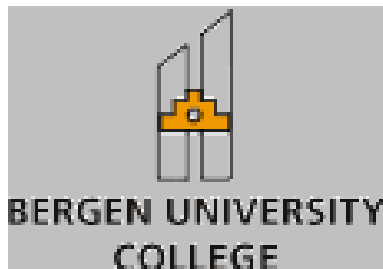


# Design and Evaluation of a Framework for Annotating Coloured Petri Net Models with Code Generation Pragmatics

Mikal Hitsøy Henriksen

---

Master's Thesis in Software Engineering



Department of Computing, Mathematics and Physics, Bergen University College  
Department of Informatics, University of Bergen  
Norway

June 1, 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 designed 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 using an ontology-based approach yields a robust and expressive environment for classifying code generation pragmatics, which fits MDE's pattern of model checking and validation. 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Model Driven Software Engineering . . . . .	1
1.2	Thesis Aims and Results . . . . .	4
1.3	Related Work . . . . .	6
1.4	Thesis Organisation . . . . .	6
<b>2</b>	<b>The WebSocket Protocol CPN Model</b>	<b>9</b>
2.1	The WebSocket Protocol . . . . .	9
2.2	The WebSocket CPN Model . . . . .	10
2.2.1	Overview . . . . .	11
2.2.2	Client Application . . . . .	14
2.2.3	The Client WebSocket Module . . . . .	19
2.2.4	Connection Module . . . . .	32
2.2.5	The Server WebSocket Module . . . . .	33
2.2.6	Server Application . . . . .	37
<b>3</b>	<b>State Space Analysis of the WebSocket Protocol</b>	<b>39</b>
3.1	State Spaces . . . . .	39
3.1.1	Strongly Connected Component Graph . . . . .	39
3.1.2	Application of State Spaces . . . . .	40
3.1.3	Visualisation . . . . .	42
3.2	State Space Report . . . . .	42
3.2.1	Statistics . . . . .	42
3.2.2	Boundedness Properties . . . . .	43
3.2.3	Home Properties . . . . .	44
3.2.4	Liveness Properties . . . . .	46
3.2.5	Larger Configurations . . . . .	47
3.2.6	Summary . . . . .	52

<b>4</b>	<b>Technology and Foundations</b>	<b>55</b>
4.1	Representing CPN Models . . . . .	55
4.2	The Eclipse Platform . . . . .	56
4.3	ePNK: Petri Net modeling framework . . . . .	58
4.4	Eclipse Modeling Framework . . . . .	60
4.4.1	Graphical Modeling Framework . . . . .	61
4.5	Access/CPN: Java interface for CPN Tools . . . . .	61
4.6	Ontologies: OWL 2 and OWL API . . . . .	61
<b>5</b>	<b>Analysis, Design and Implementation</b>	<b>63</b>
5.1	The CPN Ontology with Pragmatics . . . . .	63
5.1.1	The Ontology Plugin . . . . .	71
5.2	The Annotated CPN model type for ePNK . . . . .	71
5.2.1	CPN Type Model . . . . .	72
5.2.2	Code Generation . . . . .	73
5.2.3	Constraints . . . . .	73
5.2.4	Annotated CPN Type . . . . .	74
5.2.5	Persisting Pragmatics Sets in the Model . . . . .	77
5.2.6	Adding Pragmatics Sets . . . . .	78
5.3	Importing from CPN Tools . . . . .	83
5.4	Creating Pragmatic Annotations . . . . .	83
5.4.1	Providing a Dynamic Context Menu . . . . .	84
5.4.2	Determining Appropriate Pragmatics . . . . .	85
5.5	Defining Model Specific Pragmatics Sets . . . . .	91
5.6	Assessment of Requirements . . . . .	94
<b>6</b>	<b>Evaluation</b>	<b>95</b>
6.1	Annotating the WebSocket CPN Model . . . . .	95
6.1.1	The Kao-Chow Protocol Model . . . . .	97
6.2	Performance . . . . .	97
6.3	Limitations . . . . .	99
<b>7</b>	<b>Conclusion</b>	<b>101</b>
7.1	Future work . . . . .	101
7.2	Acknowledgments . . . . .	102
<b>A</b>	<b>Downloading Pragma/CPN</b>	<b>103</b>

# 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 [Kru04]). 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 at the time it was conceived.

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.

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 is defined with a model to describe concepts of the domain, along with associated semantics and constraints. This is often termed a metamodel
- Transformation engines and generators, that process models to produce various artifacts in an automated manner. Examples of such artifacts include documentation, deployment descriptions, alternative representations, and source code ranging from system skeletons to complete, deployable products.

To define a DSML, a language is needed with which to write the meta-model definition. This language is itself defined with a model, hence termed the meta-metamodel, and needs to be capable of describing itself. There exist many such languages, including the Meta-Object Facility (MOF) [Met06] created by the Object Management Group (OMG) to provide the basis for metamodel definition in OMG's family of modeling languages. Fig. 1.1 shows the relationships between the abstraction layers using UML as an example DSML for the domain of object-oriented software systems. The layers have been termed M0 to M3 by OMG, although the number of layers is not rigid and can be different for other DSMLs.

One of the central arguments for MDE is automatic source code generation. Several advantages come from this: Documentation and implementation are synchronised, and boilerplate code and automatic testing is managed, thus enhancing quality, productivity, reliability, maintainability, portability and reusability.

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.



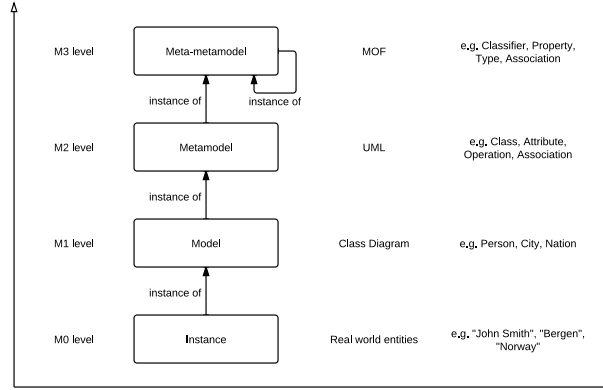


Figure 1.1: The OMG MOF Metamodelling Layers

There exist many extensions to the Petri Net formalism that define additional constructs or that change or enhance 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 includes a short introduction to CPN and state space exploration.

CPN Tools [RWL<sup>+</sup>03] is a popular graphical editor for working with CPN models, from construction and simulation, to analysis via state space exploration. CPN Tools uses the CPN ML language, an extension of Standard ML, to specify declarations and net inscriptions.

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.

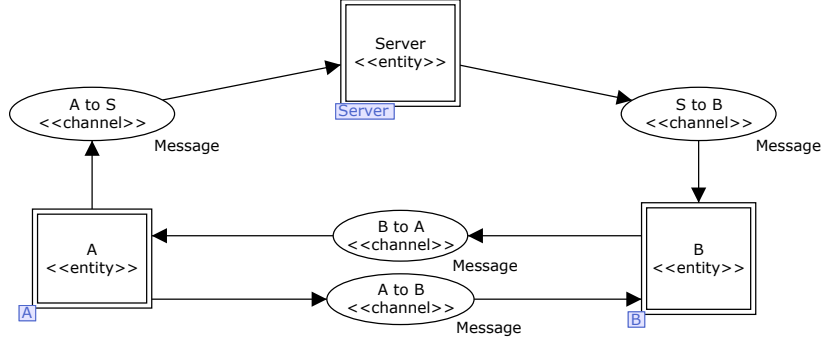


Figure 1.2: Top level module of the Kao-Chow model

## 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 for a particular protocol model.

Fig. 1.2 shows the top level module of the Kao-Chow CPN model from [FS11], with pragmatics enclosed in `<<and >>`. These pragmatics have been inserted manually using auxiliary labels and workarounds to avoid CPN ML syntax errors. 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.

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:

- Importing CPN Tools models. CPN Tools is one of the best applications available for constructing and analysing CPN models, and we wish to support models created in CPN Tools.
- Annotate model with pragmatics. The prototype should assist the user by providing only pragmatics that are applicable to the selected model element.
- Loading sets of domain-specific pragmatics to make available for the model. This is to reduce cluttering and potentially avoid performance reductions.
- Define set(s) 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 establishing 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 designed 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 and expressive semantics for classifying 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, Design and Implementation.** Gives a discussion and detailing of our solutions for the requirements described earlier. We start by showing the ontologies that define CPN and Pragmatics, as well as defining the General Pragmatics. We give details of the ePNK Petri Net Type Definition for Coloured Petri Nets and Annotated Coloured Petri Nets, and the mechanic for loading domain and model specific pragmatics into models. We show how models created with CPN Tools are imported and converted to ePNK models. We describe the algorithm used to determine appropriate pragmatics for a selected model element by using an ontology reasoner, and finally how the user can write their own model specific pragmatics sets.

**Chapter 6: Evaluation.** Here we evaluate Pragma/CPN by annotating various CPN models of protocols, including the WebSocket CPN model. We also discuss performance compared between the models, as well as our evaluation of the software technologies used.

**Chapter 7: Conclusion.** Contains a summary of the results of the thesis, as well as personal experiences, 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 to be used 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 concepts 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 of different types as defined by a list of opcodes, 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 constitutes a layer on top of TCP that does the following:

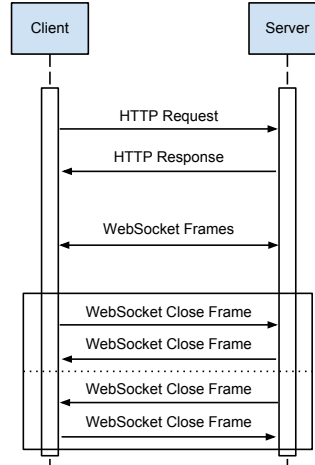


Figure 2.1: Sequence Diagram of the WebSocket protocol

- adds a Web "origin"-based security model for browsers
- adds an addressing and protocol naming mechanism to support multiple services on one port and multiple host names on one IP address;
- 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 the reliability that TCP provides
- 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 to 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.



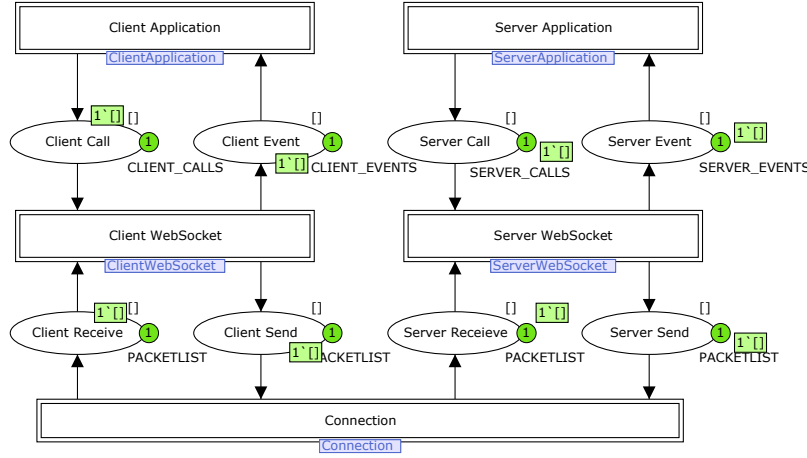


Figure 2.2: Overview module of the WebSocket CPN model

### 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 by circles and arrows respectively. The semantic details of arcs and arc expressions will be explained when presenting the **Client Application** module.

The text in the middle of each substitution transition is the name of that substitution transition, 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 for a particular substitution transition, 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 to-

gether 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 for easier recognition, but any CPN ML identifier would be 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 between the application and protocol 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, the use of which 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 example `/home/index.html`).

The Client Application can send tokens to the WebSocket layer as a

Listing 2.1: Overview colour sets

---

```

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

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

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

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

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

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

```

---

`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 in the overview and all connected submodules, and instead simply need to add the new color sets to the relevant color set declaration (for example `CLIENT_CALL`).

Both the client and server WebSocket modules send and receive `PACKET` tokens, a union of three color sets that will be explained later. `PACKETS` 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 CPN ML expressions containing variables. Variables are declared globally and can only be bound to values (colours) of the colour set they are defined for. A variable declaration has the form:

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

The set of variables declared for the simple colour sets 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 places. In order to talk about the enabling of a transition, a binding that

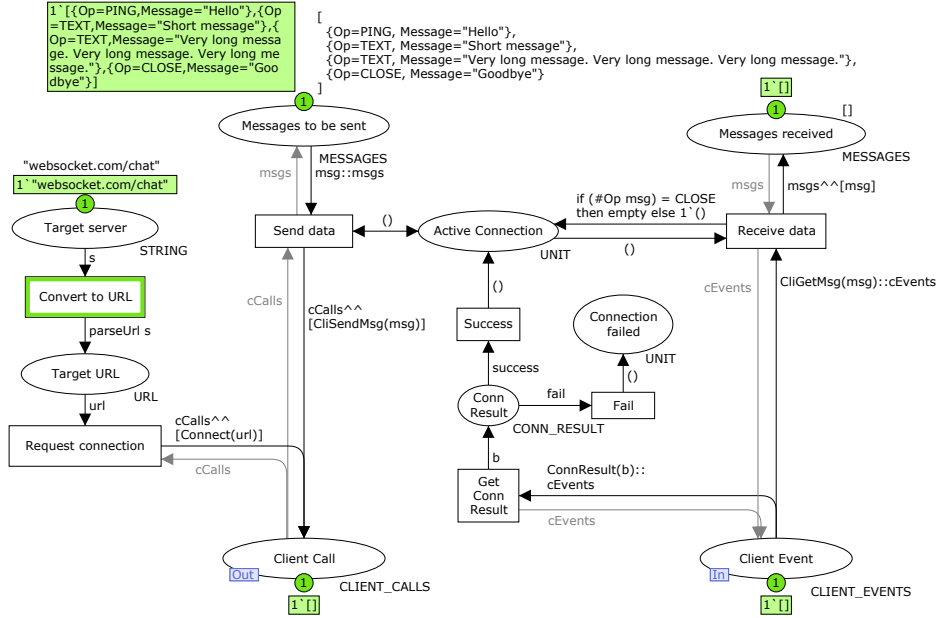


Figure 2.3: The Client Application Module

assigns 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 colour 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 colour set 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.

Variable declarations will be listed as they are encountered in the model.

Listing 2.3 shows the remaining variables used in the Client Application sub-module.

