. The Block node is a Loop and represent the loop in Fig. 5.

7.1 CPN to ATT

For technical reasons, the CPN model is first translated into an ePNK model. This translation, although tedious, is fairly straightforward and will not be discussed any further here. The first phase in generating an ATT is to add the derived pragmatics to the Petri net structure. This is done by identifying

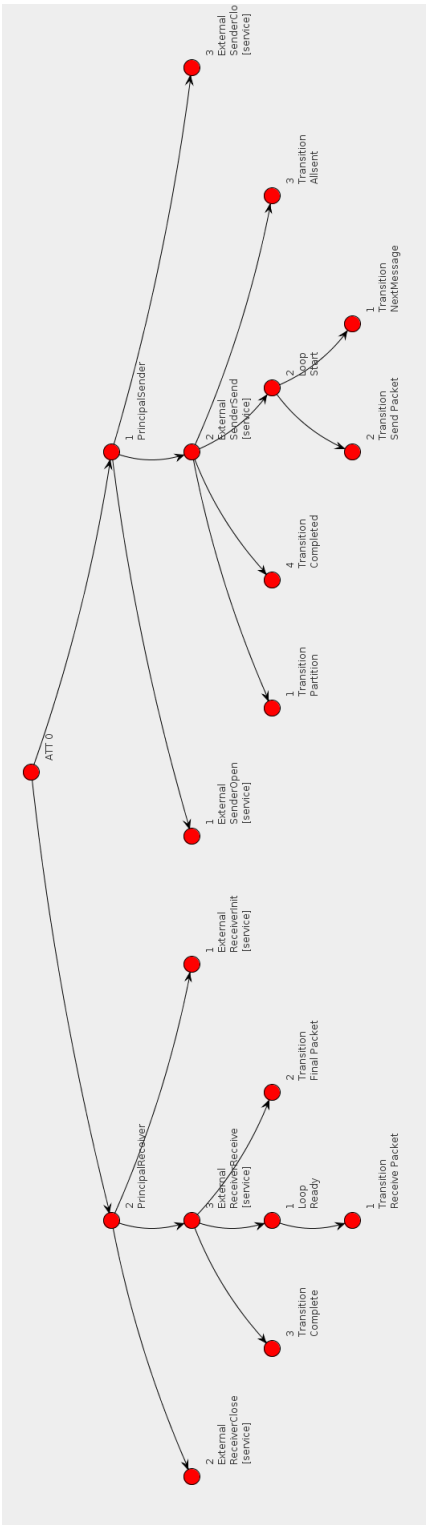


Figure 23: Abstract Template Tree

patterns for each of the derived pragmatics and adding the pragmatic wherever the pattern fits. Although the patterns themselves may include adjacent nodes, all the patterns can be matched against one node at a time, making the pattern matching fairly efficient given small patterns.

With all derived pragmatics added, the ATT can be generated by a guided walk through the model. This walk will start at the top module and for each pragmatic it will generate a corresponding node in the ATT. On the next level, the generator looks for modules with the ⟨⟨service⟩⟩ pragmatic which is transformed to methods in the code. Each service module contains exactly one transition also with the ⟨⟨service⟩⟩ pragmatic. This transition is the starting point for the method modelled by the sub-module. The subsequent set of nodes is divided up by the rules of the block structure described in Sect. 5. The ATT for the CPN in Sect. 2 is shown in Fig. 23.

7.2 Template Binding

When the ATT is generated, and in order to finally generate code, the pragmatics must be bound to templates. This is done by introducing another artefact called a template binding. A template binding contains bindings for all pragmatics that need to be translated into code for a specific platform. In addition the bindings works as a configuration by picking the appropriate templates for each pragmatic.

The bindings are given in a simple domain specific language (DSL) where each line represents a binding. A binding consists of a name followed by a left-parenthesis followed by some key value pairs where the keys can be `pragmatic` which contains the name of the pragmatic, `template` which corresponding value is the path to the template, `isContainer` which indicates whether this pragmatic denotes a container or `isMultiContainer` which is an implementation detail in our tool. The `isMultiContainer` flag indicates whether or not the container is either a loop or a conditional.

A small extract of the binding for groovy cab be seen in Listing 1.

