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 15, 2012

Supervisor
Lars Michael Kristensen

Abstract

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

One of the modeling languages that can be used in MDE is Coloured Petri Nets (CPN). It is a type of high-level Petri Net, suited 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 develop an approach 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 builds on the ePNK framework, which uses the Eclipse Modeling Framework (EMF) to provide an extensible platform for working with CPN models. The prototype is desgned as a plugin for Eclipse. It can import models created by CPN Tools and lets the user annotate the models with code generation pragmatics. The prototype has been 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 model-driven development 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 exploration, and annotation with code generation pramatics.

Contents

Chapter 1

Introduction

Software engineering is an increasingly complex discipline, with new and improved technology emerging at a rapid pace. There is no single answer on how to approach all problems, which has lead to the development of several software development paradigms. A motivation common to many of them is that software developers have always sought increasing levels of abstraction. Today's technology is at a level that potentially gives us means for automatically generating source code from conceptual domain models of applications, and substantial research is being conducted to create formal methods for unleashing 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 methods and tools developed around it (like Unified Process). However, models largely play a secondary role, performing as design tools and documentation.

Model Driven Engineering (MDE) has emerged as a new development methodology, putting the models in the center of the software development process. Developers design models that serve both as documentation and as a basis for implementation, and become a layer of abstraction over source code. This trend has been touted as a new programming paradigm, the same way object-oriented programming was when it was conceived.

MDE has two central concepts:

- Domain specific modeling languages (DSML), which are used to formalise application structure, behavior, and requirements of specific do-

mains, such as financial services, warehouse management, task scheduling, and protocol and communication software. A DSML relies on 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.

Mer

TODO

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

Using models as the core design element comes with several benefits. It allows developers to employ different methods of analysis of the models, like verifying correctness, completeness, finding race conditions, and analysing scalability. Models also act as graphical representations of the system, making them more understandable for people that are not programmers.

One of the central arguments for MDE is automatic source code generation. Several advantages come from this: Documentation and implementation are synchronised, boilerplate code and automatic testing is taken care of...

Mer

One modeling language that is being used as a DSML is Petri nets. Which kind of systems, concurrent. Enables verification and analysis, State Spaces.

Mer

There exist many extensions to the Petri Net formalism, termed high-level Petri Nets, that define additional constructs or that change or enhace concepts of Petri Nets. One of these extensions is called Coloured Petri Nets (CPN). The term Coloured comes from the fact that a token can have a colour from a defined colour set, essentially data values from a set of values. A common way of analysing Petri Nets is called state space exploration, and is a powerful mothod for automatic model verification and determination of several properties. This thesis gives a short introduction to CPN and state space exploration.

1.2 Related Work

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 develop approaches for and demonstrate how to use CPN and other Petri Net variants to model software and generate source code.

Kristensen and Westergaard [KW10] examine challenges of using CPN for automatic code generation, and propose a new Petri Net type called Process-Partitioned CPNs. They demonstrate and evaluate it by designing an implementation for the Dynamic MANET On-demand (DYMO) routing protocol.

Mortensen [Mor00] presented an extension to the Design/CPN tool to support automatic implementation of systems by reusing the model simulation algorithm, thus eliminating the usual manual implementation phase. They demonstrate the tool by implementing an access control system, and evaluate benefits of the model architecture.

Lassen and Tjell [LT07] present a method for developing Java applications from Coloured Control Flow Nets (CCFN), a specialised type of CPN. CCFN forces the modeler to describe the system in an imperative manner, making it easier to automatically map to Java code.

This thesis focuses on work done by Simonsen [FS11], who discusses some of the challenges in modelling and automatically generating software in the domain of communication protocols, using the Kao-Chow authentication protocol as an example. The ideas introduced in the paper outline a method for annotating CPNs with a set of code generation pragmatics that describe how model elements relate to and bind to source code.

Simonsen's 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 constructs for a particular platform,
- Create a configuration model for deciding implementation details, depending on the protocol model and platform model.

This research defines the domain of the work for this thesis.

1.3 Thesis Aims and Results

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 developing specialised tool support for this purpose.

This thesis seeks to satisfy this need by answering the following question: What is an approach for annotating Coloured Petri Nets with code generation pragmatics following the idea of?

The method used to answer this question is to create a prototype application framework, and evaluate it by annotating an example CPN model of the WebSocket Protocol.

The code generation pragmatics (or just "pragmatics") are categorised into three classes.

General pragmatics are used to define protocol entities, communication channels external method calls and API entry points for 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 particular domain. An example is security protocols, where examples of domain specific pragmatics relate to operations such as encryption, decryption, and nonce generation.

Model specific pragmatics apply only to the specific model instances in which they are defined, and are used to label concepts unique to that model.

The prototype builds on the ePNK framework, which uses the Eclipse Modeling Framework (EMF) to provide an extensible platform for working with CPN models. The prototype is desgned as a plugin for Eclipse, and can import models created by CPN Tools. It lets the user annotate the models with pragmatics through a tree editor. Pragmatics are defined using ontologies, and can be dynamically loaded into models. Evaluation is done by applying it to a set of protocol CPN models.

1.4 Thesis Organisation

The thesis is organised as follows:

Chapter 2: The WebSocket Protocol CPN Model Provides a description of the WebSocket protocol, the primary case study used in this thesis. An introduction to Coloured Petri Nets and the CPN Tools 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 WebSocket Protocol Gives an introduction to State Space Analysis of CPN models, and an example of how to apply it using the model of the WebSocket Protocol as an example. This establishes the produced model as correct.

Utvid litt, formuler bedre

Chapter 4: Technology and Foundations The produced prototype is built on top of a number of software frameworks and technologies. This chapter gives an introduction to these as well as the reasons for choosing them. The Eclipse Platform and its modules; the Eclipse Modeling Framework; the ePNK framework, which makes up the foundation of the prototype, and how it can be extended to support new functionality; Access/CPN, the engine used to omport models from CPN Tools; and an introduction to ontologies, the format used to specify 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 Coloured Petri Nets and Annotated Coloured 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 7: Conclusion Summary, personal experience, limitations and suggested focus of future work.

The reader is assumed to be familiar with Java programming, the basics of functional programming languages, and the TCP/IP Protocol Suite. Some basic knowledge of Petri Nets is also an advantage, but not a strict requirement as we briefly introduce the basic constructs of the CPN modeling language.

Chapter 2

The WebSocket Protocol CPN Model

2.1 The WebSocket Protocol

The primary case study for this thesis is the WebSocket protocol [FM11]. 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;

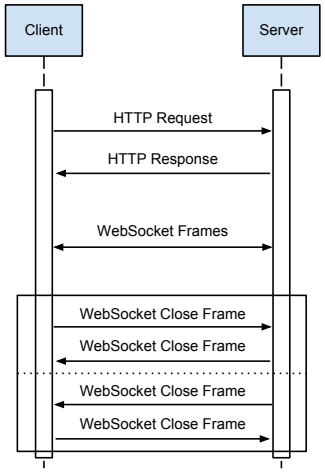


Figure 2.1: Sequence Diagram of the WebSocket protocol

- 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.

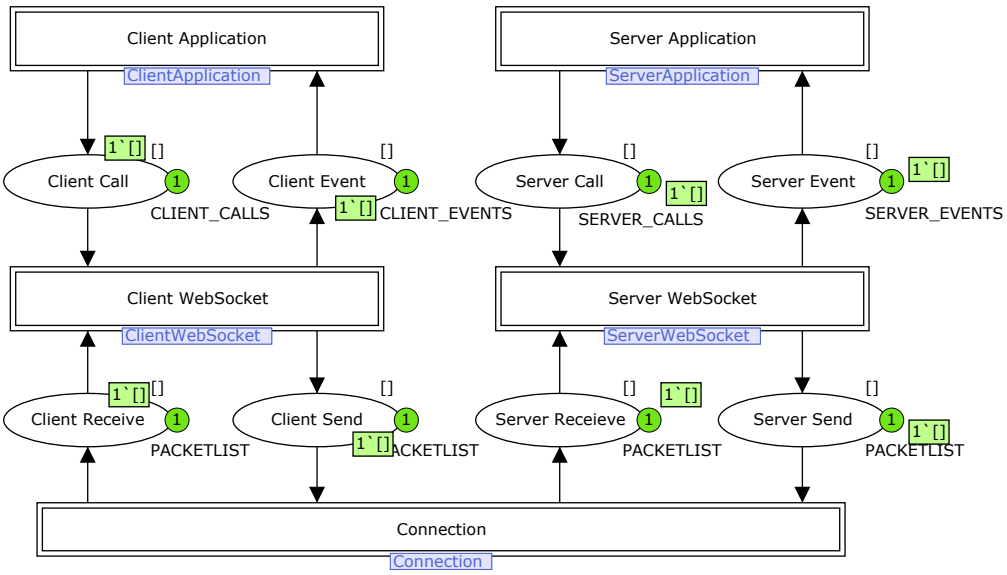


Figure 2.2: Overview of CPN model of the WebSocket protocol

2.2.1 CPN Tools

CPN Tools [RWL+03] is a popular graphical tool for working with CPN models, from construction and simulation, to analysis via state space exploration.

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 [Mil97], developed at Edinburgh University.

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.

A CPN Model is organised into pages, also called modules. Each module can also contain sub-modules, which can help keep complex and/or large models clear and manageable. The WebSocket model makes full use of this feature, and will be described in a top-down manner.

