Design and Evaluation of a Framework for Annotating Coloured Petri Net Models with Code Generation Pragmatics

Mikal Hitsøy Henriksen

Master's Thesis

Department of Computer Engineering
Bergen University College
Norway

May 7, 2012
Supervisor
Lars Michael Kristensen

Abstract

Model Driven Engineering (MDE) is a software development methodology that focuses on developing domain models that represent knowledge and activities that belong to the application domain. When applied in software development, MDE aims to provide automatic generation of source code from the domain models, which also functions as a mechanic for keeping design and implementation synchronised.

One of the modeling languages that can be used in MDE is Colored Petri Nets (CPN). It is a type of high-level Petri Net, whic is useful for for describing, analysing and validating systems that consist of communication, synchronisation, and resource sharing between concurrently executing components.

A CPN model can accurately model many types of software systems, but cannot directly be used to generate a software implementation. Research is being conducted to define a framework/method? for annotating CPN models of communication protocols to enable source code generation.

This thesis has resulted in a prototype application for annotating CPN models with code generation pragmatics. The prototype is developed as an Eclipse plugin. It builds on the ePNK framework, which uses the Eclipse Modeling Framework (EMF) to provide an extensible platform for working with CPN models. The prototype can import models created by CPN Tools.

The prototype is evaluated by applying it to a set of protocol CPN models. We show that CPN models annotated with code generation pragmatics are a viable method for use in MDE of protocol software. We also give a detailed case study of modeling the WebSocket protocol, including verification and analysis of the CPN model using State Space Generation, and annotation with code generation pramatics.

Contents

Chapter 1

Introduction

The art of software development is an increasingly complex discipline, with new and improved technology emerging at a rapid pace. There is no single answer to how to approach all problems, which has lead to the development of several software development paradigms. But a motivation common to all of them is that developers have always sought increasing levels of abstraction. Today's technology is at a level that potentially gives us means of generating source code from conceptual models of applications, and much research is being conducted to create formal methods of applying this potential.

1.1 Model Driven Software Engineering

Models and diagrams have been used in software design for a long time, and have been standardised with the introduction of Unified Modeling Language (UML) and the tools developed around it (like Unified Process). However, they largely play a secondary role, performing as design tools and documentation.

ref

Model Driven Engineering (MDE) has emerged as a new development methodology, putting the models in the center of the development process. Developers design models that act as both documentation and implementation, and become a layer of abstraction over source code. This has been hailed as a new paradigm shift, like the transition from procedural programming to object-oriented programming was.

MDE has two central concepts: - Domain specific modeling languages (DSML), which are used to formalise application structure, behavior, and requirements of specific domains, such as financial services, warehouse management, task scheduling, and protocol and communication software. A DSML uses a metamodel to describe concepts of the domain, along with associated sematics and constraints.

- Transformation engines and generators, that process models to produce various artifacts, including written documentation, deployment descriptions, alternate representations, and source code.

Meta-modeling, creating a DSML. Model for OO (figure).

Short intro to code generation from models, arguments for.

A modeling language, Petri nets. There exist many extensions to the Petri Net language High-level Coloured Petri Nets (CPN). Which kind of systems, concurrent. Enables verification and analysis, State Spaces.

1.2 Related work

There are examples that demonstrate how to use CPN and other Petri Net variants to model software and generate source code.

Reference and longer explanation of current work of Simonsen [**?**], which defines the domain for the prototype. Code Generation Pragmatics.

The approach consists of three parts:

- Annotate the CPN protocol model with pragmatics which bind the model entities to program concepts,
- create a platform model that knows how to implement specific things for a particular platform,
- and create a configuration model for deciding implementation details, depending on the protocol model and platform model.

Three classes of pragmatics.

Generic pragmatics for defining entities, communication channels external method calls and API entry points

Domain specific pragmatics (an example is encryption)

Model specific pragmatics .

1.3 Thesis Aims

The work of Simonsen is focused on the model transformation and code generation aspect. Annotation of the CPN models are done with simple text files. This approach is cumbersome, and there is a need for a specialised tool for this purpose.

This thesis seeks to satisfy this need by answering the following question: What is a good method of annotating Coloured Petri Nets with code generation pragmatics?

The method used to answer this question is to create a prototype application framework and using it to annotate a test model.

The framework should be capable of supporting the different classes of pragmatics, categorised into Generic, Domain Specific, and Model Specific Pragmatics.

It should be possible to import CPN models created in CPN Tools. Annotations are placed through a tree editor, and also, if possible, through a graphical diagram editor. The framework should provide contextual content assistance

1.4 Thesis Organisation

The thesis is organised as follows:

6-7 linjer om hvert kapittel.

Chapter 2: Background A description of the WebSocket protocol, the primary case study used in this thesis. An introduction to Colored Petri Nets and the CPN Toolls application used to create the models, with a breakdown of the CPN model of the WebSocket protocol we produced for the thesis.

Chapter 3: State Space Analysis of the CPN Web Socket Protocol An explanation of State Space Analysis of CPN models, and an example of how to apply it using the WebSocket Protocol as the subject. This establishes the produced model as correct.

Chapter 4: Technology and Foundations The produced prototype is built on top of a number of frameworks and technologies. This chapter gives an introduction to each of them, including the Eclipse Platform and its modules. We describe ePNK, which makes up the foundation of the prototype, and how it can be extended to support new functionality. Explanation of reason behind each chosen solution. Introduction to ontologies, the format used to represent pragmatics classes.

Chapter 5: Analysis and Design Start with discussion and detailing of requirements for the editor. We give details of the ePNK Petri Net Type Definition for Colored Petri Nets and Annotated Colored Petri

Nets. Describe how the implementation works. Eclipse Plugin structure. Extension Points, used to register with ePNK and extend context menus. Reasoning with ontologies, using this to decide available pragmatics.

Chapter 6: Evaluation Discuccion on which requirements have been met. Overview and explanation of test cases. Results from test cases and feedback from users. Evaluation of technology used, and opinion on maturity of MDD and the tools available.

Chapter ??: ?? Summary, personal experience, limitations and suggested focus of future work.

mer?

The reader is assumed to be familiar with Java programming.

Chapter 2

Background

Bør kanskje gjenvurdere om det skal hete “Background”

2.1 The WebSocket Protocol

The primary case study for this thesis is the WebSocket protocol [?]. From the abstract of the document:

> The WebSocket protocol enables two-way communication between a client running untrusted code running in a controlled environment to a remote host that has opted-in to communications from that code.

Fig. 2.1 shows the basic sequence of the WebSocket protocol. To establish a connection, a client sends a specially formatted HTTP request to a server, which replies with a HTTP response. Once the connection is established, the client and server can then freely send WebSocket message frames, until either endpoint sends a control frame with the opcode 0x8 for close and optionally data about the reason for closing. The other endpoint then replies with the same opcode and data, and the connection is considered closed.

From the RFC document:

> Conceptually, WebSocket is really just a layer on top of TCP that does the following:
>
> - adds a Web ”origin”-based security model for browsers
> - adds an addressing and protocol naming mechanism to support multiple services on one port and multiple host names on one IP address;

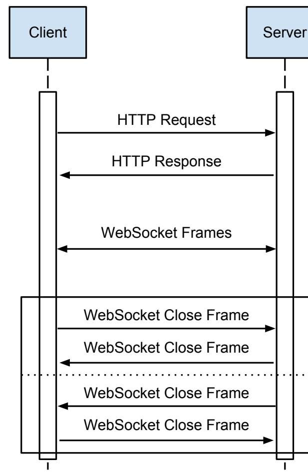


Figure 2.1: Sequence Diagram of the WebSocket protocol

- layers a framing mechanism on top of TCP to get back to the IP packet mechanism that TCP is built on, but without length limits
- includes an additional closing handshake in-band that is designed to work in the presence of proxies and other intermediaries

2.2 Coloured Petri Nets

Coloured Petri Nets (CPN) are a type of directed graph used to model processes, especially processes with an asynchronous and/or concurrent nature. Common examples are modelling networks, processes and protocols, as well as concurrent programming design.

The strength of CPN lies i the operations that can be performed with it: Simulation, verification and analysis.

More recently, research has been conducted to examine the feasibility of using CPN for software design with automatic code generation.

The structure of CPN models will be explained gradually through the case study in this chapter.

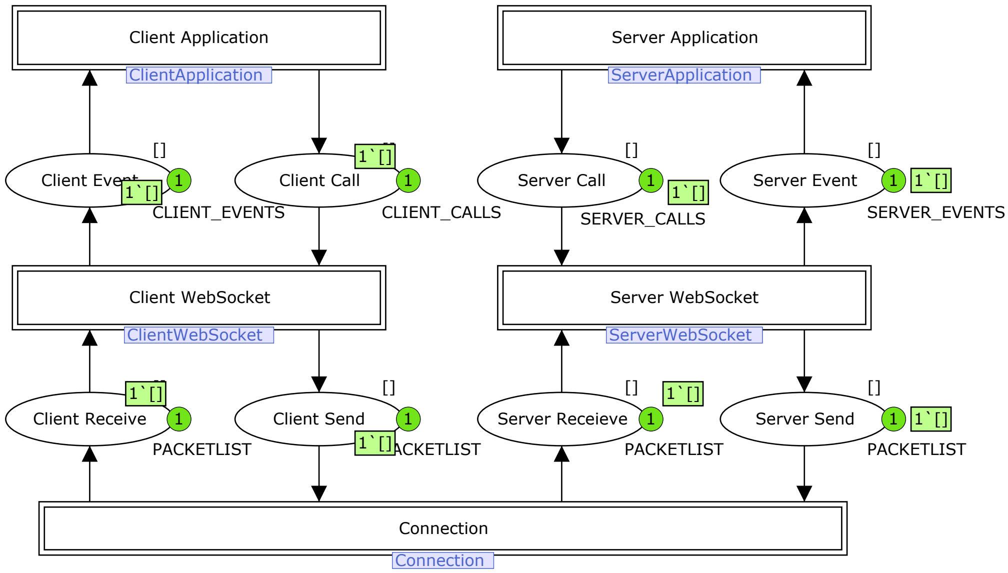


Figure 2.2: Overview of CPN model of the WebSocket protocol

2.2.1 CPN Tools

Graphical tool used to design CPN models.

2.3 The WebSocket CPN Model

The websocket protocol is the primary case study of this thesis. In this section the model that has been built is described in full detail.

2.3.1 Overview

Fig. 2.2 shows the top-level Overview module of the WebSocket CPN model. The model has been laid out to resemble part of the OSI model [**?**], where Client Application and Server Application each correspond to the top two layers Application and Presentation, Client WebSocket and Server WebSocket correspond to the Session layer, and Connection corresponds to the lower layers Transport, Network, Data Link and Physical.

Model basics

A Petri Net model consist of pages, or modules, that contain Places,

Funker dårlig å starte med Overview modulen for å forklare hvordan CPN fungerer, mangler eksempler på transitions og arcs

Transitions and Arcs.

Places are represented by circles. They can contain tokens from a specified colour set, and have markings to define initial token(s). As an example, in the lower right of Fig. 2.2 we have the Server Send place. This place can have `PACKETLIST` colour tokens (or just tokens), and has an initial marking of an empty list `[]`. It currently has one token shown in the green box, `1'[]`, which means one of an empty list. The number in the smaller circle shows the total number of tokens in this place.