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 substring and 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 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

Under normal execution of a CPN model, tokens in a particular place can be consumed in any order. But for the WebSocket CPN model, a number of places 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 controlling this. Instead, to emulate the queue behaviour, we use a list of a colour set instead of using the actual colour set 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

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;
```

---

elements being denoted the tail of the list. To access these elements, we use the `::` list constructor like this: `head::tail`. Lists are usually processed in a tail-recursive manner. 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, thus placing the element at the end of the queue. 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 confusing since information logically moves in only 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 occurred (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 `success`, 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, technically the same as two arcs going in opposite 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 initial 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.

The new colour sets 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 HTTP Version, the response

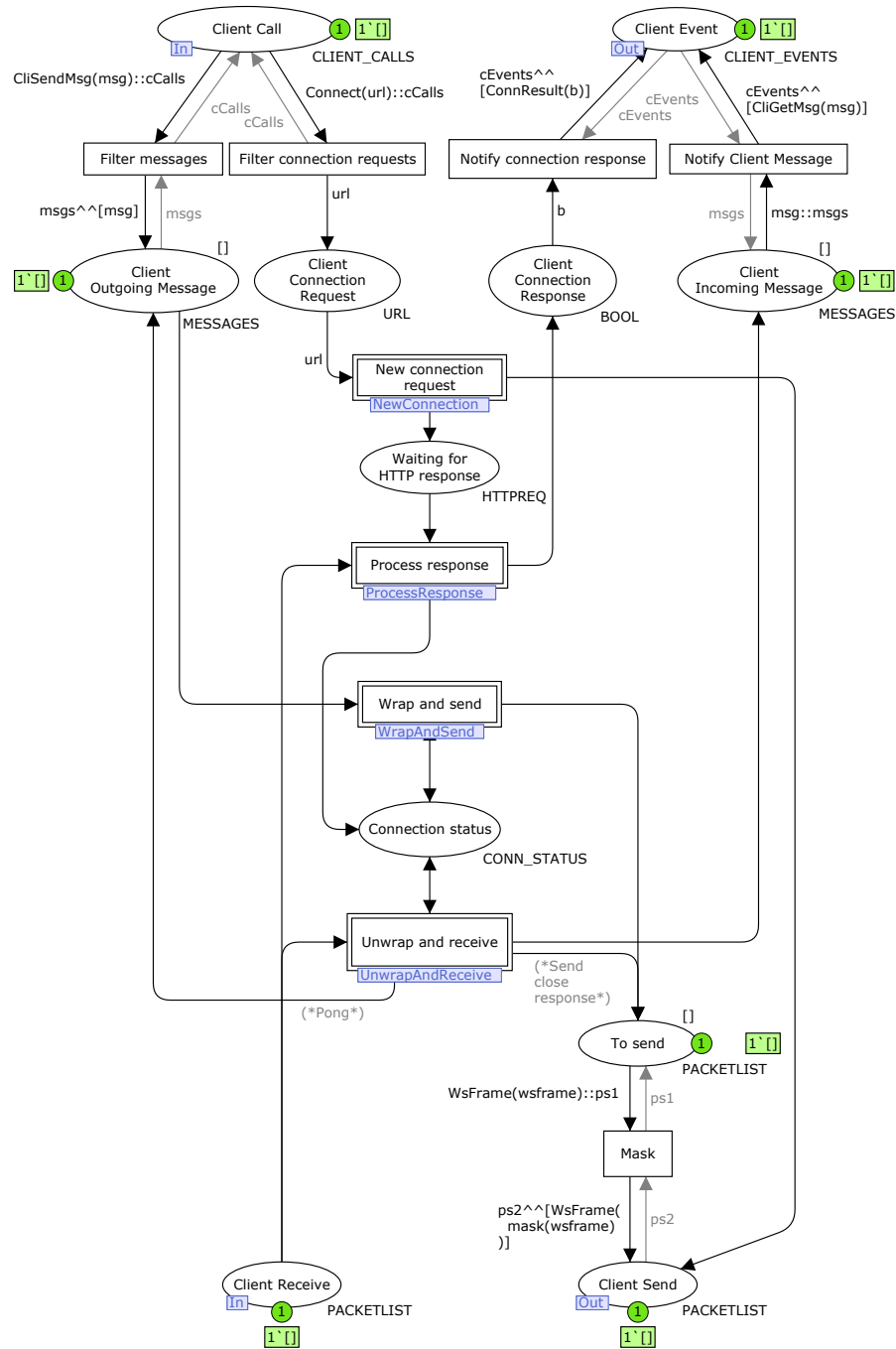


Figure 2.4: The Client WebSocket Module

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;

```

---

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 `BYTES`. 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).

All frames sent from a WebSocket client should mask their payload us-

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;

```

---

ing a four byte masking key. The motivation for this is to prevent potential network intermediaries like proxy servers from interfering with transmission of frames based on their content. Masking of frames is modeled to only set the masking bit and provide a masking key. A real-world implementation would also apply the mask to the payload according to the protocol specification, which 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, along with the `randMask` function it uses.

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;

```

---

## 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.

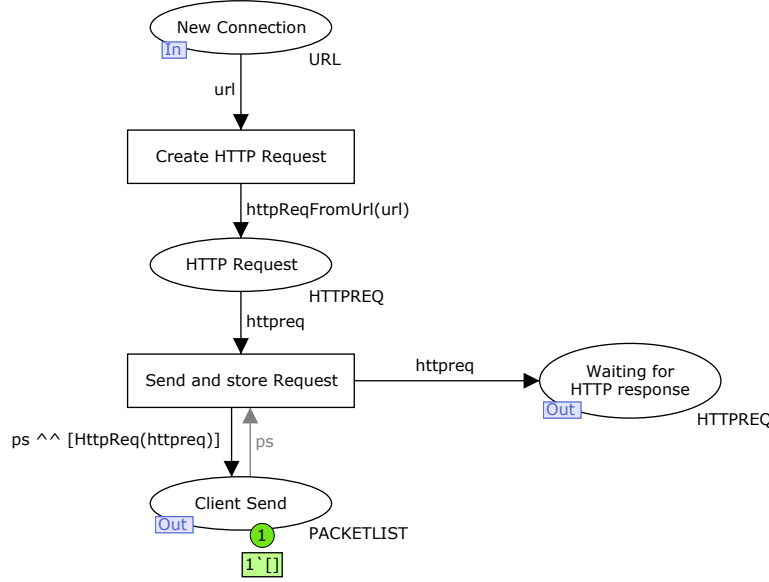


Figure 2.5: New Connection submodule

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 values `httpVersion`, `origin` and `nonce` have been statically defined, partly for simplification and partly to more easily see it during simulation. The `nonce` value should be generated randomly under real-world execution, however there should be no side effects from abstracting this in our model.

In a similar vein, the two functions `B64()` and `SHA1()` are abstract versions of the Base64 encoding algorithm [Jos06] and the SHA1 hashing algorithm [iosat02] on which the WebSocket protocol relies. We considered it unnecessary to actually implement these for the purpose of this model, and instead define them to simply wrap the string argument to show that it has been encoded or hashed.

The rationale for the elaborate set of headers is to ensure that only correctly implemented WebSocket servers will recognise the request as a WebSocket connection request and respond appropriately, while still adhering to the HTTP specification so regular web servers can understand the request.

Listing 2.9: httpReqFromUrl

---

```
val httpVersion = "HTTP/1.1";
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}
    ]
  };

```

---

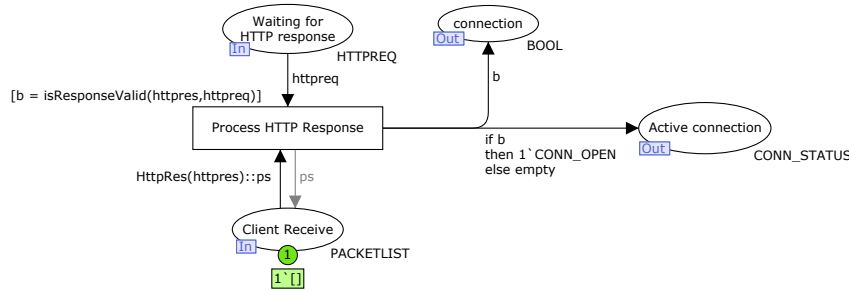


Figure 2.6: Process Response submodule

Listing 2.10: isResponseValid

---

```

val uuid = "258EAF5-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

```

---

## 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 an inscription (termed its guard) 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) (the same one should be used in every WebSocket implementation) and the nonce previously generated by the client for the `Sec-WebSocket-Key` header.

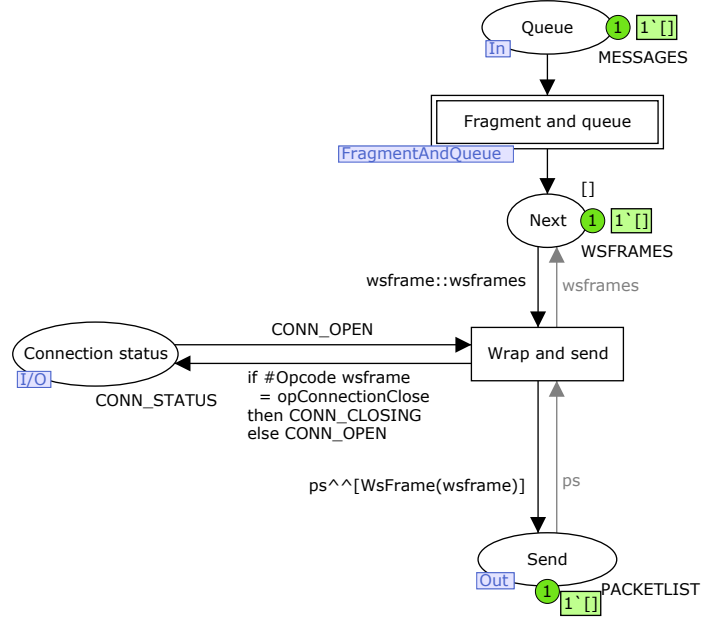


Figure 2.7: Wrap And Send submodule

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 them as WebSokert frames 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



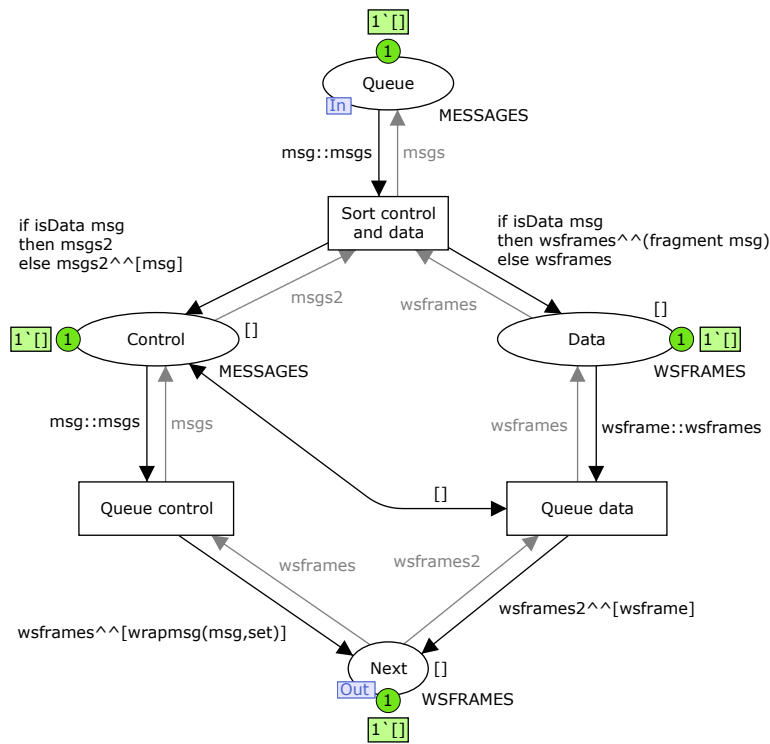


Figure 2.8: Fragment And Queue submodule

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, which has to be declared before it is used by the other two functions) to produce `WSFRAME` tokens.

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 prioritisation 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;

```

---

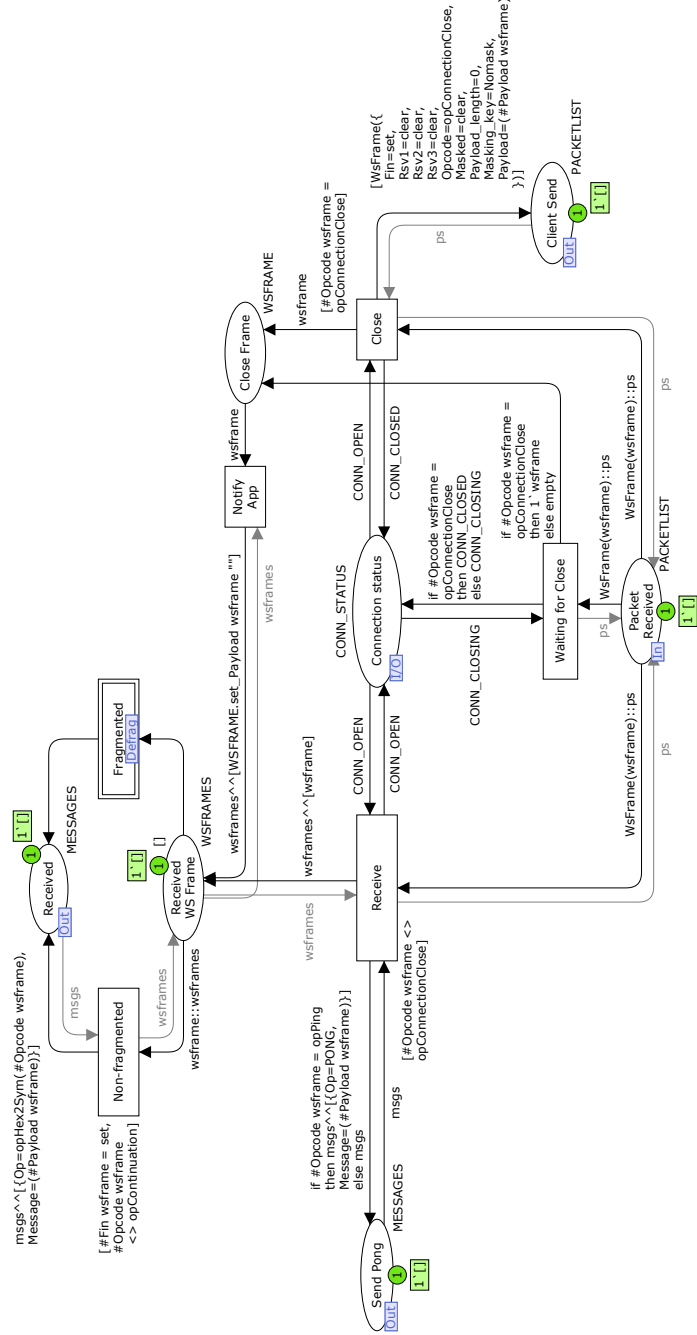


Figure 2.9: Unwrap And Receive submodule

- 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. Any frame that is 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) to concatenate the strings.

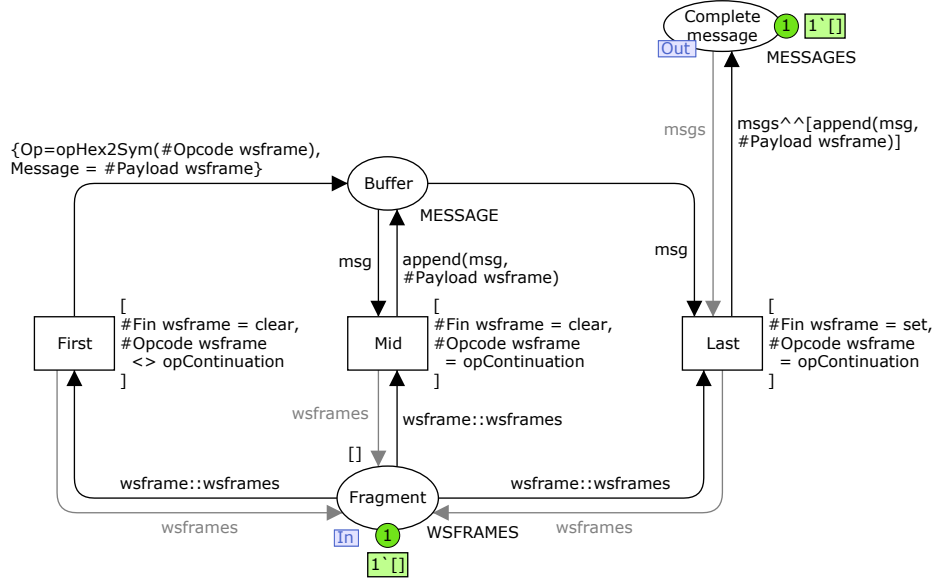


Figure 2.10: Defrag Module

Listing 2.13: append

---

```

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

```

---

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

Note that since the WebSocket protocol does not allow separate 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 sub-protocols, 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 pur-

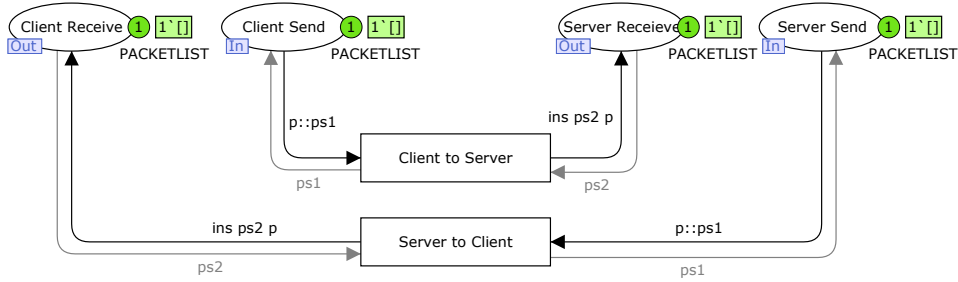


Figure 2.11: The Connection module

posely been modeled abstractly, since the transportation of data between the client and the server as well as establishing and ensuring a stable connection is assumed to be taken care of by TCP as the WebSocket protocol specifies, 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.

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;