2.3.1 Overview

Fig. 2.2 shows the top-level Overview module of the WebSocket CPN model. It consists of five sub-modules represented by double border rectanges, and several places and arcs that connect them, represented with circles and arrows respectively. The function of arcs will be explained under the next submodule.

The sub-modules have been laid out to resemble part of the OSI model [Zim80], 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.

Places are used to represent the state of the modelled system, and can contain tokens. Each token can have a data value, termed the token colour. The number of tokens and their colors in a specific place is termed the marking of that place. Similarly, the tokens in all places in the model together form the marking of the model, and thus represents the state of the system.

All colour sets, variables, symbolic constants and functions have to be declared globally for the model. Colour sets are defined with the following syntax:

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

A colour set is defined as a range or set of data values, and is declared globally for the model using CPN ML. Colour set names are always capitalised in this thesis, but any CPN ML identifier is valid. A simple example is `colset INT = int;`, representing the set of all integers. A token from this colour set could for example have the value '3'. Similar declarations are present for `STRING`, `BOOL` and `UNIT`. These are examples of simple color sets.

It is also possible to delcare more complex color sets, termed compound color sets, to form data structures of data types. One such compound structure is the list, used to define an ordered collection of tokens from one color set.

Colour sets defined in the WebSocket model will be explained as they are encountered in each sub-module. The places in the Overview use the following colour set definitions:

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.

The `MESSAGE` colour set represents the essential message sent from and to the application layers. This colour set is defined as a record, which means a

Listing 2.1: Overview colour sets

```
colset OPERATION = with TEXT | BINARY | PING | PONG | CLOSE;
colset MESSAGE = record Op: OPERATION * Message: STRING;
colset MESSAGES = list MESSAGE;

colset URL = record Protocol: STRING * Host: STRING * Port: INT
    * Path: STRING;
colset CLIENT_CALL = union Connect:URL + CliSendMsg:MESSAGE;
colset CLIENT_CALLS = list CLIENT_CALL;

colset CONN_RESULT = bool with (fail, success);
colset CLIENT_EVENT = union CliGetMsg:MESSAGE + ConnResult:
   CONN_RESULT;
colset CLIENT_EVENTS = list CLIENT_EVENT;

colset CONN_REPLY = bool with (reject, accept);
colset SERVER_CALL = union SerSendMsg:MESSAGE + ConnReply:
   CONN_REPLY;
colset SERVER_CALLS = list SERVER_CALL;

colset SERVER_EVENT = union SerGetMsg:MESSAGE + ConnRequest:
   UNIT;
colset SERVER_EVENTS = list SERVER_EVENT;

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

tuple of elements that can be referred to by name. `MESSAGE` has two elements: the Op `OPERATION` and the Message `STRING`. In the WebSocket protocol, both data- and control-frames can have messages, although the message part of control-frames does not have to be shown to the user. 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.

To be able to connect to a server, its URL needs to be known, and this concept is defined by the `URL` color set. It consists of the Protocol (for example http), the Host (www.example.com), the Port (for example 80) and the Path (for example /home/index.html).

The Client Application can send tokens to the WebSocket layer as a `CLIENT_CALL`. This is a union color set, meaning a token can be one of the declared identifiers, optionally with an associated colour set. 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. `CLIENT_CALL` has two identifiers: `Connect`, used to request a connection to the associated `URL`, and `CliSendMsg`, signifying an outbound message from the client. And similar to `MESSAGES`, `CLIENT_CALLS` is a list of `CLIENT_CALL`.

When a connection attempt is completed, the result is represented from the `CONN_RESULT` color set, which is a simple rebranding of bool values to improve readability.

The WebSocket layer needs a way to notify the client application about connection results as well as received messages, which is done with the `CLIENT_EVENT` colour set. Like `CLIENT_CALL`, this is a union colour set which in this case can be either a `MESSAGE`, or a `CONN_RESULT`, and we similarly have a list version `CLIENT_EVENTS`.

The reason we use a single place to transmit both connection information and messages between application and WebSocket layers, is that this makes it easier to extend the model later if tokens from other color sets need to be passed between the modules. The benefit is that we will not have to create extra places for this, and simply need to add the new color sets to the releveant color set declaration (for example `CLIENT_CALL`).

Both the client and server WebSocket layers send and receive `PACKET` tokens, a union of three color sets that will be explained later. `PACKET`s are abstract and not fully modeled actual network packets, as this is not relevant to how WebSocket works. This color set is also used in a list `PACKETLIST`.

2.3.2 Client Application

This module (Fig. 2.3) is meant to serve as a generic invocation of the WebSocket protocol. It also shows the rest of the essential building blocks of a CPN model.

Transitions, represented with single border rectangles, are elements that represent potential state changes in the model. They are connected to places by directed arcs. These arcs can be inscribed with expressions containing variables. Variables are declared globally and can only be bound to tokens of the colour set they are defined for. A set of variables has been declared for the simple coloursets as follows:

Several variables for the same coloursets have been created for convenience in the cases where tokens of the same colour set 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 before the transition would be enabled.

A transition is said to be enabled when each of its incoming arcs is able

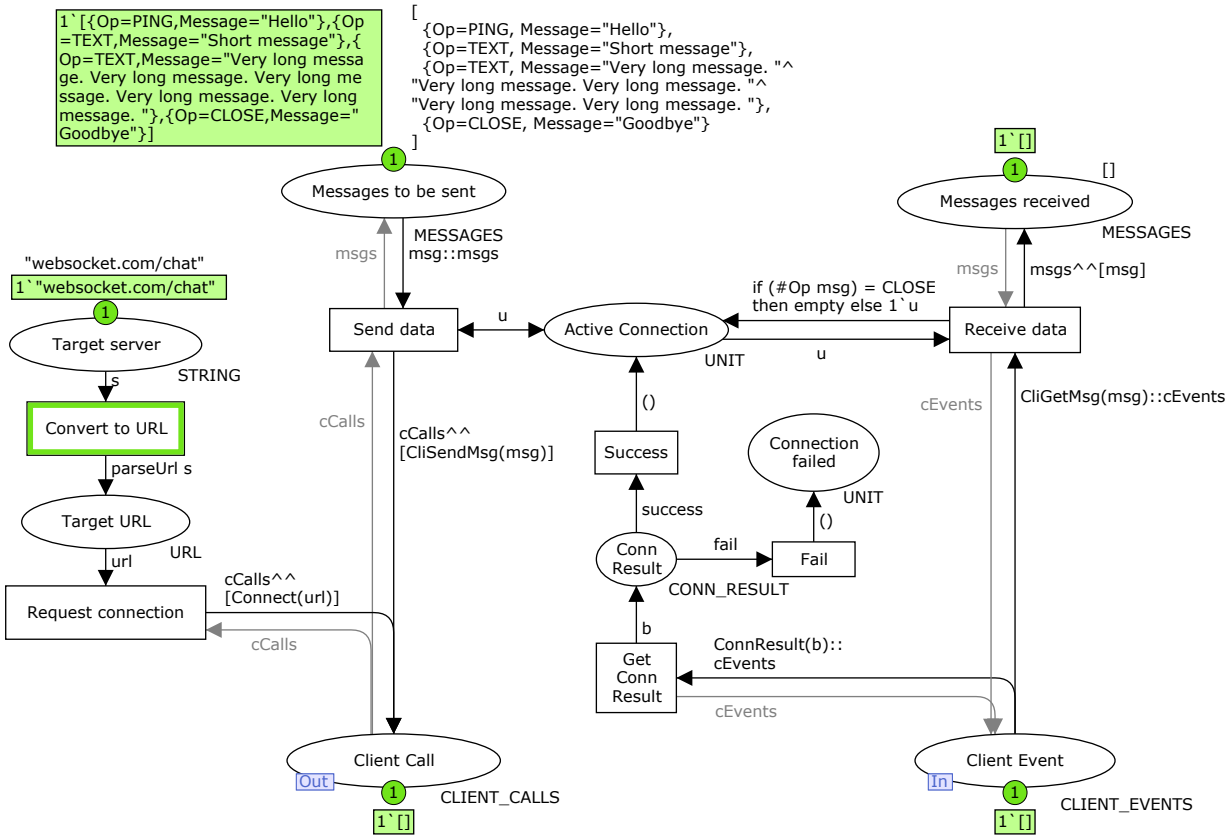


Figure 2.3: The Client Application

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;
```

to bind the variables in its expression to tokens in its corresponding place. As an example, the `Convert to URL` transition in Fig. 2.3 is enabled (signified with a green glowing border), because the incoming arc's expression is simply the variable `s`, which can be bound to the "websocket.com/chat" token in the `Target server` place.

When a transition occurs (or is fired), it will consume said tokens from the respective places. These variables can then be used to produce new tokens, according to the expression(s) inscribed on the output arc(s), which end up in the place(s)those arcs point to. To continue the earlier example, the string bound to s becomes an argument for the function `parseUrl` (explained further down), which produces an `URL` token that then ends up in the `Target URL` place.

Other 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. Listing 2.3 shows the variables used in this submodule.

Listing 2.3: Client Application Variables

```
var msg: MESSAGE;
var msgs, msgs2: MESSAGES;
var cEvents: CLIENT_EVENTS;
var cCalls: CLIENT_CALLS;
```

The function `parseUrl` converts a string into an `URL` token. Its implementation can be seen in Listing 2.4.