A token can be a single value like the integer 3, the string "Hello" or simple units which resemble tokens from regular Petri Nets. A colour set is a set of such tokens, for example all integers, the integers from 1 to 100, all lowercase letters, or all strings of length 5. These are examples of simple colour sets, as they each consist of basic data types. Compound colour sets are made by combining the simple ones in different ways to create data structures, for example an integer together with a string might describe a postal code and city. This will be described in more detail in the next section.

Transitions are represented by rectangles.

Double-bordered rectangles represent sub-modules and are called substitution transitions. These sub-modules have input and output places that connect to the places on their parent module. This enables us to keep the model structured and readable, and also allows reuse of modules in different parts of the model by instancing them, which will be described toward the end of the chapter. The details of each submodule is described in the following subsections.

Arcs are directed arrows going between a place and a transition (including substitution transitions), and will be explained in the next submodule.

The State of the model is defined by the current tokens in each place. A Transition can consume tokens from places if there is an arc from the place to the transition, and produce new tokens in places where there is an arc from the transition to the place. This changes the state of the system.

Declarations

CPN Tools uses the CPN ML language to specify declarations and net inscriptions. This language is an extension of the functional programming language Standard ML, developed at Edinburgh University.

All coloursets, variables, symbolic constants and functions have to be declared before they are used in the model. In CPN Tools these declarations can be grouped together, but they are still parsed sequentially, so if one declaration depends on another, it has to be declared after its dependency.

Colour sets are defined with the following syntax:

```
colset name = <type-specific>;
```

Names are always capitalised in this thesis, but any CPN ML identifier is valid.

The places in the Overview use the following colour set definitions:

Listing 2.1: Overview colour sets

```
colset OPERATION = with CONNECT | TEXT | BINARY | PING | PONG |
    CLOSE;

colset MESSAGE = record Op: OPERATION * Message: STRING;

colset MESSAGES = list MESSAGE;

colset CONN_RESULT = bool with (fail, success);

colset EVENT = union Msg:MESSAGE + ConnRequest + ConnResult:
   CONN_RESULT;

colset EVENTS = list EVENT;

colset CONN_REPLY = bool with (reject, accept);

colset MSG_OR_CONN_REPLY = union Msg':MESSAGE + ConnReply:
   CONN_REPLY;

colset MSGS_OR_CONN_REPLY = list MSG_OR_CONN_REPLY;

colset PACKET = union HttpReq:HTTPREQ + HttpRes:HTTPRES +
   WsFrame:WSFRAME;

colset PACKETLIST = list PACKET;
```

The `OPERATION` colour set is an enumeration that represents the different types of messages that can be passed between the application and the protocol layer. All of these correspond with opcodes used in WebSocket frames, except `CONNECT` which is used for initiating a connection.

A `MESSAGE` is a record that consists of an `OPERATION` and the `STRING` message body. Both data- and control-frames can have messages, although control-frame messages might not be shown to the user. Tokens from record colour sets can have their component tokens referenced by name, in this case `Op` and `Message`. We can also have a list of `MESSAGES` to keep them ordered. Lists will be explained in detail later where they are used and manipulated in the model.

The WebSocket layer also needs a way to notify the application about connection results, which is done with the `EVENT` colour set. This is a union colour set which can be either a `MESSAGE`, a `ConnRequest` identifier (with no other colour set associated), or a `CONN_RESULT`. A union colour set is used if a place should be able to contain tokens from different colour sets, or if such tokens should be handled in the same way at a point in the model. They can also contain simple identifiers (like `ConnRequest`). And like `MESSAGES` we can have a list of `EVENTS`.

The client application sends `MESSAGES` to the WebSocket layer and receives `EVENTS` back. The server application also receives `EVENTS`, but can send either a `MESSAGE` or a `CONN_REPLY`, which is used for connection attempts. The `CONN_REPLY` colourset is simply a boolean value with different names for true and false, for improving readability and semantics. A union of these two types must exist, `MSG_OR_CONN_REPLY`, so the same place can be reused.

TODO: Noe om hvorfor vi bruker kun en plass for dette

Both the client and server WebSocket layers send and receive `PACKET` tokens, a union of three types of data. `PACKET`s are abstract and not fully modeled actual network packets, as this is not relevant to how WebSocket works. Ther list version `PACKETLIST` These colour sets are described in the section for the Connection module.

2.3.2 Client Application

This module (Fig. 2.3) is laid out from top to bottom to loosely show the sequence of operation, as is most of the other modules.

The places at the bottom represent the interface to the WebSocket layer. To simplify the overview model and facilitate easier later modification and expansion, we have only two places that act as input and output. These are called Receive Client Events and Send Client Message, and are tagged In and Out respectively. These are connected to the respective places on the Overview, which are also connected to corresponding places in the WebSocket Library.

Variables

The arcs here have inscriptions on them containing variables. Variables are declared globally and can only be bound to tokens of the colourset they are defined for.

When a transition fires, it will bind the tokens it is consuming to the variables of the inscriptions on the arcs from the respective places. These

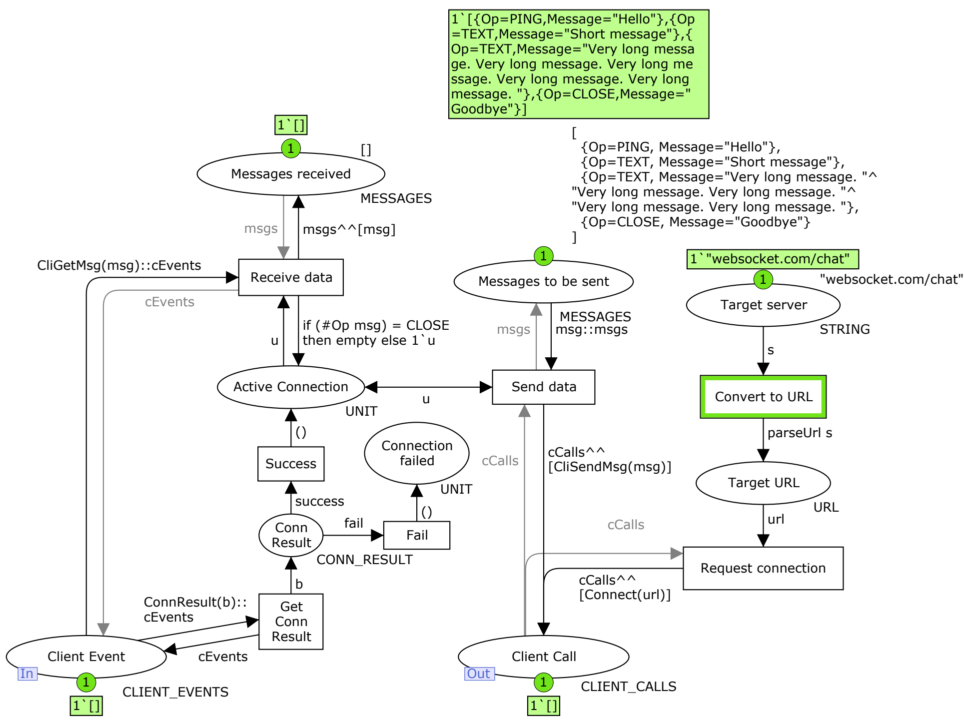


Figure 2.3: The Client Application

variables can then be used to produce new tokens, by direct manipulation and/or in conditional statemens. The expression that will produce the desired result is inscribed on the output arc.

A set of variables has been declared for the simple coloursets as follows:

Listing 2.2: Simple Colourset Variables

```
var u, u1, u2, u3: UNIT;
var b, b1, b2, b3: BOOL;
var i, j, k: INT;
var s, s1, s2, s3: STRING;
var ss, ss1, ss2: STRINGLIST;
```

Several variables for the same coloursets have been created for convenience in the cases where tokens of the same type is consumed from different places. If the same variable was used on each arc from two different places, both places would need a token with the same value.

Variable declarations will be listed as they are encountered in the model. As a general guideline, most variables are named with the first letter of its colourset for non-lists, and the same letter plus the letter s for lists. On this submodule we have the following variables:

Listing 2.3: Client Application Variables

```
var msg: MESSAGE;
var msgs, msgs2: MESSAGES;
var es: EVENTS;
```

Queues

A lot of the places in this model rely on tokens being consumed in the same order they are produced, and to enforce this, queues are used, facilitated by list colour sets. To describe a list in SML, we use square brackets `[ ]`. By themselves they represent an empty list. To describe a populated list, we write each token inside the brackets separated by commas. An example of this is seen in Fig. 2.3 on the initial marking for the `Messages to be sent` place.

To process such a list, we use the `::` operator like this: `head::tail`, where head is the first element in the list, and tail is all the following elements. To concatenate two lists, we use the `^^` operator.

To emulate the queue behaviour we need, we use a list of a colourset instead of using the actual colourset we want in that place. When we want to take an element from the front of the queue, we use the `::` operator to bind the head and tail of the list to variables, and put only the tail back to the source place. When we want to append an element to a queue, we concatenate the queue using the `^^` operator with a new list containing only the new element. To improve readability of the model, these queue operations have one arc slightly dimmed, to emphasise the flow direction of data.

An example of this is seen in Fig. 2.3 on the arc from the `Send data` transition to the `Messages to be sent` place. On the left side, the list in the Messages to be sent place is bound to the two variables msg and msgs, and only msgs is put back. On the right side, we take the list in Send Client Messages and bind it to the variable `msgs2`, and send back a concatenation of this list and a new list containing msg, which was bound earlier in the incoming arc from Messages to be sent. The total effect is that we take the first element from Messages to be sent, and add it to the end of the list in Send Client Message.

Program Flow

The Request connection transition at the top is highlighted by a green border, which means that it is enabled and ready to fire. If we were to fire this transition, it would consume the `STRING` token, named s, produce a `MESSAGE` token containing s, and add it to the queue in the Send Client Message place.

Next, the Client Application waits for a `CONN_RESULT` token, and if this is equal to `success`, we add a `UNIT` token to the Active Connection place. If any other messages are received before this, they will wait at the Receive Client Event place, since the Receive Data transition will not fire unless the Active Connection place has a `UNIT` token.

If and when the Active Connection place has a `UNIT` token, the Client Application can start sending and receiving messages. A sample of messages has been predefined at the Messages to be sent place, but this would probably be dynamic in a real application.

Finally, if the Client Application receives a `MESSAGE` where the `OPERATOR` is `CLOSE`, nothing is put back into the Active Connection place, and the connection is effectively closed.

Kanskje dette kunne vært mer abstrakt? Argument for hvorfor modellert på denne måten.

2.3.3 The Client WebSocket Module

This module consists mostly of substitution transitions. The only processing being done directly on this module is masking of all websocket frames , which is required from the client by the protocol specification. The rest is

Hvordan funker mask funksjonen?

