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 20, 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 in turn 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). CPN 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 currently 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 called Pragma/CPN 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 manipulating CPN models. Pragma/CPN is desgned as a plugin for the Eclipse platform. 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 give a detailed case study on CPN modeling of the WebSocket protocol, including verification and analysis of the CPN model using state space exploration, and annotation with code generation pragmatics.

Contents

1 Introduction **1**
1.1 Model Driven Software Engineering 1
1.2 Thesis Aims and Results 3
1.3 Related Work 5
1.4 Thesis Organisation 5

2 The WebSocket Protocol CPN Model **7**
2.1 The WebSocket Protocol 7
2.2 The WebSocket CPN Model 8
2.2.1 Overview 9
2.2.2 Client Application 12
2.2.3 The Client WebSocket Module 17
2.2.4 Connection Module 30
2.2.5 The Server WebSocket Module 31
2.2.6 Server Application 34

3 State Space Analysis of the WebSocket Protocol **37**
3.1 State Spaces 37
3.1.1 Strongly Connected Component graph 37
3.1.2 Application of State Spaces 38
3.1.3 Visualisation 40
3.2 State Space Report 40
3.2.1 Statistics 40
3.2.2 Boundedness Properties 41
3.2.3 Home Properties 42
3.2.4 Liveness Properties 44
3.2.5 Larger Configurations 44
3.2.6 Summary 49

4 Technology and Foundations **51**
4.1 Representing CPN Models 51
4.2 The Eclipse Platform 52
4.3 ePNK: Petri Net modeling framework 54
4.4 Eclipse Modeling Framework 56
4.4.1 Graphical Modeling Framework 57
4.5 Access/CPN: Java interface for CPN Tools 57
4.6 Ontologies: OWL 2 and OWL API 57

5 Analysis and Design **59**
5.1 Defining Pragmatics 59
5.2 The CPN Ontology with Pragmatics 59
5.3 The Annotated CPN model type for ePNK 60
5.4 Importing from CPN Tools 63
5.5 Creating annotations 63
5.6 Choosing Pragmatics Sets 63
5.6.1 Creating custom pragmatics 64

6 Evaluation **65**
6.1 Requirements 65
6.2 Test cases 65
6.3 User Feedback 65

7 Conclusion **67**
7.1 Results 67
7.1.1 Limitations 67
7.2 Further work 67
7.3 Acknowledgments 67

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 every challenge in modern software engineering, 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 software development 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) [Gro11] and the methods and tools developed for and around it (like Rational Unified Process [rup]). However, models largely play a secondary role, primarily playing the role of design tools and documentation.

Model Driven Engineering (MDE) [Ken02] 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 provides a layer of abstraction over source code. This trend has been touted as a new programming paradigm, in the same way object-oriented programming was when it was conceived.

MDE is based on two central concepts:

- Domain specific modeling languages (DSML), which are used to formalise application structure, behavior, and requirements of specific domains, such as financial services, warehouse management, task scheduling, and protocol and communication software. A DSML 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 documentation, deployment descriptions, alternative 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 may also act as graphical representations of the system, making them more understandable for people that are not programmers. This is for instance important when soliciting requirements from customers.

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. Petri Nets are a type of directed graph used to model processes, especially with an asynchronous and/or concurrent nature. Common example domains are communication networks and protocols, as well as concurrent programming design. Petri Nets are very well suited for MDE, as they have robust methods for computer-aided verification and analysis.

There exist many extensions to the Petri Net formalism 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 (type), essentially data values from a set of values. Combined with the capability to model synchronicity, concurrency and communication that Petri Nets give, the functional programming language Standard ML [Mil97] is used as a foundation to define colours and colour sets in a compact manner, and to provide basic data types and manipulation implementation.

A common way of analysing Petri Nets is called state space exploration, and is a powerful method for automatic model verification and determination of several properties. This thesis gives a short introduction to CPN and state space exploration.

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 Standard ML, developed at Edinburgh University.

Mer om CPN Tools

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 automatically generate source code.

1.2 Thesis Aims and Results

This thesis focuses on work done by Simonsen [FS11], who discusses some of the challenges in modelling and automatically generating software. His work is focused on the domain of communication protocols, and uses the Kao-Chow authentication protocol as an example. The ideas introduced in the paper sketch 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 describes how to implement specific constructs for a particular platform,
- Create a configuration model for capturing implementation details, including the mapping from the protocol model to the platform model.

Fig. 1.1 shows the top level module of the Kao-Chow CPN model, with pragmatics enclosed in <<>>.

The work of Simonsen [FS11] is focused on the model transformation and code generation aspect. Annotation of the CPN models in [FS11] are done with simple text files. This approach is cumbersome, and there is a need for developing specialised tool support for this purpose.

Based on the very initial ideas of [FS11], the aim of this thesis is to investigate how to support annotation of CPN models with code generation pragmatics in a flexible and model-centric manner. By model-centric we mean that annotation of CPN models should be closely integrated with editing of CPN model elements.

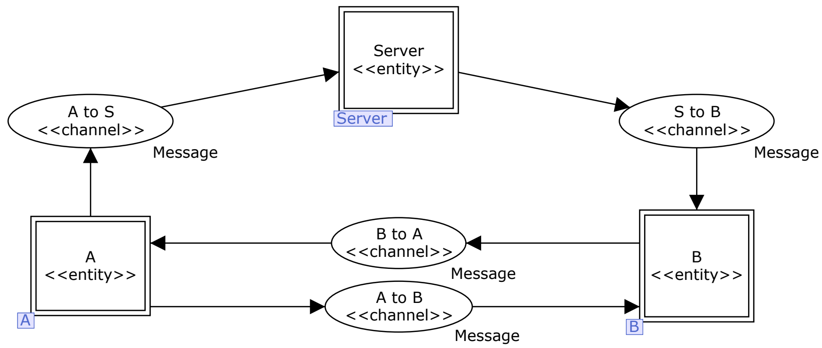


Figure 1.1: Top level module of the Kao-Chow model

The research method used to answer this question is to create a prototype application framework, and evaluate it by annotating a selected set of CPN models of communication protocols, including a detailed case study on the application of the WebSocket protocol. The resulting application has been named Pragma/CPN. This name was chosen to resemble the naming format of other tools that are built around CPN Tools, including Access/CPN and Design/CPN.

The requirements for the prototype can be divided into four main items:

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

The code generation pragmatics (or just "pragmatics") in the approach developed in this thesis 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.

Pragma/CPN builds on the ePNK framework [Kin11], 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 the Eclipse platform, 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, which gives advanced capabilites to classify pragmatics and dynamically deduce appropriate pragmatics for individual model elements. These ontologies can be dynamically loaded into models, which lets the user write their own ontology containing model specific pragmatics.

1.3 Related Work

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.

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. The chapter gives an introduction to Coloured Petri Nets (CPNs) and the CPN Tools application used to create the models, combined with a detailed presentation of the CPN model of the WebSocket protocol produced as part of this 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. This validates that the constructed CPN model of the WebSocket protocol is behaviourally correct.

Chapter 4: Technology and Foundations. Pragma/CPN 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 Pragma/CPN); Access/CPN (the engine used to import models from CPN Tools). This chapter also provides an introduction to ontologies, the format used to specify pragmatics classes.

Chapter 5: Analysis and Design. Gives a discussion and detailing of requirements for Pragma/CPN. 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.

Bedre flyt her

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 as part of describing the WebSocket protocol in Chapter 2.

Chapter 2

The WebSocket Protocol CPN Model

The primary case study developed in this thesis for evaluation of the Pragma/CPN framework is the WebSocket protocol [FM11]. The WebSocket protocol is designed to provide two-way communication between a client running untrusted code running in a controlled environment (e.g. a web browser) to a remote host that has opted-in to communications from the untrusted code.

Section 2.1 briefly introduces the basic operation and concepts of the WebSocket protocol. Section 2.2 presents the constructed CPN model of the WebSocket protocol and introduces the basic conocepts of the CPN modelling language.

2.1 The WebSocket Protocol

Fig. 2.1 shows the basic sequence of message exchanges in 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 reason, and the connection is considered closed.

Conceptually, WebSocket constitues a layer on top of TCP that does the following:

- adds a Web "origin"-based security model for browsers

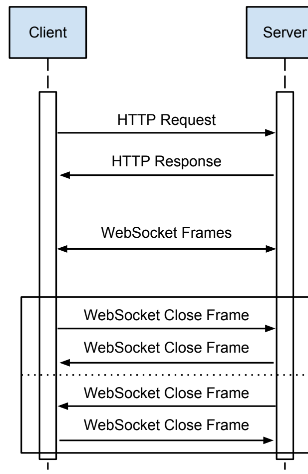


Figure 2.1: Sequence Diagram of the WebSocket protocol

- adds an addressing and protocol naming mechanism to support multiple services on one port and multiple host names on one IP address;

- 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 and with reliability

- includes an additional closing handshake in-band that is designed to work in the presence of proxies and other intermediaries

Further details on the operation of the WebSocket protocol will be provided in the next section when presenting the constructed CPN model.

2.2 The WebSocket CPN Model

A CPN model is organised into pages, also called modules. Each module may in turn contain sub-modules, which provides a way keep complex and/or large models clear and manageable. The WebSocket model makes full use of this hierarchy feature, and we describe the WebSocket CPN model in a top-down manner in the following subsections.