The places at the bottom, `Client Call` and `Client Event` represent the interface to the WebSocket layer, and are paired with the corresponding places on the Overview, which are also paired with corresponding places in the WebSocket Library. The tokens in these places will be mirrored between the sub-module and the super-module. The term for this is ports and sockets, where the port is in the sub-module and the socket is in the super-module. The ports are labeled with port-type tags (In and Out) to signify the direction that tokens move (although this has no technical

Listing 2.4: parseUrl and related functions

```
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;
```

significance in CPN Tools).

Queues

A lot of the places in this model rely on tokens being consumed in the same order they are produced, in other words, the places should behave like queues. However, CPN Tools does not have a mechanism specifically for this purpose. Instead, 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, and use operations defined for lists to access the first and last elements of the list.

To describe a list in CPN ML, 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.

Lists work similarly to other functional programming languages, defined as the head, which is the first element, and the tail, which is the list of all remaining elements. To access these elements, we use the `::` operator like this: `head::tail`. Lists are usually processed in a tail-recursive mannner. The `^^` operator is a convenience for concatenating two lists.

To model a queue with a list, 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. 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. In both cases this means there will be two arcs between the place and the transition: one to bind the target queue to a variable, and one that adds or removes an element. The two arcs can be misleading since information only logically moves one direction. To improve readability of the model, these queue operations have one arc colored gray, to emphasise the flow direction of information.

Examples of this is seen in Fig. 2.3 on the arcs connecting `Client Call` and `Client Event` places at the bottom, as well as the `Messages to be sent` and `Messages received` places at the top.

Program Flow

After the `Convert to URL` transition has occured, the `Request connection` transition should be enabled. If we were to fire this transition, it would consume the `URL` token from `Target URL`, bind the value of that token to the variable `url`

, create a Connect identifier (from the CLIENT_CALL union color set) containing url, and add it to the queue in the Client Call place.

Next, the Client Application waits for a CONN_RESULT token and place it in the Conn Result place. The expressions on the arcs going out from this place use the literal values success and fail instead of variables. If the token has the value succes, the Success transition is enabled, and it adds a UNIT token to the Active Connection place.

If and when the Active Connection place has a UNIT token, the Client Application can start sending and receiving messages. The arc between Active Connection and Send data is a two-way arc, which works like two arcs going in opposte directions with the same expression. The result is that the transition is only enabled when there is a token available, but the token will not be consumed when the transition occurs.

A sample of messages has been set as the intial value of the Messages to be sent place, to simulate an example execution of the program.

Finally, looking at the arc from Receive Data to Active Connection, 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.

2.3.3 The Client WebSocket Module

This module consists mostly of sub-modules. The only processing being done directly on this module is at the top, where messages and connection info is separated, and at the bottom, where masking of all websocket frames occurs, which the client is required to do by the protocol specification. The rest is plumbing between the submodules.

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

The declarations for HTTP requests and responses (Listing 2.5) are modeled after the the HTTP 1.1 standard. An HTTPREQ, or HTTP Request, consists of an initial REQUEST_LINE (following the standard format of an HTTP_VERB , a Path and a Version), and a number of HEADERS (key-value tuples). An HTTPRES, or HTTP Response, begins with a RESPONSE_LINE (consisting of the Version, the response Status, and a status Message), with a subsequent HEADERS list. A real response would usually also have a body, but this is not used in the WebSocket protocol, so there is no need to model it.

The declarations for WebSocket frames (Listing 2.6) have been modeled to approximate the actual memory structure of such frames. To this end, the colour sets BIT and BYTE have been created, where BIT is a relabeling of

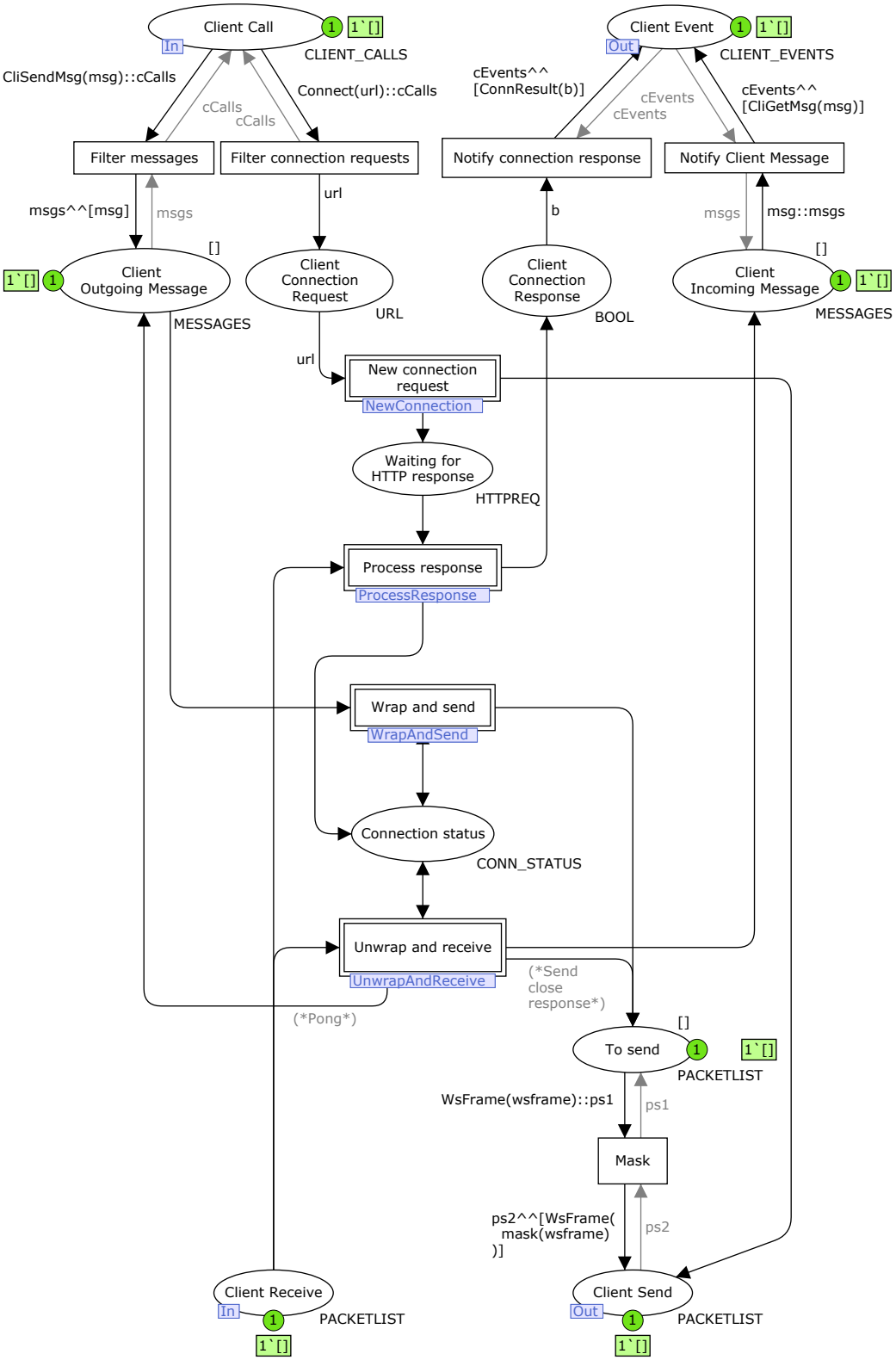


Figure 2.4: The Client WebSocket Module

Listing 2.5: HTTP colour sets

```
colset HTTP_VERB = with GET | POST | PUT | DELETE | HEAD;
colset REQUEST_LINE = record Verb: HTTP_VERB * Path: STRING *
    Version: STRING;
colset HEADER = record Key: STRING * Value: 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;
```

boolean values, and `BYTE` is an integer range. A `MASK` is a list with exactly 4 `BYTE`s. To model the optionality of masks, `MASKING` is either `Nomask` or `Mask` with an associated `MASK`.

Next, we declare a number of static values for convenience, which correspond to WebSocket frame operation identifiers, termed opcodes.

According to its specification, the `WSFRAME`, or WebSocket frame, consists of four control bits (only the first one is in use to mark final frames), an `Opcode` to describe the type of frame, a Masked bit to mark if the frame payload is masked, `Payload_length` to declare the number of bytes in the payload, an optional `Masking_key`, and the `Payload` itself.

Finally, we also have to declare the variables to be used in arc expressions (Listing 2.7).