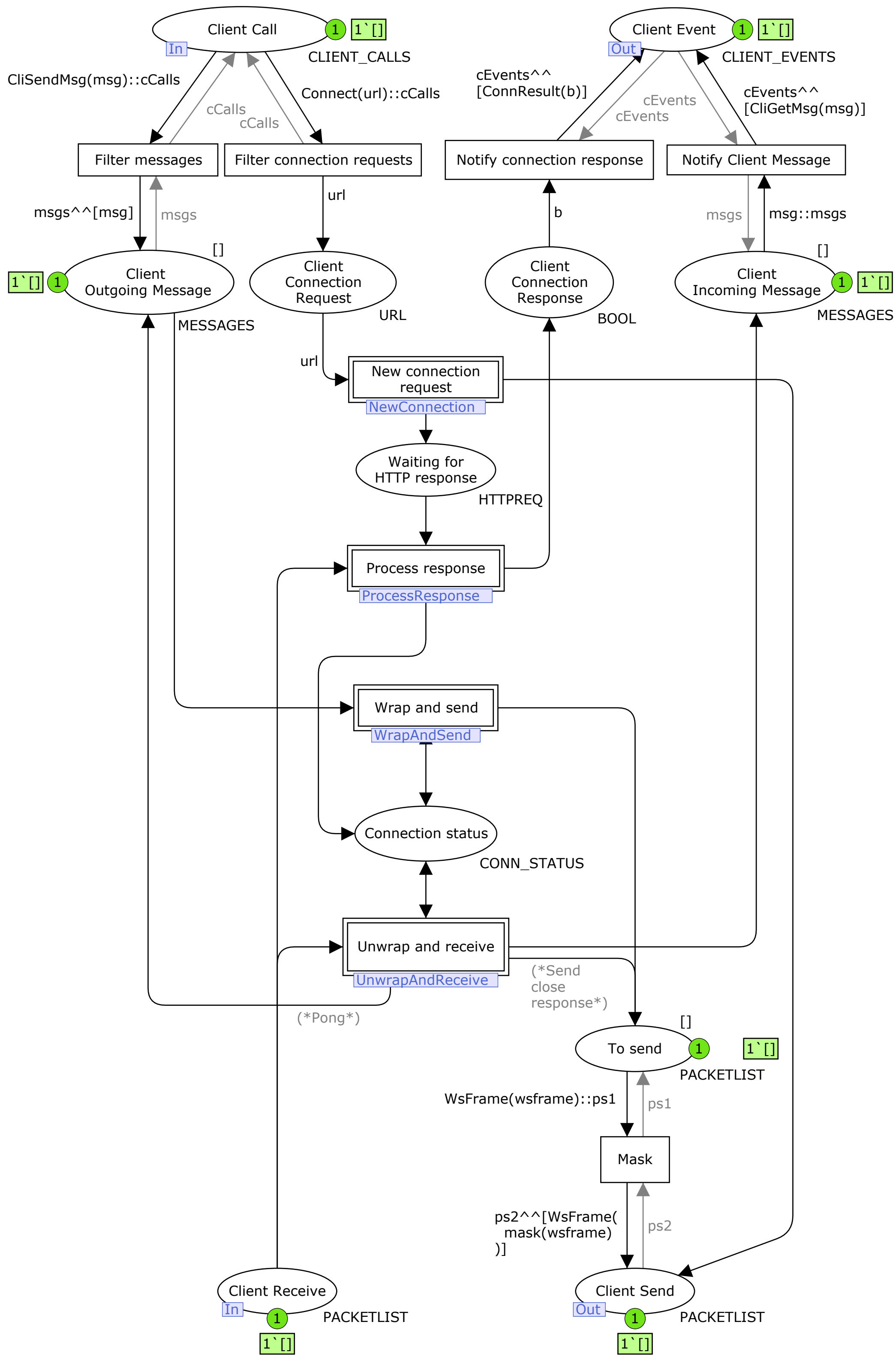


Figure 2.4: The Client WebSocket Module

plumbing between the submodules.

The new coloursets are for HTTP requests and responses, WebSocket frames, and packets. They have corresponding variables.

Listing 2.4: Library colour sets

```
colset HTTP_VERB = union GET + POST + PUT + DELETE + HEAD;

colset REQUEST_LINE = record
Verb: HTTP_VERB *
Path: STRING *
Version: STRING;

colset HEADER = product STRING*STRING;

colset HEADERS = list HEADER;

colset HTTPREQ = record
RequestLine: REQUEST_LINE *
Headers: HEADERS;

colset RESPONSE_LINE = record
Version: STRING *
Status: INT *
Message: STRING;

colset HTTPRES = record
ResponseLine: RESPONSE_LINE *
Headers: HEADERS;

colset MASK = list BYTE with 4..4;

colset MASKING = union
Nomask + Mask:MASK;

val opContinuation = 0x0;
val opText = 0x1;
val opBinary = 0x2
val opConnectionClose = 0x8;
val opPing = 0x9;
val opPong = 0xA;

colset WSFRAME = record
Fin: BIT *
Rsv1: BIT *
Rsv2: BIT *
Rsv3: BIT *
Opcode: INT *
Masked: BIT *
```

```
Payload_length: INT *
Masking_key: MASKING *
Payload: STRING;

colset WSFRAMES = list WSFRAME;

colset PACKET = union
HttpReq:HTTPREQ +
HttpRes:HTTPRES +
WsFrame:WSFRAME;

colset PACKETLIST = list PACKET;
```

Listing 2.5: Library variables

```
var wsframe: WSFRAME;
var wsframes, wsframes2: WSFRAMES;
var httpreq: HTTPREQ;
var httpres: HTTPRES;
var p: PACKET;
var ps, ps1, ps2: PACKETLIST;
```

There are also static values declared here, which correspond to WebSocket frame operation identifiers.

New connection

This module is fairly straightforward. We take the first `MESSAGE` in the list and create an HTTP request, which is then queued to be sent by the client, as well as keeping a copy of the request for validation purposes when the response arrives.

The transition here has a guard inscription enclosed in square brackets. It is used for two things: The first line limits the transition to only accepting messages with the `CONNECT` operation, while the second line binds the url variable by parsing the string in the message into a `URL` token by using the `parseurl()` function. This function makes use of a utility function `split()` which splits a string on the first occurrence of a given character. To do this, it uses a recursive function `split2()` to iterate over the characters. The functions have to be declared in reverse order to satisfy dependency.

Listing 2.6: URL Declarations

```
colset URL = record
Protocol: STRING *
Host: STRING *
Port: INT *
```

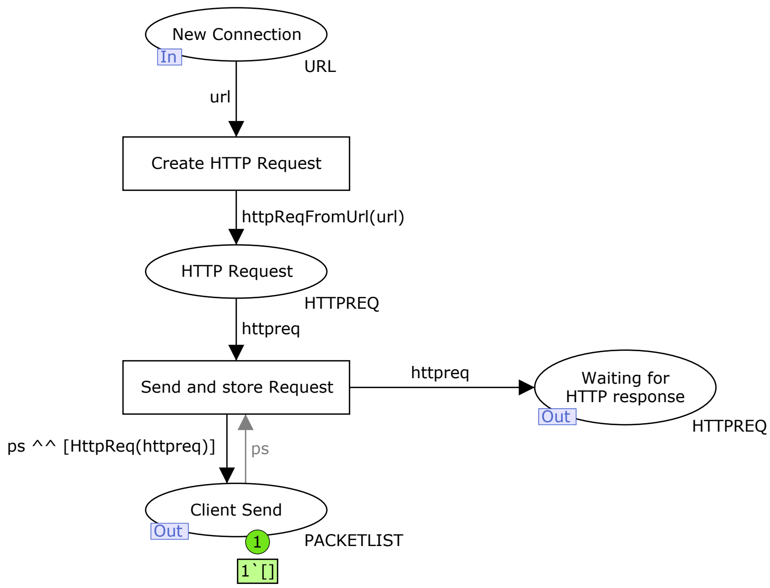


Figure 2.5: New Connection submodule

```
Path: STRING;

var url: URL;

fun split2 (s, t, i) =
(* Recursively scan for character t in string s starting as
    position i, split
if match *)
  let
    val ss = String.extract(s, i, NONE)
  in
    if String.isPrefix t ss then
      [substring(s, 0, i),
      String.extract(s, i + String.size t, NONE)]
    else split2(s, t, i+1)
  end

fun split (s, t) =
  (* Split string s on character c *)
  if String.isPrefix t s then
    [String.extract(s, String.size t, NONE)]
  else if String.isSubstring t s then
    split2 (s, t, 1)
  else [s]
```

```
fun parseUrl (s) =
 let
   val proto'rest = split (s, "://")
   val proto'rest =
     if length proto'rest = 1
     then "ws" :: proto'rest
     else proto'rest
   val pro = List.hd(proto'rest)

   val host'path = split (List.nth(proto'rest, 1), "/")
   val pat = if length host'path = 2
     then "/" ^ List.nth(host'path, 1)
     else "/"

   val host'port = split (List.hd(host'path), ":")
   val hos = List.hd(host'port)

   val port'default = case pro of
     "wss" => 443
     | _ => 80
   val por = if length host'port = 1
   then port'default
   else let
     val port'str = List.nth(host'port, 1)
     val port'int'opt = Int.fromString port'str
   in
     Option.getOpt(port'int'opt, port'default)
   end
 in
   {Protocol=pro,
     Host=hos,
     Port=por,
     Path=pat}
 end;
```

The function httpReqFromUrl takes a URL argument and uses it to produce a HTTPREQ token with headers according to the WebSocket protocol requirements. Not all headers defined by the protocol are used, because it is assumed this application is not browser based, and therefore specifying an origin does not make sense.

Skriv hvorfor det er en rimelig antagelse. (Det er det kanskje ikke)

Listing 2.7: httpReqFromUrl

```
fun httpReqFromUrl (url:URL) =
{
  RequestLine={
    Verb=GET,
    Path=(#Path url),
    Version=httpVersion
```

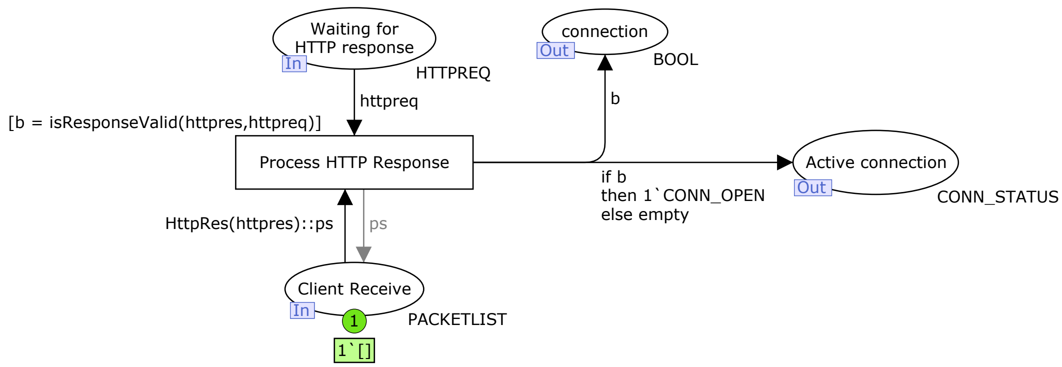


Figure 2.6: Process Response submodule

```
  },
  Headers=[
    ("Host", (#Host url)),
    ("Upgrade", "websocket"),
    ("Connection", "Upgrade"),
    ("Sec-WebSocket-Key", (B64 nonce)),
    ("Sec-WebSocket-Version", "13")
  ]
};
```

Process response

On this module, the transition fires when a HTTPRES token is received from the server (and the HTTPREQ token from earlier is still waiting in its place). The boolean variable b is bound to the result from the isResponseValid function, which checks if the server's reply is valid and conforms to the WebSocket protocol specification.