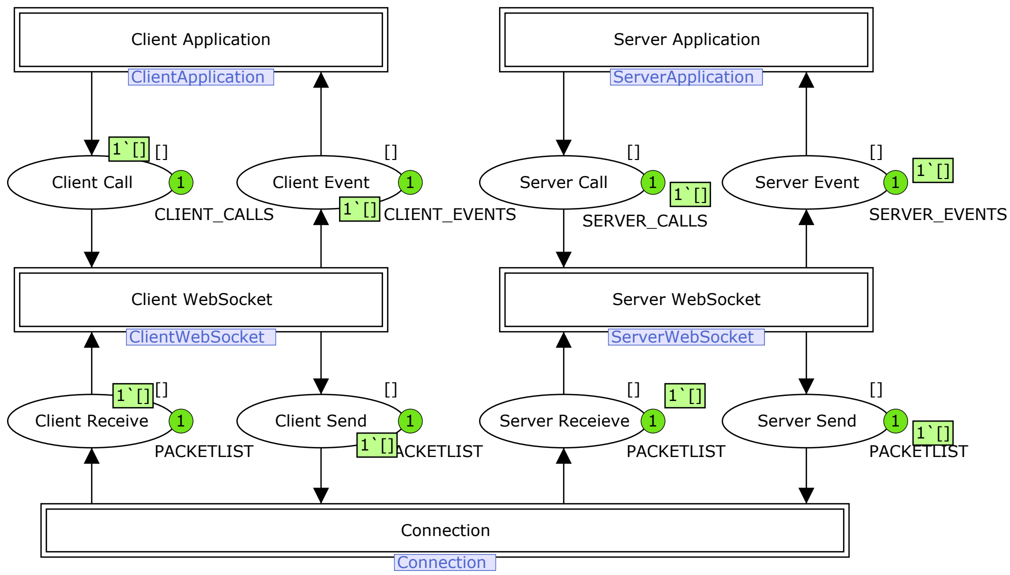


Figure 2.2: Overview module of the WebSocket CPN model

2.2.1 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 rectangles called substitution transitions, and several places and arcs that connect them, represented with circles and arrows respectively. The function of arcs and arc expressions will be explained when presenting the Client Application module.

The text in the middle of each substitution trasition is the name of that node, while the text in the small blue box attached to the bottom is the name of the associated sub-module. These names are often the same, although module names cannot contain spaces.

The substitution transitions 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 layers. They will be referred to as the application layer, the protocol layer and the network layer respectively.

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 colours 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. The number of tokens currently in a place is represented by the number in the small green circle attached to the place, and the details of the token colours are shown in the small green box next to it. Each place also specifies a colour set and an initial marking.

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 the colour set definition `colset INT = int;` representing the set of all integers. A token from this colour set could for example have the value '3'. Similar standard declarations are present for `STRING`, `BOOL` and `UNIT`. These are examples of simple color sets.

It is also possible to declare complex color sets, termed compound color sets, to form data structures / data types. One such compound type 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 module use the colour set definitions provided in Listing 2.1.

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

The `MESSAGE` colour set represents the messages sent from and to the application layers. This colour set is defined as a record, which means a 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 also have a list of `MESSAGES` to keep them ordered. Lists are used to keep tokens ordered; their use will be explained in the following subsection.

For the client to connect to a server in the WebSocket protocol, the URL of the server 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 exam-

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;
```

ple /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 obtained using one of the constructors of the union color set with an argument that matches the constructor. A union colour set is typically 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 constructors: `Connect` (used to request a connection to the associated `URL`), and `CliSendMsg` (signifying an outbound message from the client). 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 boolean values `fail` (false) and `success` (true) in order 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`. We similarly have a list version `CLIENT_EVENTS`.

The reason we use a single place to transmit both connection information and messages between the application and protocol 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 do not fully model actual network packets, as this level of detail is not required in order to model the operation of the WebSocket protocol. This color set is also used in a list `PACKETLIST`.

2.2.2 Client Application

This module (Fig. 2.3) is designed 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 bordered rectangles, are elements that capture state changes in the model. They are connected to places by directed arcs. These arcs can be inscribed with expressions containing variables. A variable declaration has the form:

```
var <name> = <color-set>;
```

Variables are declared globally and can only be bound to values (colours) of the colour set they are defined for. The set of variables declared for the simple coloursets are provided in Listing 2.2, and some of them can be seen in use on arcs in Fig. 2.3.

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;
```

The execution of a CPN model consists of enabled transitions removing tokens connected from its input places and adding tokens to its output

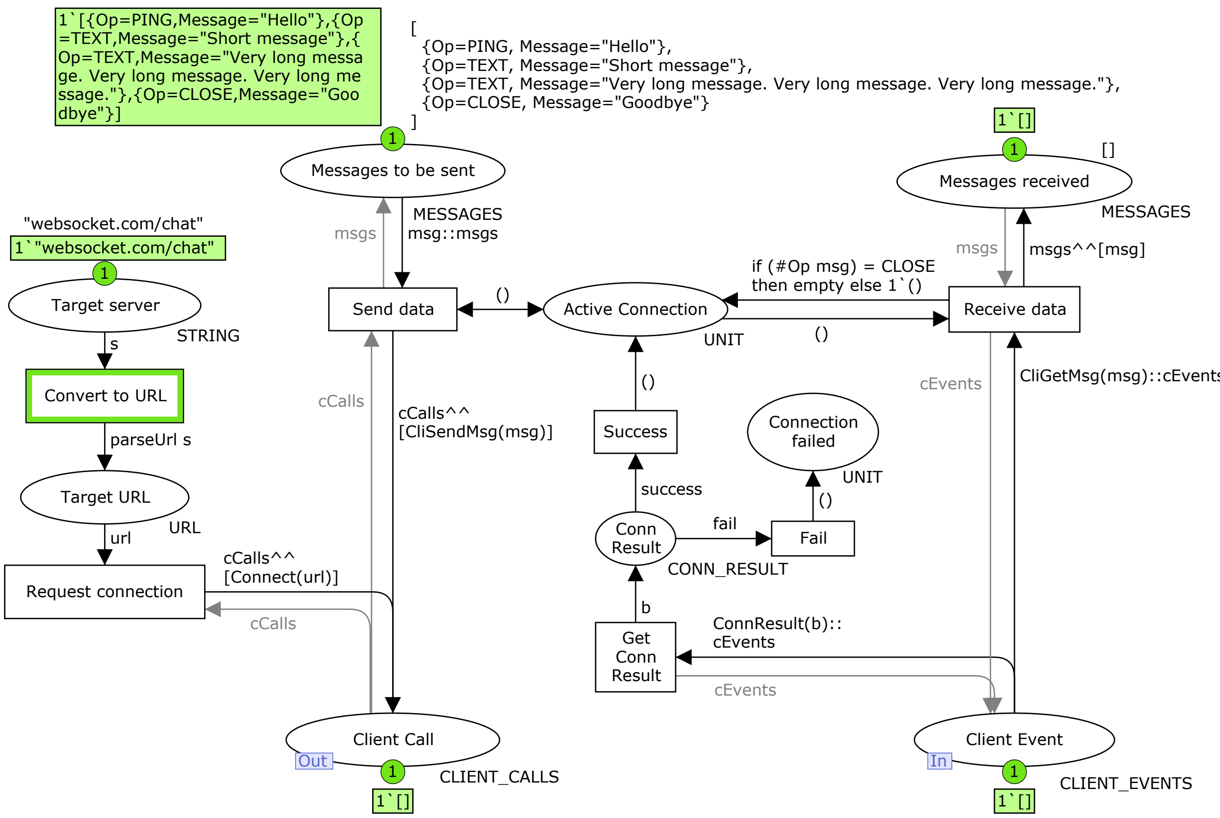


Figure 2.3: The Client Application Module

places. In order to talk about the enabling of a transition, a binding that assignes values to the variables of the transition must be provided. The variables of a transition are the variables occurring in the arc expressions of arcs connected to the transition. Assigning values to the variables of a transition allow the arc expressions to be evaluated in order to determine the value of tokens to be removed and added. A binding of a transition is enabled when there are tokens available in the input places that match the color sets that the variables has been declared for.

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 string value "websocket.com/chat" of the token in the Target server place. The variable s is then used as an argument for the function `parseUrl` (explained further down), which produces an `URL` token that is added to the Target URL place.

Several variables have been created for each colourset for convenience in the cases where tokens of the same colour set but with different colours is consumed from separate places. If the same variable was used on each arc from two separate places, both places would need a token with the same colour before the transition would be enabled.

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 the Client Appliaction 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. It makes use of a `split` function, which splits a string on a given substringand returns the result as a list. `parseUrl` will make sure all the parts of the returned `URL` token are given values, and provides default values for any part that is missing from its string argument.

The places at the bottom of Fig. 2.3, Client Call and Client Event represent the interface to the WebSocket layer, and are paired with the accordingly named places on the Overview module, which are also paired with corresponding places in the Client WebSocket module. 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

Listing 2.4: The parseUrl function

```
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;
```

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 significance in CPN Tools).

Queues

A number 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 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 `[ ]`, with the empty list specified as `[]`. 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. This list consists of records representing WebSocket messages.

Lists work similarly to other functional programming languages, with the first element being denoted the head of the list, and the remaining elements being denoted the tail of the list To access these elements, we use the :: list constructorlike this: `head::tail`. Lists are usually processed in a tail-recursive mannner. The `^^` operator is used 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.

Control Flow

After the Convert to URL transition has occured (see Fig. 2.3), the Request connection transition will 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 the `url`, and add it to the queue in the Client Call place.

Next, the Client Application waits for a `ConnResult` token to appear on place Client Event and place it in the Conn Result place. The expressions on the arcs going out from the Conn Result 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. This models that the WebSocket connection has now been established.

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 marking 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.

2.2.3 The Client WebSocket Module

This module (shown in Fig. 2.4) 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 connection request submodule handles creation and sending of HTTP requests based on the URL specified by the application. The Process response submodule processes the expected HTTP reply from the server. The Wrap and send submodule forms WebSocket frames from the messages sent from the client application, and the Unwrap and receive submodule does the reverse with WebSocket frames coming from the server. All of these submodules will be detailed in the coming sections.