```

---

The `Wrap` and `Send` and the `Unwrap` and `Receive` substitution transitions are bound to the same submodules as the ones in the Client WebSocket module. In effect this allows us to reuse the submodules for wrapping and

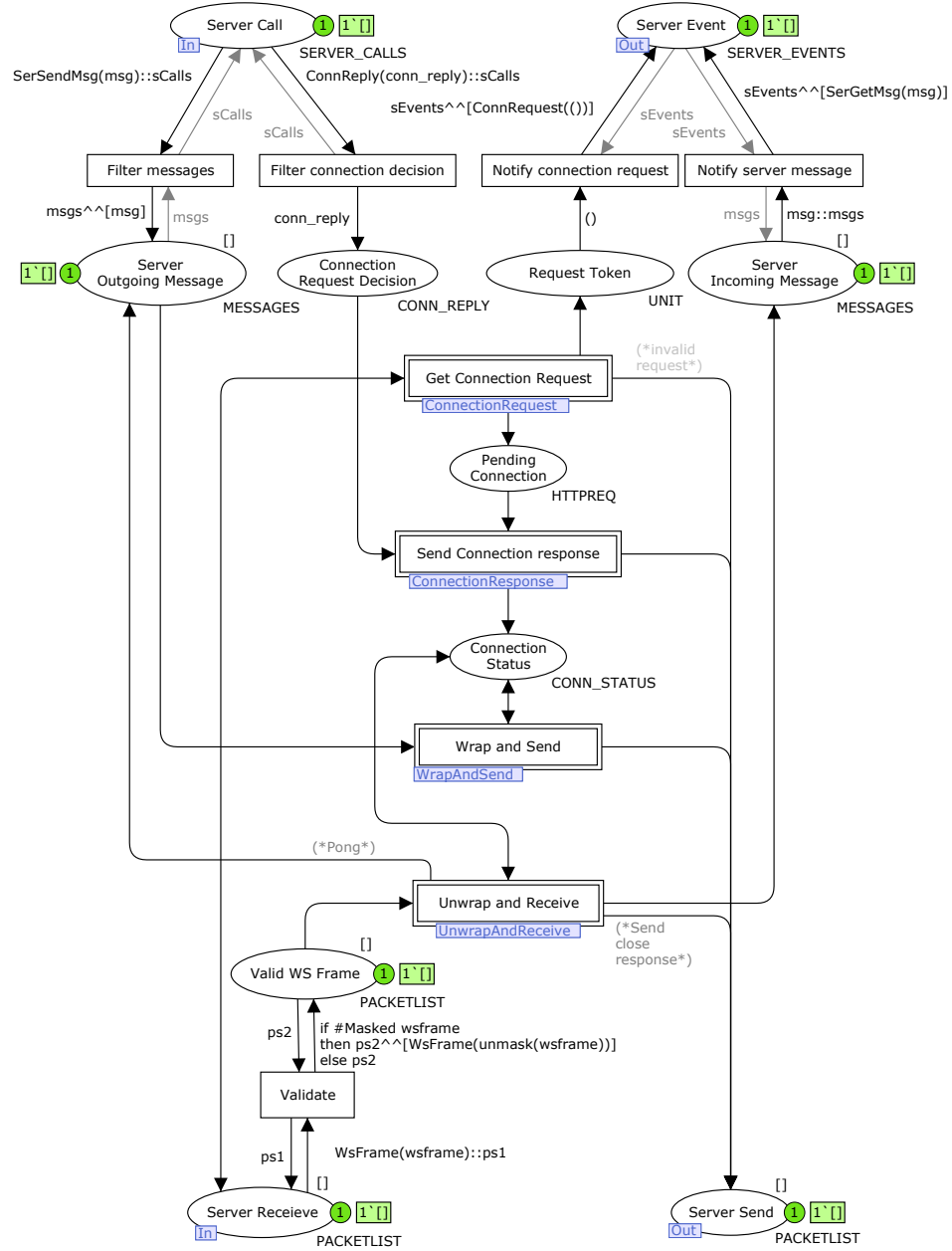


Figure 2.12: The Server WebSocket Module



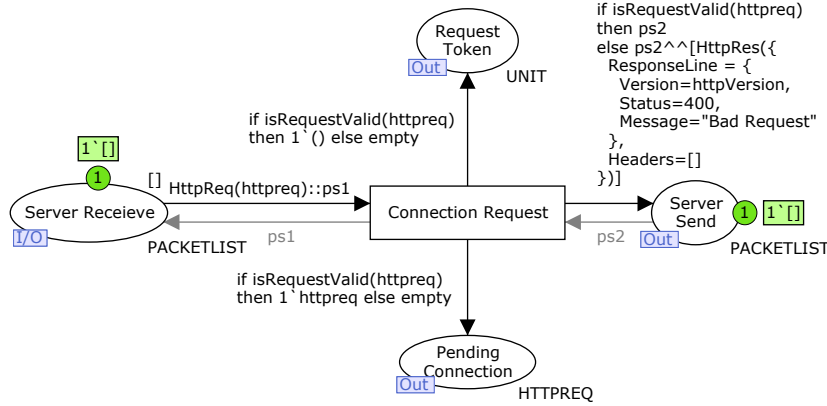


Figure 2.13: Get Connection Request

unwrapping WebSocket frames (since the mechanics for this is exactly the same for the client and the server), while during simulation they will use different instances of the bound module, and are thereby able to 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 2.15. 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 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.

### 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` (see Listing 2.1) 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 specification of the WebSocket Protocol. As discussed in Section 2.2.3 under the `New Connection` and `Process Response` modules, this involves generating a `Sec-WebSocket-Accept` header based on the `Sec-WebSocket-Key` header from the client, and

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

```

---

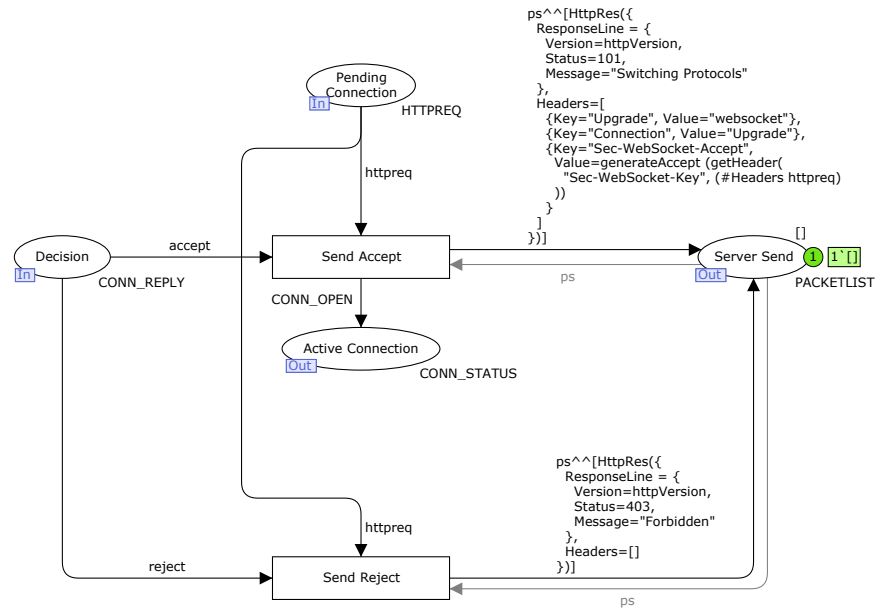


Figure 2.14: Send Connection Response

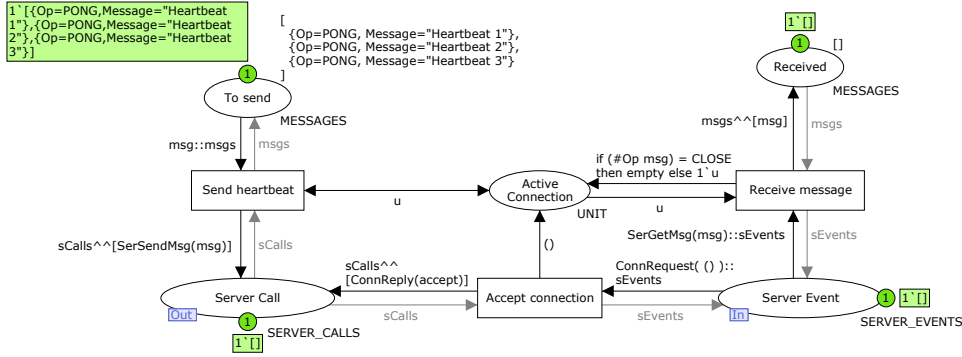


Figure 2.15: The Server Application Module

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 messages as its initial marking, to illustrate the capability of PONG frames to be used as a heartbeat<sup>1</sup> (without PING being involved). Otherwise, the mechanics of sending and receiving messages is the same as in the Client Application. While this module has intentionally been modelled to be simple and abstract, a real world application would have more logic here, but the interface to the protocol layer would be the same.

<sup>1</sup>A heartbeat is commonly used to explicitly declare that a connection is still active.



## 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, as well as 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 will by default calculate the state space in breadth-first order.

Once generated, the state space graph can be visualised directly in CPN Tools. For example, 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

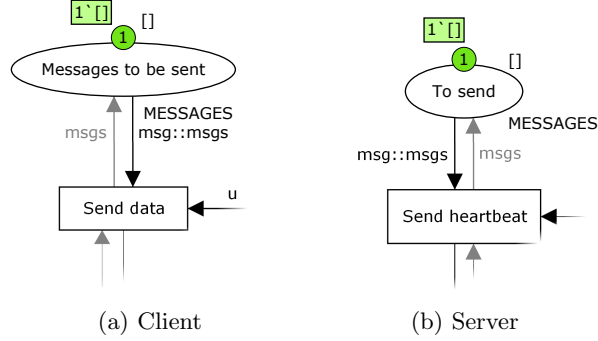


Figure 3.1: Configuration: No messages

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 that 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 that 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 colour set. 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 colour set, 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 generator 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.

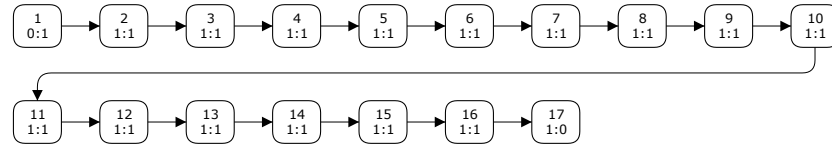


Figure 3.2: State space for configuration with no messages

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 occurrences. 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 is shown on Listing 3.2 and 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



Listing 3.2: State Space Report: Statistics

---

State Space		
Nodes:	17	
Arcs:	16	
Secs:	0	
Status:	Full	
Scc Graph		
Nodes:	17	
Arcs:	16	
Secs:	0	

---

node than there are arcs, 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, shown in Listing 3.3, 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.

Listing 3.3: State Space Report: Best Integer Bounds

---

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
[...]		

---

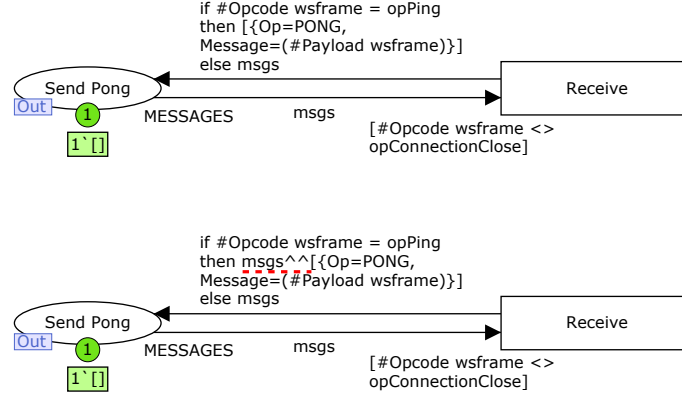


Figure 3.3: Problem with the Pong reply modeling

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: 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, shown in Listing 3.4, 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, shown in Listing 3.5, shows all home markings. A home marking is a marking that can always be reached from any other reachable state in the state space.

We see that there is one such marking defined by node 17. From earlier

Listing 3.4: State Space Report: Best Multi-set Bounds

---

```

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
  [...]

```

---

Listing 3.5: State Space Report: Home Properties

---

```

Home Markings
[17]

```

---

Listing 3.6: State Space Report: Liveness Properties

---

```

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

```

---

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.

### 3.2.4 Liveness Properties

This section of the state space report, shown in Listing 3.6, 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 effectively 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

Listing 3.7: 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.

---

transition. Our example has no such transition, which follows trivially from the fact that there is a dead marking.

The state space report also contains fairness properties, but this does not apply to our model since the 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 Web-Socket protocol model. Below we present the results from considering more complex configurations. For each configuration we only present select elements 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 results are shown in Listing 3.7

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

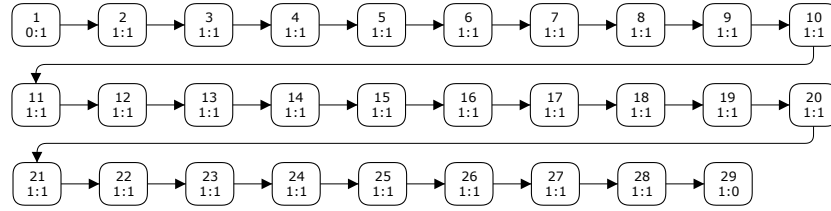


Figure 3.4: One message

Listing 3.8: One ping then one message

---

```

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

```

```

Home Markings
[475]

```

```

Dead Markings
[475]

```

---

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

### 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 in Listing 3.8 that the number of nodes has increased by an order of magnitude. This is due to the fact that for each position the first message can have, the other message can be anywhere from not sent yet to at the same place, yielding an exponential increase in possible states.

CPN Tools supports exporting state spaces to a format supported by Graphviz [EGK<sup>+</sup>02], 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.

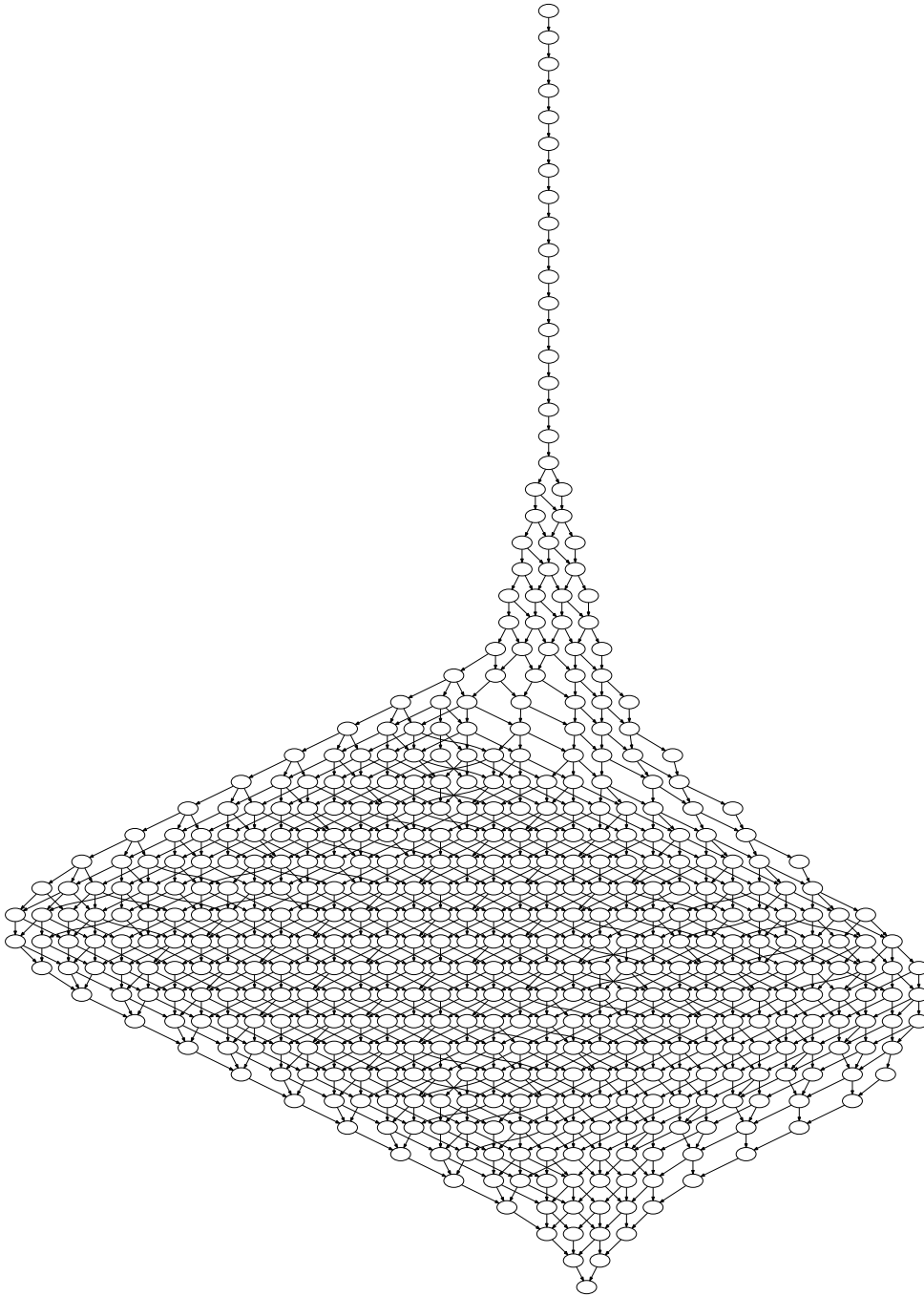


Figure 3.5: State space for one ping, one message configuration

Listing 3.9: One message then one ping

---