Masking of frames is modeled to only set the masking bit and provide a masking key. An actual implementation would also apply the mask to the payload according to the specification, however this involves applying the XOR-operation on each octet in the payload with each octet in the masking key. XOR is not defined in CPN ML and would thus have to be modeled manually. Since the effects of masking the payload do not change the overall execution of the protocol, we opted to omit it completely, and let the presence of a masking key abstractly represent that the payload has been masked. The `mask` function is thus defined:

New connection

This module is fairly straightforward. We take an available `URL` and create an HTTP request, which is then queued to be sent by the client, as well

Listing 2.6: WebSocket colour sets

```
colset BIT= bool with (clear, set);
colset BYTE = int with 0x00..0xFF;
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;
```

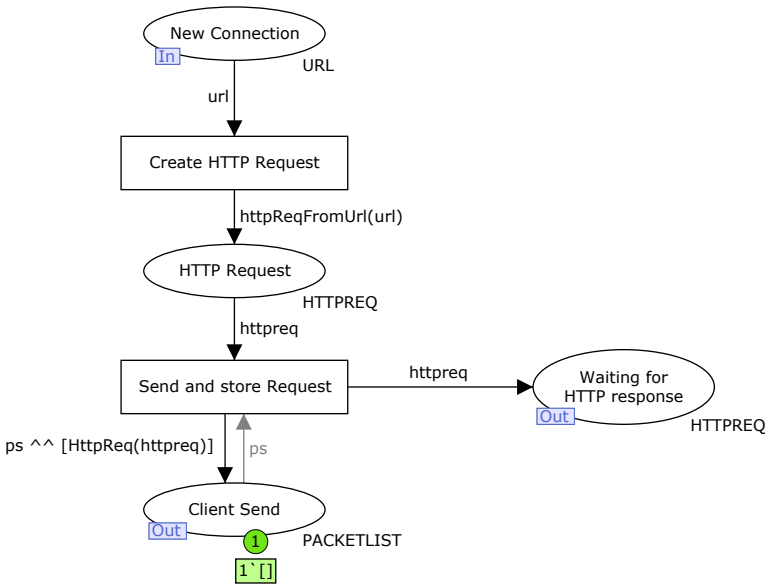


Figure 2.5: New Connection submodule

Listing 2.7: WebSocket Module Variables

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

Listing 2.8: Masking functions

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

fun mask (ws:WSFRAME) = let
 val ws1 = WSFRAME.set_Masked ws set
 val ws2 = WSFRAME.set_Masking_key ws1 (randMask())
in
 ws2
end;
```

as keeping a copy of the request for validation purposes when the response arrives.

The function httpReqFromUrl (Listing 2.9) takes a URL argument and uses it to produce a HTTPREQ token with headers according to the WebSocket protocol requirements. All the required headers are specified, but optional headers are not, as this is a generic implementation. The nonce is statically defined, partly to keep it simple and partly to more easily see it during simulation.

The two functions B64() and SHA1() are abscract versions of the Base64 encoding algorithm [Jos06] and the SHA1 hasing algorithm [iosat02]. We felt it was unneccessary to actually implement these for the purpose of this model, and instead decided to simply wrap the string argument to show that it had been encoded or hashed.

Process response

On this module, the transition enables when a `HTTPRES` token arrives from the server (and the `HTTPREQ` token from earlier is still waiting in its place). The transition has a guard inscription where the boolean variable b is bound to the result from the isResponseValid function (Listing 2.10), which checks if

Listing 2.9: httpReqFromUrl

```
val origin = "http://www.example.com";
val nonce = "nonce";
val uuid = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";

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

fun httpReqFromUrl (url:URL) =
        {
          RequestLine={
            Verb=GET,
            Path=(#Path url),
            Version=httpVersion
          },
          Headers=[
            {Key="Host", Value=(#Host url)},
            {Key="Upgrade", Value="websocket"},
            {Key="Connection", Value="Upgrade"},
            {Key="Sec-WebSocket-Key", Value=(B64 nonce)},
            {Key="Sec-WebSocket-Version", Value="13"},
            {Key="Origin", Value=origin}
          ]
};
```

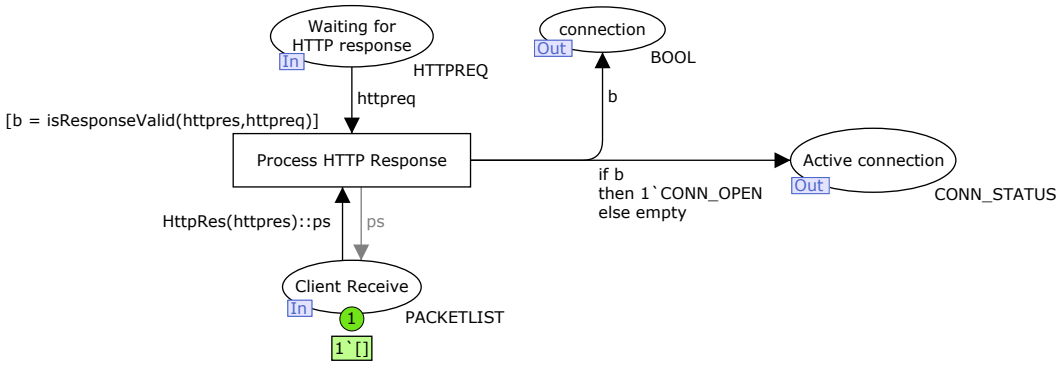


Figure 2.6: Process Response submodule

Listing 2.10: 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) = "Upgrade" andalso
  getHeader("Sec-WebSocket-Accept", headers) = accepttoken
end
```

the server's reply is valid and conforms to the WebSocket protocol specification.

Since b is a boolean variable, we can use it directly to notify the Client App through the `connection` place. If b is true, a `CONN_OPEN` token is put in the Active Connection place.

Wrap and send

This module (Fig. 2.7) takes new messages, wraps and optionally fragments them in the `Fragment and queue` submodule, 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 will also prevent sending of subsequent frames.

Fragment and queue This module is shown in Fig. 2.8. The `Sort control and data` uses the isData function (Listing 2.11) to filter the two types of messages.

Listing 2.11: isData

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

Control frames should never be fragmented and can thus be directly wrapped with the `wrapmsg` function. Data frames with long payloads should be fragmented. This is taken care of by the `fragment` function, which uses

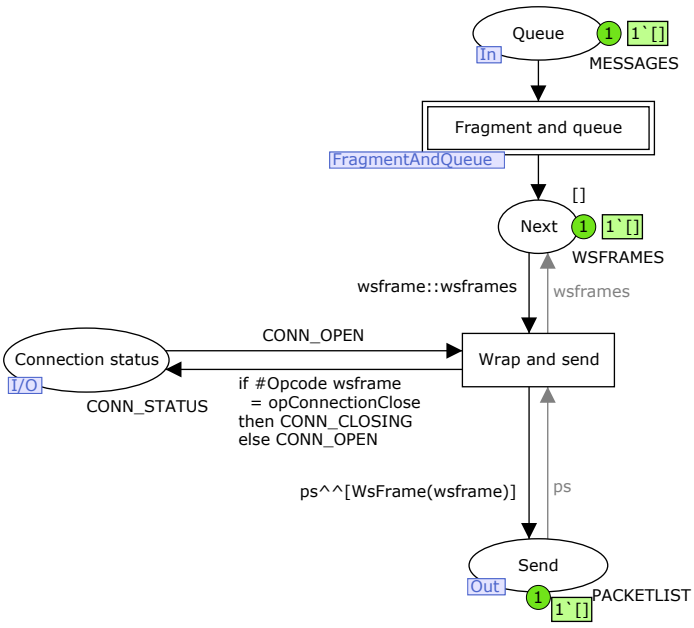


Figure 2.7: Wrap And Send submodule

an inner recursion to split the payload. Both of these functions employ the `wrap` function to produce `WSFRAME` tokens, and therefore have to be declared after it. These three functions are shown in Listing 2.12.

The WSFRAME tokens are then queued 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 presence 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

This module, shown in Fig. 2.9, handles reception of frames and extracting their payload into a message.