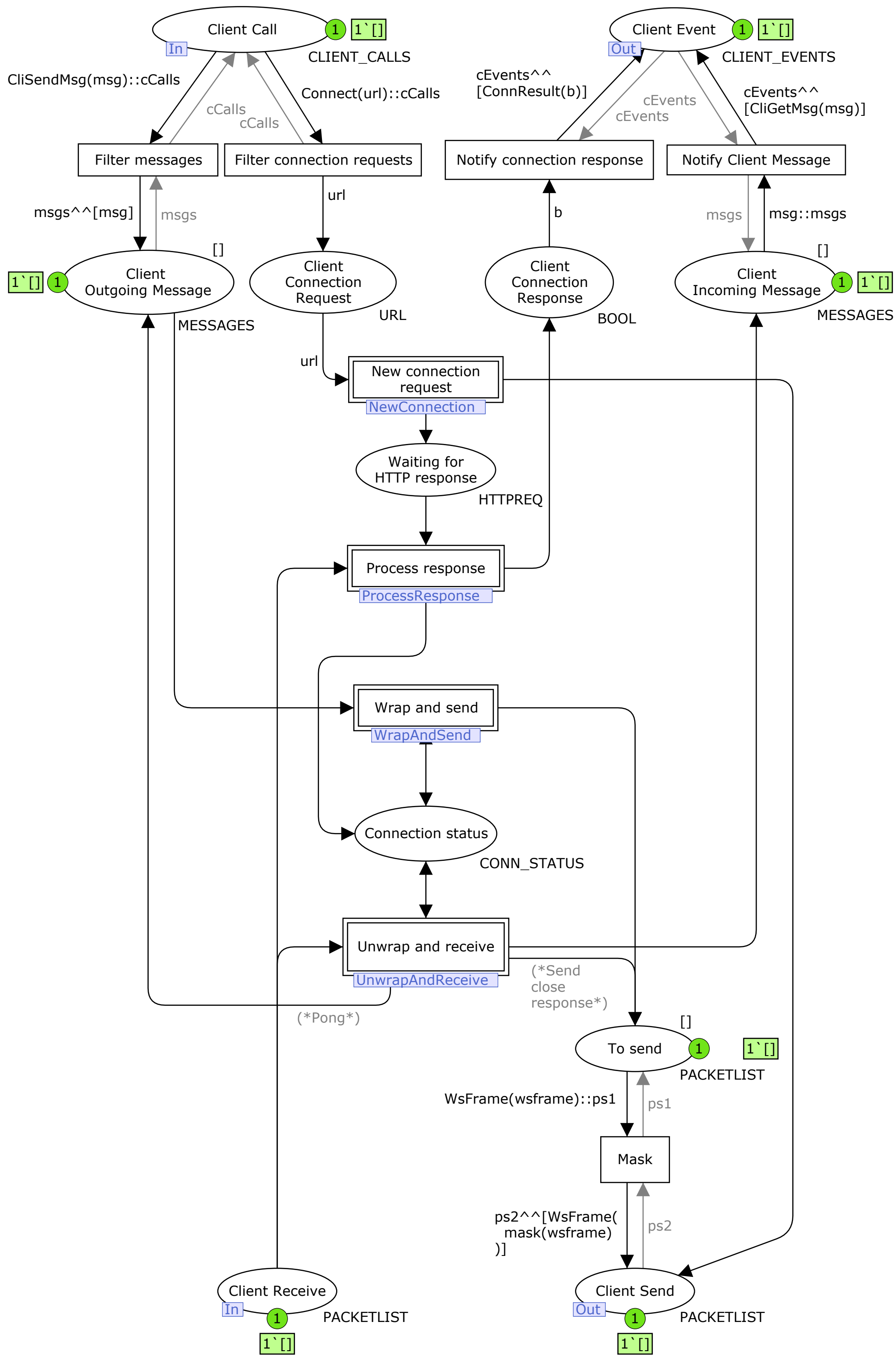


Figure 2.4: The Client WebSocket Module

The new coloursets used in this module 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.

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;
```

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, hence this has been abstracted away in the CPN model.

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

Next, we declare a number of symbolic 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

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;
```

a masking key. An actual implementation would also apply the mask to the payload according to the protocol 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 shown in Listing 2.8.

New Connection Module

This module is shown in Fig. 2.5. We take an available `URL` from the New Connection place and create an HTTP request, which is then queued to be sent by the client, as well as keeping a copy of the request for validation purposes when the response arrives.

The 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, 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;
```

partly to keep it simple and partly to more easily see it during simulation.

The two functions `B64()` and `SHA1()` are abstract versions of the Base64 encoding algorithm [Jos06] and the SHA1 hasing algorithm [iosat02] on which the WebSocket protocol relies. We considered it 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 Module

On this module, shown in Fig. 2.6, the transition Process HTTP Response is enabled when a `HTTPRES` token arrives from the server (and the `HTTPREQ` token from earlier is still waiting in the place Waiting for HTTP Response). 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 the server's reply is valid and conforms to the WebSocket protocol specification. This involves generating a hash token that proves the server has interpreted the original request in accordance with the WebSocket specification. The hash is computed using `B64` and `SHA1` on a static universally unique identifier (UUID) and the random nonce hash generated for the `Sec-WebSocket-Key` header.

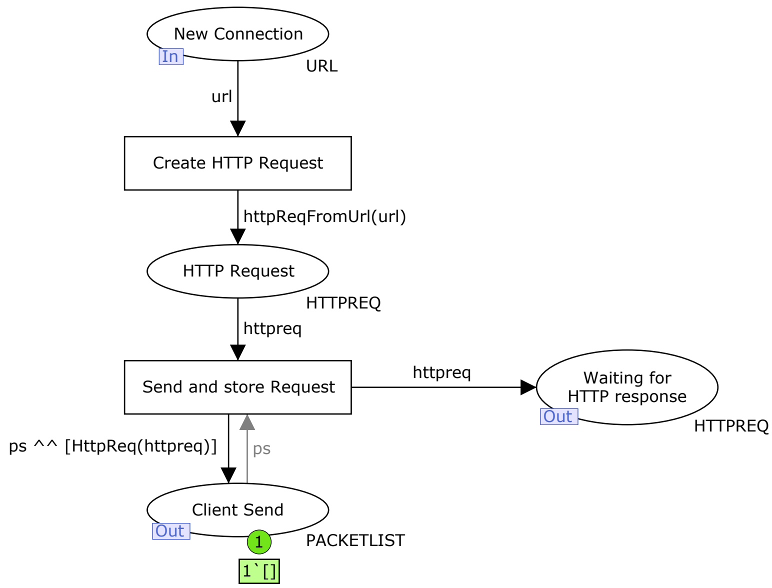


Figure 2.5: New Connection submodule

Listing 2.9: httpReqFromUrl

```
val origin = "http://www.example.com";
val nonce = "nonce";

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

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}
          ]
};
```

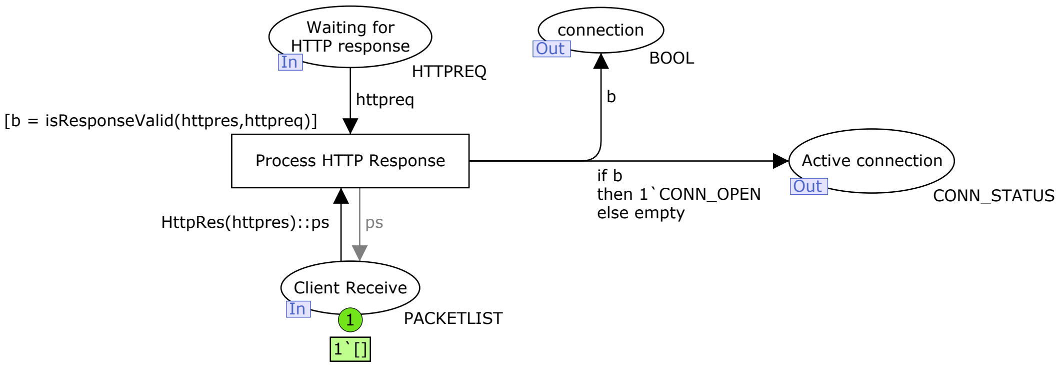


Figure 2.6: Process Response submodule

Listing 2.10: isResponseValid

```
val uuid = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
fun generateAccept str = B64(SHA1(str^uuid));