```

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

Home Markings
None

Dead Markings
[512,513]

```

---

**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. Results are shown in Listing 3.9.

Note that we now no longer have any home marking, and instead have two dead markings, representing the two possible orderings of the two messages. 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 to be sent 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 in Listing 3.10 that this has more states than sending two simple messages. We can also use the Best Upper Multi-set Bounds section, shown in Listing 3.11 to find the largest collection of fragments.

We are also able to inspect every other possible 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



Listing 3.10: One long message

---

```

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

```

```

Home Markings
  [813]

```

```

Dead Markings
  [813]

```

---

Listing 3.11: Upper Multi-set Bounds - long message fragments

---

```

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

```

---

Listing 3.12: Ping Text Close

---

```

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

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

```

---

Listing 3.13: Large configuration

---

```

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

```

---

three messages, state space calculation becomes noticeably time-consuming. Listing 3.12 shows the results.

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, which we show in Listing 3.13.

The report also showed that it had not detected 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 work 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

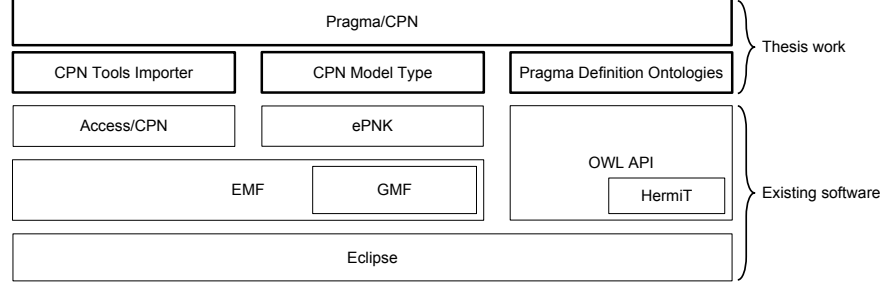


Figure 4.1: Application Overview Diagram

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:

- At the bottom of this we have the Platform Runtime, based on the

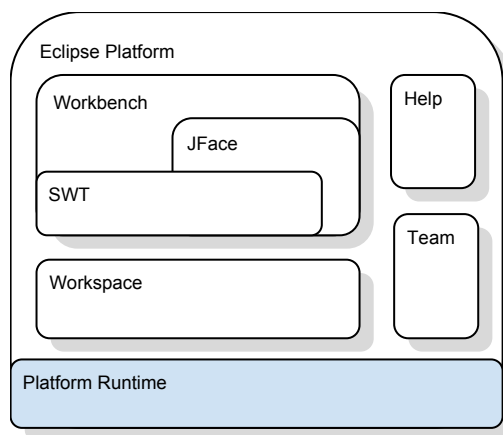


Figure 4.2: The Eclipse RCP

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.

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, as well as add tools, functionality and

services. For example, this thesis was written in  $\text{\LaTeX}$  using the Texlipse plugin, and managed with the Git version control system through the EGit plugin.

Plugins connect to each other through the so-called extension points they declare, which includes a schema to define requirements and instructions. Where one plugin declares an extension point, another plugin can extend that point according to its defined schema.

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



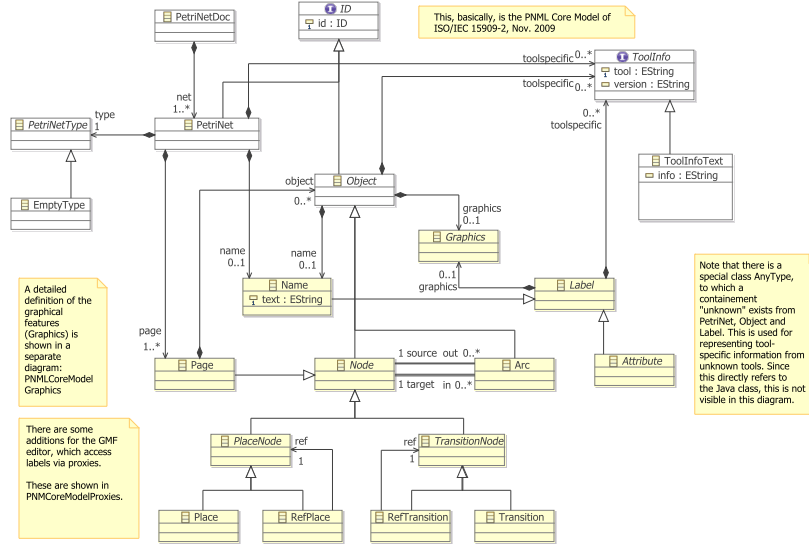


Figure 4.3: PNML Core Model

- 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:

**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 of HLPNG are not compatible with the CPN ML syntax from CPN Tools, and for our prototype, these structured labels are not necessary with regard to pragmatics, as we do not define any pragmatics for this prototype that depend on this level of detail. This might be desired in a future version, where for example pragmatics are available depending on things like the colour set of a place or the variables on an arc, but initially this is considered to be outside the scope of this thesis; simple string representations of inscriptions are sufficient.

In light of this, our decision was therefore to develop our own Petri Net type that matches the structure of CPN Tools models as well as supporting annotation with pragmatics, 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 automatically generate some or all of the Java interface, implementation, as well as any XML serialization for the modelled 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 by ePNK to generate its 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 preferable, 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, as they relate to information science, 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 describing individuals and known information about them.

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 its meaning and draw conclusions that are not explicitly asserted. Use of ontologies is also prevalent in other fields, like in medicine to describe and relate symptoms and diseases.

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 [BM04]. There also exist other syntaxes, like Manchester syntax which improves readability, and Functional syntax which emphasises formal structure.

To give a clearer impression of ontologies, consider this small example ontology: Person is a class. Man and Woman are subclasses of Person. Parent is also a subclass of Person, and can be described with the property hasChild with values of type Person. Mother is a class equivalent to the intersection of Woman and Parent. Anne and Bob are individuals, and we assert that Anne is a Woman and that Anne hasChild Bob.

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. From the above example, we can infer that Bob is a Person, and Anne is a Parent, from the definition of the hasChild property. Thus we can further infer that Anne is also a Mother, since she is both a Woman and a Parent.

OWL 2 uses an open world assumption, meaning that a fact does not have to be explicitly asserted for it to be true. We discuss the consequences of this more in the next chapter.

The code generation pragmatics are defined as OWL 2 ontologies, primarily using Functional syntax. Together with an ontology describing the structure of CPN models, this allows us to describe pragmatics using the full expressiveness of OWL 2, and enables complex restrictions to be defined for placement of pragmatics on model elements. This acts as a basis to enable the Pragma/CPN model editor to determine appropriate pragmatics for selected model elements, as well as for model validation.

To parse these ontologies, we use OWL API [HB09], the reference Java implementation for OWL 2, which is capable of reading ontologies in any of the standard OWL 2 syntaxes, as well as programmatically creating and modifying ontologies. Implementation details are given in the next chapter.

There exists many reasoner engine implementations, with some that support OWL API. Dentler et al. [DCTTDK11] highlight the different advantages and weaknesses of several such reasoners. We give our own evaluation of some of them in context of Pragma/CPN in Chapter 6. The selected reasoners were chosen for being freely available, and include Pellet [SPG<sup>+</sup>07], HermiT [SMH08], FaCT++ [TH06] and TrOWL [TPR10].

## Chapter 5

# Analysis, Design and Implementation

Our analysis of available software solutions and platforms in the previous chapter showed that the basic primitives 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 development in the context of this thesis on the Eclipse Platform.

### 5.1 The CPN Ontology with Pragmatics

As mentioned in section 4.6, Pragmatics are defined and modeled as ontologies, using the OWL 2 Web Ontology Language. We have created two ontologies to act as a basis for specifying individual pragmatics: One that defines Coloured Petri Nets, and one that defines basic classes for pragmatics. The ontology defining CPN will be explained in several parts, with line numbering continuing across parts.

Listing 5.1 shows the first fragment of the CPN ontology. Every ontology

Listing 5.1: The CPN Ontology: Opening

---

```
1 Prefix(:=<http://hib.no/ontologypetrinets/cpn/>)
2 Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
3 Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
4
5 Ontology(<http://hib.no/ontologypetrinets/cpn/>
```

---

must be declared with `Ontology()` containing an Internationalized Resource Identifier (IRI) and a series of declarations and axioms. IRI is a generalization of the Uniform Resource Identifier (URI) scheme to allow all Unicode characters to be used. An IRI can point to an actual resource on the Internet, but is not required to do so. Each concept in an ontology is also identified with an IRI, but to avoid repetition we define “:” as a prefix representing this document’s IRI (line 1). Prefixes can also be declared for other ontologies, and some (like `owl:` and `xsd:`) are predefined (and stated on lines 2 and 3 for clarity).

Inside the `Ontology()` declaration we write declarations and axioms to describe the domain. Listing 5.2 shows the first such declarations. Whitespace and indentation is syntactically irrelevant and only used for readability. We first declare the class `:ColouredPetriNet` (line 7). We then declare the object property `:pages`, and restrict its domain to `:ColouredPetriNet` and its range to `:Page`<sup>1</sup> (lines 8-10). This means that there exists a property `:pages`, that instances of `:ColouredPetriNet` can have this property, and that the value of the property must be a `:Page`. The class `:Page` is defined just below on line 12, but the order of declarations are not important, as the whole document must be loaded before any processing or reasoning can be performed on the ontology. Also note that there are two kinds of properties: Object (which range must be a class) and Data (which range must be a datatype primitive, like string or integer, defined in the `xsd:` ontology).

We then continue declaring all model elements in a similar manner. A `:Page` can have many `:elements`. An `:Element` has a `:page`. Line 21 states `SubClassOf(:Element ObjectMaxCardinality(1 :page))`, meaning an `:Element` can have at most 1 `:page`. Line 22 states `InverseObjectProperties(:page :elements)`, defining these properties as inversely related. An example consequence of this is if we define an individual `:Element` and specify its `:page`, it is implicitly known that the `:Page` contains that individual in its `:elements`. An `:Element` can also have an `:id`.

Lines 29-42 describe the class `:Node` as a subclass of `:Element` with a `:name` property and `:in` and `:out` properties for connected `:Arcs`.

Listing 5.3 shows declarations relating to the `:Place` class, and follows the same pattern as the earlier classes to describe the properties `:sort`<sup>2</sup> and `:initialMarking`.

---

<sup>1</sup>Note that both Access/CPN and ePNK use the term Page instead of Module, so for convenience we will also use the term Page in the implementation of Pragma/CPN.

<sup>2</sup>The term Sort comes from Description Logic, which is sometimes used in relation to CPN inscriptions. Sort is used in Access/CPN instead of the term Colour Set. We opted to do the same.

Listing 5.2: The CPN Ontology: Base Classes

---

```

7 Declaration(Class(:ColouredPetriNet))
8 Declaration(ObjectProperty(:pages))
9 ObjectPropertyDomain(:pages :ColouredPetriNet)
10 ObjectPropertyRange(:pages :Page)
11
12 Declaration(Class(:Page))
13 Declaration(ObjectProperty(:elements))
14 ObjectPropertyDomain(:elements :Page)
15 ObjectPropertyRange(:elements :Element)
16
17 Declaration(Class(:Element))
18 Declaration(ObjectProperty(:page))
19 ObjectPropertyDomain(:page :Element)
20 ObjectPropertyRange(:page :Page)
21 SubClassOf(:Element ObjectMaxCardinality(1 :page))
22 InverseObjectProperties(:page :elements)
23
24 Declaration(DataProperty(:id))
25 DataPropertyDomain(:id :Element)
26 DataPropertyRange(:id xsd:string)
27 SubClassOf(:Node DataMaxCardinality(1 :id))
28
29 Declaration(Class(:Node))
30 SubClassOf(:Node :Element)
31 Declaration(DataProperty(:name))
32 DataPropertyDomain(:name :Node)
33 DataPropertyRange(:name xsd:string)
34 SubClassOf(:Node DataMaxCardinality(1 :name))
35
36 Declaration(ObjectProperty(:in))
37 ObjectPropertyDomain(:in :Node)
38 ObjectPropertyRange(:in :Arc)
39
40 Declaration(ObjectProperty(:out))
41 ObjectPropertyDomain(:out :Node)
42 ObjectPropertyRange(:out :Arc)

```

---

Listing 5.3: The CPN Ontology: Place Class

---

```

44 Declaration(Class(:Place))
45   SubClassOf(:Place :Node)
46
47   Declaration(DataProperty(:sort))
48   DataPropertyDomain(:sort :Place)
49   DataPropertyRange(:sort xsd:string)
50   SubClassOf(:Place DataMaxCardinality(1 :sort))
51
52   Declaration(DataProperty(:initialMarking))
53   DataPropertyDomain(:initialMarking :Place)
54   DataPropertyRange(:initialMarking xsd:string)
55   SubClassOf(:Place DataMaxCardinality(1 :initialMarking))

```

---

Listing 5.4: The CPN Ontology: Transition Classes

---

```

57 Declaration(Class(:TransitionNode))
58   SubClassOf(:TransitionNode :Node)
59
60 Declaration(Class(:Transition))
61   SubClassOf(:Transition :TransitionNode)
62
63   Declaration(DataProperty(:guard))
64   DataPropertyDomain(:guard :Transition)
65   DataPropertyRange(:guard xsd:string)
66   SubClassOf(:Transition DataMaxCardinality(1 :guard))
67
68 Declaration(Class(:SubstitutionTransition))
69   SubClassOf(:SubstitutionTransition :TransitionNode)
70
71   Declaration(ObjectProperty(:subpage))
72   ObjectPropertyDomain(:subpage :SubstitutionTransition)
73   ObjectPropertyRange(:subpage :Page)
74   SubClassOf(:Node ObjectMaxCardinality(1 :page))
75   SubClassOf(:SubstitutionTransition ObjectMaxCardinality(1 :
       subpage))

```

---



Listing 5.5: The CPN Ontology: Arc Class

---