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

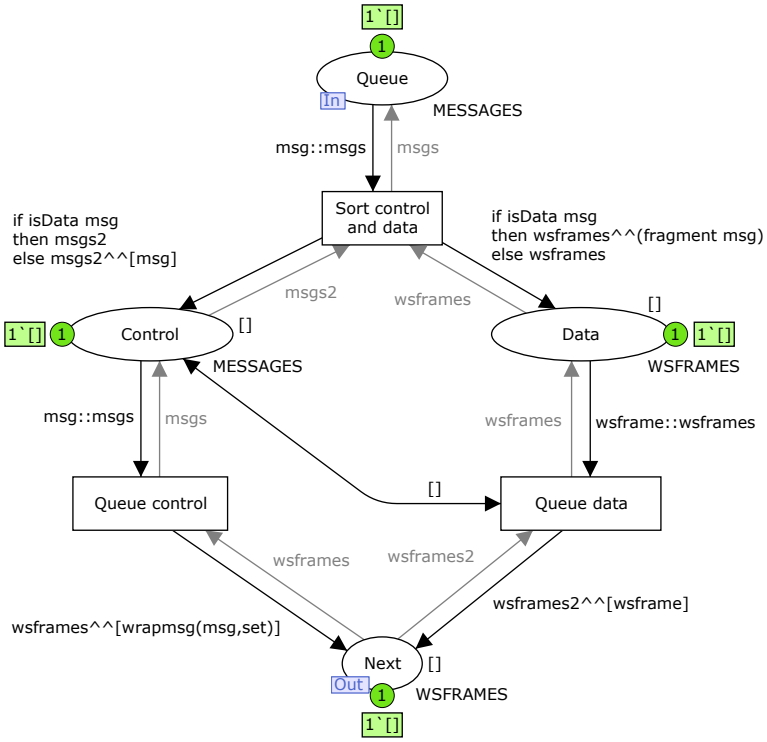


Figure 2.8: Fragment And Queue submodule

Listing 2.12: 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;
```

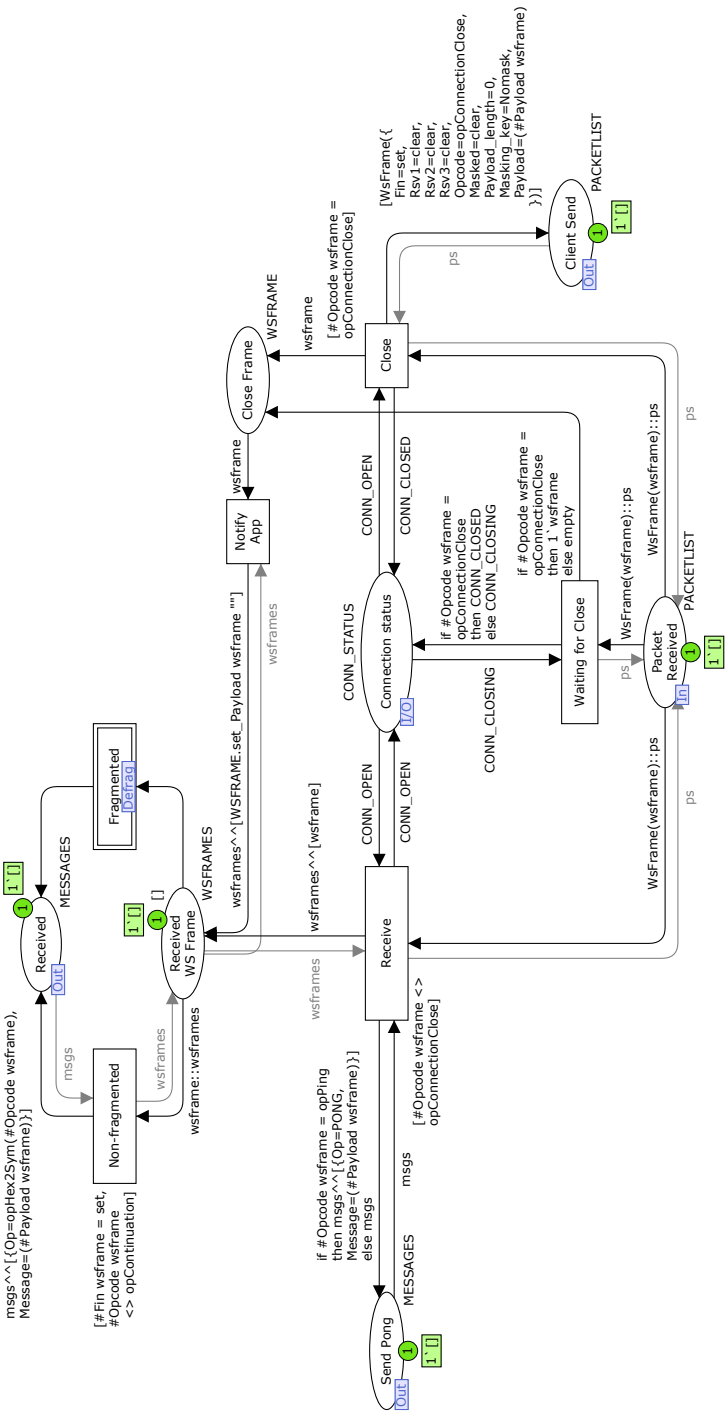


Figure 2.9: Unwrap And Receive submodule

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 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 as response, 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`.

Both the second and third paths will queue the received frame to notify the application, but the payload is stripped as it should not be exposed to the user according to the specification.

The received frame is now in the `Received WS Frame` place. It is now 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, this is part of a fragmented message and is processed in the Defrag submodule.

Defragmenting fragmented frames Fig. 2.10 shows this module. Frames that are part of a fragmented message can have its order determined by its fin bit and its opcode.

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.

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 since the WebSocket protocol does not allow fragmented mes-

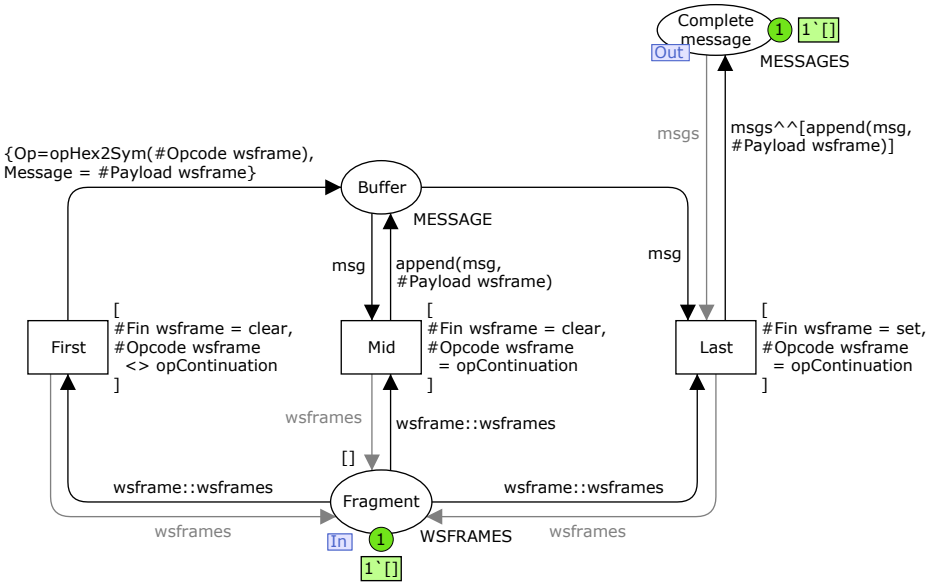


Figure 2.10: Defragment

Listing 2.13: append

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

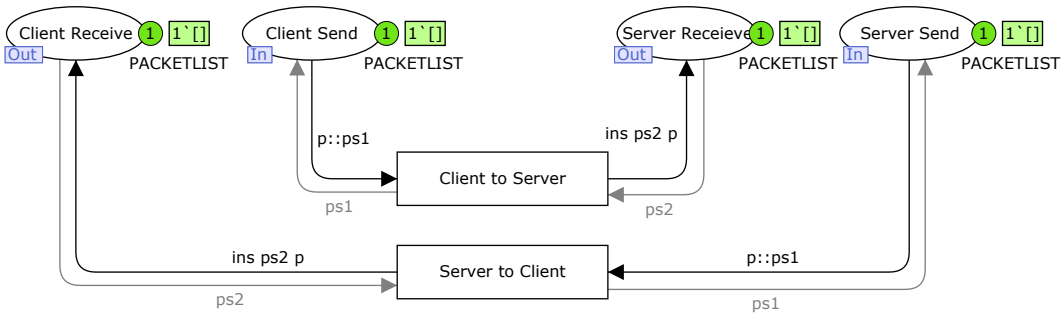


Figure 2.11: The Connection module

sages to be interleaved, and the TCP protocol guarantees preservation of order,it can be assumed that consecutive fragment frames belong to the same message and are in the correct order. Fragment inerleaving can be defined by subprotocols, but this is not relevant to this model.

2.3.4 Connection

Fig. 2.11 shows the connection layer. 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. It has purposely been modeled abstractly, since 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 module (Fig. 2.12) is very similar to the Client-side equivalent. 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 `unmask` function is shown in Listing 2.14.

The `Wrap and Send` and the `Unwrap and Receive` submodules are the same as the ones for the Client WebSocket module. To be more precise, the Client

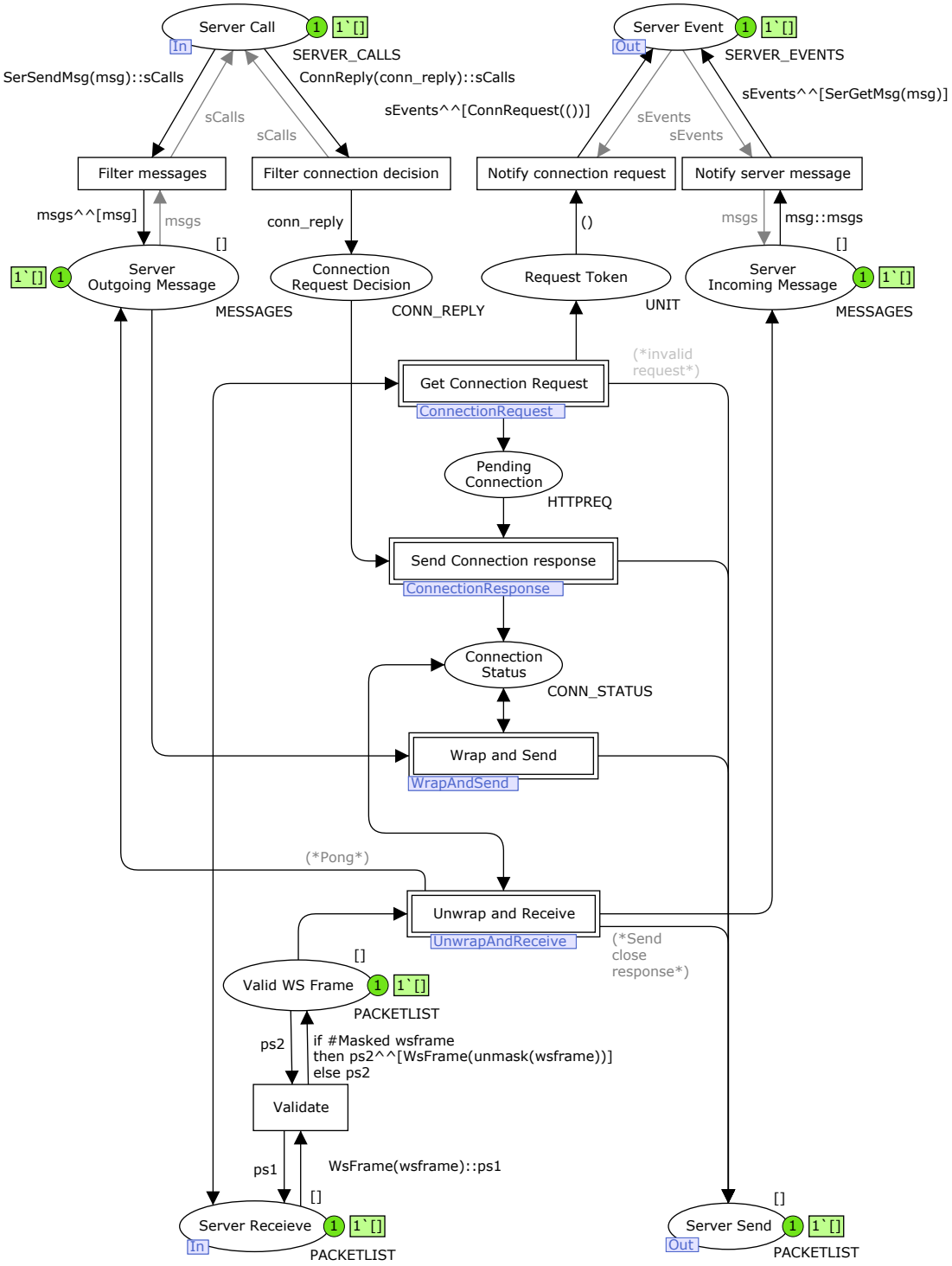


Figure 2.12: The Server WebSocket Module

Listing 2.14: unmask