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
```

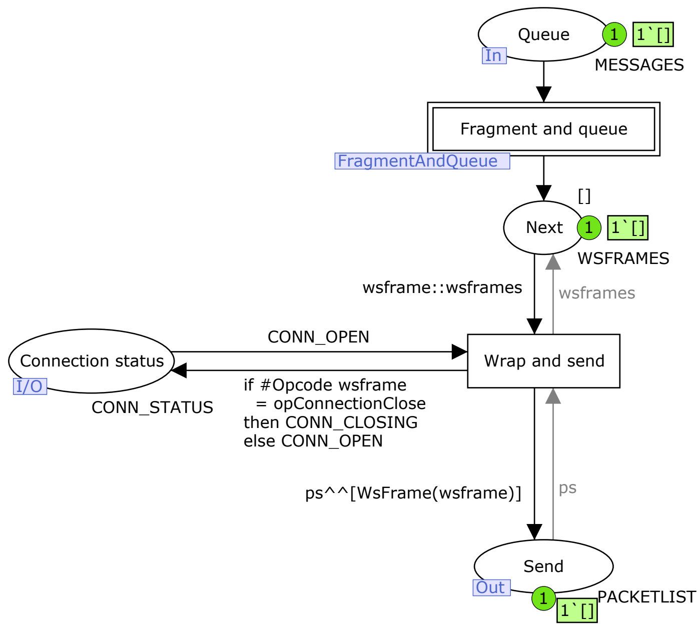


Figure 2.7: Wrap And Send submodule

Since `b` is a boolean variable, we can use it directly to notify the Client App through the connection place whether the response was valid and thus if the connection was successful or not. Additionally, if `b` is `true`, a `CONN_OPEN` token is put in the Active Connection place to signify that the connection is open.

Wrap and Send Module

This module (Fig. 2.7) takes new messages from place Queue, wraps and optionally fragments them in the Fragment and queue submodule, and sends them if there is an open connection as indicated by the token on the Connection status place. 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 Module 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 that can be sent with the WebSocket protocol: Data (meaning actual information the application wants to send, either binary or

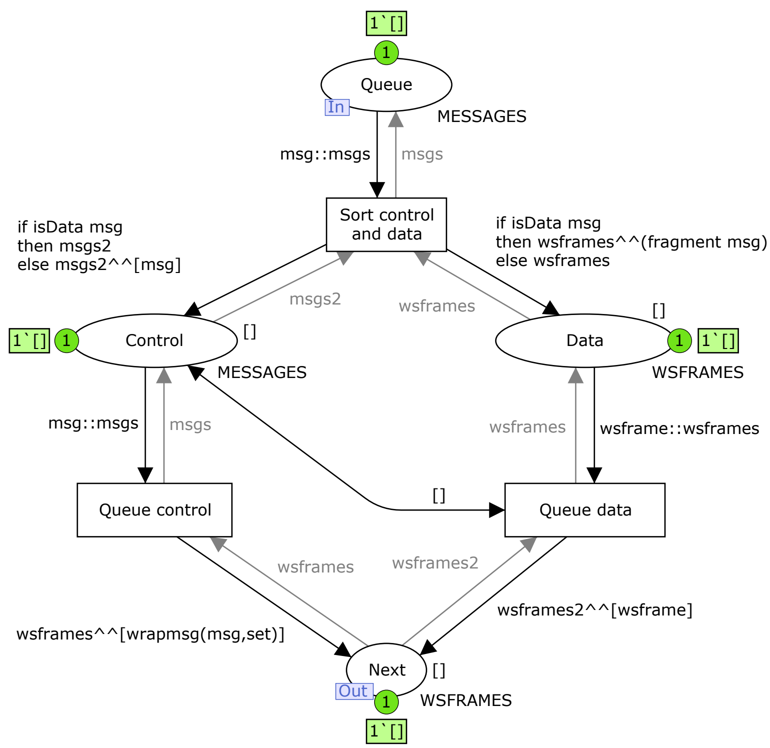


Figure 2.8: Fragment And Queue submodule

textual) and Control (ping, pong and close).

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 (Listing 2.12). Data frames with long payloads should be fragmented. This is taken care of by the `fragment` function, which uses an inner recursion to split the payload. Both of these functions employ the `wrap` function (Listing 2.12) to produce `WSFRAME` tokens, and therefore have to be declared after it.

After Sort control and data, the WSFRAME tokens are queued one by one into the data queue or the control frame queue depending on its type. 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 between fragmented data frames.

Unwrap and Receive Module

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 (bottom) can be processed in three different manners:

- The first one is modeled by the Receive transition, and happens if the connection is in the `CONN_OPEN` state (checked on the arc from place Connection Status) 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. This is modeled by the arc expression on the arc going to Send Pong.

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) =
    (* Recurse over s, adding ws frames to
    accumulator acc and returning it when
    remaining s will fit max fragment size,
    and manage opcode and fin bit *)
    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;
```

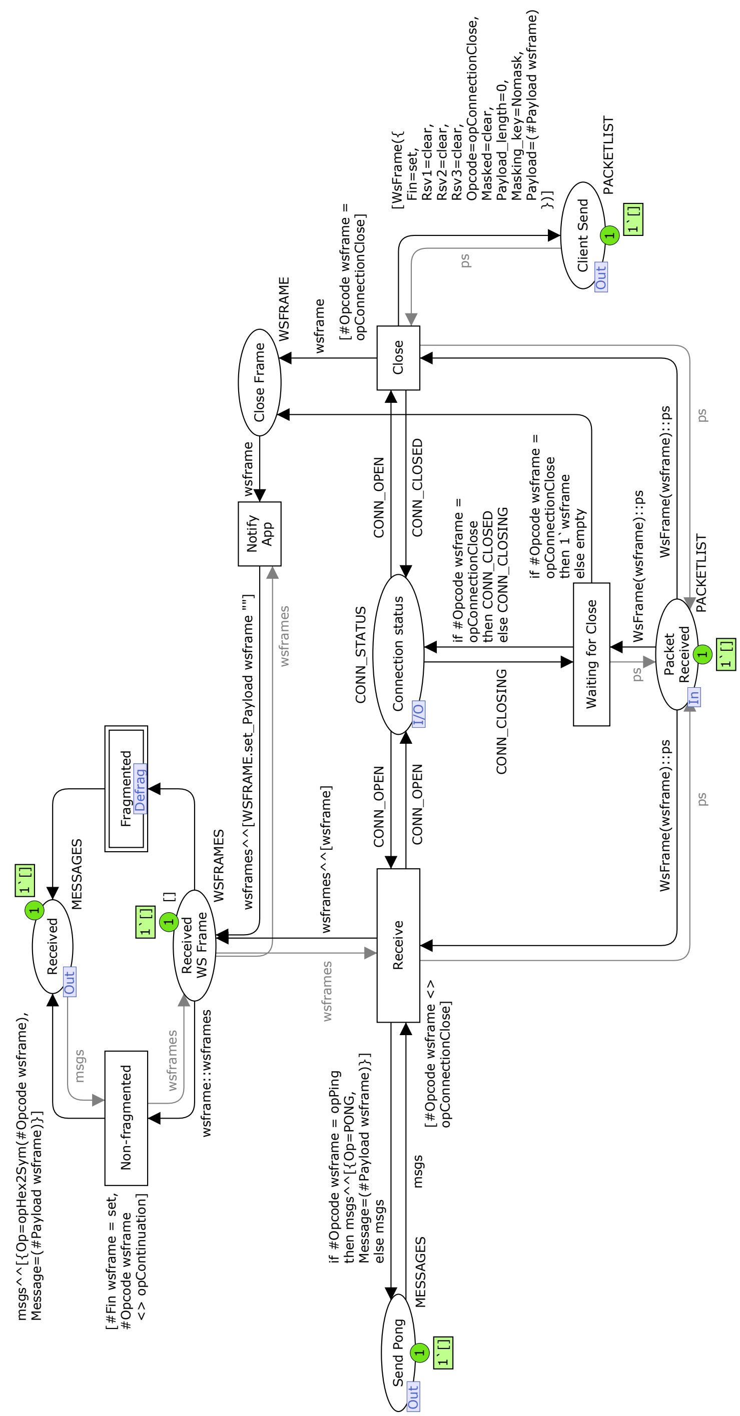


Figure 2.9: Unwrap And Receive submodule

- The second manner a frame can be processed is modeled by the Close transition, and happens if the connection is in the `CONN_OPEN` state (checked on the arc from Connection Status) and the frame is a close frame (checked in the guard of the Close transition). A close frame is created and set to be sent as response, and the connection state is changed to `CONN_CLOSED`, since the client has both received and sent a close frame. Note that the packetlist from Client Send bound to the variable `ps` 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 case happens if the connection state is in `CONN_CLOSING`, which means a close frame has been sent and we are waiting for a reply. This is modeled by the Waiting for Close transition. 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 situation described above will queue the received close frame in the Close Frame place to notify the application, but the payload is stripped through the Notify App transition's outgoing arc expression 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` and placed as a token on place Received. If either or both of those conditions are not true, this is part of a fragmented message and is processed in the Defrag submodule associated with the Fragmented substitution transition.

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 not set and the `Opcode` is not continuation, this is the first frame in the series. A new `MESSAGE` is created in the Buffer place with the opcode and payload from the WebSocket frame.

If the `Fin` bit is not set and the `Opcode` is continuation, this frame belongs in the middle of the sequence. The payload is appended to the `MESSAGE` in the Buffer place using the `append()` function (Listing 2.13).

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

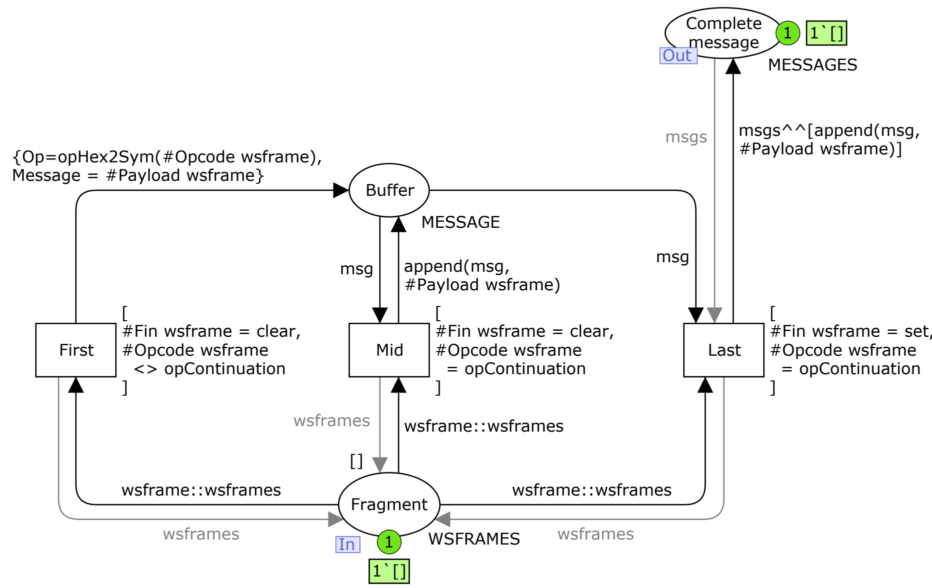


Figure 2.10: Defrag Module

Listing 2.13: append

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

final Complete message place.

Note that since the WebSocket protocol does not allow fragmented messages 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 interleaving can be defined by subprotocols, but this is not relevant to this model.