Listing 2.8: isResponseValid

```
fun isResponseValid (res:HTTPRES, req:HTTPREQ) = let
  val rline = #ResponseLine res
  val headers = #Headers res
  val accepttoken = generateAccept(
    getHeader("Sec-WebSocket-Key",
      (#Headers req)))
in
  #Status rline = 101 andalso
  getHeader("Upgrade", headers)
    = "websocket" andalso
  getHeader("Connection", headers)
```

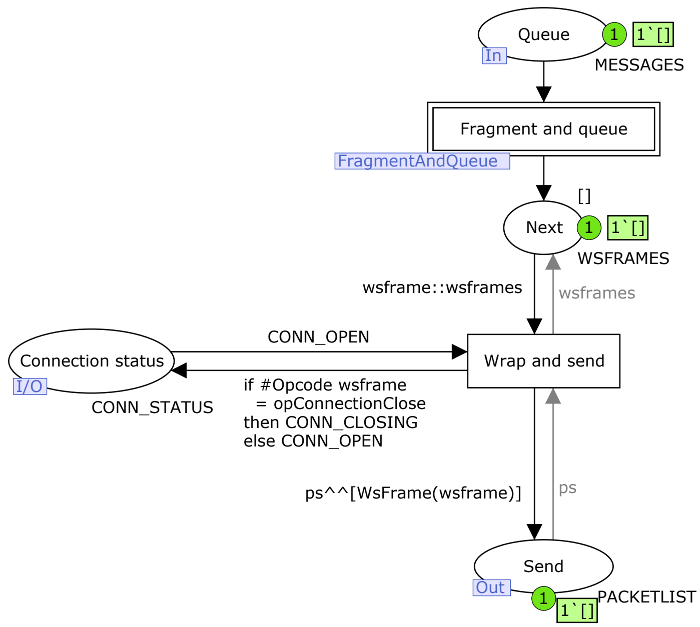


Figure 2.7: Wrap And Send submodule

```
    = "Upgrade" andalso
  getHeader("Sec-WebSocket-Accept",headers)
    = accepttoken
end
```

It sends the result directly to the Client App, and puts a `CONN_OPEN` token in the Active Connection place if b is true.

Wrap and send

This module takes new messages, wraps and optionally fragments them in the Fragment and queue submodule (explained below), and sends them if there is an open connection. If a Close frame is being sent,the connection state will be changed to `CONN_CLOSING`, which prevents further sending.

Fragment and queue The top transition on this has two tasks: Making sure we do not process CONNECT messages, and filtering data and control frames into different places using the isData function.

Listing 2.9: isData

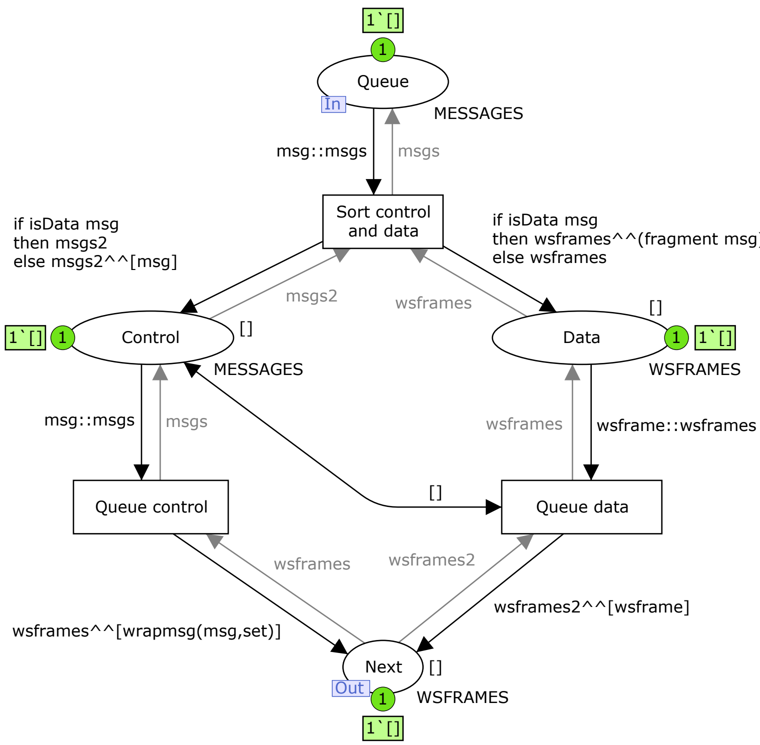


Figure 2.8: Fragment And Queue submodule

```
fun isData (msg:MESSAGE) =
(#Op msg = TEXT) orelse
(#Op msg = BINARY);
```

Control frames should never be fragmented, but data frames could be. This is taken care of by the fragment function, which also converts the message into a WSFRAME token. The control frames are converted in the same way by the wrapmsg function. Both of these functions rely on the wrap function, and therefore have to be declared after it.

Listing 2.10: wrap wrapmsg and fragment

```
fun wrap (opc,payload,fin) = { Fin=fin,
  Rsv1=clear,
  Rsv2=clear,
  Rsv3=clear,
  Opcode=opc,
  Masked=clear,
  Payload_length=(String.size payload),
  Masking_key= Nomask,
  Payload=payload
}

fun wrapmsg (msg:MESSAGE,fin) =
  wrap(opSym2Hex(#Op msg),
    (#Message msg), fin);

fun fragment (msg:MESSAGE) = let
  fun loop (opc, s, acc) =
    if (String.size s) > fragSize
    then loop(
      opContinuation,
      String.extract(s,fragSize,NONE),
      acc^^[wrap(opc,
        String.substring(s,0,fragSize),
        clear
      )]
    )
    else
      acc^^[wrap(opc, s, set)];
  in
    loop(
      opSym2Hex(#Op msg),
      (#Message msg),
      [])
  end;
```

The WSFRAME tokens are then queued randomly one by one from the data queue or the control frame queue. This allows control frames to be injected between the parts of a fragmented data frame, as required by the WebSocket protocol specification. Control frames are prioritised, by the prescence of a two-way arc from the Control place to the Queue data transition, inscribed with `[]`, which prevents it from being enabled if the list in the Control place is not empty. This priorisation is allowed but not required by the WebSocket protocol specification, but is included here to emphasise that control frames can be sent even between two fragmented data frames.

Unwrap and receive

Received WebSocket frames that arrive in the Packet Received place at the bottom can take three paths.

The first is to the left, and happens if the connection is in the `CONN_OPEN` state (checked on the arc) and the frame is not a close frame (checked in the guard of the Receive transition). The WebSocket frame is put in the Received WS Frame place, and if it is a Ping frame, a Pong frame is immediately queued for sending with identical message body.

The received frame is then checked on two points for fragmentation: If the Fin bit is set and the opcode is not continuation, it is not part of a fragmented message and converted directly to a MESSAGE. If either or both of those conditions are not true, tihs is part of a fragmented message and is processed in the Defrag submodule (explained below).

The second path a frame can take is to the right, and happens if the connection is in the `CONN_OPEN` state (checked on the arc) and the frame is a close frame (checked in the guard of the Receive transition). A close frame is created and set to be sent, and the connection state is changed to `CONN_CLOSED`, since we have both received and sent a close frame. Note that the packetlist from Client Send is not appended to but instead discarded, because we can not expect the other end to process any more frames other than a close frame since it has already sent a close frame of its own.

The third path is upwards and happens if the connection state is `CONN_CLOSING`, which means a close frame has been sent and we are waiting for a reply. Any payload is ignored, and the connection state stays the same until a close frame is received, in which case the connection state is set to `CONN_CLOSED`.

Defragmenting fragmented frames Frames that are part of a fragmented message should always arrive in a specific order, and correspond to each of the transitions on this module.