```

77 Declaration(Class(:Arc))
78   SubClassOf(:Arc :Element)
79
80 Declaration(ObjectProperty(:dest))
81   ObjectPropertyDomain(:dest :Arc)
82   ObjectPropertyRange(:dest :Node)
83   SubClassOf(:Arc ObjectMaxCardinality(1 :dest))
84
85 Declaration(ObjectProperty(:source))
86   ObjectPropertyDomain(:source :Arc)
87   ObjectPropertyRange(:source :Node)
88   SubClassOf(:Arc ObjectMaxCardinality(1 :source))
89
90 InverseObjectProperties(:in :dest)
91 InverseObjectProperties(:out :source)
92
93 Declaration(DataProperty(:expression))
94   DataPropertyDomain(:expression :Arc)
95   DataPropertyRange(:expression xsd:string)
96   SubClassOf(:Arc DataMaxCardinality(1 :expression))

```

---

Listing 5.6: The CPN Ontology: Arc Class

---

```

99 DisjointClasses(:Page :Element)
100 DisjointClasses(:Node :Arc)
101 DisjointClasses(:Place :TransitionNode)
102 DisjointClasses(:Transition :SubstitutionTransition)

```

---

Listing 5.4 shows declarations relating to the `:Transition` and `:SubstitutionTransition` classes.

Listing 5.5 shows declarations relating to the `:Arc` class. The properties `:dest` and `:source` are declared as inverse of `:in` and `:out` (from `:Node`).

Finally, Listing 5.6 declare classes as disjoint. This is necessary, as ontologies by default use open world assumptions, meaning that even though something is not explicitly asserted, it may still be true. Thus, without these declarations, it is possible for an individual to for instance be a `:Place` and an `:Arc` at the same time. Thus, we need to “close the world” by defining classes as disjoint, meaning a member of one class is never a member of any of the other classes in the statement. To fully enforce Closed World Assumption (CWA) further measures need to be taken, which we describe in Section 5.4.2.

Listing 5.7 provides the base ontology for pragmatics. The purpose of the base is to link CPN elements and pragmatic annotations. The specific pragmatics that can be attached to CPN elements will be defined in subsequent ontologies.

Listing 5.7: The Basic Pragmatics Ontology

---

```

1 Prefix(:=<http://k1s.org/OntologyRestrictedNets/basic/>)
2 Prefix(basic:=<http://k1s.org/OntologyRestrictedNets/basic/>)
3 Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
4 Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
5 Prefix(cpn:=<http://hib.no/ontologypetrinets/cpn/>)
6
7 Ontology(<http://k1s.org/OntologyRestrictedNets/basic/>
8   Import( <http://hib.no/ontologypetrinets/cpn/> )
9
10 Declaration(Class(:Pragmatic))
11
12 DisjointClasses(cpn:Element cpn:Page :Pragmatic)
13
14 Declaration( ObjectProperty( :belongsTo ) )
15 ObjectPropertyDomain( :belongsTo :Pragmatic )
16 ObjectPropertyRange( :belongsTo ObjectUnionOf(cpn:Element cpn:
    Page) )
17
18 Declaration( ObjectProperty( :hasPragmatic ) )
19 ObjectPropertyDomain( :hasPragmatic ObjectUnionOf(cpn:Element
    cpn:Page) )
20 ObjectPropertyRange( :hasPragmatic :Pragmatic )
21
22 InverseObjectProperties(:belongsTo :hasPragmatic)
23
24 )

```

---

Listing 5.7 illustrates that in an ontology document it is possible to import other ontologies by referring to their IRI. We describe later how this is taken into account when loading ontology documents from projects and plugins. In this ontology, we import the CPN Ontology (line 8) and also assign it the prefix `cpn:` (line 5).

The ontology in Listing 5.7 is much shorter than the CPN ontology, declaring only one new class `:Pragmatic`, as well as its disjointness with other classes and its `:belongsTo` property. It also declares the property `:hasPragmatic` for `cpn:Element`, and declares the two properties as the inverse of each other.

Next, we have the ontology for the Generic pragmatics that provide the fundamental set of pragmatics for the protocol domain, shown in Listing 5.8. This is the first ontology that defines actual usable pragmatics. Each of these

pragmatics are defined using two declarations, where the first asserts it as a subclass of `basic:Pragmatic`, and the second is a form of property restriction. In the case of the first pragmatic, line 14 states that a `:Principal` can only belong to a `cpn:SubstitutionTransition`. We will give a brief explanation of the purpose and usage each of the pragmatics in Chapter 6, where we apply them to various CPN models. Note that this is not an exhaustive list of all generic pragmatics, and is only meant to be a working example.

Listing 5.8: The Generic Pragmatics Ontology

---

```

1  Prefix(:=<http://org.k1s/orn/nppn/>)
2  Prefix(basic:=<http://k1s.org/OntologyRestrictedNets/basic/>)
3  Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
4  Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
5  Prefix(cpn:=<http://hib.no/ontologypetrinets/cpn/>)
6
7  Ontology(<http://org.k1s/orn/nppn/>
8    Import( <http://k1s.org/OntologyRestrictedNets/basic/> )
9    Import( <http://hib.no/ontologypetrinets/cpn/> )
10
11
12
13  SubClassOf( :Principal basic:Pragmatic )
14  SubClassOf( :Principal ObjectAllValuesFrom( basic:belongsTo
15    cpn:SubstitutionTransition ) )
16
17  SubClassOf( :Channel basic:Pragmatic )
18  SubClassOf( :Channel ObjectAllValuesFrom( basic:belongsTo cpn
19    :Place ) )
20
21  SubClassOf( :Id basic:Pragmatic )
22  SubClassOf( :Id ObjectAllValuesFrom( basic:belongsTo cpn:
23    Place ) )
24
25  SubClassOf( :External basic:Pragmatic )
26  SubClassOf( :External ObjectAllValuesFrom( basic:belongsTo :
27    TransitionConnectedToId ) )
28
29  SubClassOf( :Send basic:Pragmatic )
30  SubClassOf( :Send ObjectAllValuesFrom( basic:belongsTo cpn:
31    Transition ) )
32
33  SubClassOf( :Recieve basic:Pragmatic )
34  SubClassOf( :Recieve ObjectAllValuesFrom( basic:belongsTo cpn
35    :Transition ) )
36
37  SubClassOf( :OpenChannel basic:Pragmatic )
38  SubClassOf( :OpenChannel ObjectAllValuesFrom( basic:belongsTo

```

```

        cpn:Transition ) )
33
34 SubClassOf( :CloseChannel basic:Pragmatic )
35 SubClassOf( :CloseChannel ObjectAllValuesFrom( basic:
        belongsTo cpn:Transition ) )
36
37 DisjointClasses(:Id :External )
38
39
40 EquivalentClasses( :IdPlace ObjectIntersectionOf(
41     cpn:Place
42     ObjectSomeValuesFrom( basic:hasPragmatic :Id)
43     ObjectExactCardinality( 1 basic:hasPragmatic )
44 )
45 )
46
47 EquivalentClasses( :ArcFromId ObjectIntersectionOf(
48     cpn:Arc
49     ObjectSomeValuesFrom( cpn:source :IdPlace)
50     ObjectExactCardinality( 1 cpn:source :IdPlace)
51 )
52 )
53
54 EquivalentClasses( :TransitionConnectedToId
    ObjectIntersectionOf(
55     cpn:Transition
56     ObjectSomeValuesFrom( cpn:in :ArcFromId)
57 )
58 )
59 )

```

---

The `:External` pragmatic (line 22 and 23) shows why ontologies are well suited for defining pragmatics. It can belong to `:TransitionConnectedToId`, which is a complex class defined through a chain of class assertions: We first define `:IdPlace` on line 40 to be equivalent to the set of individuals that are a `cpn:Place`, and has a `:hasPragmatic` property with a value that is an instance of `:Id`, and has exactly one `:hasPragmatic` property associated. In the same manner we further define `:ArcFromId` on line 47 as anything that is a `cpn:Arc` coming from an `:IdPlace`. Finally on line 54, a `:TransitionConnectedToId` must have an incoming `:ArcFromId`. All together this describes the circumstance where an `:External` pragmatic can be placed, which is on any transition with an incoming arc from a place annotated with an `:Id` pragmatic. We demonstrate the practical effect of this when annotating models in Chapter 6.

### 5.1.1 The Ontology Plugin

The ontologies described above are not subject to change by the user of Pragma/CPN, and should be included automatically. We need a way to conveniently provide them to Pragma/CPN, as well as utilities for managing them. As mentioned, the IRI identifying the ontology can point to an actual resource on the Internet, but we want to avoid relying on an Internet connection.

To solve this we created a separate Eclipse plugin called Ontologies. It contains the ontology documents, utility classes for listing all ontologies the plugin provides, a custom `PluginIRIMapper` which can translate the IRI of import statements to the ontology documents available in the plugin, and an `OntologyLoader` utility which creates an OWL API ontology manager loaded with a `PluginIRIMapper`.

It also includes the OWL API and several reasoner engine library files, and exports their packages to let other plugins use them. This also gives a central point of upgrading libraries should it be desired in the future.

## 5.2 The Annotated CPN model type for ePNK

While designing the ePNK Petri Net Type model for Pragma/CPN, it was decided to separate it into two parts: One to define a CPN Type that corresponds to CPN Tools including the various inscriptions and model constraints, and one for defining an Annotated CPN Type, extending from the first and capturing how pragmatics relate to elements of the CPN Type model. This also adds the benefit that the CPN Type can be used in other applications.

A custom Petri Net Type needs to be contained in an Eclipse Plugin project. Such a project will contain configuration files that define the properties and capabilities of the plugin. These files include `MANIFEST.MF` for declaring plugin name, version and dependencies, `plugin.xml` for defining extensions and extension points, and `build.properties` for defining build parameters. The Eclipse Plugin IDE includes an editor for editing all three files in a convenient user interface with guides and content assistance.

After creating a plugin 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<sup>3</sup>). We will refer to the PNML Core Model as the Core Model.

---

<sup>3</sup>These ePNK Types were briefly introduced in Section 4.3

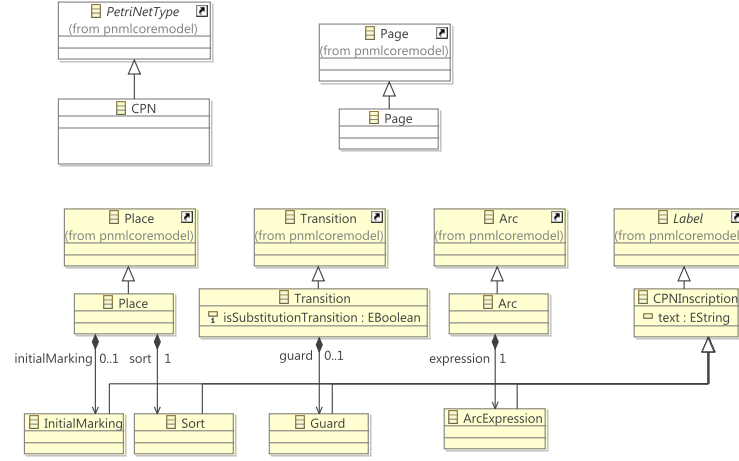


Figure 5.1: CPN model type diagram

### 5.2.1 CPN Type Model

The CPN Type model is shown in Fig. 5.1, and defines the structure and constraints of Coloured Petri Nets. The first thing a new Type model should define is a subclass of the `PetriNetType` class from the Core Model (we will refer to this as the Core Type), with the name of the new 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 used to identify the Type, and is what will appear in the menu to let a user extend a model with the new Type.

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` uses reflection to check a Type Model for subclasses with the same name as classes in the Core Type, and load these dynamically instead of the base classes. This can be seen in Fig. 5.1, where `Place`, `Arc`, `Transition` and `Page` are all subclassed from the classes referenced from the Core Model, with added relations to inscription classes.

Due to limitations of the PNML specification, which we discuss in Section 6.3, we need to simplify the representation of substitution transitions by defining an attribute on the `Transition` class to be able to designate an instance of it as a substitution transition. We do not want to use the `RefTransition` from the Core Type for this, as it is intended to reference another transition, and since we are unable to change or remove this aspect

of `RefTransition` (for the same reasons discussed in the previous paragraph) it could potentially confuse users.

To represent the different inscriptions of a CPN model, we define classes that inherit the `Label` class from the Core Model. From a data modeling perspective, it would make more sense to make inscriptions simple string attributes of the classes they belong to, but the `Label` class includes functionality required for displaying in the graphical diagram editor of ePNK. The different inscription classes are identical implementation-wise, but have different names to be distinguishable in the user editor context menu for adding child elements.

### 5.2.2 Code Generation

After creating the CPN Type model, EMF can generate source code for interfaces and implementations of the new classes. This is done by creating a “genfile” linked to the EMF model. The genfile can define meta-info 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 an ePNK Type we only need Model code and Edit code. the Model code includes interfaces and corresponding basic implementations of the classes as well as factories for instantiating them, while the Edit code contains classes for presenting and manipulating the model 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 supplying a unique id, a descriptive name, and the fully qualified classpath to the `PetriNetType` subclass. The resulting element in `plugin.xml` is shown in Listing 5.9.

### 5.2.3 Constraints

We define one constraint for the CPN Type: An Arc must go between a Place and a Transition. Technically this is done via the EMF extension point “`org.eclipse.emf.validation.constraintProviders`”. There are several ways of

Listing 5.9: ePNK Petri Net Type Definition Extension

---

```

<extension
  id="org.cpntools.pragma.epnk.pnktypes.cpndefinition"
  name="CPN"
  point="org.pnml.tools.epnk.pntd">
  <type
    class="org.cpntools.pragma.epnk.pnktypes.
      cpndefinition.impl.CPNImpl"
    description="Coloured Petri Net from CPN Tools">
  </type>
</extension>

```

---

defining extensions for this point, some of which are explained in [Kin11]. Our extension declaration is shown in Listing 5.10, and is configured to use a Java class to perform validation (defined in the `<constraint>` element's class attribute).

The class `ArcSourceAndTargetLimitation` we have specified is shown in Listing 5.11. The `IValidationContext` argument provides both the target model element to be validated, and facilities for creating an appropriate return object representing failure or success. The algorithm for checking the arc's source and target is straightforward.

#### 5.2.4 Annotated CPN Type

The Annotated CPN Type model is shown in Fig. 5.2. It extends the CPN model to enable annotation of model elements. It also handles references to ontology documents that define sets of pragmatics.

As with the CPN Type Model, we have subclassed the classes of model elements we need to extend. All of these need to be extended to support the same feature: being annotated with pragmatics. Ideally, we would have changed a superclass of these classes, for instance `Node` (see Fig. 4.3). But this will not work, since model merging is not possible (as discussed earlier), and subclassing `Node` will not work, since existing classes will not be subclasses of our new superclass and therefore not inherit anything from it.

We needed to devise a way to extend a group of classes without access to their superclass, and ideally without copying code between the classes. The solution was to define an interface that all of the target classes implement<sup>4</sup>.

---

<sup>4</sup>Note that in Fig. 5.2 the arcs that signify this are easily mistaken as multiple inheritance. Only the small (I) icon on the `OntologyMember` element shows that this is an interface, and not a class.



Listing 5.10: constraintProvider Extension

---

```

1  <extension
2      point="org.eclipse.emf.validation.constraintProviders"
3      >
4      <category
5          id="org.pnml.tools.epnk.validation"
6          name="CPN Validation">
7      </category>
8      <constraintProvider
9          cache="true"
10         mode="Batch">
11      <package
12          namespaceUri="http://org.pnml.tools/epnk/
13          pnmlcoremodel">
14      </package>
15      <constraints
16          categories="org.pnml.tools.epnk.validation">
17      <constraint
18          class="org.cpntools.pragma.epnk.pnktypes.
19          cpndefinition.validation.
20          ArcSourceAndTargetLimitation"
21          id="org.cpntools.pragma.epnk.pnktypes.
22          cpndefinition.validation.
23          ArcSourceAndTargetLimitation"
24          isEnabledByDefault="true"
25          lang="Java"
26          mode="Batch"
27          name="Arc source and target limitation"
28          severity="ERROR"
29          statusCode="301">
30      <target
31          class="Arc:http://org.cpntools/pragma/epnk
32          /pnktypes/cpndefinition">
33      </target>
34      <description>
35          An Arc must go between a Place and either a
36          Transition or a Page.
37      </description>
38      <message>
39          The source and target of arc {0} are not
40          compatible.
41      </message>
42      </constraint>
43      </constraints>
44      </constraintProvider>
45  </extension>

```

---

Listing 5.11: Constraint Implementation

---