2.2.4 Connection Module

Fig. 2.11 shows the connection layer as modeled by the Connection module. 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

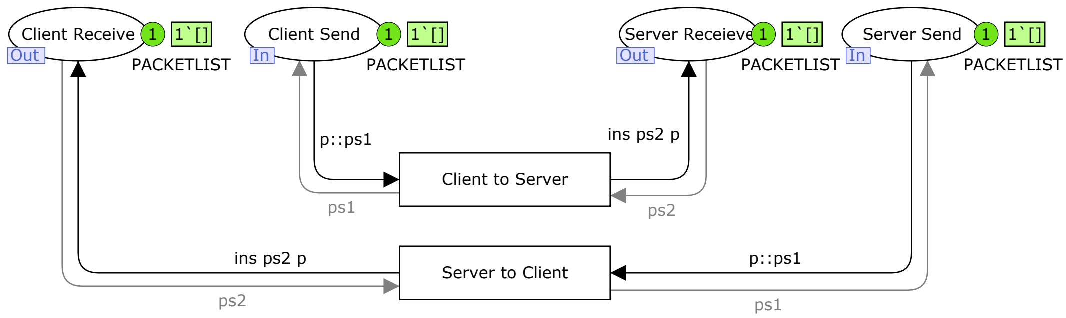


Figure 2.11: The Connection module

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;
```

is assumed to be taken care of by TCP as the WebSocket protocol specifies, hence is not necessary to model TCP in detail to capture the operation of the WebSocket protocol.

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.

2.2.5 The Server WebSocket Module

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 WebSocket module and Server WebSocket module both have instances of the same submodules, so that editing a submodule model affects both parent

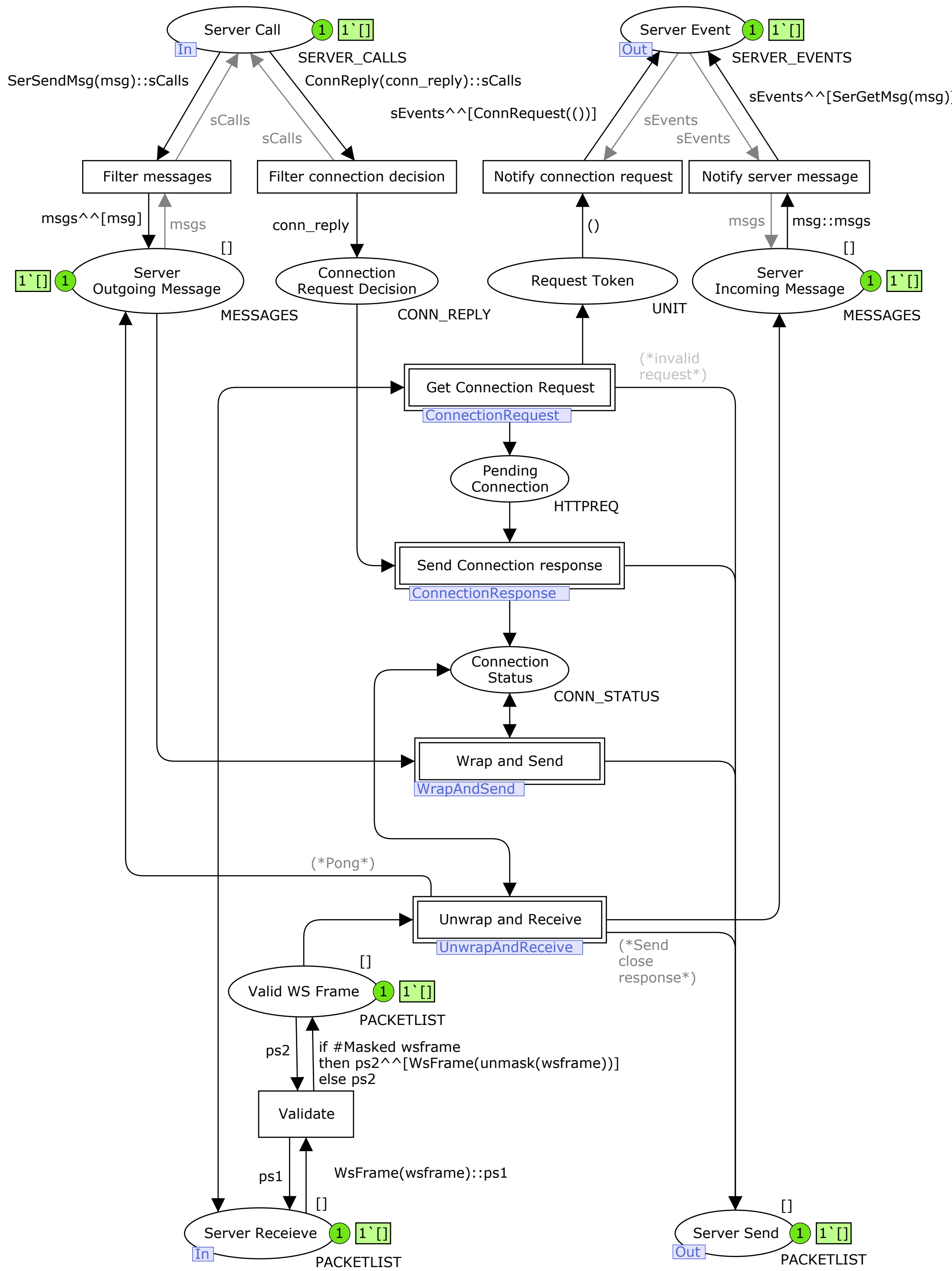


Figure 2.12: The Server WebSocket Module

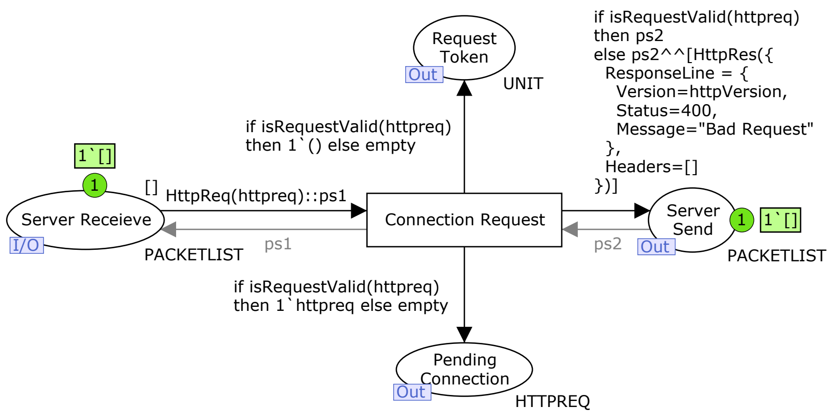


Figure 2.13: Get Connection Request

Listing 2.15: isRequestValid

```
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
```

modules, while during a simulation they can have different states.

Get Connection Request Module

This module is shown in Fig. 2.13, and is a very abstract representation of how connection requests are received. It checks the incoming request using `isRequestValid`, shown in Listing **??**. If the incoming request is valid, a simple `UNIT` token is sent to the application via the Request Token place. In an implementation, much more data might have to be provided to the application, for example IP address and possibly authentication information from headers, but for this model it is sufficient to model that some data is being sent, while the details are abstracted away.

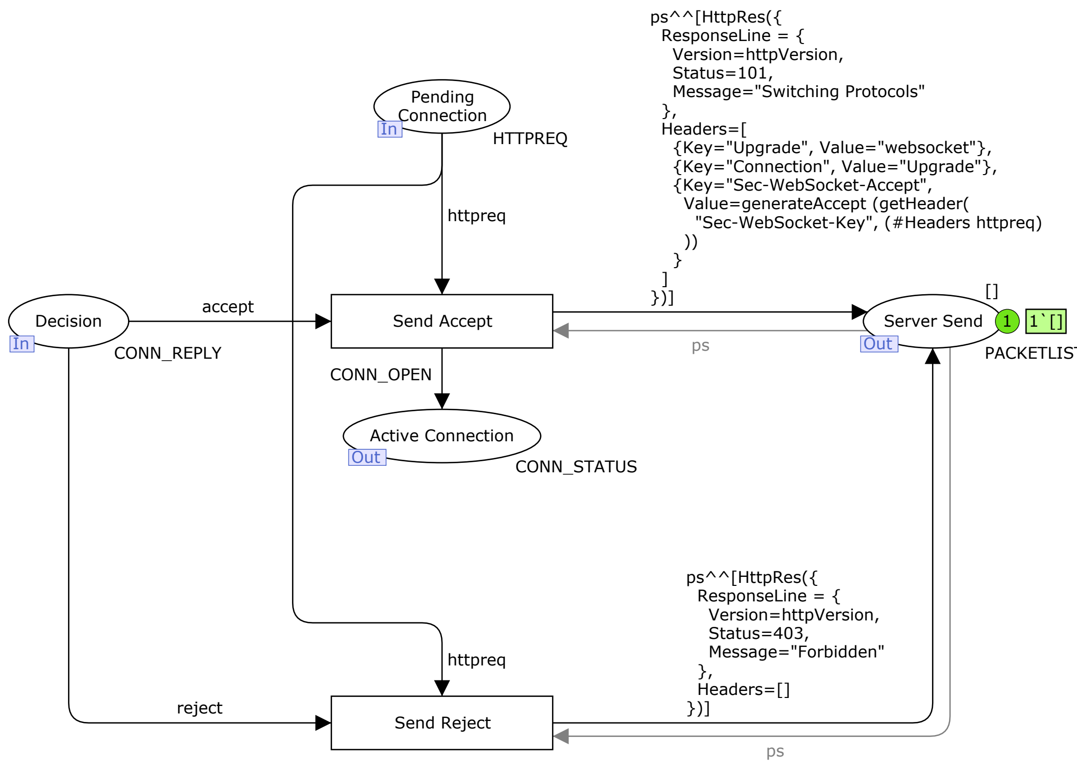


Figure 2.14: Send Connection Response

Send Connection Response Module

This module is shown in Fig. 2.14. When the Server Application has decided what to do with an incoming connection, it will send a `CONN_REPLY` to the Server Websocket module. 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 based on the Sec-WebSocket-Key header form the client, and is done to prove this is a WebSocket sever. The header is generated with the `generateAccept` function which was explained in Listing 2.10.

2.2.6 Server Application

The Server Application has three tasks: Accept or reject incoming connections, and sending and retrieval of data. The To send place has three

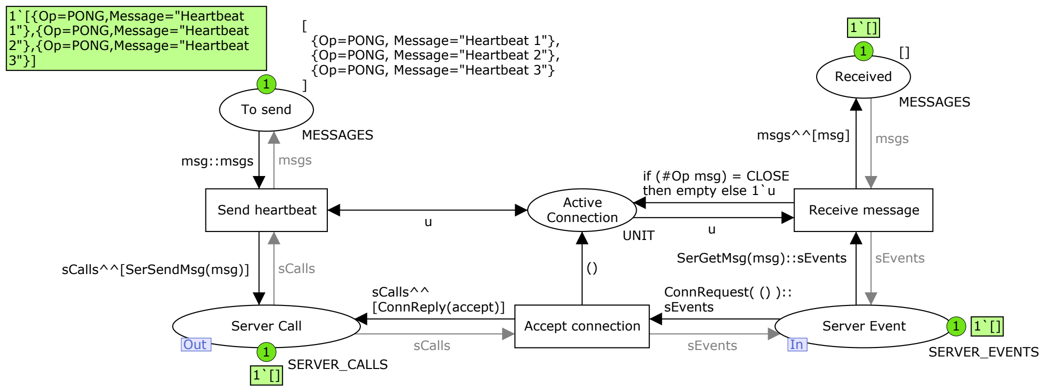


Figure 2.15: The Server Application Module

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 protocol layer 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 a CPN model, and in our case the communication protocol being modeled.

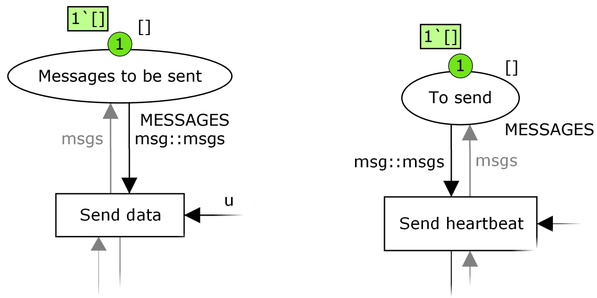
3.1 State Spaces

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.

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 display 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 written in CPN ML 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, the analysis of certain properties becomes simpler and faster, such as determining reachability, determining 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 of the state space 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 on a place, 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 types 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 the choice is non-deterministic, this construct is not 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 obtained, since it is impossible for the state space genrator of CPN Tools to make sure that 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.

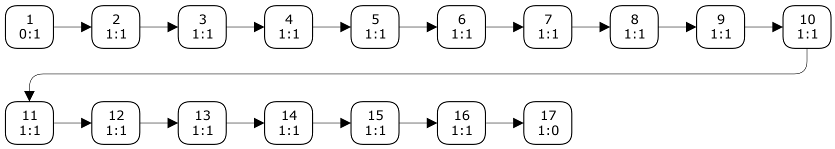


Figure 3.2: State space fpr configuration with no messages

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 CPN model.

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

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

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