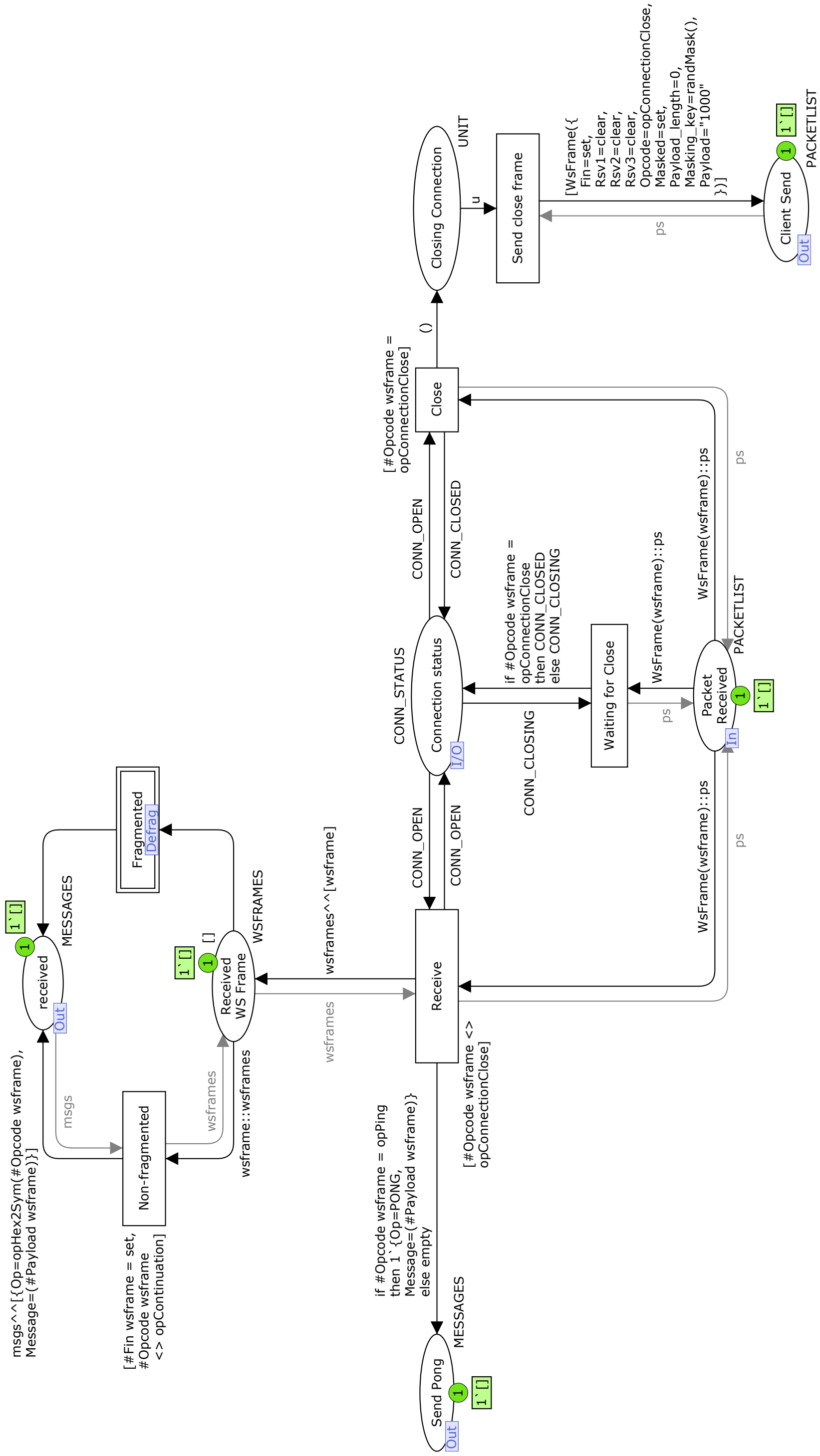


Figure 2.9: Unwrap And Receive submodule

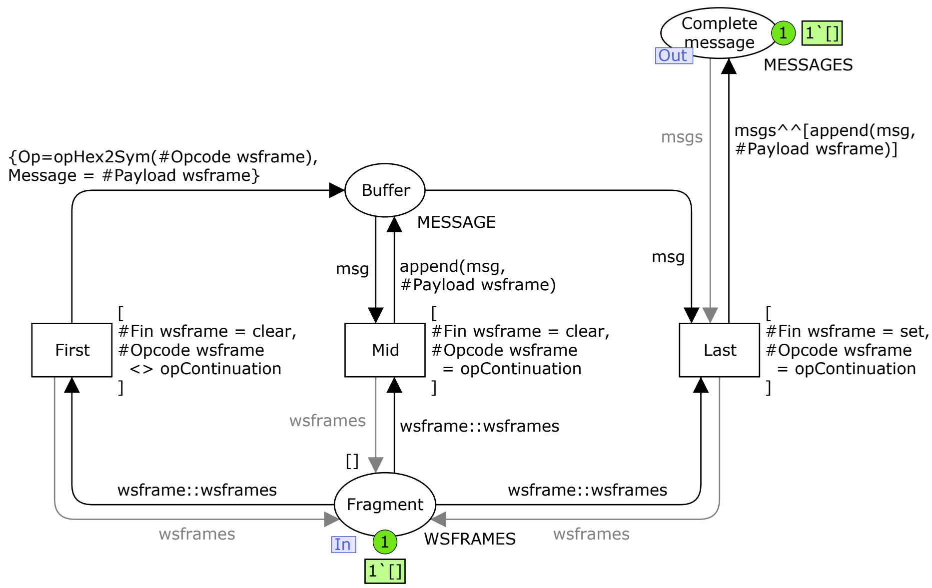


Figure 2.10: Defragment

If the Fin bit is clear and the opcode is not continuation, this is the first frame in the series. A mew `MESSAGE` is created with the opcode and payload from the WebSocket frame and put in the Buffer place.

If the Fin bit is clear and the opcode is continuation, this frame belongs in the middle of the sequence. The payload is appended to the `MESSAGE` in the Buffer place using the `append()` function.

Listing 2.11: append

```
fun append (msg:MESSAGE, s) =
  {Op = #Op msg,
  Message = (#Message msg)^s}
```

If the Fin bit is set and the opcode is continuation, this is the last frame of the sequence. We append the payload to the message and put it in the final Completed message place.

Note that the WebSocket protocol does not allow fragmented messages to be interleaved, so it can be assumed that consecutive fragments belong to the same message. Fragment inerleaving can be defined by subprotocols, but this is not relevant to this model.

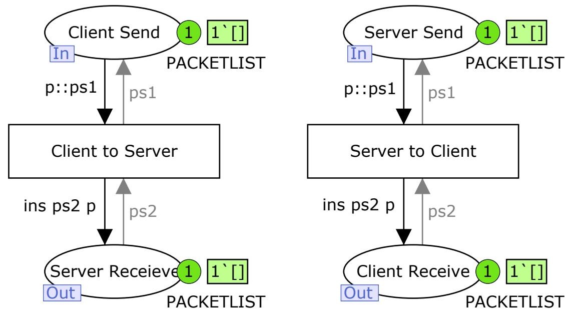


Figure 2.11: The Connection module

2.3.4 Connection

Fig. 2.11 shows the connection; a very simple model. The packets that come from the Client Send place go to the Server Receive place, and from the Server Send place to the Client Receive place. The transportation of data between the client and the server as well as establishing and ensuring a stable connection is assumed to be taken care of by TCP, as the WebSocket protocol specifies, and is not necessary to model in detail to show how WebSocket works.

The packets are also not converted to pure bits or bytes. This abstraction was made since the inner workings of the TCP layer is not relevant to the WebSocket Protocol.

2.3.5 The Server WebSocket Layer

The Server WebSocket layer is very similar to the Client version. The main differences are that instead of masking outgoing frames, we are checking incoming frames for a mask and unmasking them, and that we are checking for incoming connections and replying to them based on what the Server Application decides.

The Server Application can send two types of coloursets: `MESSAGE` and `CONN_REPLY`. A special colourset has been made to accomodate this, `MSG_OR_CONN_REPLY` .

The submodules that belong to the `Wrap and Send` and the `Unwrap and Receive` substitution transitions are the same as the ones for the Client Library. To be more precise, the Client Library and Server Library both have instances of the same submodule, so that editing the submodule model affects both parent modules while simulating them lets them have different

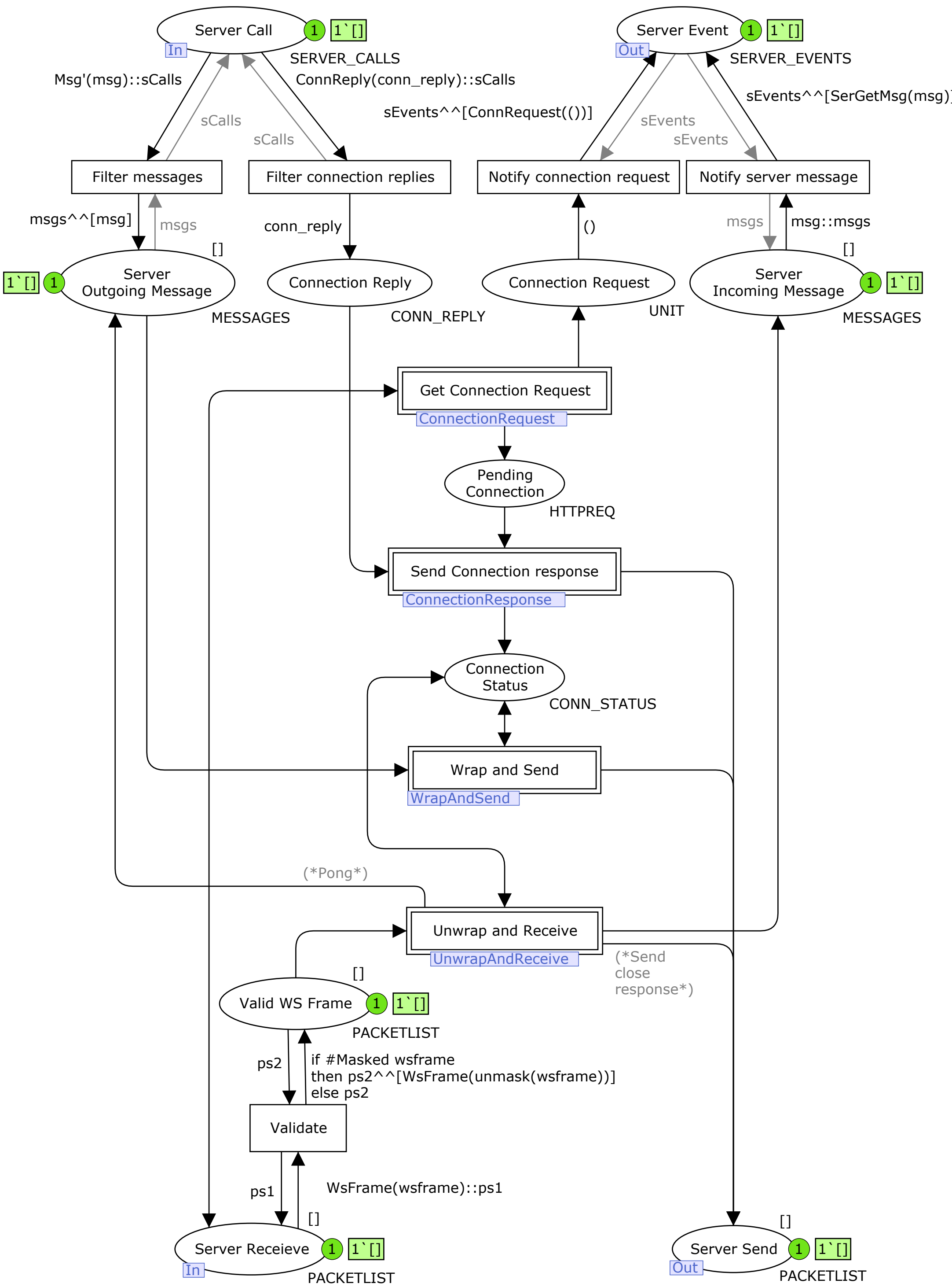


Figure 2.12: The Server WebSocket Layer

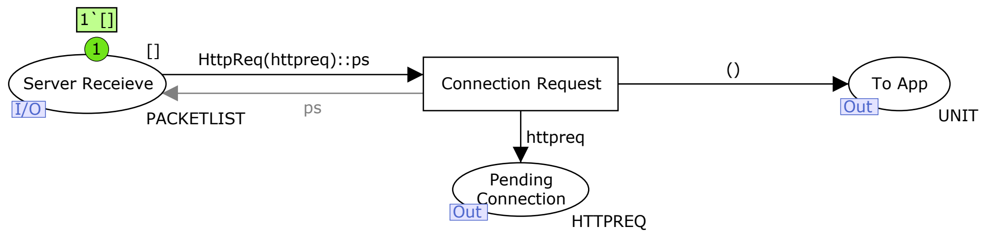


Figure 2.13: Get Connection Request

states.

Get Connection Request

This is a very abstract representation of how connection requests are received. The library simply sends a ConnRequest token to the app. In a real world app, much more info might have been sent, for example IP address and possibly authentication info, but for this model it is enough to show that some info is being sent, while the details are abstracted away.

Send Connection Response

When the Server Application has decided what to do with an incoming connection, it will send a `CONN_REPLY` to the library. If the answer is `accept`, we create a `CONN_OPEN` token in the Active Connection place and send a HTTP response back to the client, properly formatted according to the rules of the WebSocket Protocol. This involves generating a Sec-WebSocket-Accept header, which is done with the generateAccept() function:

Listing 2.12: generateAccept

```
fun B64 str =
  "B64("^str^")";

fun SHA1 str =
  "SHA1("^str^")";

fun generateAccept str =
  B64(SHA1(str^uuid));
```

The two functions B64() [?] and SHA1() [?] are abscract versions of the Base64 encoding algorithm and the SHA1 hasing algorithm. We felt it was unneccessary to actually implement these for the purpose of this model, and