```
fun unmask (ws:WSFRAME) = let
   val ws1 = WSFRAME.set_Masked ws clear
   val ws2 = WSFRAME.set_Masking_key ws1 Nomask
in
   ws2
end;
```

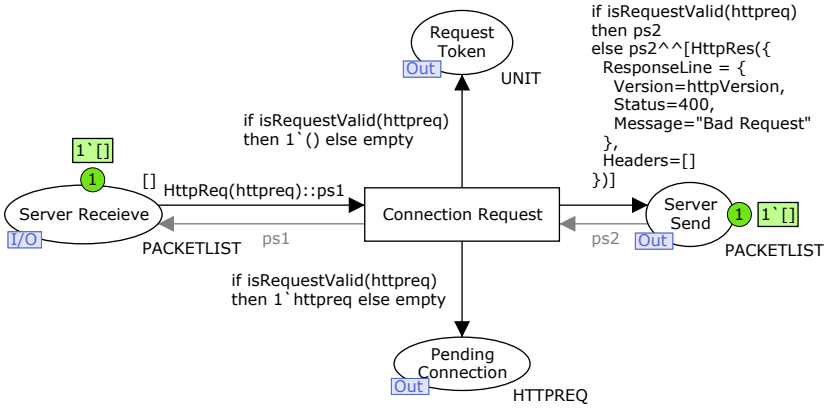


Figure 2.13: Get Connection Request

WebSocket module and Server WebSocket module both have instances of the same submodules, so that editing a submodule model affects both parent modules, while during simulation they can have different states.

Get Connection Request

Shown in Fig. 2.13, this is a very abstract representation of how connection requests are received. It checks the incoming request using `isRequestValid`, shown in Listing **??**. If it is, a simple `UNIT` token is sent to the app. In a real world app, much more data might have been sent, for example IP address and possibly authentication info from headers, but for this model it is enough to show that some data is being sent, while the details are abstracted away.

Listing 2.15: generateAccept

```
fun isRequestValid(req:HTTPREQ) = let
          val rline = #RequestLine req
          val headers = #Headers req
        in
          #Verb rline = GET andalso
          getHeader("Upgrade", headers)
            = "websocket" andalso
          getHeader("Connection", headers)
            = "Upgrade" andalso
          getHeader("Origin", headers)
            = origin
        end
```

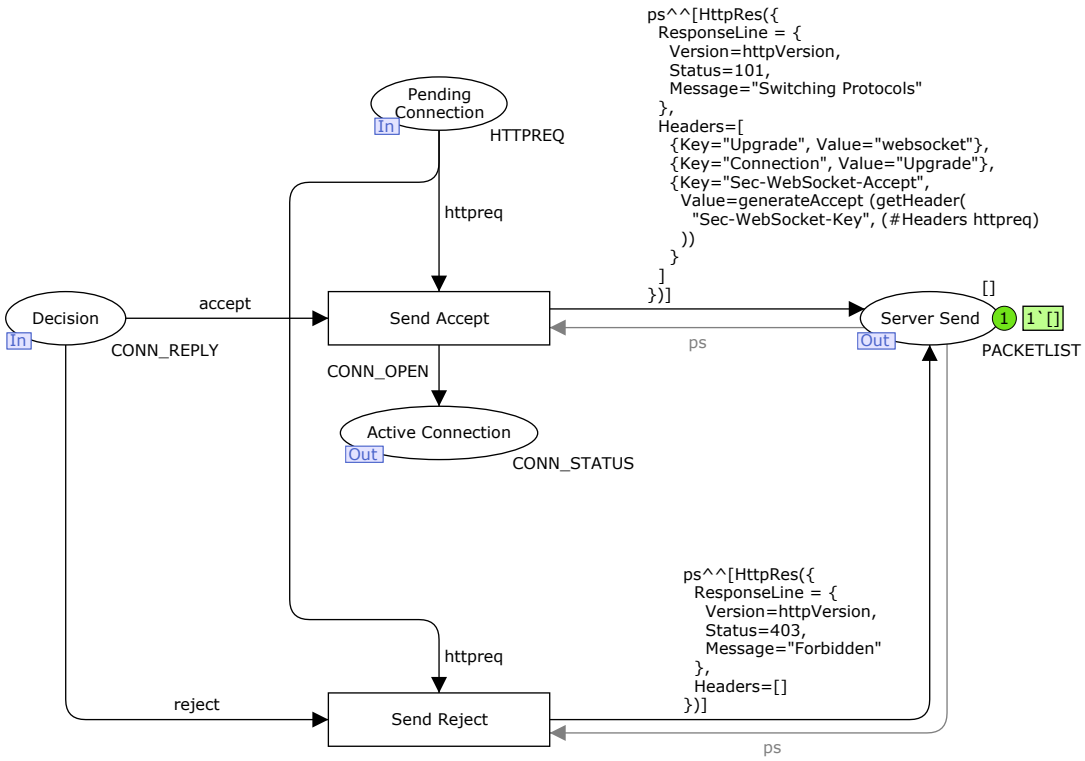


Figure 2.14: Send Connection Response

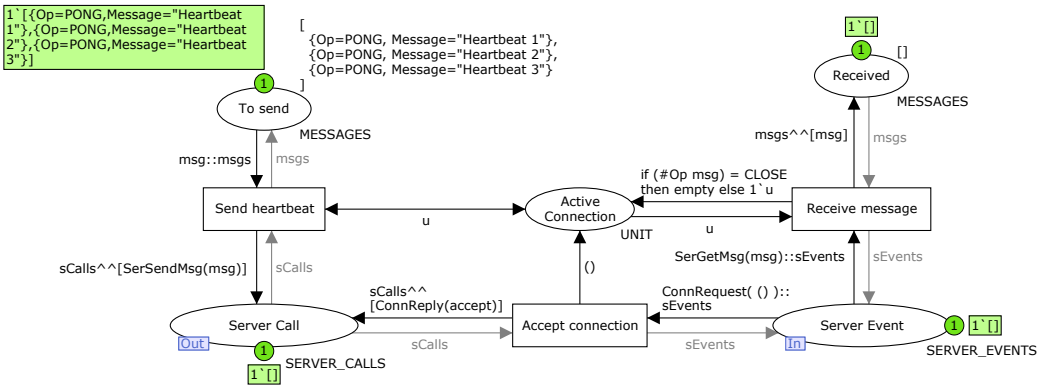


Figure 2.15: The Server Application Module

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 speccification of the WebSocket Protocol. This involves generating a Sec-WebSocket-Accept header, which is done with the `generateAccept` function whic was already explained in Listing 2.9.

2.3.6 Server Application

The Server Application has three tasks: Accept or reject incoming connections, and sendind and retrieval of data. The `To send` place has three messages as its initial marking, to illustrate the capability of PONG frames to be used as a heartbeat (without PING being involved). Otherwise, the mechanics of sending and receiving messages is the same as in the Client Application. 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 WebSocket 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

(a) Client (b) Server

Figure 3.1: Configuration: No messages

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 (see Fig. 2.3 and Fig. 2.15). We started by removing all messages to be sent (shown in Fig. 3.1). 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.

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 colour 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 colourset. The ran() function picks a value ranging over the colour set, but since is non-deterministic, it isn't suited for state space generation.

2. By using a free variable ranging over a colour 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 colour 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 colourset, 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.

Colour sets that use values from a large or unbounded range, or from continuous ranges like floating point numbers, are considered large colour sets, and using random values from such colour sets can make it impossible (or impractical) to generate a complete state space. It can be worked around by instead using small colour 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.

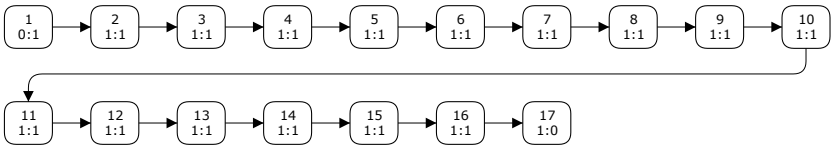


Figure 3.2: No messages

Listing 3.1: Fixed masking key

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

3.1.3 Visualisation

CPN Tools can visualise a state space once it has been calculated. Fig. 3.2 shows the state space for the no messages configuration. Rounded squares represent markings, and arcs represent transition occurences. Clicking on the small triangle in the node will display a node descriptor which shows the marking that is associated with the node. Similarly, clicking on a state space arc will display an arc descriptor which describes the binding element associated with the arc.

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.

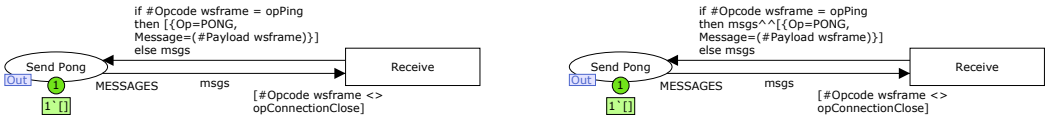
We also see that the SCC Graph has the same number of nodes and arcs, meaning that there are no cycles in the state space (although this was already known from the fact that it is a tree).

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   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 each 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.



(a) Before (b) After

Figure 3.3: Fixing Pong reply

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. 3.3 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 agrees with the visualisation shown earlier, that there is only one possible sequence of transition occurrences 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.

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 [JK09] chapter 7 for more information.

3.2.5 Larger Configurations

One short message

The next step is to gradually increase the number of messages to be passed between the endpoints. We start by configuring the client to send a single message: `{Op=TEXT, Message="Short␣message"}`.

Here is part of the generated report:

```
 Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  29
     Arcs:   28
     Secs:   0
     Status: Full

  Home Markings
     [29]

  Dead Markings
     [29]

  Live Transition Instances
     None

     No infinite occurrence sequences.
```

The number of markings has not increased by much, and the other properties are largely the same, except there are fewer dead transition instances.. The visualisation (Fig. 3.4) shows there is still only one chain of occurrences.

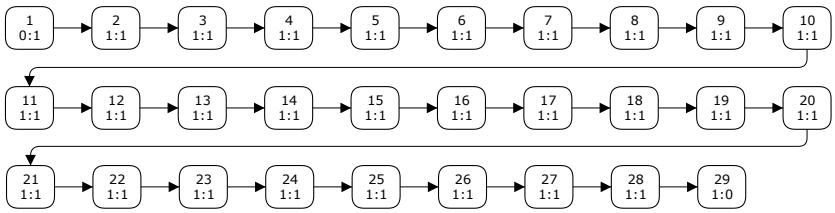


Figure 3.4: One message

One ping, then one message

When we add another message to be sent (a ping, which will also result in a pong being sent back), we see that the number of nodes has increased by an order of magnitude:

```
State Space
   Nodes:  475
   Arcs:   1140
   Secs:   1
   Status: Full

Home Markings
   [475]

Dead Markings
   [475]
```

CPN Tools supports exporting state spaces to a format supported by Graphviz, an open source application for visualising graphs. We have used Graphviz to visualise this state space, shown in Fig. 3.5. This clearly demonstrates the effect of the state space explosion problem.

One message, then one ping

By reversing the order of the two messages, the state space gets slightly larger, due to the fact that WebSocket could send the ping frame first, since it is a control frame.

```
State Space
   Nodes:  513
   Arcs:   1141
   Secs:   1
   Status: Full

Home Markings
   None
```

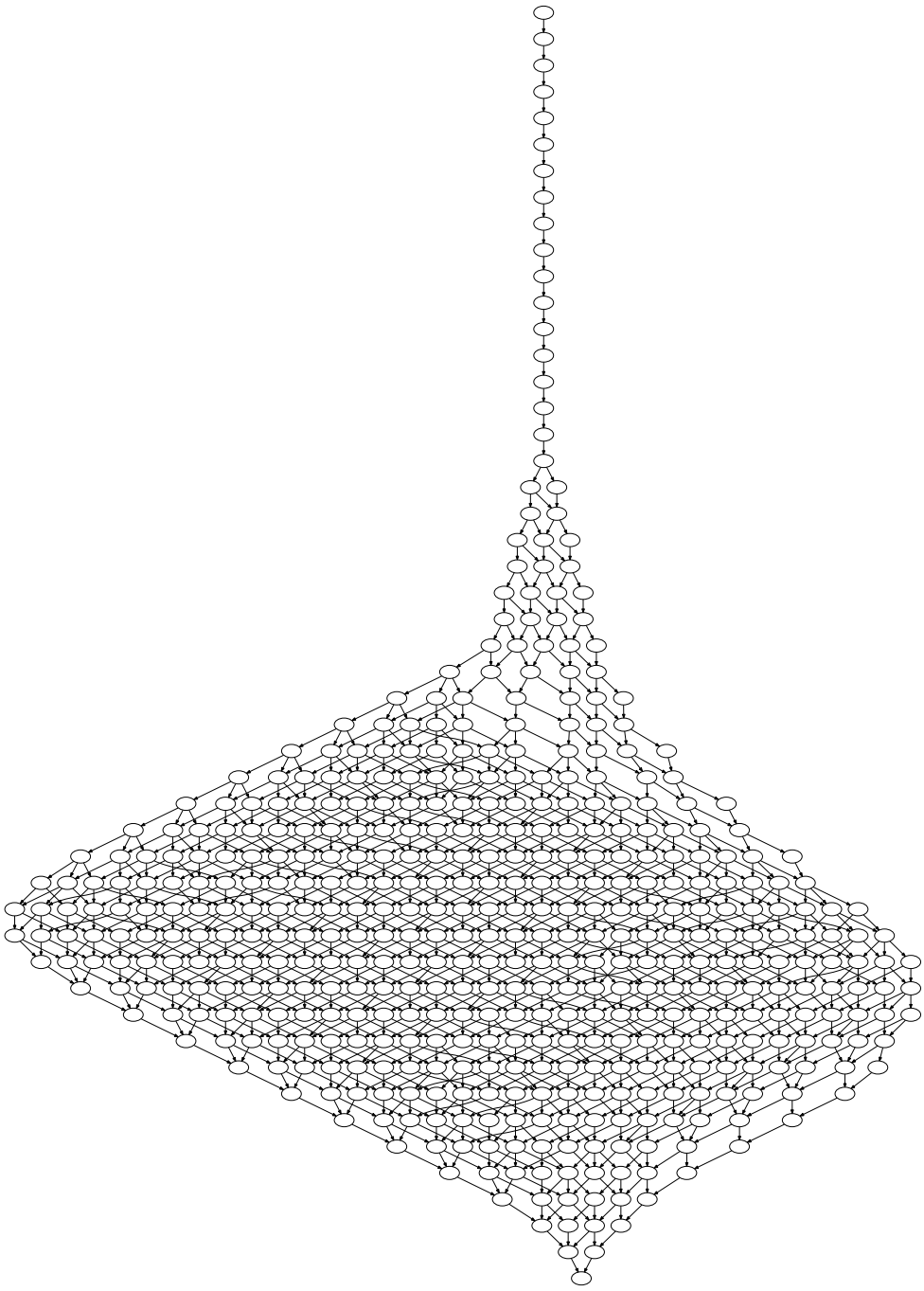


Figure 3.5: One ping, one message

```
  Dead Markings
     [512,513]
```

Note that we now no longer have any home marking, and instead have two dead markings, representing the two possible orderings of the two messages.

One long message

We now set a message that is large enough to require fragmenting. `{Op=TEXT, Message="Very␣long␣message.␣Very␣long␣message.␣Very␣long␣message.␣Very␣long␣message.␣Very␣long␣message.␣"}`

```
  State Space
     Nodes:  813
     Arcs:   2331
     Secs:   2
     Status: Full

   Home Markings
     [813]

  Dead Markings
     [813]
```

We see this has more states than sending two simple messages. We can also use the Best Upper Multi-set Bounds section to find the largest collection of fragments.

```
1`[WsFrame({Fin=clear,Rsv1=clear,Rsv2=clear,Rsv3=clear,Opcode
    =1,Masked=set,Payload_length=20,Masking_key=Mask([0,0,0,0])
    ,Payload="Very long message. V"}),
WsFrame({Fin=clear,Rsv1=clear,Rsv2=clear,Rsv3=clear,Opcode=0,
    Masked=set,Payload_length=20,Masking_key=Mask([0,0,0,0]),
    Payload="ery long message. Ve"}),
WsFrame({Fin=clear,Rsv1=clear,Rsv2=clear,Rsv3=clear,Opcode=0,
    Masked=set,Payload_length=20,Masking_key=Mask([0,0,0,0]),
    Payload="ry long message. Ver"}),
WsFrame({Fin=clear,Rsv1=clear,Rsv2=clear,Rsv3=clear,Opcode=0,
    Masked=set,Payload_length=20,Masking_key=Mask([0,0,0,0]),
    Payload="y long message. Very"}),
WsFrame({Fin=set,Rsv1=clear,Rsv2=clear,Rsv3=clear,Opcode=0,
    Masked=set,Payload_length=15,Masking_key=Mask([0,0,0,0]),
    Payload=" long message. "})]
```

We are also able to inspect every other possible combination, and can thus confirm that messages are being split correctly.

Ping, text, close

With three messages, state space calculation becomes noticeably time-consuming.