```
  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
   [...]
```

which means this graph is a tree (in fact it consists of just a single path as was shown in Fig. 3.2)

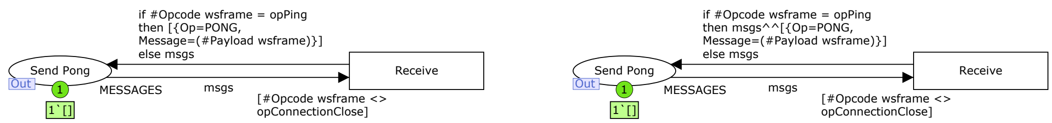
The `Secs` field shows that it took less than one second to calculate this state space, while the `Status` field specifies whether the report is generated from a partial or full state space. In this case, the state space is fully generated.

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 of the state space report 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.

Many places have 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 (i.e. the number of elements in the list) that are in each place, because technically there is just one token there:



(a) Before (b) After

Figure 3.3: Problem with the Pong reply modeling

the list itself. However, it quickly lets us know if something is wrong as well, since any values other than 0 or 1 here would indicate a problem.

In fact, an error in the model was discovered this way, in the Unwrap and Receive module, where the Pong reply was creating a new list instead of appending to the old one in outgoing messages. This caused the Client Outgoing Messages place to have 2 tokens at once. Fig. 3.3 shows the location of the error before and after fixing it.

The Best Upper Multi-set Bounds will show for each place a set of every token that exists in that place at some point in the state space. 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.

The Best Lower Multi-set Bounds is the opposite, and shows the smallest set of tokens that exists at any point in the state space. This is either the empty list `[]` or simply `empty` for all places.

3.2.3 Home Properties

This section of the state space report shows all home markings. A home marking is a marking that can always be reached no matter where we are in the state space. 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 in Fig. 3.2 that there is only one possible sequence of transition occurrences to establish a connection. CPN Tools allows us to import the state from this node into the editor, and by manually inspecting the various markings of the model to verify the desired state of an open connection exists with no side effects, we can confidently say that the model works correctly with this configuration.

```
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
   [...]
```

```
Home Markings
   [17]
```

```
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
```

3.2.4 Liveness Properties

This section of the state space report describes so-called liveness properties of the state space. Some of the transitions have been omitted for readability.

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 from any reachable marking we 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 sincethe state space does not contains cycles. We will not go into detail about this, and instead refer to [JK09], chapter 7 for more information.

3.2.5 Larger Configurations

Above we have considered the simplest possible configuration of the WebSocket protocol model.Below we present the results from considering more complex configurations. For each configuration we only present select ele-

Listing 3.2: One message

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

Home Markings
   [29]

Dead Markings
   [29]

Live Transition Instances
   None

   No infinite occurrence sequences.
```

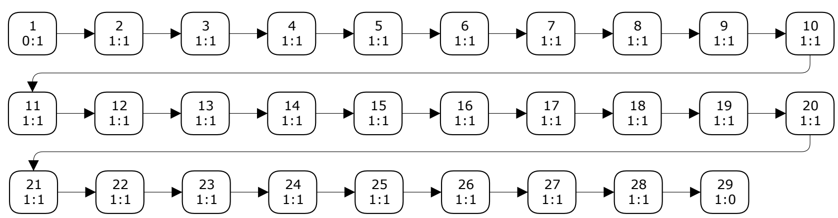


Figure 3.4: One message

ments from the state space report

Configuration 1: 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"}`.

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, and by manual inspection of state 29 we verified that this state had the desired outcome without side effects (connection open and message received).

Listing 3.3: One ping then one message

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

Home Markings
   [475]

Dead Markings
   [475]
```

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

Home Markings
   None

Dead Markings
   [512,513]
```

Configuration 2: 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:

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. The home marking 475 was again manually verified as correct.

Configuration 3: 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.

Note that we now no longer have any home marking, and instead have

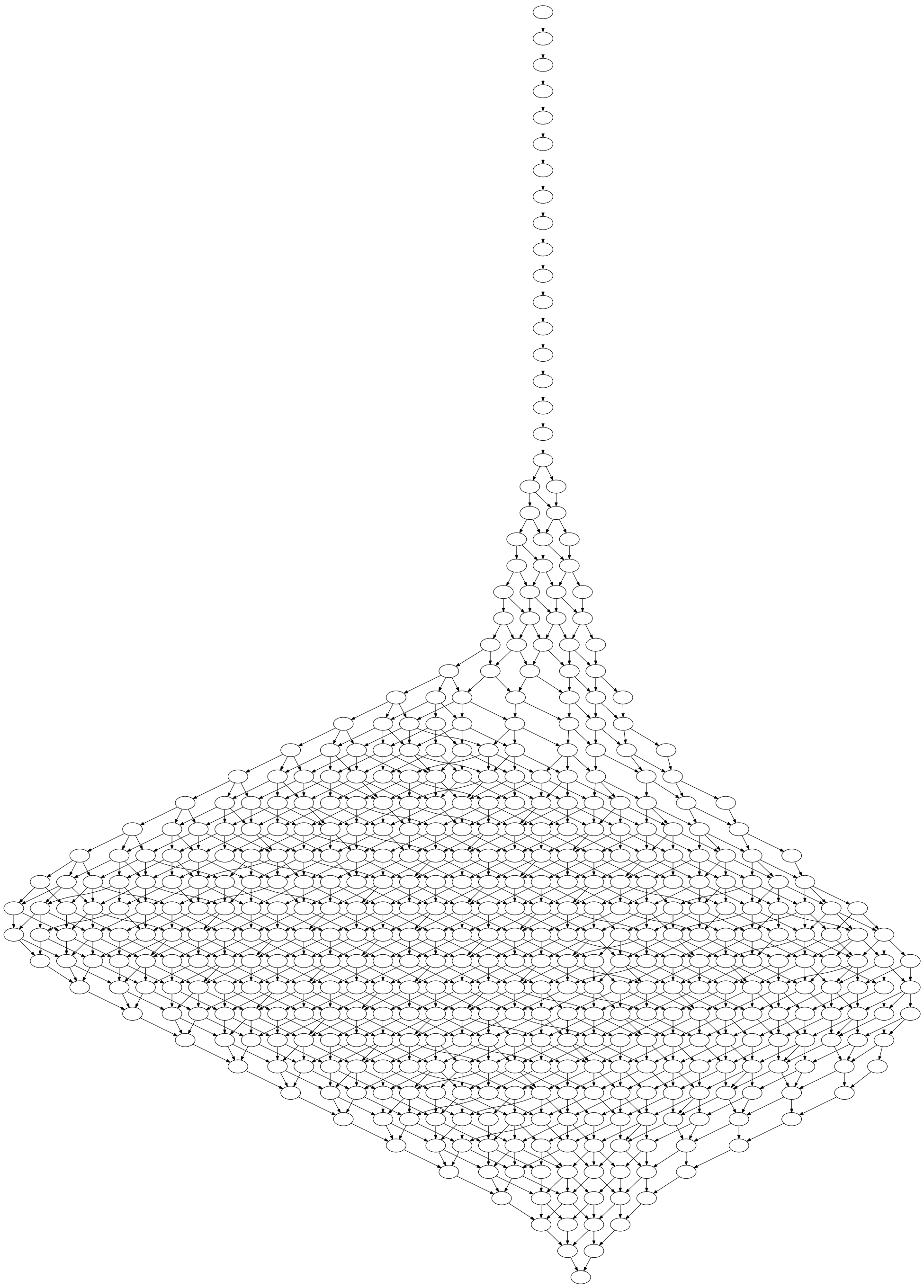


Figure 3.5: State space for one ping, one message configuration

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

 Home Markings
   [813]