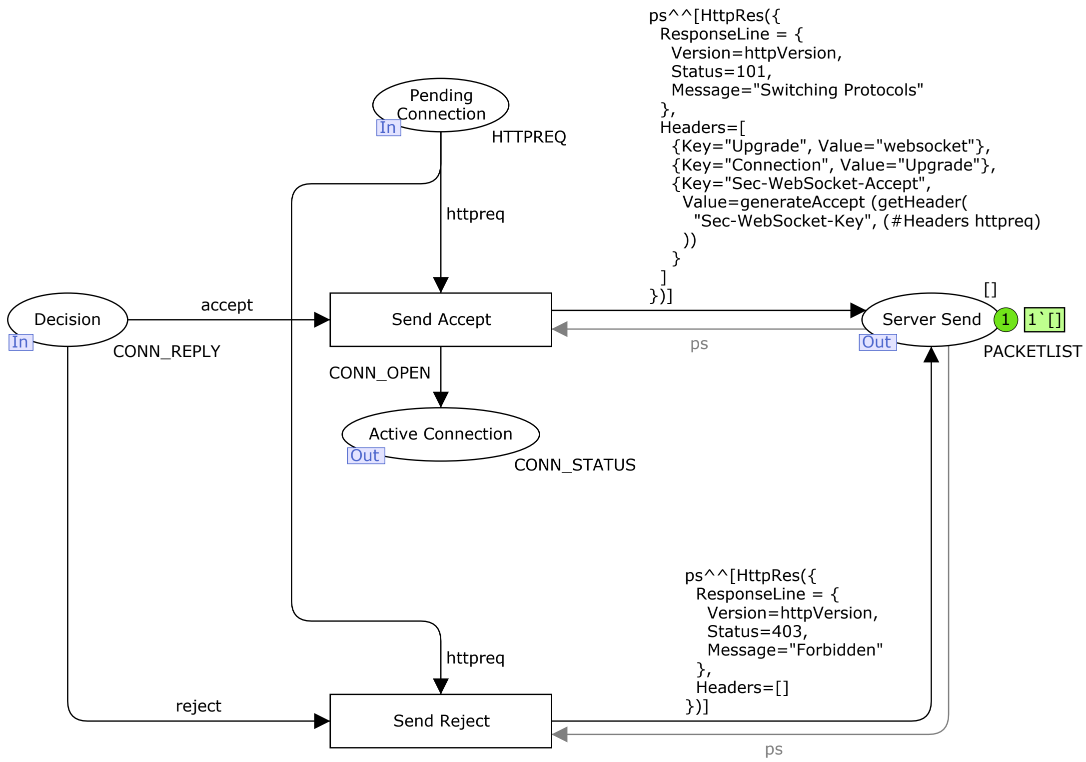


Figure 2.14: Send Connection Response

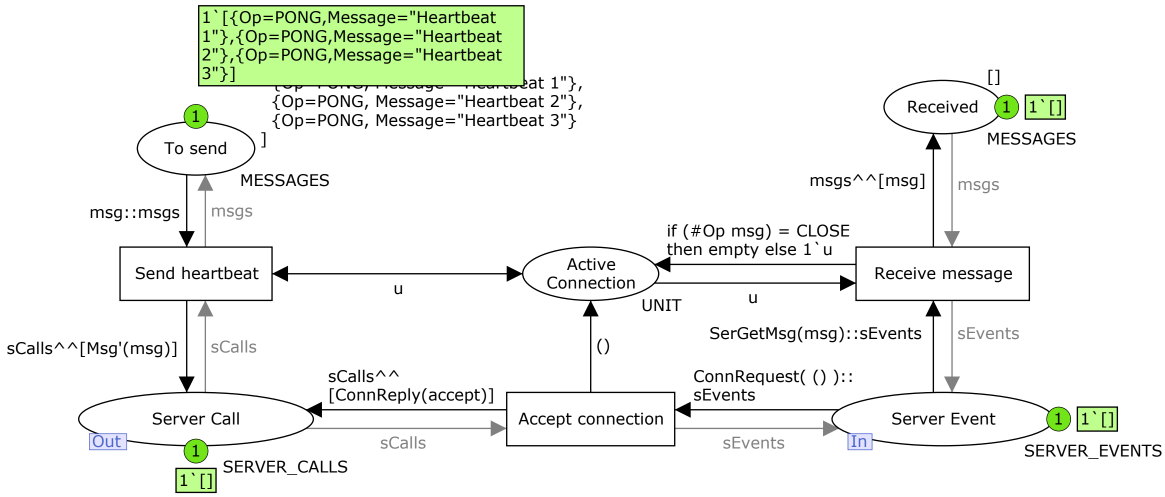


Figure 2.15: The Server Application Module

instead decided to simply wrap the string argument to show that it had been encoded or hashed.

2.3.6 Server Application

The Server Application has three tasks: Accept or reject incoming connections, and send and retrieve data. The colourset `MSGS_OR_CONN_REPLY` that was defined in Listing 2.1 is seen here. It has three queued messages, which illustrate the capability of PONG frames to be used as a heartbeat (without PING being involved). A real world application would have more logic here, but the interface to the library would be the same.

Chapter 3

State Space Analysis of the CPN Web Socket Protocol

One of the advantages of Coloured Petri Nets is the ability to conduct state space analysis, which can be used to obtain information about the behavioural properties of a CPN model, and which can be used to locate errors and increase confidence in the correctness of the CPN model.

3.1 State Spaces

A state space is a directed graph where each node represents a reachable marking (a state) and each arc represents an occurring binding element (a transition firing with values bound to the variables of the transition). CPN Tools by default generates the state space in breadth-first order.

TODO: Figur av SS initiell modell, med forklaring

Once generated, the state space can be visualised directly in CPN Tools. Starting with the node for the initial state, one can pick a node and show all nodes that are reachable from it, and in this way explore the state space manually. This can be very tedious and unmanageable for complex state spaces, though, and instead it is usually better to use queries to automate the analysis based on state spaces.

3.1.1 Strongly Connected Component graph

In graph theory, a strongly connected component (SCC) of a graph is a maximal subgraph where all nodes are reachable from each other. An SCC graph has a node for each SCC of the graph, connected by arcs determined by the arcs in the underlying graph between nodes that belong to different SCCs. An SCC graph is acyclic, and an SCC is said to be trivial if it consists of only one node from the underlying graph.

By calculating the SCC graph of the state space, some of the further analysis becomes simpler and faster, such as determining reachability, cyclic behaviour, and checking so-called home and liveness properties.

3.1.2 Application of State Spaces

The biggest drawback of state space analysis is the size a state spaces may become very large. The number of nodes and arcs often grows exponentially with the size of the model configuration. This is also known as the state explosion problem.

This can be remedied by picking smaller configurations that encapsulate different parts of the system. This was necessary with the WebSocket Protocol model, as the complete state space took too long to generate with our original configuration. We started by removing all messages to be sent. This means the only thing that should happen during simulation is the opening handshake. This configuration is used to explain the State Space Report in the next section. After this, we gradually added different types of messages to the client and/or server applications. These configurations will be discussed at the end of the chapter.

illustrasjon

Another aspect that must be considered prior to state space analysis is situations where an unlimited number of tokens can be generated, thus making the state space infinite. This can be remedied by modifying the model to limit the number of simultaneous tokens in the offending place.

Additionally, a model that incorporates random values is not always suited for computing a state space. The generated state space depends on the random values chosen, so the state space generator needs to be able to deterministically bind values to arc expression variables.

For small color sets (generally defined as discrete sets usually with less than 100 possible values), binding of random values in arc expressions can occur in two ways:

1. By calling ran() on the colorset. The ran() function picks a value ranging over the color set, but since is non-deterministic, it isn't suited for state space generation.

2. By using a free variable ranging over a color set in the arc expression. A free variable is a variable that does not get assigned a value in an expression. It will bind to a value picked at random from the color

set during simulation just like the ran() function, but also lets the state space generator pick each of the possible bindings from the values available in the colorset, and thus generate all possible successive states.

For arc expressions that use type 1, it is usually possible to change or adapt it into type 2.

Color sets that use values from a large or unbounded range, or from continuous ranges like floating point numbers, are considered large color sets, and using random values from such color sets can make it impossible (or impractical) to generate a complete state space. It can be worked around by instead using small color sets as described above. The CPN Tools manual has examples on how to do this.

If these issues are not taken into account, a complete state space can not be achieved, since it's impossible for the state space genrator to make sure all possible values have been considered, and occurrence sequences might diverge if the same occurrence can happen in different orders but with different random values.

For the WebSocket Protocol model, this was a problem for the masking key in WebSocket frames, which is supposed to be a random 4-byte string, giving 2^{32} or almost 4.3 billion possible values. To generate state spaces for this model, the randomisation function used was simply changed to always return four zeros. This is a reasonable abstraction since the specific value of the masking key does not affect the operation of the protocol. The result is shown in Listing 3.1, with the old code commented out.

Listing 3.1: Fixed masking key

```
fun randMask() = Mask([
 Byte(0),Byte(0),Byte(0),Byte(0)
 (* BYTE.ran(), BYTE.ran(), BYTE.ran(), BYTE.ran() *)
]);
```

3.2 State Space Report

Once a partial or complete state space has been generated, CPN Tools lets the user save a state space report as a textual document. The report is organised into parts that each describe different behavioural properties of the state space.

To explain each section of the state space report, a simple report for the WebSocket protocol has been generated, in a configuration where no messages are set to be sent. Thus, the only thing that will happen is that a connection will be established. Later in the chapter we will consider more elaborate configurations of the WebSocket protocol.

3.2.1 Statistics

The first section of the report describes general statistics about the state space.

```
State Space
   Nodes:  17
   Arcs:   16
   Secs:   0
   Status: Full

Scc Graph
   Nodes:  17
   Arcs:   16
   Secs:   0
```

This state space has 17 possible markings, with 16 enabled transition occurrences connecting them. There is one more node than there are arcs, which means this graph is a tree.

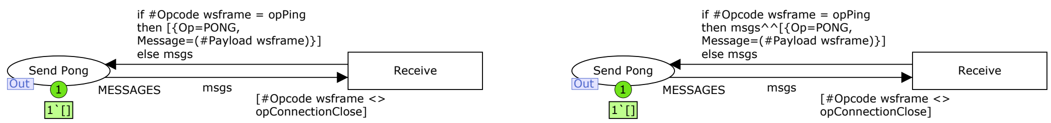
The `Secs` field shows that it took less than one second to calculate this state space, while the `Status` field tells whether the report is generated from a partial or full state space. In this case the state space is fully generated.

We also see that the SCC Graph has the same number of nodes and arcs, meaning that there are no cycles in the state space (although this was already known from the fact that it is a tree).

3.2.2 Boundedness Properties

The second section describes the minimum and maximum number of tokens for each place in the model, as well as the actual tokens these places can have. The text has been reformatted and truncated (indicated by [...]) for readability.

```
Best Integer Bounds
                           Upper      Lower
   ClientApplication
        Active_Connection
                           1          0
        Conn_Result
                           1          0
        Connection_failed
```



(a) Before (b) After

Figure 3.1: Fixing Pong reply

```
                            0           0
     Messages_received
                            1           1
     Messages_to_be_sent
                            0           0
[...]
```

Many places show a lower and upper bound of 1. This shows a weakness in the approach of using lists to facilitate ordered processing of tokens: We cannot see the actual number of tokens that are in the place, because technically there is just a list there. However, it quickly lets us know if something is wrong as well, since any values other than 0 or 1 here indicate a problem.

In fact, an error in the model was discovered this way, in the Unwrap and Receive module, where the pong reply was creating a new list instead of appending to the old one in outgoing messages. This caused the Client Outgoing Messages place to have 2 tokens at once. Fig. 3.1 shows the location of the error before and after fixing it.

```
Best Upper Multi-set Bounds
   ClientApplication
        Active_Connection
                      1`()
        Conn_Result
                      1`success
        Connection_failed
                      empty
        Messages_received
                      1`[]
        Messages_to_be_sent
                      empty
   [...]
   ClientWebSocket
        Connection_status
                      1`CONN_OPEN
   [...]
   ServerWebSocket
```

```
          Connection_Status
                         1`CONN_OPEN
     [...]

  Best Lower Multi-set Bounds
     ClientApplication
          Active_Connection
                         empty
          Conn_Result
                         empty
          Connection_failed
                         empty
          Messages_received
                         1`[]
          Messages_to_be_sent
                         empty
     [...]
     ClientWebSocket
          Connection_status
                         empty
     [...]
     ServerWebSocket
          Connection_Status
                         empty
     [...]
```

Apart from that, we see that both the client and the server has an open connection at some point, as the `Connection_status` place in the `ClientWebSocket` and `ServerWebSocket` modules have both had a `CONN_OPEN` token.