```
State Space
   Nodes:  6129
   Arcs:   19625
   Secs:   14
   Status: Full

     Dead Markings
   6 [6129,6112,6029,5960,5632,...]
```

Here we have six dead markings. In half the cases, the close is sent before the text. For each of those halves, the three cases consist of the pong frame either successfully arriving at the client, not being received by the client due to the connection being closed, or not being sent from the server for the same reason. This was verified by manual inspection in CPN Tools.

Even larger configurations

We tried adding one more message to the server application, but after running for 5 hours the state space calculation had still not been able to compute a complete state space. Fortunately, it is still possible to create a report for the partial graph.

```
State Space
   Nodes:  165748
   Arcs:   707380
   Secs:   18000
   Status: Partial
```

The report also showed that it had not found any cycles, which reinforces the claim that the model works as it should.

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

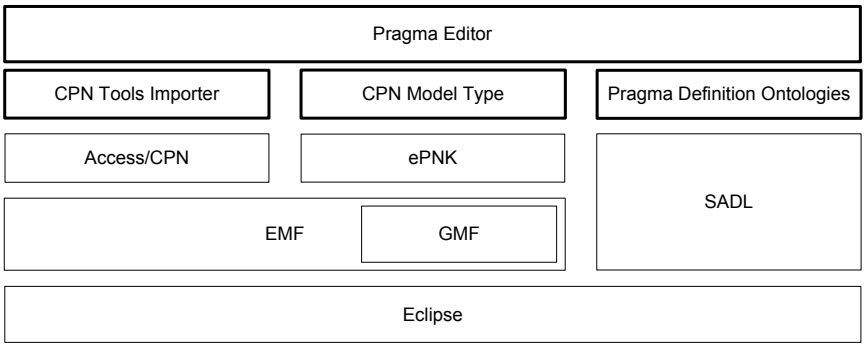


Figure 4.1: Application Overview Diagram

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 [ecl] 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

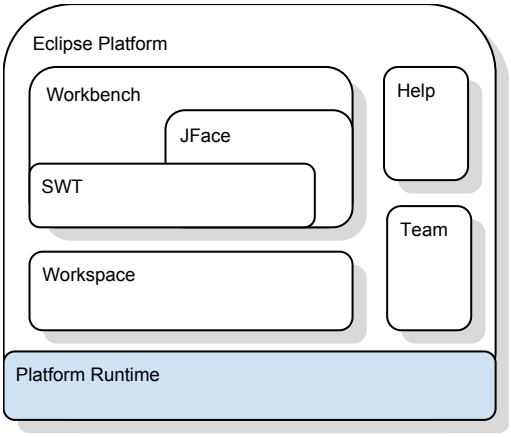


Figure 4.2: The Eclipse RCP

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 [ISO11] 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

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

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

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 colourset 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 [OWL09] is the World Wide Web Consortium (W3C) recommended standard for representing ontologies. The primary exchange syntax for OWL 2 is RDF/XML [W3C04]. 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 [owl], 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.

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.

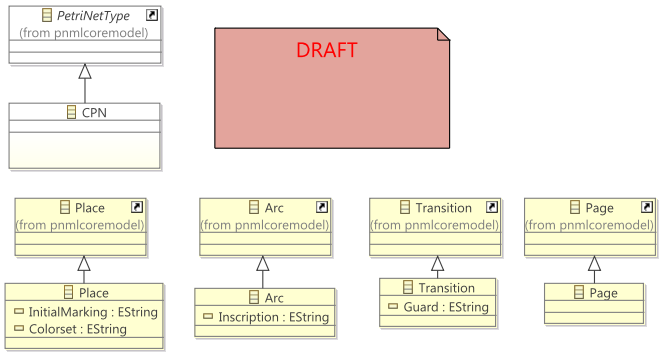


Figure 5.1: CPN model type diagram

There exists two ontologies that function as a base for pragmatics: One that defines Coloured 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

The CPN model, shown in Fig. 5.1, defines the structure and constraints of Coloured 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

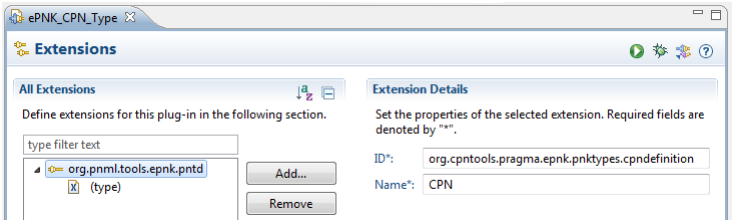


Figure 5.2: Plugin Manifest part 1

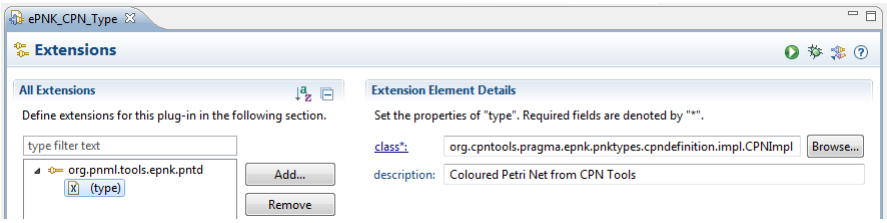


Figure 5.3: Plugin Manifest part 2

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 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

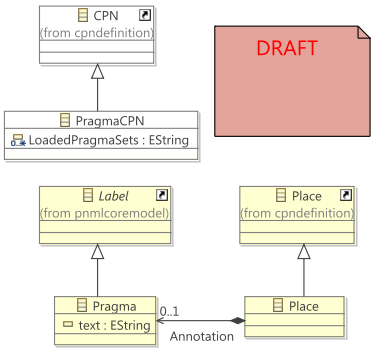


Figure 5.4: Annotated CPN model type diagram

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.

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

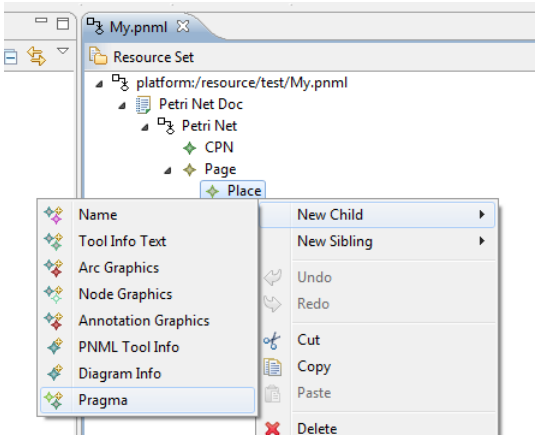


Figure 5.5: Adding Pragma as child to Place

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

Chapter 7

Conclusion

7.1 Results

7.1.1 Limitations

7.2 Further work

7.3 Acknowledgments

Bibliography

[ecl] Eclipse ide.

[FM11] I. Fette and A. Melnikov. The websocket protocol. Internet-Draft draft-ietf-hybi-thewebsocketprotocol-15, Internet Engineering Task Force, September 2011.

[FS11] Kent Inge Fagerland Simonsen. On the use of pragmatics for model-based development of protocol software. In Michael Duvigneau, Daniel Moldt, and Kunihiko Hiraishi, editors, *Petri Nets and Software Engineering. International Workshop PNSE'11, Newcastle upon Tyne, UK, June 2011. Proceedings*, volume 723 of *CEUR Workshop Proceedings*, pages 179–190. CEUR-WS.org, June 2011.

[iosat02] National institute˜of˜standards˜and technology. FIPS 180-2, secure hash standard, federal information processing standard (FIPS), publication 180-2. Technical report, DEPARTMENT OF COMMERCE, August 2002.

[ISO11] ISO. *ISO 15909-2:2011 Systems and software engineering – High-level Petri nets – Part 2: Transfer format*, 2011.

[JK09] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

[Jos06] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006.

[KW10] L. Kristensen and M. Westergaard. Automatic structure-based code generation from coloured petri nets: a proof of concept. *Formal Methods for Industrial Critical Systems*, pages 215–230, 2010.

[LT07] K. B. Lassen and S. Tjell. Translating colored control flow nets into readable java via annotated java workflow nets. *Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 127–136, 2007.

[Mil97] R. Milner. *The definition of standard ML: revised*. The MIT press, 1997.

[Mor00] K. Mortensen. Automatic code generation method based on coloured petri net models applied on an access control system. *Application and Theory of Petri Nets 2000*, pages 367–386, 2000.

[owl] Owl api.

[OWL09] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. Available at `http://www.w3.org/TR/owl2-overview/`.

[RWL+03] Anne Ratzer, Lisa Wells, Henry Lassen, Mads Laursen, Jacob Qvortrup, Martin Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In Wil van der Aalst and Eike Best, editors, *Applications and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 450–462. Springer Berlin / Heidelberg, 2003.

[W3C04] W3C. *RDF/XML Syntax Specification*, 2004.

[Zim80] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

List of Figures

Listings