Contents

Chapter 1

State Space Analysis of the CPN Web Socket Protocol

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

1.1 State Spaces

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

TODO: Figur av SS initiell modell, med forklaring

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

1.1.1 Strongly Connected Component graph

In graph theory, a strongly connected component (SCC) of a graph is a maximal subgraph where all nodes are reachable from each other. An SCC graph has a node for each SCC of the graph, connected by arcs determined

by the arcs in the underlying graph between nodes that belong to different SCCs. An SCC graph is acyclic, and an SCC is said to be trivial if it consists of only one node from the underlying graph.

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

1.1.2 Application of State Spaces

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

This can be remedied by picking smaller configurations that encapsulate different parts of the system. This was necessary with the WebSocket Protocol model, as the complete state space took too long to generate.

Diskutere mer hva som måtte gjøres med WebSocket, illustrasjon, konkret eksempel fra modellen

Another aspect that must be considered prior to state space analysis is situations where tokens can be generated an unlimited amount of times 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.

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. Otherwise, a complete state space can not be achieved, since occurrence sequences might not converge after branching, and it's impossible to make sure all possible values have been considered.

Vanskelig å formulere...

For small color sets (generally defined as discrete sets usually with less than 100 possible values), binding random values to arc expressions can be done in two ways: By calling ran() on the colorset, or by using a free variable in the arc expression. The former, ran(), is non-deterministic and However, a free variable, which is a variable that does not get assigned a value in an expression, will also bind to a value picked at random from the color set during simulation, and also lets the state space generator pick each of the possible bindings from the values available in the colorset and thus generate all possible successive states.

Color sets that use values from a large or unbounded range, or from continuous ranges like floating point numbers, are considered large color sets, and using random values from such color sets can make it impossible

or impractical to generate a complete and correct state space. It can be worked around by providing single or tiny sets of fixed preset values. The CPN Tools manual has examples on how to do this.

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.

1.2 State Space Report

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

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

1.2.1 Statistics

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

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

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

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

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

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

1.2.2 Boundedness Properties

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

```
Best Integer Bounds
                            Upper       Lower
   ClientApplication
        Active_Connection
                            1           0
        Conn_Result
                            1           0
        Connection_failed
                            0           0
        Messages_received
                            1           1
        Messages_to_be_sent
                            0           0
   [...]
```

Many places show a lower and upper bound of 1. This shows a weakness in the approach of using lists to facilitate ordered processing of tokens: We cannot see the actual number of tokens that are in the place, because technically there is just a list there.

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

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

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

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

1.2.3 Home Properties

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

Vise marking

```
Home Markings
   [17]
```

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

1.2.4 Liveness Properties

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

```
Dead Markings
   [17]


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

Live Transition Instances
   None
```

from where? for which? formuler

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

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

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

The state space report also contains fairness properties, but this does not apply to our model since it contains no cycles. We will not explain more about this here, and instead refer to [JK09] chapter 7 for more information.

1.3 TODO:

Skrive om resten av analysene.

Flette inn dette:

1.3.1 Error discovery

An error in the model was discovered this way, in the Unwrap and Receive module, where the pong reply was adding a new list instead of appending to the old one in outgoing messages.

Flere detaljer - hvordan viste fielen seg? Figur, før og etter

Chapter 2

Technology and Foundations

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

TODO: Krav

2.1 Representing CPN Models

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

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

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

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

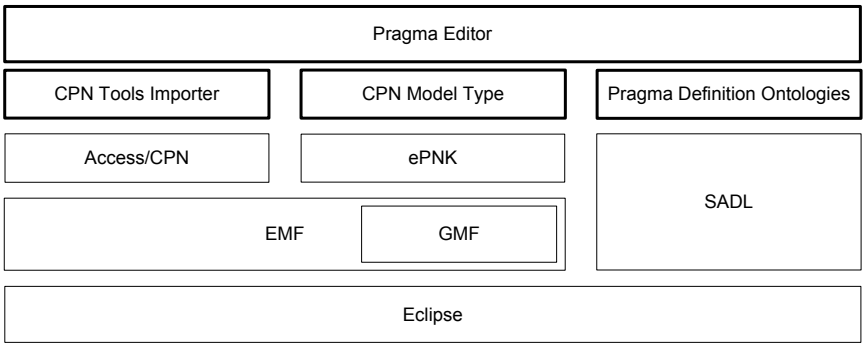

Figure 2.1: Application Overview Diagram

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

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

Fig. 2.1 shows the different elements that make up the application. The elements with bold frames are the ones newly created for this thesis, while the rest below are the existing solutions used and built upon. These will be described in the following sections, from the bottom up.

2.2 Eclipse IDE

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

The Eclipse IDE is built on the Eclipse Rich Client Platform (RCP) shown in Fig. 2.2. At the bottom of this we have the Platform Runtime, based on the OSGi framework, which provides the plugin architecture.

TODO

The Workspace

THe Workbench

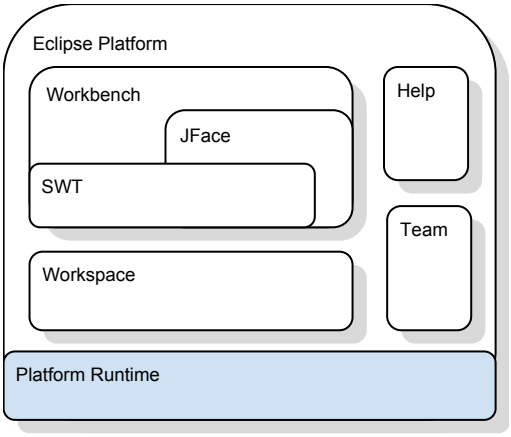

Figure 2.2: The Eclipse RCP

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

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

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.