3.2.3 Home Properties

This section shows all home markings. A home marking is a marking that can always be reached no matter where we are in the state space.

```
  Home Markings
     [17]
```

We see that there is one such marking defined by node 17. From earlier we know that the state space is a tree, and if this node is always reachable it must be a leaf and all the other nodes must be in a chain. This tells us that there is only one possible sequence of transitions to establish a connection. We can then confidently say that the model works correctly with this configuration.

3.2.4 Liveness Properties

This section describes liveliness of the state space. Some of the transitions have been omitted for readability.

```
Dead Markings
   [17]


Dead Transition Instances
   ClientApplication'Fail 1
   ClientApplication'Receive_data 1
   ClientApplication'Send_data 1
   ClientWebSocket'Filter_messages 1
   .....

Live Transition Instances
   None
```

A dead marking is a marking from where no other markings can be reached. In other words, there are no transitions for which there are enabled bindings, and the system is effectivley stopped. For our example, we have a single dead marking, and it is the same as our home marking, confirming that this is a leaf node in the tree.

from where? for which? formuler

We also get a listing of dead transition instances, which are transitions that never have any enabled bindings in a reachable marking and are thus never fired. This can be useful to detect problems with a model, but in this example it is expected for many of the transitions, since we are not sending any kind of messages in the configuration considered.

Last, there are live transition instances. A transition is live if we from any reachable marking can find an occurrence sequence containing the transition. Our example has no such transition, which follows trivially from the fact that there is a dead marking.

The state space report also contains fairness properties, but this does not apply to our model since it contains no cycles. We will not go into detail about this, and instead refer to [?] chapter 7 for more information.

3.3 TODO:

Skrive om resten av analysene.

Chapter 4

Technology and Foundations

One of the first decisions that had to be made for this thesis was whether to base the work on some existing platform or to create a new one from scratch. In this chapter we will describe the reasoning behind the design choices we made, and give an overview of the technologies that have been used.

4.1 Representing CPN Models

A central design decision is how to represent the CPN models. A simple but easy way of manipulating a CPN model is by representing it as a tree, with pages as nodes, and places, transitions and arcs as child nodes with properties describing how they connect in the actual CPN model. Simple tree editors are a feature of most GUI software platforms. Even so, we realised early that writing everything from scratch would take much longer than adapting an existing platform.

There are of course many complete implementations of Petri Net tools in different languages and toolkits, but few of them are open source, or written with extensibility in mind. If we were to base our work on an existing platform, it would have to be open and extendable.

To narrow our search, we limited our options to solutions in languages we had experience with: Java, c++/Qt and Ruby. Java is a popular language, and we already have Access/CPN, a part of the CPN Tools project, which can parse .cpn files and represent the model as Java objects.

The ePNK framework, an extendable framework for working with Petri Nets in a graphical manner, and that makes it possible to specify your own Petri Net type. It is built on the Eclipse Modeling Framework (EMF) (which

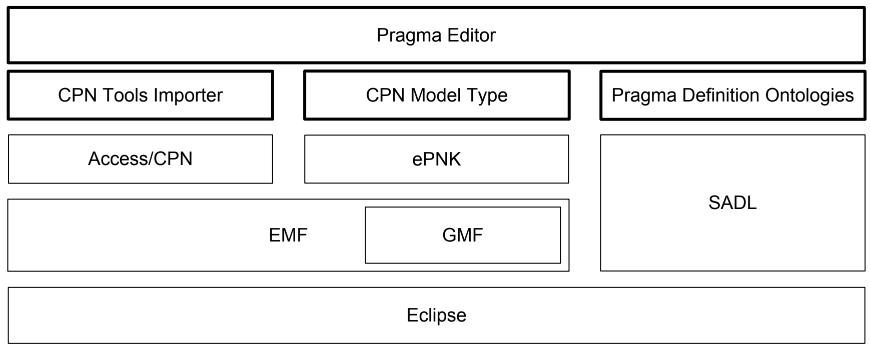


Figure 4.1: Application Overview Diagram

Access/CPN is also built on).

We also needed a way to represent pragmatics. It was suggested to try an ontology-based approach, and we selected Semantic Application Design Language (SADL), another Eclipse plugin that lets us easily define and work with ontologies.

Fig. 4.1 shows the different elements that make up the application. The elements with bold frames are the ones newly created for this thesis, while the rest below are the existing solutions used and built upon. These will be described in the following sections, from the bottom up.

4.2 Eclipse IDE

Eclipse IDE [?] is an open source, cross-platform, polyglot development environment. Its plugin framework makes it highly extendable and customisable, and especially makes it easy for developers to quickly create anything from small custom macros, to advanced editors, to whole applications. The Eclipse IDE is open source, and part of the Eclipse Project, a community for incubating and developing open source projects.

The Eclipse IDE is built on the Eclipse Rich Client Platform (RCP) shown in Fig. 4.2. At the bottom of this we have the Platform Runtime, based on the OSGi framework, which provides the plugin architecture.

The Standard Widget Toolkit (SWT) gives efficient and portable access to the user-interface facilities of the operating systems on which it is implemented. JFace is a User Interface framework built on SWT. The Workbench

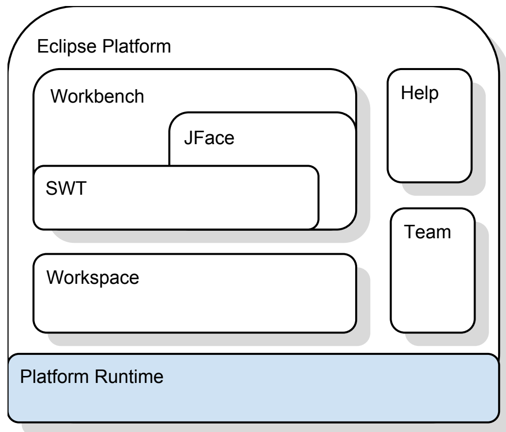


Figure 4.2: The Eclipse RCP

uses these two frameworks to provide a scalable multi-window environment.

The Workspace defines API for creating and managing resources (projects, files, and folders) that are produced by tools and kept in the file system.

The Team plugin is a foundation for collaboration and versioning systems. It unifies many operations that are common between version control systems.

The Help plugin is a web-app-based help system that supports dynamic content.

There are other utilities as well, like search tools, build configuration, and the update manager which keeps plugins up to date as well as handles installation of new plugins.

Together these plugins form a basic generic IDE. Other plugins build on this to specialise the environment for a programming language and/or type of application.

Plugins are the building blocks of Eclipse, and there exists a wide range of plugins that add tools, functionality and services. For example, this thesis was written in LaTeX using the Texlipse plugin, and managed with the Git version control system through the EGit plugin.

It is possible to package Eclipse with sets of plugins to form custom distributions of Eclipse that are tailored for specific environments and programming languages. The principal Eclipse distribution is the Eclipse Java IDE, which is one of the most popular tools for developing Java applications, from small desktop applications, to mobile apps for Android, to web applications, to enterprise-scale solutions. Another examples is Aptana Studio, aimed at Ruby on Rails and PHP development.

Publishing a custom plugin is simple. By packaging it and serving it on a regular web server, anyone can add the web server URL to the update manager in Eclipse, and it will let you download and install it directly, as well as enabling update notifications.

4.3 Eclipse Modeling Framework (EMF)

EMF is a framework for Model Driven Development (MDD) in Java. It is an Eclipse plugin that is part of the Eclipse Platform, and is open source. The principle of MDD is to define model structure by creating a metamodel to specify which entities can be created and how they relate to each other. By providing modeling and code generation tools, EMF allows developers to create model specifications (metamodels) that can be used to generate code for Java classes, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. These capabilities make EMF ideal for obtaining a simple editor that can be used to manipulate CPN models.

EKSEMPEL

4.3.1 Graphical Modeling Framework (GMF)

GMF builds on EMF to provide graphical viewing and editing of models. It uses metamodels created with EMF to generate implementations of views and editors that can create and edit the respective models. This can be used to create an editor that looks and works similarly to CPN Tools, and also lets you annotate it with pragmatics.

Uklart, bør skrives på ny, med eksempel

4.4 ePNK: Petri Net modeling framework

ePNK is an Eclipse plugin both for working with standard Petri Net models, and a platform for creating new tools for specialised Petri Net types, which is exactly what we need for our annotated Petri Net type. It uses EMF and GMF to work with the Petri Net models and provide generic editors for custom Petri Net variants.

There are several reasons why ePNK is a good choice as a foundation for the prototype:

- It saves models using the ISO/IEC 15909 [?] standard file format Petri Net Markup Language (PNML),

- It is currently actively developed,
- It is designed to be generic and easily extendable by creating new model types, and
- It includes both a tree editor and a graphical editor, provided through GMF.

4.4.1 ePNK Model Type Definitions

To simplify this section, we will first define some terminology:

Plugin - An Eclipse Plugin

Type - An ePNK Model Type Definition

Type Model - The EMF model defining a Type. In MDD terms: The metamodel.

Finne på bedre navn så det ikke er så lett å blande disse

Model - A model created by a user

A standard Petri Net model created in ePNK is initially only defined by the PNML Core Model Type. This Type is intended to be as generic as possible, and only defines the basic classes that most Petri Net variants contain, like Pages, Places, Transitions and Arcs. The only constraint defined is that an arc must go between two nodes on the same page.

The user can specialise a model to extend it with features of a mode advanced Petri Net type. This is done by adding a Type as a child of the Petri Net Document node in the model, available in the right-click menu. Only one Type can exist in a Petri Net.

Once a Type is added, ePNK will use class reflection to dynamically load any associated plugin(s), and the menus for adding new objects to the model will include any new classes and functionality that the Type defines.

In addition to the PNML Core Model Type, ePNK includes definitions for two subtypes of Petri Nets. The first is P/T-Nets (Place/Transition Nets), which expand on the core model with a few key items: initial markings for places as integers, inscriptions on arcs, and constraining arcs to only go between a place and a transition.

The second type included with ePNK is High level Petri Nets (HLPNG). This type adds Structured Labels which are used to represent model declarations, initial markings, arc expressions and transition guards. These are parsed and validated using a syntax that is inspired from (but not the same as or compatible with) CPN ML from CPN Tools. It is possible to write invalid data in these labels and still save the document, as they will only be marked as invalid by the editor to inform the user.

Neither of these two types conform exactly to the Coloured Petri Nets created by CPN Tools. HLPNG comes close, but is missing a few things like ports and sockets (RefPlaces can emulate this), and substitution transitions. Also, the structured labels are not compatible with CPN ML syntax from CPN Tools, and for our prototype, these structured labels are not necessary with regard to annotations. They might be useful in a future version, where for example pragmatics are available depending on things like the colorset of a place or the variables on an arc, but initially this is considered to be out of the scope of this thesis.

Our decision was therefore to develop our own Petri Net type that matches the type supported by CPN Tools.

4.5 Access/CPN: Java interface for CPN Tools

CPN Tools has a sister project called Access/CPN. This is an EMF-based tool to parse .cpn files and represent them as an EMF-model. The .cpn files saved by CPN Tools are XML-based, which makes them easy to parse, but having an existing solution for this is preferrable.