Dead Markings
   [813]
```

```
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. "})]
```

two dead markings, representing the two possible orderings of the two messages. In this situation it is possible to use the `HomeSpace` query to see if the markings belong to a so-called home space, meaning for any reachable marking in the state space, it is possible to reach at least one of the markings in the home space. The full query used is `HomeSpace(ListDeadMarkings())`, and it returned `true` when executed in this instance.

Configuration 4: 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. "}`

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. We are also able to inspect every other possible

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

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

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

combination, and can thus confirm that messages are being split correctly.

Configuration 5: Ping, text, close

The client is now set to send three messages of different types: A ping (which will solicit a pong), a short text string, and a close message. With three messages, state space calculation becomes noticeably time-consuming.

Here we have six dead markings. In three of the cases, the close is sent before the text. For each of those three, 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. We also verified that this set is a home space.

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

3.2.6 Summary

State space analysis has proved to be very useful by uncovering problems with the WebSocket CPN model, and effectively allowing us to conclude that the model is now valid. All but two of the transitions are enabled at some point; the two exceptions are Fail in the ClientApplication module and Send Reject in the ConnectionResponse module, which correctly are never enabled. This means we have full coverage of the model; no part of it is unused and unaccounted for.

Chapter 4

Technology and Foundations

One of the fundamental decisions that had to be made for the wwork conducted in this thesis was whether to base Pragma/CPN on an existing platform or to create a new platform from scratch. In this chapter we describe the reasoning behind the choices made, taking into account the requirements stated in Chapter 1, and we give an overview of the technologies that have been selected.

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 modules as nodes, and places, transitions and arcs as child nodes of pages with properties describing how they are connected in the CPN model being represented. Simple tree editors are a feature of most GUI toolkits, but we realised early that creating a CPN model editor from scratch would take much longer than adapting an existing platform.

There already exist 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 Pragma/CPN 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 a library called Access/CPN [WK09] (a part of the CPN Tools project) can parse CPN Tools files, and represent the model using Java objects.

The ePNK framework [Kin11], an extendable framework for working with Petri Nets in a graphical manner, and that makes it possible to specify

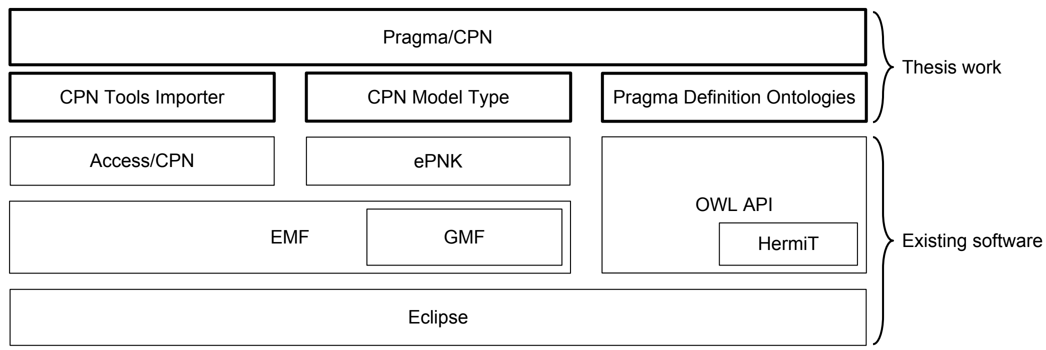


Figure 4.1: Application Overview Diagram

your own Petri Net type. It is built on the Eclipse Modeling Framework (EMF) [SBMP08] (which Access/CPN is also built on).

A central part of representing the CPM model is also to devise a means for representing the code generation pragmatics that can be attached to CPN model elements. It was suggested to take an ontology-based approach, and we selected the reference Java implementation OWL API [HB09] to define and work with ontologies, together with the HermiT ontology reasoner [SMH08].

Fig. 4.1 shows the different elements that make up Pragma/CPN. The elements with a thick border are the ones created as part of this thesis, while the rest represent existing solutions used and built upon. These will be described in the following sections.

4.2 The Eclipse Platform

The Eclipse Rich Client Platform (RCP) [dW04] is an open source, cross-platform, polyglot integrated development tools platform. Its plugin framework makes it highly extendable and customisable, and especially makes it easy for developers to quickly create tools and solutions ranging from small custom macros, to advanced editors, to complete applications. The Eclipse Platform is a part of the Eclipse Project, a community for incubating and developing open source projects.

The architecture of the Eclipse RCP shown in Fig. 4.2. We will give a short explanation of each element:

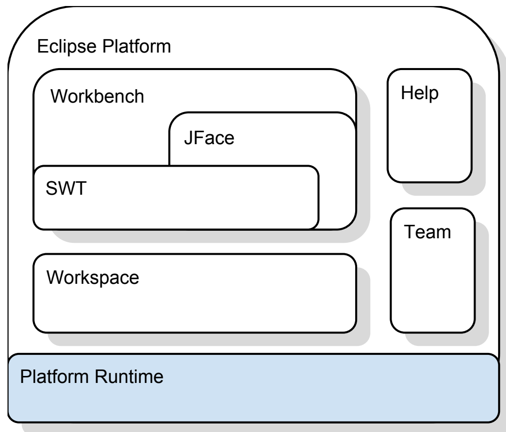


Figure 4.2: The Eclipse RCP

- At the bottom of this we have the Platform Runtime, based on the OSGi framework [All07], which provides the fundamental plugin architecture.

- The Standard Widget Toolkit (SWT) gives efficient and portable access to the user-interface facilities of the underlying operating systems on which it is implemented. JFace is a user interface framework built on SWT. The Workbench is built using these two frameworks to provide a scalable multi-window editing 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 and handles installation of new plugins. Together these plugins form a basic generic IDE.

Additional plugins exist that build on this to . Plugins are the building blocks of Eclipse, and there exists a wide range of plugins that can specialise the environment for a particular programming language and/or type of application, and 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 using specialised tools in Eclipse (that themselves are plugins) and placing 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 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 CPN type. It uses EMF and GMF (described later) 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 Pragma/CPN:

- It saves models using the ISO/IEC 15909-2 [ISO11] standard file format Petri Net Markup Language (PNML),
- It is currently actively developed by researchers in the model-based software engineering research group at the Danish Technology University,
- 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 for CPN models, provided through GMF.

Below we provide a description of key concepts in ePNK, and for this purpose we will use the following terminology:

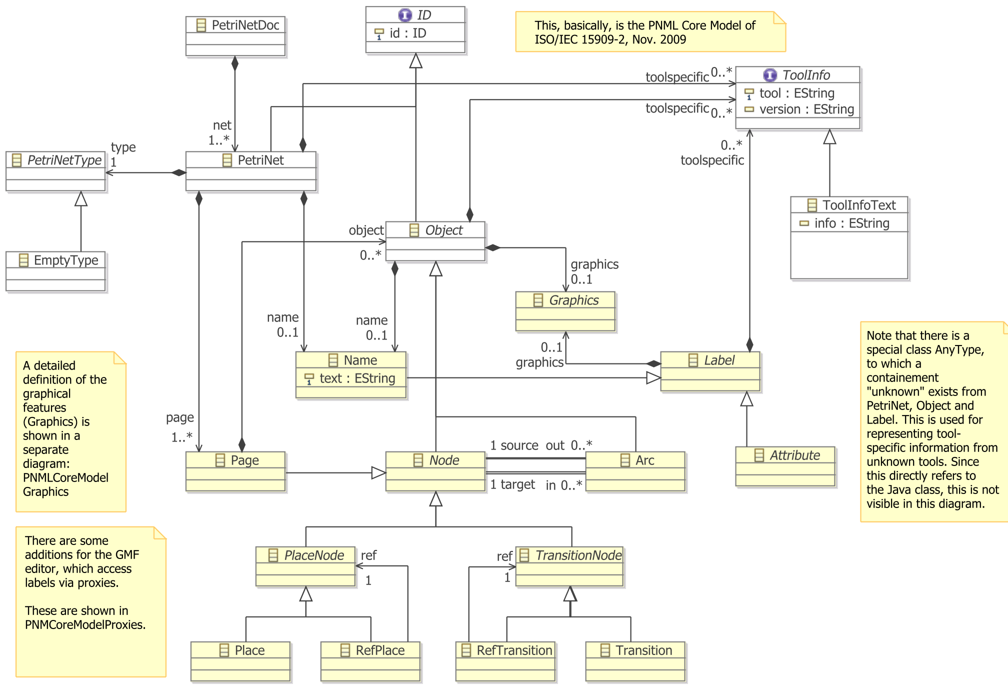


Figure 4.3: PNML Core Model

Type - An ePNK Model Type Definition is an extension to ePNK that declares and implements a particular Petri Net extension.

Type Model - The EMF model defining a Type (as defined above). In MDA terms, it corresponds to the metamodel.

Model - A model created by a user as an instance of a Type Model.

A standard Petri Net Model created in ePNK is initially only defined by the PNML Core Model Type. Its Type Model is shown in Fig. 4.3. This Type is intended to be generic, and only defines the basic classes that most Petri Net variants contain, like Pages (modules), Places, Transitions and Arcs. The only constraint defined is that an arc must go between two nodes on the same page.

The user can choose to extend a Model with features and capabilities of a more advanced Petri Net Type. This is done by adding a Type as a child of the Petri Net Document node in the Model. 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 Type 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 Net Graphs (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 the CPN ML syntax from CPN Tools, and for our prototype, these structured labels are not necessary with regard to annotations, as there does not exist any pragmatics that depend on this level of detail yet. 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 outside the scope of this thesis; simple string representation is sufficient.