Listing 1: Extract of Groovy bindings

```
Id(pragmatic: 'Id')
Cond(pragmatic: 'Cond', template: '.../groovy/cond.tmpl',
     isContainer: true, isMultiContainer: true)
startLoop(pragmatic: 'startLoop', template: '.../groovy/loop.tmpl',
     isContainer: true, isMultiContainer: true)
endLoop(pragmatic: 'endLoop', template: '.../groovy/endLoop.tmpl')
```

8 Code Generation

As a first step towards evaluating the approach described in the previous sections, we have generated software for the example in Sect. 2. The first step is to translate the CPN model into an ATT. The second step is to translate the ATT to code.

8.1 ATT to Groovy Code

Generating the actual code is a matter of traversing the ATT and invoking the associated templates for each node. Each pragmatic is bound to a template by a mapping. Listing 1 shows a part of the mapping for Groovy programs. As can be seen in the listing, the mapping is fairly straightforward linking between pragmatic name and its corresponding template file.

When a pragmatic is transformed to code its template is run through a template engine together with a number of parameters given by the pragmatics definition and the CPN structure. The templates are sown together by replacing a special tag in the container templates, `%%yield%%`, with the text of the underlying templates in order.

As an example of a container template, the template for a Loop pragmatic for the Groovy language is given in Listing 2. The template creates a while-loop which continues while the `__LOOP_VAR__` variable is true. The body of the loop is populated by replacing the `%%yield%%` directive with the code generated by the templates of the sub-nodes in the ATT. The `__LOOP_VAR__` is updated at the end of the loop by the ⟨⟨endLoop⟩⟩ pragmatic which is always present as the last child element of a loop. The ⟨⟨send⟩⟩ pragmatic is an example of an operation and is used to send a message over a channel. Listing 3 shows the template for the ⟨⟨send⟩⟩ pragmatic which requires two parameters: one is the name of the socket that the message should be sent on, and the other is the variable that holds the message to be sent.

Listing 2: Template for loops in Groovy.

```
%%VARS:__LOOP_VAR__%%
__LOOP_VAR__ = true
while(__LOOP_VAR__){
        %%yield%%
}
```

Listing 3: Template for sending a message.

```
${params[0]}.getOutputStream()
        .newObjectOutputStream()
        .writeObject(${params[1]})
%%VARS:${params[1]}%%
```

8.2 The Generated Code

As an example of the generated code the loop in the senderSend service in the Sender principal is shown in Listing 4. The loop is started by defining a variable, `__LOOP_VAR__`. This is done because the Groovy language does not support do-while loops. After the `__LOOP_VAR__` is defined the loop is entered. Inside the loop the next fragment is code from the template bound to the ⟨⟨pop⟩⟩ pragmatic. This code removes the first element from OutgoingMessage and assigns it to m. Then the code for the ⟨⟨setToken⟩⟩ pragmatic on the arc between the transition NextMessage and the place Created. This code sets the `__TOKEN__` variable in the code according to the conditional statement in the pragmatic: if OutgoingMessage is empty then the message is prefixed by 1, otherwise it is prefixed by zero. The next pragmatic that is found on the control flow path is

the ⟨⟨send⟩⟩ pragmatic on the transition Send Packet. The socket Receiver is used to send the value of the __TOKEN__ variable. In the code, this is done using an ObjectOutputStream sending the list that is now in the __TOKEN__ variable. Finally the template associated with the ⟨⟨endLoop⟩⟩ pragmatics has generated the code for updating __LOOP_VAR__ according to the conditional expression given as a parameter to the ⟨⟨endLoop⟩⟩ pragmatic.

Listing 4: The generated code for the loop of the senderSend service

```
__LOOP_VAR__ = true
while(__LOOP_VAR__){
  def m = OutgoingMessage.remove(0)
  if(OutgoingMessage.size() == 0){
    __TOKEN__ = [1,m]
  } else {
    __TOKEN__ = [0,m]
  }
  Receiver.getOutputStream().newObjectOutputStream().
          writeObject( __TOKEN__)
  __TOKEN__
  __LOOP_VAR__ = ( 0 == __TOKEN__[0] )
}
```

The readability of the generated code is, of course, difficult to measure with confidence without extensive studies using human subjects, although some metrics exists [4]. However, assuming that humans are able to create code that is fairly readable to humans attempting to emulate human generated code by creating code using templates, also written by humans, is expected to create fairly readable code. Also the sequential nature of service level modules makes it possible to create templates so that the resulting code is readable.