The model definition used by Access/CPN is very similar to that of ePNK. This will be discussed more in the next chapter.

4.6 Ontologies: OWL 2 and OWL API

Ontologies are a way to present information and meta-information so that it can be understood by computers. Essentially, this is done by defining classes that have properties, relations and constraints, and then present information with these classes.

There is a lot of ongoing research on this subject, especially to create a semantic web, that is extending web pages to provide meta-information about the content they contain and enabling software to understand it and reason about it. The OWL 2 Web Ontology Language [?] is the World Wide Web Consortium (W3C) recommended standard for representing ontologies. The primary exchange syntax for OWL 2 is RDF/XML [?]. There also exist other syntaxes, like Manchester syntax which improves readability,and Functional syntax which emphasises formal structure.

The power of ontologies lies in the potential to reason about the facts they present, and infer implicit facts.

TODO: Lite eksempel på ontology.

The code generation pragmatics are defined as OWL 2 ontologies, primarily using Functional syntax. To parse and reason with these ontologies, we use OWL API [?], the reference Java implementation, which is capable of reading ontologies in any of the OWL 2 syntaxes.

4.7 Summary

After picking these technologies, since all the componets have Eclipse in common, it was an easy decision to develop our project as an Eclipse plugin. This also let us centralise all our development in Eclipse.

Chapter 5

Analysis and Design

5.1 Requirements

Before we describe the Eclipse plugin that has been developed, we discuss the requirements. The requirements can be divided into four main classes:

Finne på navn på plugin. ePNK CPN Pragma Plugin?

Mer detaljer

- Importing models created in CPN Tools.
- Annotate model with pragmatics.
- Load sets of domain-specific pragmatics to make available for the model.
- Define set of model-specific pragmatics while annotating the model

5.2 Defining Pragmatics

TODO: Detaljert info om pragmatikker og hva de er

A Pragmatic is an annotation on a model element that describes how it should be translated into code. Each pragmatic has restrictions on which model elements it can annotate.

Pragmatics come in three types: General, domain-specific and model-specific.

In the context of network protocols, general pragmatics include concepts like channels, principals, and operations like opening or closing a connection, and sending or receiving data.

Domain-specific pragmatics are pragmatics that apply to all (or many) protocols within a potential domain. An example is security protocols, where

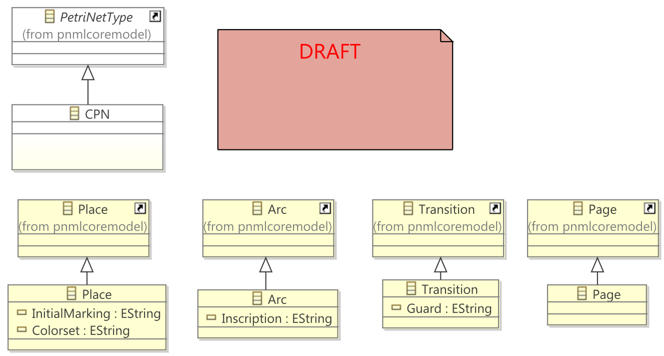


Figure 5.1: CPN model type diagram

examples of domain specific praagmatics relate to operations such as encryption, decryption, and nonce generation.

Model-specific pragmatics apply only to the model in which they are defined. Examples in the context of the WebSocket Protocol are masking and unmasking of frames, and integrity checking of received frames.

5.3 The CPN Ontology with Pragmatics

As mentioned in section 4.6, Pragmatics are defined and modeled as ontologies, using OWL 2 Web Ontology Language.

There exists two ontologies that function as a base for pragmatics: One that defines Colored Petri Nets, and one that defines base classes for pragmatics

TODO: Beskrive ontologiene

5.4 The Annotated CPN model type for ePNK

While designing the Type model for the Plugin, it was decided to separate it into two parts: One to define CPN Type, and one for defining Annotated CPN Type, extending from the first. This also adds the benefit that the pure CPN Type can be used for other applications.

Oppdater figurer!

A custom Petri Net Type is made by first creating an Eclipse Plugin project. In this project, an EMF model should be created. This is the Type Model, and should inherit the PNML Core Model from ePNK, or any other model that already does this (such as the P/T-Net or HLPNG Types).

Glossary for P/T-Net and HLPNG

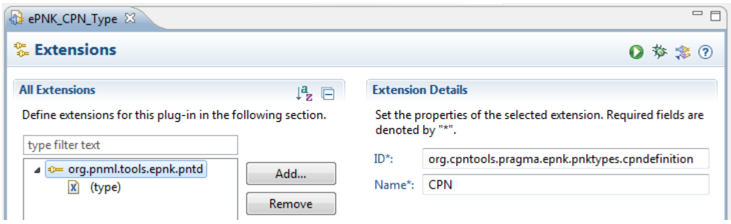


Figure 5.2: Plugin Manifest part 1

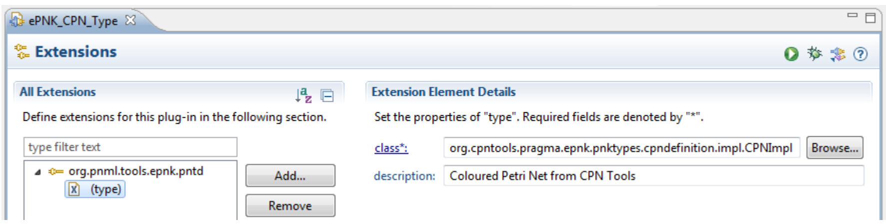


Figure 5.3: Plugin Manifest part 2

The CPN model, shown in Fig. 5.1, defines the structure and constraints of Colored Petri Nets. The first thing this new Model Type should define is a subclass of the PetriNetType class with the name of the new Model Type, which in our case is CPN. This can be seen in the top left corner of the diagram in Fig. 5.1. This class is what identifies the Type, and is what will appear in the menu to let a user extend a model with the new Type.

add fig?

After creating this model, EMF can generate source code for interfaces and implementations of the new class. This is done by creating a "genfile" linked to the EMF model. The genfile can define metainfo such as the base package of generated source files, and configuration parameters for the individual classes. EMF can then generate different groups of code, but for a ePNK Type we only need Model code and Edit code. the Model code includes interfaces and corresponding basic implementations of the classes as well as factories for instantiating them, while the Edit code contains classes for presenting the Model code in an editor.

After generating the code for the Type Model, the source file for the implementation of the PetriNetType subclass needs two minor modifications to work with ePNK: The constructor must be made public (it is protected by default), and the toString method must be implemented to conform to the PetriNetType interface. This method should return a string that textually represents the net type, usually simply its formal name.

Before ePNK will recognise the plugin and the Type model, the plugin

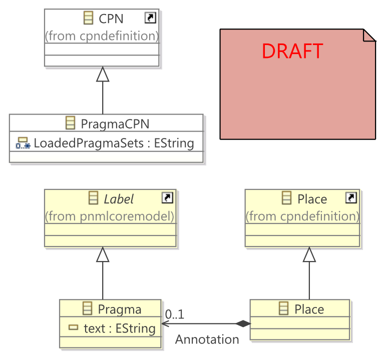


Figure 5.4: Annotated CPN model type diagram

manifest needs to be edited to define this plugin as an extension to the org.pnml.tools.epnk.pntd extension point of ePNK. All that is needed to configure this is supplyting a unique id, a descriptive name, and the fully qualified classpath to th ePetriNetType subclass. Fig. 5.2 and Fig. 5.3 shows the finished configuration for the CPN model type.

Kan bytte ut figurene med XML-data istedet

EMF does not support merging of models, meaning it is not possible to define new properties or relations directly on the original classes of the Core Model. Thus, in order to change the functionality of existing classes such as Place, Arc and Page, they have to be subclassed. ePNK will use reflection to check a Type Model for subclasses with the same name as classes in the Core Type, and load these dynamically instead of the base classes. This can be seen in Fig. 5.1, where Place, Arc, Transition and Page are all subclassed from the classes referenced from the Core Model, with added attributes needed to represent Coloured Petri Nets.

TODO: Constraints

The Annotated CPN model, shown in Fig. 5.4, extends the CPN model to enable annotation of model elements. It also handles saving and loading of ontologies that define sets of pragmatics.

TODO: Intro Pragma Net Type

These subclasses then have references to the Pragma class. The references are configured to act as containment, meaning Pragma instances are created as children of the related classes. TODO: Smooth overgang. As we can see in Fig. 5.5, Pragma objects are now available as children to Places.

The Pragma class inherits from Label, which lets it be represented visually as a text label in the graphical editor of ePNK.

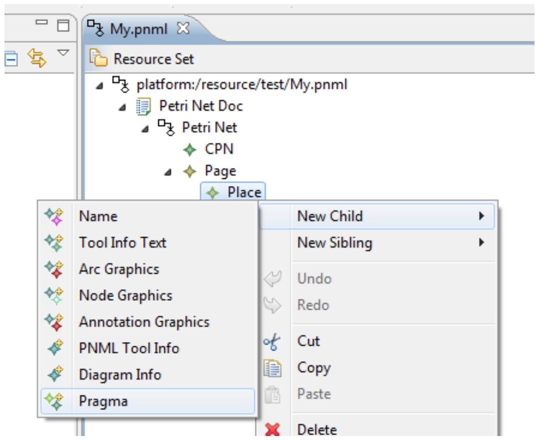


Figure 5.5: Adding Pragma as child to Place

5.5 Importing from CPN Tools

Access/CPN is a framework that can parse CPN models saved by CPN Tools and represent the model with EMF classes. Access/CPN has many additional features related to the semantics of CPN, but only the model importer is of relevance for the work in this thesis.

Access/CPN also uses EMF to represent models internally. The EMF model for CPN models that Access/CPN defines uses many of the same class names as ePNK, which makes it tedious to write and read code working between the two frameworks due to the need to use fully qualified classpaths to avoid name collisions. Initially we planned to extract the parser source code from Access/CPN and rewriting it to use the new ePNK CPN Type classes. This plan was later discarded in favor of depending on the Access/CPN plugin, as Access/CPN has continually been improved during development of the Plugin , and is now also capable of parsing graphics data. And by depending on Access/CPN instead of writing our own parser, we can also benefit from further updates more easily.

Sett inn navn

The conversion is straightforward, the .cpn file is loaded with Access/CPN, and the resulting model is then converted object by object to the ePNK CPN type.

5.6 Creating annotations

Labels on nodes, arcs and inscriptions. Choose from list. Possibly write freehand with content assist. Validation, with problem markers (Eclipse feature).

5.7 Choosing Pragmatics Sets

Where to store? Model, Project, Plugin Model pros: Will need anyway for model-specific pragmatics Could be associated with net type as a property (like HLPNG does) or as a sub node somewhere Have URI string and version to check. cons: Keeping base pragmatics up to date a problem Namespace-based? Already required for ontology

5.7.1 Creating custom pragmatics

Dynamically supported in content assist If ontology-based, use SADL editor På sikt eget verktøy Hva kan det settes på, hvilke attributter har det. Oversette til ontologi

Chapter 6

Evaluation

6.1 Requirements

6.2 Test cases

Simple protocol (TCP)

Kao-chow authentication protocol

The Edge Router Discovery Protocol (ERDP) for mobile ad-hoc networks

The WebSocket Protocol

Hvordan skal bruk på test cases demonstreres? Grafisk

6.3 User Feedback