```

1 public class ArcSourceAndTargetLimitation extends
   AbstractModelConstraint {
2 public IStatus validate(IValidationContext ctx) {
3     EObject eObj = ctx.getTarget();
4     if (eObj instanceof Arc) {
5         Arc arc = (Arc) eObj;
6         Node source = arc.getSource();
7         Node target = arc.getTarget();
8
9         if (source != null && target != null) {
10             PlaceNode pn = null;
11             Node other = null;
12             if(source instanceof PlaceNode) {
13                 pn = (PlaceNode) source;
14                 other = target;
15             } else if (target instanceof PlaceNode) {
16                 pn = (PlaceNode) target;
17                 other = source;
18             }
19             if( pn == null || // there was no Place
20                other instanceof PlaceNode) // both are Places
21                 return ctx.createFailureStatus(new Object[] {arc});
22         }
23     }
24     return ctx.createSuccessStatus();
25 }
26 }

```

---

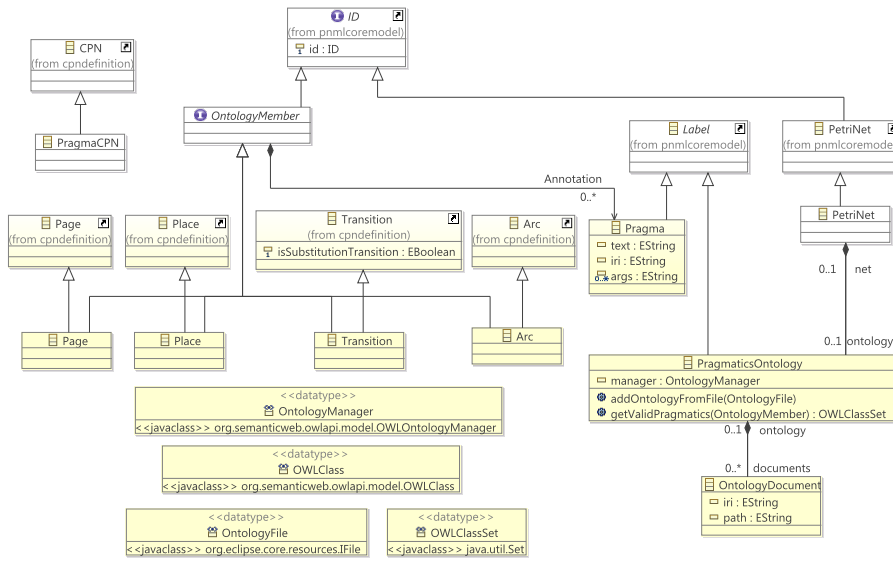


Figure 5.2: Annotated CPN model type diagram

This is the `OntologyMember` interface. `OntologyMember` has a reference to the `Pragma` class. The reference is configured to act as containment, meaning `Pragma` instances are created as children of `OntologyMember` instances.

A `Pragma` instance represents a pragmatic annotation, and stores the IRI of the ontology class to which it belongs. Pragmatics can also define configuration arguments. Like the inscriptions from the CPN Type, `Pragma` inherits the `Label` class to be able to be displayed in the diagram editor. The `text` attribute is configured to be virtual<sup>5</sup>, and manually implemented to return the pragmatics class enclosed in `<<and >>`, including arguments if present.

### 5.2.5 Persisting Pragmatics Sets in the Model

We have already shown how pragmatics sets can be defined using ontology documents. But we also need a way to include them in the models we want to annotate. In other words: a model needs to store which pragmatic sets it utilises. This is encapsulated in the `OntologyDocument` class (Fig. 5.2, lower right), which can store the IRI of the ontology and the path to the

<sup>5</sup>This means no field is generated and nothing is serialised, and is often used for attributes that are purely derived from other attributes.

document that contains it. This is sufficient to be able to reload the ontology if the CPN model is closed and later reopened.

These `OntologyDocument` instances need to be serialised somewhere in the model. Our first idea was to model them as contained in the `PragmaCPN` class (i.e. the `Type`), however due to the fact that this element is serialised only as an attribute this was not possible. We decided to create a new class which instance is contained by the `PetriNet` class. This is the `PragmaticsOntology` class (Fig. 5.2, right), and the intention is to have a central entity to manage the ontologies of the entire net, including loading and reasoning. For these purposes we defined one property to contain the OWL API ontology manager, as well as two operations for including new ontology documents and deducing which pragmatics are valid for a given `OntologyMemeber`. These will be described later in the chapter.

### 5.2.6 Adding Pragmatics Sets

There needs to be a convenient way to add domain and model specific pragmatics sets to a model from ontology documents. We decided to extend the context menu of the `PragmaticsOntology` class for this purpose.

Eclipse defines several extension points for extending every part of the UI. Menus in particular have more than one way of being extended. The ePNK manual suggests using the `org.eclipse.ui.popupMenus` extension point and has an example of how to implement such an extension. By following this example we have created the menu command “Add Ontology” under the submenu `Pragmatics`. The extension is listed in Listing 5.12.

Every menu contribution should supply an `id` attribute to allow it to be referenced by external code. This is the case for all the elements in this listing, so we explain it here to avoid repetition.

The extension defines an `<objectContribution>` element (line 3-8), stating that this menu item should only appear in the context menu when an object of the specified type is selected. The `adaptable` attribute is only relevant to `IResource` objects, but is a required attribute with a default value of `false`. The `objectClass` is the target class that will have its context menu extended with the new menu, in this case our `PragmaticsOntology` class.

We then define a `<menu>` element (line 9-14) that will create a submenu intended to group together all menu contributions related to pragmatics. The `label` is simply the label, and `path` specifies where in the parent menu it should be placed, with the value “additions” being recommended. The menu has a `separator` child element (line 11-13) that acts as an anchor for positioning items.

Listing 5.12: Add Ontology Menu Extension

---

```

1    <extension
2        point="org.eclipse.ui.popupMenus">
3        <objectContribution
4            adaptable="false"
5            id="ePNK Annotated CPN Type.edit."
6            objectContribution1"
7            objectClass="org.cpntools.pragma.epnk.pnktypes.
8                pragmacpndefinition.PragmaticsOntology">
9            <menu
10                id="org.cpntools.pragma.epnk.pnktypes.
11                    pragmacpndefinition.actions.standardmenu"
12                label="Pragmatics"
13                path="additions">
14                <separator
15                    name="group1">
16                </separator>
17            </menu>
18            <action
19                class="org.cpntools.pragma.epnk.pnktypes.
20                    pragmacpndefinition.menu.AddOntologyAction"
21                enablesFor="1"
22                id="org.cpntools.pragma.epnk.pnktypes.
23                    pragmacpndefinition.actions.
24                    AddOntologyAction"
25                label="Add Ontology"
26                menubarPath="org.cpntools.pragma.epnk.pnktypes.
27                    pragmacpndefinition.actions.standardmenu/
28                    group1">
29            </action>
30        </objectContribution>
31    </extension>

```

---

Listing 5.13: AddOntologyAction setActivePart() and selectionChanged()

---

```

private PragmaticsOntology ontology;
private Shell shell;

@Override
public void selectionChanged(IAction action, ISelection
    selection) {
    ontology = null;
    if (selection instanceof IStructuredSelection) {
        IStructuredSelection structuredSelection = (
            IStructuredSelection) selection;
        if (structuredSelection.size() == 1
            && structuredSelection.getFirstElement() instanceof
                PragmaticsOntology) {
            ontology = (PragmaticsOntology) structuredSelection.
                getFirstElement();
        }
    }
    action.setEnabled(ontology != null);
}

@Override
public void setActivePart(IAction action, IWorkbenchPart
    targetPart) {
    shell = targetPart.getSite().getWorkbenchWindow().getShell();
}

```

---

Last, there is the `action` element (starting line 15), defining the menu item that performs the action. It specifies the `class` that performs an action, which should be a class implementing the `IObjectActionDelegate` interface. We have implemented this as the `AddOntologyAction` class, described below. The `enablesFor` attribute specifies that the action is enabled (clickable) when exactly one element is selected. `label` is the displayed label. `menubarPath` defines placement by referencing the `id` and `separator` we defined for `<menu>`.

As soon as a `PragmaticsOntology` element is right-clicked and the menu is about to be displayed, an `AddOntologyAction` instance will be created and configured with `setActivePart()` (which allows us to get the current shell to be able to display error messages) and `selectionChanged()` (which gives us the current model selection). These two methods are shown in Listing 5.13.

When the menu item is clicked, the `run()` method will be called, shown in Listing 5.14. This method uses standard Eclipse classes to open a file selection dialog. This dialog shows the current project's resources and asks

Listing 5.14: AddOntologyAction run()

---

```

@Override
public void run(IAction action) {
    ElementTreeSelectionDialog dialog = new
        ElementTreeSelectionDialog(
            shell, new WorkbenchLabelProvider(),
            new BaseWorkbenchContentProvider());
    dialog.setTitle("Select an Ontology");
    dialog.setMessage("Select the Ontology document you want to
        include in the net:");

    dialog.setInput(getProject(ontology));
    dialog.open();

    if (dialog.getReturnCode() == Window.OK) {
        Object o = dialog.getFirstResult();
        if (o instanceof IFile) {
            IFile f = (IFile) o;
            ontology.addOntologyFromFile(f);
        } else {
            IStatus status = new Status(IStatus.ERROR, "PragmaCPN", 0,
                "Invalid file", null);
            ErrorDialog.openError(shell, "Invalid file",
                "The selected file was not valid.", status);
        }
    }
}

```

---

the user to select an ontology document. It does not filter files on their extension, as there exist editors that save ontologies with other extensions than those defined by W3C.

If the user selects a file and clicks OK, the `run()` method continues by calling `addOntologyFromFile()` on the selected `PragmaticsOntology`. This method is shown in Listing 5.15. All modification of an EMF model should happen through an `EditingDomain`. This is a facility that integrates with Eclipse to provide undo/redo capabilities, as well as ensuring model changes are performed in an atomic manner. We use it by creating an appropriate `Command` and executing it on the `EditingDomain`'s `CommandStack`. In this case we create an `AddCommand` to add an `OntologyDocument` as a child of the `PragmaticsOntology`.

Listing 5.15: addOntologyFromFile()

---

```

public void addOntologyFromFile(IFile file) {
    try {
        TransactionalEditingDomain domain = TransactionUtil.
            getEditingDomain(this);

        if (domain != null) {
            OWLOntology ont = loadOntology(file);
            //System.out.println(ont);
            if(ont.getOntologyID().getOntologyIRI() == null){
                MessageDialog.openError(getShell(), "Load Error",
                    "The file does not seem to contain a valid ontology.");
                unloadOntology(ont);
                return;
            }
            OntologyDocument loadedDoc = PragmacpndefinitionFactory.
                eINSTANCE.createOntologyDocument();
            loadedDoc.setIri(ont.getOntologyID().getOntologyIRI().
                toString());
            loadedDoc.setPath(file.getFullPath().toPortableString());
            //System.out.println(getManager().getOntologies().size());
            domain.getCommandStack().execute(
                AddCommand.create( domain, this,
                    PragmacpndefinitionPackage.PRAGMATICS_ONTOLOGY__DOCUMENTS
                        , loadedDoc
                ));
        }
    } catch (OWLOntologyCreationException e) {
        e.printStackTrace();
        String s = "";
        String msg = e.getMessage();
        if(msg != null && msg.length()>0)
            s = "Reason: " + msg;
        MessageDialog.openError(getShell(), "Load Error",
            "The file could not be loaded. "+s);
    } catch (CoreException e) {
        e.printStackTrace();
    }
}

```

---



### 5.3 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 class 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 Pragma/CPN, and is now also capable of parsing graphics data. And by depending on Access/CPN instead of writing our own parser, we can benefit from further updates and improvements.

The Eclipse Plugin IDE includes several template plugin projects, and one of them is for creating an Import Wizard, that is a step-by-step dialog for importing resources into the Eclipse workbench, for instance copying a file from the file system to a project, or importing entire projects from version control. The template also pre-configures the plugin manifest with an extension that makes the wizard available in Eclipse's Import Dialog. This is shown in Fig. 5.3 (a)

The import wizard for importing CPN Tools models only has one step, which requires the user to select the .cpn file from the file system, and select a destination in the workspace for the converted model. It is shown in Fig. 5.3 (b). After selecting the source file, the wizard suggests a new name for the converted model based on the original file name but with the .pnml extension used by ePNK.

The conversion process is straightforward, the .cpn file is loaded with Access/CPN, and the resulting model is then converted object by object to the ePNK Annotated CPN type. The final model instance also includes a `PragmaticsOntology` instance.

### 5.4 Creating Pragmatic Annotations

We want it to be fast and easy to annotate model elements, and an immediate example to follow is the way child and sibling elements are added to

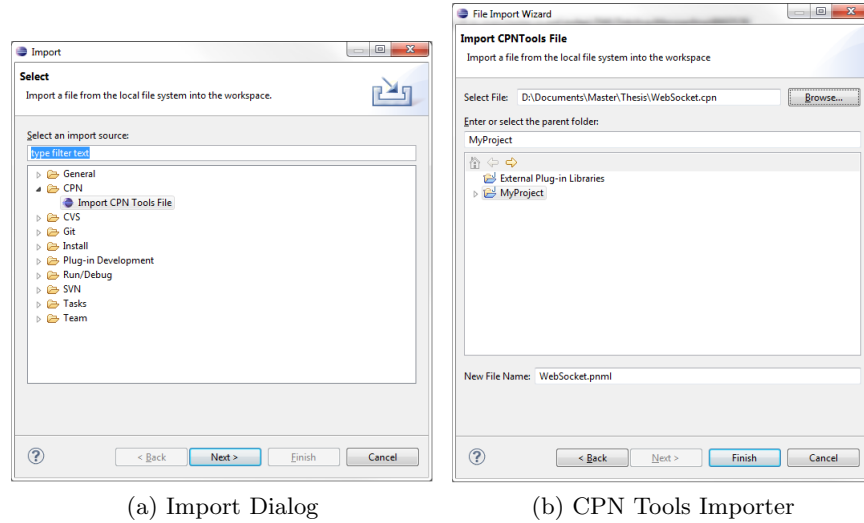


Figure 5.3: The Import Wizard

the model through context menus. It is possible to create an empty Pragma instance as a child of an `OntologyMember` instance without any manual programming; ePNK and EMF dynamically takes care of that. The user can then manually enter the IRI of the desired pragmatic.

However, we have also implemented a new menu entry similar to `New Child` and `New Sibling` that is dynamically populated with all valid pragmatics for the selected model element. The routine for providing the menu entry can be divided into two parts, and we will explain each part in the following subsections.

#### 5.4.1 Providing a Dynamic Context Menu

In Section 5.2.6 we described a simple way of extending popup menus with static entries through the `org.eclipse.ui.popupMenus` extension point. This point can however not be used to provide a menu with dynamically generated menu items, which we need for domain- and model-specific pragmatics. Instead, we need to use the `org.eclipse.ui.menus` extension point, which allows more generic menu extensions.

The extension we created is shown in Listing 5.16. The `locationURI` attribute is this time set to the URI of the editor itself. The menu is defined by a `<dynamic>` element pointing to the class `AnnotationContributionMenu` (described shortly), and a `<visibleWhen>` element constraining this menu to

only be visible when we have selected exactly one `OntologyMember` instance in the model.

The first part of the `AnnotationContributionMenu` class is shown in Listing 5.17. To be able to be used as a dynamic menu, it should implement the `IContributionItem` interface, and there exist several classes that implement this interface that can be further subclassed for the desired purpose. In this case, we subclass the `CompoundContributionItem` class which is designed to provide a dynamic list of items by calling `getContributionItems()` when the menu is about to be shown.

Aside from some utility methods (which we have omitted in the listing), we call `getValidPragmatics` (described in the next section) on the model's `PragmaticsOntology` instance (if one exists). If we get a populated set back, we generate an `ActionContributionItem` for each class in the set. An `ActionContributionItem` is a simple type of contribution item that will execute its assigned `IAction` when clicked. Each of the items are given an `AddPragmaAction` instance, a private class designed to provide a more convenient interface for creating `CreateChildAction` instances. The `AddPragmaAction` class is shown in Listing 5.18. This type of action wraps the low-level API of the `CommandStack` and related classes to yield a more concise way to encapsulate changes to a model, in this case adding a `Pragma` child to the selected element. The current selection should also be provided in case it needs to be restored, for example if the user chooses to undo it.

### 5.4.2 Determining Appropriate Pragmatics

As mentioned above, the `PragmaticsOntology` is responsible for determining appropriate pragmatics for the current model selection through the `getValidPragmatics()` method. We will describe it in parts throughout this section.

In broad terms, the algorithm we designed is naïve in its nature: Every time the method is called (which happens when the Add Pragmatics submenu is expanded), it will use OWL API to create a new ontology from scratch to describe the current state of the model. It then adds one new pragmatic instance to the ontology, and for each subclass defined for `Pragmatic`, it checks if the ontology is still consistent if the pragmatic is of that class, and if so, this pragmatic class is valid.

Listing 5.19 begins with some static values which will be used throughout the method. The `PrefixManagers` purpose is equivalent to the `Prefix()` statements used in written ontologies. Continuing, we have the beginning of the `getValidPragmatics()` method, which takes as its argument the target model entity to receive a pragmatic. This must be an `OntologyMember`, so

Listing 5.16: Add OntologyPragmatic Dynamic Menu Extension

---