8.3 Running the Generated Code

The full code for the example above is shown in Appendix A. The code has been formatted to fit in this paper but is otherwise unaltered. Each principal is represented by a single class and each service is a method in that class.

To execute the generated protocol software we created simple programs that uses the API provided by the protocol. These programs for the sender and receiver are shown in List. 5 and 6 respectively. When the programs are run, the Receiver predictably prints out the line “The following message was received: ”the quick brown fox jumps over the lazy dog”” which was the expected result in this case. Inspection of the network traffic confirms that the message is indeed broken up into smaller frames before sending. In the interest of brevity we do not include the network dump here.

Listing 5: The code for running the sender module

```
def sender = new Sender()
sender.SenderOpen([port: 31337, host:'localhost'])
sender.SenderSend("the␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog")
sender.SenderClose()
```

Listing 6: The code for running the receiver module

```
def reciever = new Receiver()
reciever.ReceiverInit(31337)
def res = reciever.ReceiverReceive()
println "\n\n\nThe␣following␣message␣was␣recieved:␣"+
        "\"$res\"\n\n\n"
```

9 Conclusions and Future Work

In this technical report, we have outlined a technology, which allows us to automatically generate code for protocol software from models. To this end, we introduced a simple, but complete example of a communication protocol. This running example was used for discussing our modelling notation and methodology – CPN models with some extensions – and the concepts and techniques for generating the code. There is a prototypical implementation of a code generator, which is based on these ideas. Actually, there were different variants of such a prototype over time, and experimenting with these variants helped consolidating the concepts. The main purpose of this technical report is to present a consolidated and consistent version of the notation, concepts, and techniques of our approach.

The main objective of this approach is that the code can be generated from what we call *descriptive models*. Descriptive models are typically used for understanding and explaining how a protocol works on a high level of abstraction. Descriptive models focus on the concepts and not on the technical details and, in many cases, these models can be used – with some additional tweaking – also for analysing protocols and for verifying their correctness. Typical practise today is that models are used for analysing a protocol and its specification and for the verification of the protocol. Then, the protocol software is implemented manually based – more or less – on these models. Our approach makes it possible to use, basically, the same descriptive model for analysis and verification as well as for code generation – in both cases, the models are moderately extended.

In our approach, we chose to use Coloured Petri Nets (CPNs) [6] as a modelling notation for descriptive models since they have successfully been used for modelling, analysing, and verifying various kinds of systems [5] for a long time now. Over the time, specific modelling styles, principles, and disciplines have developed for using CPN for that purpose. These styles and principles are mostly used informally – sometimes not even mentioned at all. In our approach, we needed to make them into more rigorous rules.

Since descriptive models are conceptual in nature and on a high level of abstraction, they often do not capture some technical aspects and implementation details. Examples of such information not contained in descriptive models are the API and the interface for calling the services or operations of a protocol. Our approach caters for that by so-called *pragmatics* that can be added to different elements of the model. This way, it is possible to attach additional information without compromising the overall structure of the original model. And our example shows, that all relevant technical information can be added to the model in this way. Our approach comes with some predefined pragmatics which are of general use. But, the approach is open for adding more pragmatics

if need should be. Moreover, pragmatics can be used for adding more technical information, which, in principle, could be derived automatically, but for which no tool support is implemented yet. This way, it is possible to gradually extend the degree of automation of our approach without changing the approach itself.

Another important objective of our approach is the generation of code for different target languages and platforms. To this end, the *abstract template trees* (*ATTs*) and template bindings were introduced; by replacing the set of templates and template bindings, code for a different platform can be generated. In a way, a set of templates along with a template binding can be considered as a characterization of a target platform – working out the details is future work, however. Also part of future work is to refine the way the $\langle\langle \mathsf{LCV} \rangle\rangle$ places are handled by the code generator including making them thread-safe and guaranteeing that they will always be properly set after a service has finished.

In this technical report, we have shown that the approach works for a simple example and for one target platform only. An evaluation for larger examples and different target platforms is still missing – working out larger examples is future work and part of the evaluation process. Likewise, we still need to show that the same CPN models can be used for verification as well as for automatic code generation. Though verification is not the main focus, future work will, in the least, demonstrate that verification from the model is possible in principle. A first step towards verification was taken already in [8].