Our decision was therefore to develop our own Petri Net type that matches the type supported by CPN Tools, and we present this Type as part of the next chapter.

4.4 Eclipse Modeling Framework

EMF is an open source framework for Model Driven Architecture (MDA) in Java. It is an Eclipse plugin that is part of the Eclipse Platform. MDA is an industry architecture proposed by the OMG that aims to unify some of the industry best practices in software architecture, modeling, metadata management, and software transformation technologies that allow a user to develop a modeling specification once and target multiple technology implementations by using precise transformations/mappings.

EMF is an example of the use of MDA to enable creation of a UML model representation of a tool or application and to use this model to automate (some or all) of the Java interface, implementation, as well as any XML serialization for the modeled objects. Other generated artifacts include a set of adapter classes that enable viewing and command-based editing of models, and a basic editor. This serves as the foundation used to build ePNK, including CPN model editing and PNML serialisation.

4.4.1 Graphical Modeling Framework

GMF builds on EMF to provide tools to implement more advanced graphical viewing and editing of models. It works by creation of model transformations that use the metamodels created with EMF to generate implementations of graphical views and editors that plug in to the Eclipse workbench. It is used in ePNK to generate the graphical diagram editor.

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 (files created with CPN Tools) and represent them as an EMF-model. The .cpn files saved by CPN Tools are XML-based, which makes them easy to parse. However, having an existing solution for this is preferrable, as we can rely on it to keep up to date with new versions of CPN Tools.

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

4.6 Ontologies: OWL 2 and OWL API

Ontologies are a way to formally represent and structure information and knowledge within a domain as a set of concepts. 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 in the context of the Semantic Web [BLHL01], 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 use a reasoner engine to explore and describe the domain, draw conclusions, 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. We use HermiT [SMH08] as a reasoner engine. The use of ontologies in the context of pragmatics is described in detail in the next chapter.

Chapter 5

Analysis and Design

After picking the technologies described in the previous chapter, since many of the componets have Eclipse in common, it was an easy decision to develop Pragma/CPN as an Eclipse plugin. This also let us centralise all our development in Eclipse.

Our analysis of available software solutions and platforms showed that the basic p that are required to develop the desired code generation pragmatic framework are available as part of the Eclipse eco system. It is therefore natural to develop Pragma/CPN as an Eclipse Plugin and base all software develpoment in the context of this thesis on the Eclipse Platform.

5.1 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.

5.2 The CPN Ontology with Pragmatics

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

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

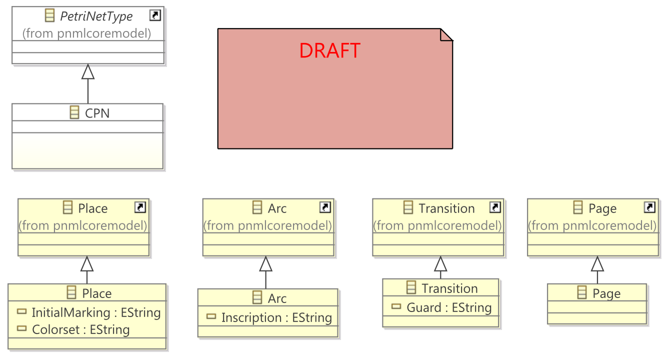


Figure 5.1: CPN model type diagram

TODO: Beskrive ontologiene

An ontology document can import other ontologies

5.3 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 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

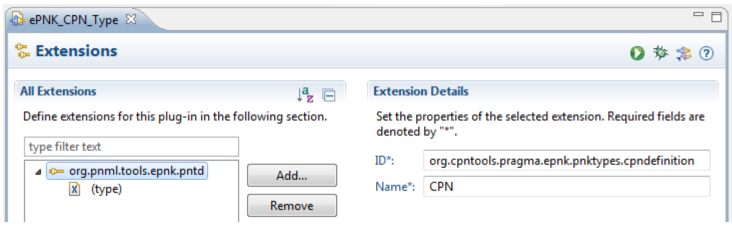


Figure 5.2: Plugin Manifest part 1

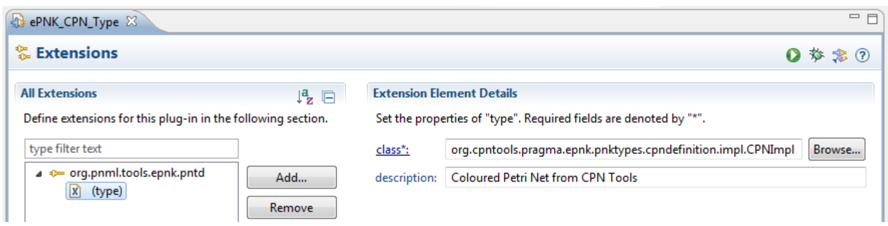


Figure 5.3: Plugin Manifest part 2

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 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

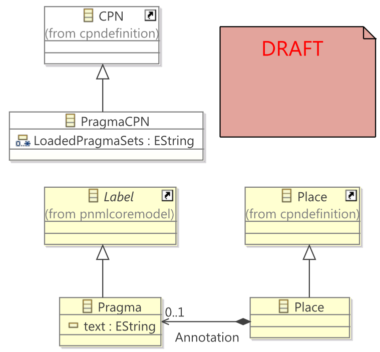


Figure 5.4: Annotated CPN model type diagram

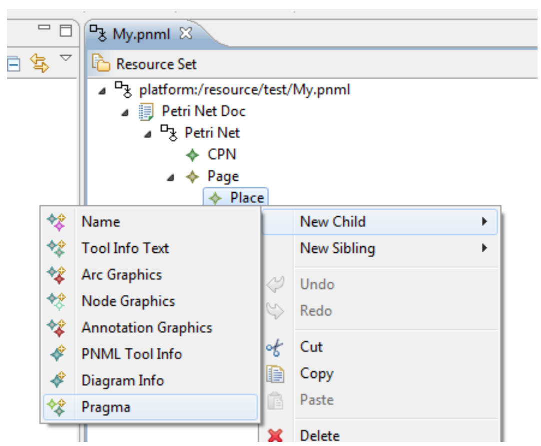


Figure 5.5: Adding Pragma as child to Place

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.

5.4 Importing from CPN Tools

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

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

Sett inn navn

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

5.5 Creating annotations

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

5.6 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

5.6.1 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

[All07] O.S.G. Alliance. Osgi service platform, core specification, release 4, version 4.1. *OSGi Specification*, 2007.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[dW04] J. desRivieres and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[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.

[Gro11] Object M. Group. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.4.1. Technical report, Object Modelling Group, August 2011.

[HB09] M. Horridge and S. Bechhofer. The owl api: a java api for working with owl 2 ontologies. *Proc. of OWL Experiences and Directions*, 2009, 2009.

[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.

[Ken02] S. Kent. Model driven engineering. In *Integrated Formal Methods*, pages 286–298. Springer, 2002.

[Kin11] E. Kindler. epnk: A generic pnml tool-users' and developers' guide: version 0.9. 1. Technical report, DTU Informatics, Building 321, Kgs. Lyngby, Denmark, 2011.

[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/`.

[rup] Ibm rational unified process.

[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.

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[SMH08] R. Shearer, B. Motik, and I. Horrocks. Hermit: A highly-efficient owl reasoner. In *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008)*, pages 26–27, 2008.

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

[WK09] Michael Westergaard and Lars Kristensen. The access/cpn framework: A tool for interacting with the cpn tools simulator. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and Theory of Petri Nets*, volume 5606 of *Lecture Notes in Computer Science*, pages 313–322. Springer Berlin / Heidelberg, 2009.

[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

1.1 Top level module of the Kao-Chow model 4

2.1 Sequence Diagram of the WebSocket protocol 8
2.2 Overview module of the WebSocket CPN model 9
2.3 The Client Application Module 13
2.4 The Client WebSocket Module 18
2.5 New Connection submodule 22
2.6 Process Response submodule 23
2.7 Wrap And Send submodule 24
2.8 Fragment And Queue submodule 25
2.9 Unwrap And Receive submodule 28
2.10 Defrag Module 30
2.11 The Connection module 31
2.12 The Server WebSocket Module 32
2.13 Get Connection Request 33
2.14 Send Connection Response 34
2.15 The Server Application Module 35

3.1 Configuration: No messages 38
3.2 State space fpr configuration with no messages 40
3.3 Problem with the Pong reply modeling 42
3.4 One message 45
3.5 State space for one ping, one message configuration 47

4.1 Application Overview Diagram 52
4.2 The Eclipse RCP 53
4.3 PNML Core Model 55

5.1 CPN model type diagram 60
5.2 Plugin Manifest part 1 61

5.3 Plugin Manifest part 2 61
5.4 Annotated CPN model type diagram 62
5.5 Adding Pragma as child to Place 62

Listings

2.1 Overview colour sets 11
2.2 Simple Colourset Variables 12
2.3 Client Application Variables 14
2.4 The parseUrl function 15
2.5 HTTP colour sets 19
2.6 WebSocket colour sets 20
2.7 WebSocket Module Variables 21
2.8 Masking functions 21
2.9 httpReqFromUrl 22
2.10 isResponseValid 23
2.11 isData 26
2.12 wrap wrapmsg and fragment 27
2.13 append 30
2.14 unmask 31
2.15 isRequestValid 33
3.1 Fixed masking key 40
3.2 One message 45
3.3 One ping then one message 46