```

1  <extension
2      point="org.eclipse.ui.menus">
3      <menuContribution
4          locationURI="popup:org.pnml.tools.epnk.
5              pnmlcoremodel.presentation.
6              PnmlcoremodelEditorPNML?before=additions">
7          <menu
8              label="Annotate with">
9              <dynamic
10                  class="org.cpntools.pragma.epnk.pnktypes.
11                      pragmacpndefinition.menu.
12                      AnnotationContributionMenu"
13                  id="org.cpntools.pragma.epnk.pnktypes.
14                      pragmacpndefinition.menu.
15                      AnnotationContributionMenu">
16              </dynamic>
17              <visibleWhen>
18                  <and>
19                      <count
20                          value="1">
21                      </count>
22                      <iterate>
23                          <instanceof
24                              value="org.cpntools.pragma.epnk.
25                                  pnktypes.pragmacpndefinition.
26                                      OntologyMember">
27                              </instanceof>
28                          </iterate>
29                      </and>
30                  </visibleWhen>
31              </menu>
32          </menuContribution>
33      </extension>

```

---

Listing 5.17: AddOntologyAction run()

---

```

1  public class AnnotationContributionMenu extends
    CompoundContributionItem {
2
3  private TransactionalEditingDomain domain;
4
5  protected IContributionItem[] getContributionItems() {
6      OntologyMember member = getSelectedItem();
7      domain = TransactionUtil.getEditingDomain(member);
8      PragmaticsOntology ontology = resolveOntology(member);
9      if(ontology == null)
10         return createDisabledMenu("Missing PragmaticsOntology");
11
12     Set<OWLClass> pragmas;
13     try {
14         pragmas = ontology.getValidPragmatics(member);
15     } catch (CausesInconsistencyException e) {
16         IContributionItem[] item = {new ActionContributionItem(
17             new ShowInconsistencyErrorAction(e.getSet(), getWorkbench
18                 ().getShell())
19         )};
20         return item;
21     }
22     if(!pragmas.isEmpty()) {
23         List<IContributionItem> items = new ArrayList<
24             IContributionItem>();
25         for(OWLClass pragma : pragmas) {
26             items.add(new ActionContributionItem(
27                 AddPragmaAction.create(domain, getSelection(),
28                     pragma.getIRI().toString())));
29         }
30         return items.toArray(new IContributionItem[items.size()]);
31     }
32     return createDisabledMenu("No applicable pragmatics");
33 }

```

---

Listing 5.18: AddPragmaAction

---

```
private static class AddPragmaAction extends CreateChildAction
{
    private AddPragmaAction(EditingDomain editingDomain,
        ISelection selection, Object descriptor) {
        super(editingDomain, selection, descriptor);
    }

    public static AddPragmaAction create(EditingDomain
        editingDomain,
        IStructuredSelection selection, String pragmaIRI) {
        Pragma p = PragmacpndefinitionFactory.eINSTANCE.createPragma
            ();
        p.setIri(pragmaIRI);
        CommandParameter desc = new CommandParameter(selection.
            getFirstElement(), PragmacpndefinitionPackage.Literals.
            ONTOLOGY_MEMBER__ANNOTATION, p);
        AddPragmaAction action = new AddPragmaAction(editingDomain,
            selection, desc);
        action.setText(pragmaIRI);
        return action;
    }
}
```

---

Listing 5.19: getValidPragmatics Part 1

---

```

static final private String cpnurl = "http://hib.no/
ontologypetrinets/cpn/";
static final private String basicurl = "http://k1s.org/
OntologyRestrictedNets/basic/";
static final private String nppnurl = "http://org.k1s/orb/
nppn/";
static private String localurl = "http://local.model/";
static private PrefixManager cpnPrefix = new
    DefaultPrefixManager(cpnurl);
static private PrefixManager basicPrefix = new
    DefaultPrefixManager(basicurl);
static private PrefixManager nppnPrefix = new
    DefaultPrefixManager(nppnurl);
static private PrefixManager localPrefix = new
    DefaultPrefixManager(localurl);

public Set<OWLObject> getValidPragmatics(OntologyMember member
    ) throws CausesInconsistencyException {
    // Create an empty ontology
    OWLOntology ontology;
    try {
        ontology = createModelOntology();
    } catch (OWLOntologyCreationException e) {
        e.printStackTrace();
        return Collections.emptySet();
    }
}

```

---

the variable is named `member`.

We call `createModelOntology()`, which creates an empty ontology, imports the CPN, Basic and General Pragmatics ontologies as well as any ontologies loaded in the model, and for each model element declares an individual with appropriate class and properties. The name used to declare the individual is the entity's id attribute, which must be present for the model to be valid. Existing pragmatics are also asserted as individuals, using the object hash as an id.

Listing 5.20 shows the second part of the `getValidPragmatics` method. Interaction with the OWL API happens by using an `OWLDataFactory` to create objects representing different concepts. The factory is acquired from an `OntologyManager`, which is acquired from the `Ontologies` plugin (discussed in Section 5.1.1). We also create a `ReasonerFactory`, in this case for a `PelletReasoner`.

Listing 5.20: getValidPragmatics Part 2

---

```
// Get factories
OWLDataFactory dataFactory = getManager().getOWLDataFactory()
;
PelletReasonerFactory reasonerFactory = PelletReasonerFactory
.getInstance();

// Create mock pragmatic and get target member
OWLObjectProperty hasPragmatic = dataFactory.
    getOWLObjectProperty(":_pragma", localPrefix);
OWLObjectPropertyAssertionAxiom hasPragmaticAxiom =
    dataFactory.getOWLObjectPropertyAssertionAxiom(hasPragmatic,
        memberInd, pragmaInd);
getManager().applyChange(new AddAxiom(ontology,
    hasPragmaticAxiom));
```

---

Next, we create an object representing the pragmatic we are about to add to the model, as well as one for the target member. We then assert the hasPragmatic axiom from the member to the pragmatic. Last, we use the manager to add this axiom to our ontology.

Listing 5.21 shows the next steps, consisting of creating axioms for enforcing Closed World Assumption (CWA). As discussed in the last chapter, OWL 2 uses an Open World Assumption, which states that any fact can be true unless explicitly asserted to be false. This implies there may exist other individuals and property instances other than the ones we have asserted to exist. This will prevent us from properly evaluating complex pragmatics like :External that define their availability based on the existence of other pragmatics.

In order to reason under CWA, three measures must be taken:

- Classes that are disjoint must be asserted as such. Otherwise having an individual be both a Place and an Arc at the same time would not entail an inconsistency. This needs to be designed manually, and is already included in the ontologies we have described.
- The universe of individuals must be limited to only include the indi-



viduals we have declared. The class `owl:Thing` represents every individual, asserted or otherwise. We must assert it to be equivalent to the set of the individuals we have explicitly declared (which is trivial to retrieve from the ontology).

- It is necessary to assert every property that is not explicitly stated to not exist. We do this for the `:hasPragmatic` property by first asserting all individuals to be different, and then for each individual set the cardinality of the `:hasPragmatic` property to be exactly the number of already declared values for that individual on that property. Combined, these assertions effectively limit the existence of properties to those we explicitly declared.

The last part of `getValidPragmatics` is shown in Listing 5.22. We now create a reasoner instance for the ontology that has been constructed. We first use it to check if the ontology is consistent, which is done by determining if the ontology entails that `owl:Thing` is equivalent to `owl:Nothing`, in essence meaning there exists a contradiction. This is done to ensure that the new pragmatic does not violate conditions for any existing pragmatics.

Next, we determine all subclasses of `basic:Pragmatic`, and iterate over them. For each one, we create an axiom asserting the new pragmatic belongs to this subclass, check if the ontology is still consistent, and if so then include it in the list of valid pragmatics. We then remove the subclass axiom to prepare for the next subclass.

Finally, when all subclasses have been checked, we dispose of the reasoner (to save memory) and return the subclasses we have determined to be valid.

## 5.5 Defining Model Specific Pragmatics Sets

Creating a set of model specific pragmatics (or any set of pragmatics) is as simple as writing an ontology for it, but this is not easy if one is not already familiar with ontologies and the OWL 2 specification and syntaxes. There exist a number of tools for writing ontologies, both textual and graphical. We currently use a basic textual editor for Eclipse with rudimentary syntax highlighting and content assistance tailored for pragmatics, which will be included in the release of Pragma/CPN. We examine some ideas on how the editor can be improved in Chapter 7.

Listing 5.21: getValidPragmatics Part 3

---

```

// Axioms for closed world assumption:
//   Limiting individuals to those declared
Set<OWLNamedIndividual> allInds = ontology.
    getIndividualsInSignature(); System.out.println(allInds.
        size());
OWLObjectOneOf oneOfAllInds = dataFactory.getOWLObjectOneOf(
    allInds);
OWLEquivalentClassesAxiom cwaAxiom = dataFactory.
    getOWLEquivalentClassesAxiom(dataFactory.getOWLThing(),
    oneOfAllInds);
getManager().applyChange(new AddAxiom(ontology, cwaAxiom));
//   Ensuring they are different individuals
OWLDifferentIndividualsAxiom diffInds = dataFactory.
    getOWLDifferentIndividualsAxiom(allInds);
getManager().applyChange(new AddAxiom(ontology, diffInds));
//   Asserting object property cardinality, effectively
//   limiting property values to those explicitly declared
Set<OWLClassAssertionAxiom> cardAxioms = new HashSet<
    OWLClassAssertionAxiom>();
for(OWLNamedIndividual ind : allInds){
    Set<OWLIndividual> pragmas = ind.getObjectPropertyValues(
        hasPragmatic, ontology);
    OWLObjectExactCardinality cardExp = dataFactory.
        getOWLObjectExactCardinality(pragmas.size(),
        hasPragmatic);
    OWLClassAssertionAxiom cardAxiom = dataFactory.
        getOWLClassAssertionAxiom(cardExp, ind);
    cardAxioms.add(cardAxiom);
    getManager().applyChange(new AddAxiom(ontology, cardAxiom));
}

```

---

Listing 5.22: getValidPragmatics Part 4

---

```

// Create a reasoner and an explainer
OWLReasonerConfiguration config = new SimpleConfiguration(new
    ConsoleProgressMonitor());
PelletReasoner reasoner = reasonerFactory.
    createNonBufferingReasoner(ontology, config);

// Check for inconsistency
if(!reasoner.isConsistent()) {
    reasoner.dispose();
    // Throw exception
    throw new CausesInconsistencyException();
}

// Get all Pragmatic subclasses
Set<OWLClass> classes = new HashSet<OWLClass>();
    OWLClass pragmaClass = dataFactory.getOWLClass(":"
        Pragmatic", basicPrefix);
NodeSet<OWLClass> pragmaSubClasses = reasoner.getSubClasses(
    pragmaClass, false);

// Iterate the subclasses
for(OWLClass pragmaClass : pragmaSubClasses.getFlattened()) {
    // Declare the mock pragma individual as subclass of current
    // iteration item
    OWLClassAssertionAxiom newClass = dataFactory.
        getOWLClassAssertionAxiom(pragmaClass, pragmaInd);
    getManager().applyChange(new AddAxiom(ontology, newClass));
    // Check if ontology is still consistent, and include class
    // if it is
    if(reasoner.isConsistent()) {
        classes.add(pragmaClass);
    }
    // Remove the subclass declaration axiom
    getManager().applyChange(new RemoveAxiom(ontology, newClass)
        );
}