This technical report focuses on the modelling concepts and notations of our approach as well as the concepts and techniques for generating the code from these models. In order to stay focused, we blatantly ignore other code generation approaches and related work in general. This will be discussed in a separate paper.

References

[1] C. Baier and J-P Katoen. *Principles of Model Checking*. MIT Press, 2008.

[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 *Lecture Notes in Computer Science*, pages 210–290. Springer, 2004.

[3] J. Billington, S. Vanit-Anunchai, and G. E. Gallasch. Parameterised coloured Petri net channel models. *Transactions on Petri Nets and Other Models of Concurrency*, 2009.

[4] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 121–130, New York, NY, USA, 2008. ACM.

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

[6] Kurt Jensen. Coloured Petri nets and invariant methods. *Theoretical Computer Science*, 14:317–336, 1981.

[7] S. Kent. Model Driven Engineering. In *Proc. of Integrated Formal Methods*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.

[8] Lars M. Kristensen and Kent I. F. Simonsen. Towards a CPN-based modelling approach for reconciling verification and implementation of protocol models. In 8^{th} *International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, September 2012.

[9] L.M. Kristensen and K. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *Proc. of Advanced Course on Petri Nets*, LNCS. Springer, 2012. To appear.

A The Generated Code

Listing 7: The generated code for the sender module

```
class Sender {
def Idle = true
def Open
def Receiver

  def SenderOpen(server){
    if(!Idle){
      throw new Exception('unfulfilled␣precondition:␣Idle')
    }
    /*vars: [socket]*/
    def socket
    socket = new Socket(server.host, server.port)
    this.Receiver =  socket
    Open = true
  }

  def SenderSend(msg){
    if(!Open){
      throw new Exception('unfulfilled␣precondition:␣Open')
    }
    /*vars: [msg, OutgoingMessage, __LOOP_VAR__, __TOKEN__]*/

    def OutgoingMessage
    def __LOOP_VAR__
    def __TOKEN__

    OutgoingMessage = msg.getChars().toList().collate(5)
                 .collect{ new String(it.toArray(new char[0])) }

    __TOKEN__ = null//()

    __LOOP_VAR__ = true
    while(__LOOP_VAR__){
      def  m = OutgoingMessage.remove(0)

      if(OutgoingMessage.size() == 0){
        __TOKEN__ = [1,m]
      } else {
        __TOKEN__ = [0,m]
      }
```

```
      Receiver.getOutputStream().newObjectOutputStream()
              .writeObject( __TOKEN__)
      __TOKEN__
      __LOOP_VAR__ = ( 0 == __TOKEN__[0] )
    }

    __TOKEN__ = null//()
    return

    __TOKEN__

    Open = true

  }


  def SenderClose(){
    if(!Open){
      throw new Exception('unfulfilled␣precondition:␣Open')
    }
        /*vars: []*/
    Idle = true
  }
}
```

Listing 8: The generated code for the receiver module

```
class Receiver {
  def Idle = true
  def Ready
  def Sender

  def ReceiverInit(rport){
    if(!Idle){
      throw new Exception('unfulfilled␣precondition:␣Idle')
    }
        /*vars: [socket]*/
    def socket
    socket = new ServerSocket(rport).accept()
    this.Sender =  socket
    Ready = true
  }


  def ReceiverClose(){
    if(!Ready){
      throw new Exception('unfulfilled␣precondition:␣Ready')
    }
    /*vars: []*/
    Idle = true
  }
```

```
  def ReceiverReceive(){
    if(!Ready){
      throw new Exception('unfulfilled precondition: Ready')
    }
    /*vars: [mss, __LOOP_VAR__, __TOKEN__]*/

    def mss
    def __LOOP_VAR__
    def __TOKEN__

    mss =  []

    __LOOP_VAR__ = true
    while(__LOOP_VAR__){
      def p =  Sender.getInputStream().newObjectInputStream()
                     .readObject()
      __TOKEN__ = p
      mss.add(p)
      __TOKEN__ = p
      __LOOP_VAR__ = ( 0 == __TOKEN__[0] )
    }

    __TOKEN__ = null//()
    StringBuffer sb = new StringBuffer()
    mss.each{ sb.append(it[1].toString() ) }
    __TOKEN__ = sb.toString()

    __TOKEN__
    return __TOKEN__

    Ready = true
  }

}
```