reasoner.dispose();
return classes;
}

```

---

## 5.6 Assessment of Requirements

We have successfully met all four requirements that were outlined in Section 1.2.

- By using Access/CPN, we have made Pragma/CPN capable of importing models created with CPN Tools.
- Models can be annotated using our extension of the ePNK framework. Pragmatics are provided using the strong expressiveness of ontologies, and by using an ontology reasoner we show only relevant pragmatics based on the selected model element.
- The editor only uses ontologies specified in the target model, letting the user choose which domain specific pragmatics to have available.
- The user is able to define model specific pragmatics by writing an ontology to describe them, which can then be included in the model the same way as domain specific pragmatics.

## Chapter 6

# Evaluation

In this chapter we will first demonstrate how to use Pragma/CPN by annotating the WebSocket CPN model from Chapter 2. We also briefly show examples from other CPN models of protocols. We discuss performance of the reasoner engine, and explain some of the limitations of Pragma/CPN

### 6.1 Annotating the WebSocket CPN Model

We begin by showing an overview of the full Eclipse window in Fig. 6.1. We have already imported the WebSocket CPN model created with CPN Tools, using the Import Wizard presented in Section 5.3. We have opened both the tree editor and the graphical diagram editor, and laid them out side by side.

The diagram editor cannot precisely replicate the look of CPN Tools, but is sufficient to easily recognise the intended layout. The elements can also be manually moved to clean up mistakes made by the Import Wizard.

At the bottom, the Properties View is open, showing the properties of the selected transition. Any of the properties can be directly edited here.

To annotate an element with a pragmatic, we right-click it and open the Annotate With submenu, shown in Fig. 6.2 (a). The menu shows only pragmatics that are applicable to this particular place, in this case Id and Channel. By selecting one of the available pragmatics, it will be added as a child of the place. The final result is seen in Fig. 6.2 (b), showing both the tree editor and the Properties View.

In Fig. 6.3, we show the diagram view of the Overview module with several pragmatics annotations applied. The Client WebSocket and Server Websocket substitution transitions have been annotated with `<<Princi-`

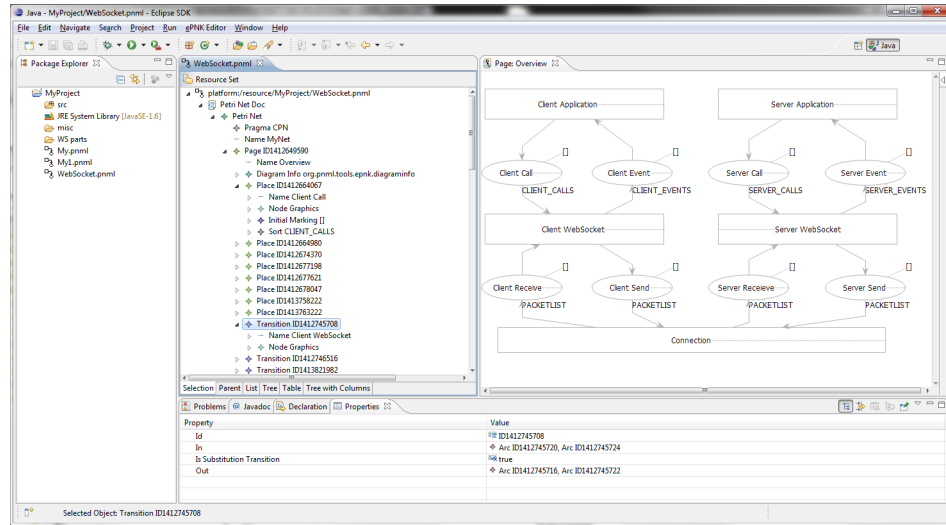


Figure 6.1: Eclipse Overview

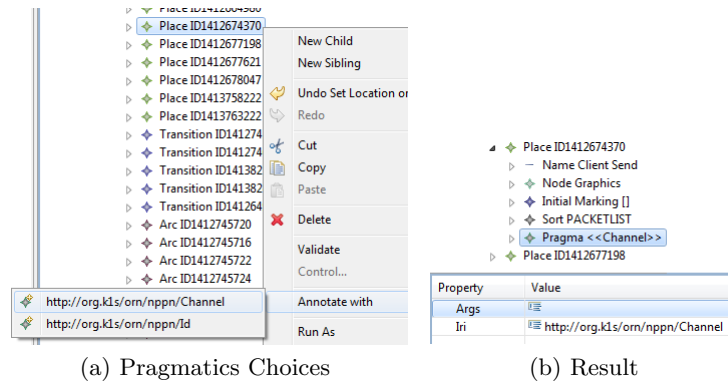


Figure 6.2: Annotating Place with Channel Pragmatic

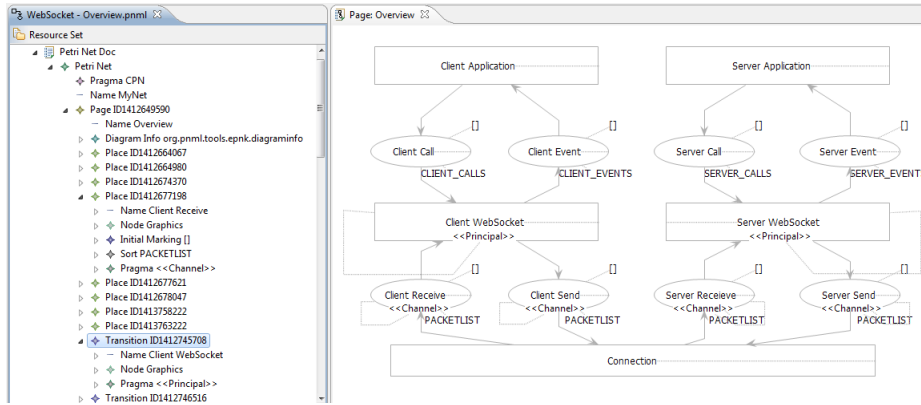


Figure 6.3: Overview Module Annotated

pal>> to signify that these are physical endpoints of the protocol. The places between them have the <<Channel>> pragmatic, representing

After creating the model-specific pragmatics ontology, we must load it into the model. This is done by right-clicking the PragmaticsOntology in the model, choosing Add Ontology under the Pragmatics menu (Fig. 6.4 (a)), and selecting the desired ontology document from the project browser (Fig. 6.4 (b)).

When we now open the Annotate With submenu, we see the new pragmatic is available.

### 6.1.1 The Kao-Chow Protocol Model

## 6.2 Performance

We discovered that the Annotate With submenu takes several minutes to appear. This is due to the axioms that are added to the ontology to enforce closed world assumption, causing the reasoner engine to take a long time to classify the ontology. These axioms consist of the axiom stating that owl:Thing is equivalent to the asserted individuals and the axiom stating that asserted individuals are different. As a test, we temporarily commented out these two axioms, and the Annotate With submenu would then show up almost instantly. However, its contents would not always be correct, for instance the :External pragmatic would be incorrectly available on all transitions. Another workaround is to split the model into several files with one page in each file, which is what we did to be able to annotate the

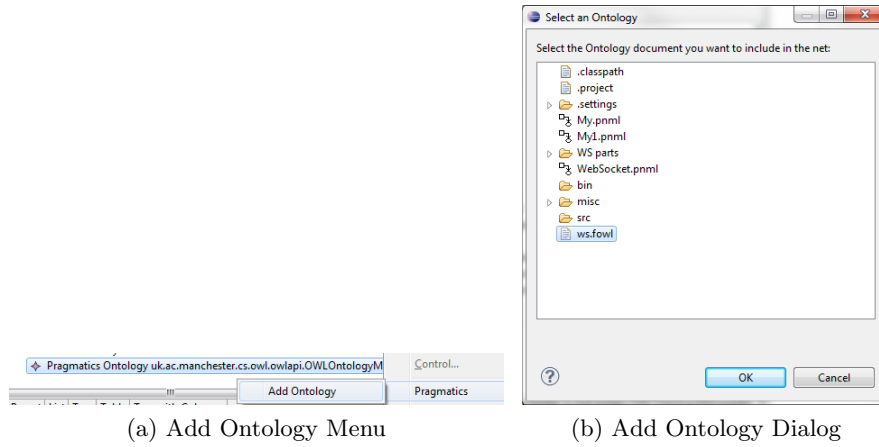


Figure 6.4: Steps for Adding Pragmatics Ontology

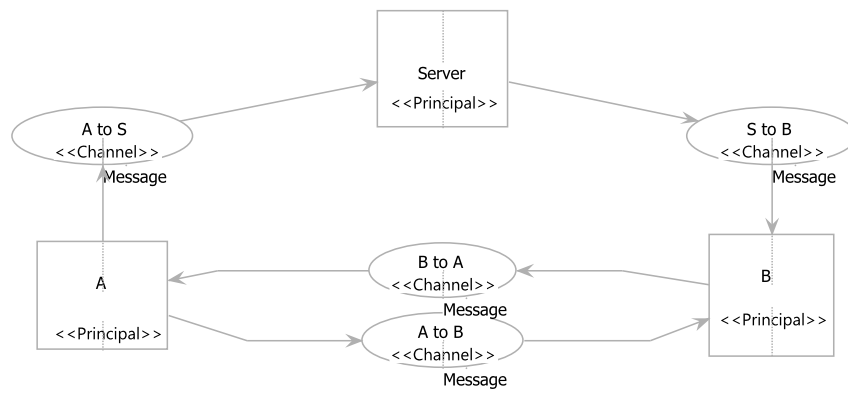


Figure 6.5: KC



WebSocket CPN model. Hypothetically it is possible to design pragmatics with constraints spanning

A delay of more than a couple of seconds on a frequently used part of an interactive user interface is not acceptable. We found through gradually constructing a test model that the delay becomes noticeable when the model has 25 elements, giving a delay of about one second. The Kao-Chow model has 96 elements, which requires about 7 seconds to compute the valid pragmatics. The WebSocket model has 372 elements, and takes several minutes to classify.

An obvious inefficiency in our approach is that we are constantly recreating the entire ontology representing the current state of the model. It is worth noting that EMF has an extensive notification framework for reacting to model changes, and it should be possible to leverage this to efficiently maintain an accurate ontology of the loaded CPN model. However, the crucial axioms that enforce CWA cannot be edited in place, and would thus still need to be removed and added every time the model is changed.

We believe this problem needs to be addressed at the level of the reasoner engine implementation. Thanks to the design of OWL API, switching reasoners is fast and simple. However, none of the four reasoner engines we tested (listed in Section 4.6) gave acceptable performance. Still, as a proof of concept for the usefulness and expressiveness of ontologies, our approach is sufficient.

## 6.3 Limitations

The graphical diagram editor of ePNK has some shortcomings and we do not consider it fully mature. Arc appearance is not serialised, making it impossible to accurately copy layout from CPN Tools. Labels are not attached to their parent elements, and by default are placed at coordinates 0,0. As ePNK is continually being updated, we look forward to improvements to this part of the framework.

Accurate modeling of module instancing (the fact that a module can be a submodule of several modules) is impractical in the current version of ePNK. This is again due to the fact that merging EMF models is not possible, meaning we must subclass existing leaf classes for model elements and therefore inherit their properties. A subclass of RefPlaces could be used to represent either a port place or a socket place, but either case presents problems as a RefPlaces can only reference a single other place.

As a port place, it could be on a module that has several instances, so

it would either need to contain a reference for each instance (which is not possible due to being limited to one) or the module itself would need to be duplicated (which is impractical from a user perspective since all annotations would have to be placed once on each module instance). As a socket place, it could be connected to several submodules (as is the case with the Connection status place in the WebSocket CPN model) and would need to have a reference to the ports in each of them.

## Chapter 7

# Conclusion

With Pragma/CPN, we have constructed a fully functional proof of concept that demonstrates the advantages and disadvantages of using ontologies to manage code generation pragmatics and applying them to a CPN model. As shown in Chapter 5, this prototype meets all of the requirements in Section 1.2. Through our evaluation by using it to annotate CPN models of different sizes, we have shown that using ontologies gives a highly expressive means to defining pragmatics as well as providing automatic validation of the annotated model. However ontology reasoning using CWA yields decreasing performance as model size increases, to a point that makes our prototype unusable for the WebSocket CPN model. Still, we believe our approach has potential, and warrants further optimisation and research.

### 7.1 Future work

Continue the work of Simonsen, defining more general and domain-specific pragmatics. Also, assert required configuration arguments.

Finding or developing ontology reasoner tailored for CWA.

A specialised tool for creating model specific pragmatics would have been ideal. Such a tool would give simple mechanics using GUI controls for specifying which model elements a pragmatic can be attached to, and which parameters it has. The Plugin Manifest editor is a good example of what we have in mind. However, this could not be included for this thesis due to time constraints.

Implement structured labels for the CPN Type. Could enable placement of pragmatics inside expressions.

When importing, let user choose between CPN and Pragma CPN. Alternatively, mechanic for upgrading the Type.

Accurately model sub-modules, ports and sockets. This enables more advanced pragmatics; for instance the Principal pragmatic could be specified to only be available for substitution transitions on top-level modules.

## 7.2 Acknowledgments

Thank supervisor Lars Michael Kristensen

Kent Inge Fagerland Simonsen

Michael Westergaard for help with Access/CPN

Ekkart Kindler for extensive help with ePNK

## Appendix A

# Downloading Pragma/CPN

At the time of writing we are using a development version of ePNK that is yet to be published, and are therefore unfortunately unable to publish Pragma/CPN as a plugin that can be installed from an update site through the Eclipse Plugin Manager. However, the source code is available at <https://github.com/stylpe/Thesis/tree/master/workspace> and can be imported into Eclipse as a collection of Eclipse Projects.

ePNK is scheduled to release version 0.9.3 later this year, and at that time we will add instructions for installing Pragma/CPN on that URL.



# 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.
- [BM04] D. Beckett and B. McBride. *RDF/XML syntax specification (revised)*, 2004.
- [DCTTDK11] K. Dentler, R. Cornet, A. Ten Teije, and N. De Keizer. Comparison of reasoners for large ontologies in the owl 2 el profile. *Semantic Web*, 2(2):71–87, 2011.
- [dW04] J. desRivieres and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.
- [EGK<sup>+</sup>02] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, pages 594–597. Springer, 2002.
- [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 Kunihiro 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.
- [Kru04] P. Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.
- [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.



- [Met06] Meta object facility (MOF) 2.0 core specification. <http://www.omg.org/spec/MOF/2.0/>, 2006. Version 2.
- [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.
- [OWL09] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. Available at <http://www.w3.org/TR/owl2-overview/>.
- [RWL<sup>+</sup>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.
- [SPG<sup>+</sup>07] E. Sirin, B. Parsia, B.C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.
- [TH06] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. *Automated Reasoning*, pages 292–297, 2006.

- [TPR10] Edward Thomas, Jeff Z. Pan, and Yuan Ren. TrOWL: Tractable OWL 2 Reasoning Infrastructure. In *the Proc. of the Extended Semantic Web Conference (ESWC2010)*, 2010.
- [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	The OMG MOF Metamodelling Layers . . . . .	3
1.2	Top level module of the Kao-Chow model . . . . .	4
2.1	Sequence Diagram of the WebSocket protocol . . . . .	10
2.2	Overview module of the WebSocket CPN model . . . . .	11
2.3	The Client Application Module . . . . .	15
2.4	The Client WebSocket Module . . . . .	20
2.5	New Connection submodule . . . . .	23
2.6	Process Response submodule . . . . .	25
2.7	Wrap And Send submodule . . . . .	26
2.8	Fragment And Queue submodule . . . . .	27
2.9	Unwrap And Receive submodule . . . . .	30
2.10	Defrag Module . . . . .	32
2.11	The Connection module . . . . .	33
2.12	The Server WebSocket Module . . . . .	34
2.13	Get Connection Request . . . . .	35
2.14	Send Connection Response . . . . .	36
2.15	The Server Application Module . . . . .	37
3.1	Configuration: No messages . . . . .	40
3.2	State space for configuration with no messages . . . . .	42
3.3	Problem with the Pong reply modeling . . . . .	44
3.4	One message . . . . .	48
3.5	State space for one ping, one message configuration . . . . .	49
4.1	Application Overview Diagram . . . . .	56
4.2	The Eclipse RCP . . . . .	57
4.3	PNML Core Model . . . . .	59
5.1	CPN model type diagram . . . . .	72

5.2	Annotated CPN model type diagram . . . . .	77
5.3	The Import Wizard . . . . .	84
6.1	Eclipse Overview . . . . .	96
6.2	Annotating Place with Channel Pragmatic . . . . .	96
6.3	Overview Module Annotated . . . . .	97
6.4	Steps for Adding Pragmatics Ontology . . . . .	98
6.5	KC . . . . .	98

# Listings

2.1	Overview colour sets . . . . .	13
2.2	Simple Colourset Variables . . . . .	14
2.3	Client Application Variables . . . . .	16
2.4	The <code>parseUrl</code> function . . . . .	17
2.5	HTTP colour sets . . . . .	19
2.6	WebSocket colour sets . . . . .	21
2.7	WebSocket Module Variables . . . . .	22
2.8	Masking functions . . . . .	22
2.9	<code>httpReqFromUrl</code> . . . . .	24
2.10	<code>isResponseValid</code> . . . . .	25
2.11	<code>isData</code> . . . . .	28
2.12	<code>wrap wrapmsg</code> and <code>fragment</code> . . . . .	29
2.13	<code>append</code> . . . . .	32
2.14	<code>unmask</code> . . . . .	33
2.15	<code>isRequestValid</code> . . . . .	36
3.1	Fixed masking key . . . . .	42
3.2	State Space Report: Statistics . . . . .	43
3.3	State Space Report: Best Integer Bounds . . . . .	43
3.4	State Space Report: Best Multi-set Bounds . . . . .	45
3.5	State Space Report: Home Properties . . . . .	45
3.6	State Space Report: Liveness Properties . . . . .	46
3.7	One message . . . . .	47
3.8	One ping then one message . . . . .	48
3.9	One message then one ping . . . . .	50
3.10	One long message . . . . .	51
3.11	Upper Multi-set Bounds - long message fragments . . . . .	51
3.12	Ping Text Close . . . . .	52
3.13	Large configuration . . . . .	52
5.1	The CPN Ontology: Opening . . . . .	63
5.2	The CPN Ontology: Base Classes . . . . .	65

5.3	The CPN Ontology: Place Class . . . . .	66
5.4	The CPN Ontology: Transition Classes . . . . .	66
5.5	The CPN Ontology: Arc Class . . . . .	67
5.6	The CPN Ontology: Arc Class . . . . .	67
5.7	The Basic Pragmatics Ontology . . . . .	68
5.8	The Generic Pragmatics Ontology . . . . .	69
5.9	ePNK Petri Net Type Definition Extension . . . . .	74
5.10	constraintProvider Extension . . . . .	75
5.11	Constraint Implementation . . . . .	76
5.12	Add Ontology Menu Extension . . . . .	79
5.13	AddOntologyAction setActivePart() and selectionChanged() .	80
5.14	AddOntologyAction run() . . . . .	81
5.15	addOntologyFromFile() . . . . .	82
5.16	Add OntologyPragmatic Dynamic Menu Extension . . . . .	86
5.17	AddOntologyAction run() . . . . .	87
5.18	AddPragmaAction . . . . .	88
5.19	getValidPragmatics Part 1 . . . . .	89
5.20	getValidPragmatics Part 2 . . . . .	90
5.21	getValidPragmatics Part 3 . . . . .	92
5.22	getValidPragmatics Part 4 . . . . .	93