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

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

It is possible to package Eclipse with sets of plugins to form custom editions of Eclipse that are tailored for specific environments and programming languages. Aptana Studio is one example, aimed at Ruby on Rails and PHP development.

2.3 Eclipse Modeling Framework (EMF)

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

EKSEMPEL

2.3.1 Graphical Modeling Framework (GMF)

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

2.4 ePNK: Petri Net framework

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

There are several reasons why ePNK is a good choice:

referanse?

- It saves models using the ISO/IEC 15909 standard file format Petri Net Markup Language (PNML),
- It is currently actively developed,
- It is designed to be generic and easily extendable by creating new model types, and
- It includes both a tree editor and a graphical editor, provided through GMF.

ePNK includes definitions for the core PNML model type, as well as two subtypes of Petri Nets. The first is P/T-Nets (Place/Transition Nets), which expand on the core model with a few key items: initial markings for placesas integers, inscriptions on arcs, and constraining arcs to only go between a place and a transition (this is not enforced in PNML, as there are Petri Net variants that allow this).

The second type included with ePNK is High level Petri Nets (HLPNG). This type adds Structured Labels which are used to represent model declarations, initial markings, arc expressions and transition guards. These are parsed and validated using a syntax that is inspired from (but not the same as) 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 substitutin transitions. Also, the structured labels are not compatible with CPN ML syntax from CPN Tools, and for our prototype, these structured labels are not necessary with regard to annotations. They might be useful in a future version, where for example pragmatics are available depending on things like the colorset of a place or the variables on an arc, but initially this is considered to be out of the scope of this thesis.

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

2.5 Access/CPN: Java interface for CPN Tools

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

The model definition used by Access/CPN is very similar to that of ePNK.

Diskutere mer i implementation?

2.6 Semantic Application Design Language (SADL): Ontologies

Ontologies are a way to present information and meta-information so that it can be understood by computers. Essentially, this is done by defining classes

that have properties, relations and constraints, and then present information with these classes.

There is a lot of ongoing research on this subject, especially to create a semantic web, that is extending web pages to provide meta-information about the content they contain and enabling software to understand it and reason about it. The Web Ontology Language (OWL) is the standard for representing ontologies.

Ref, sjekk om det er ISO

SADL is an Eclipse plugin that defines an english-like syntax for defining ontologies, and comes with a text editor that features syntax highlighting, parsing and validation. This is useful as we can possibly reuse the editor in our plugin for defining model-specific pragmatics. SADL ontologies can be compiled to OWL format.

It also has tools to parse and reason with these ontologies, which we will use to filter and validate which pragmatics are available for different model entities.

TODO: Lite eksempel på ontology.

2.7 Summary

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

Chapter 3

Analysis and Design

3.1 Requirements

Before we describe the plugin itself, the requirements need to be detailed. There are four main parts to this:

- Loading models created in CPN Tools.
- Annotate model with pragmatics.
- Load sets/classes of pragmatics to add to model
- Create model specific pragmatics on the fly

3.2 Test cases

(NYI, trenger eksempler)
Simple protocol
Kao-chow

3.3 Defining Pragmatics

TODO: Detaljert info om pragmatikker og hva de er

A Pragmatic is an annotation on a model element that describes how it should be translated into code.

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

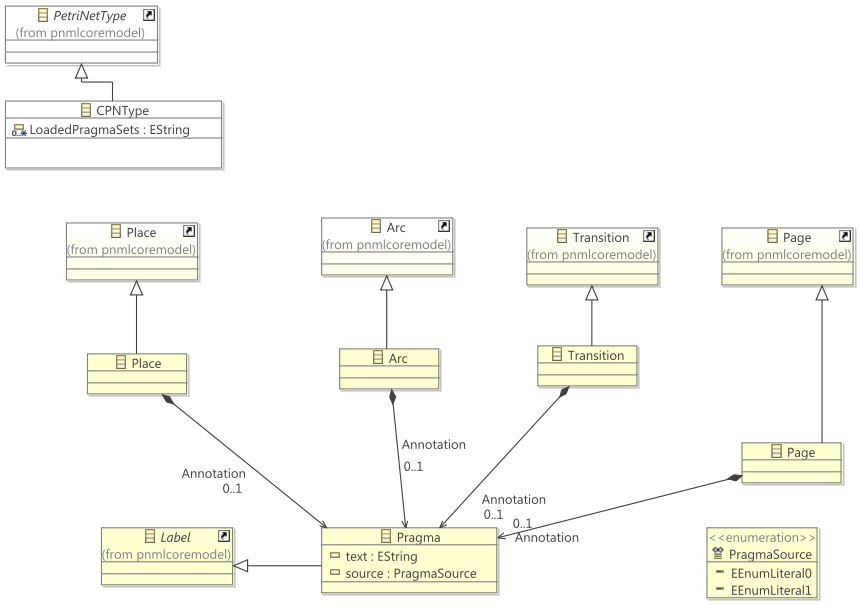

Figure 3.1: Model type diagram

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

Domain-specific pragmatics are things that apply to all (or many) protocols within a domain. An example is security protocols, where domain specific operations are encryption and decryption, nonce generation, etc.

Model-specific pragmatics apply only to the model in which they are defined. The WebSocket Protocol has a few examples, like masking and unmasking of frames, and verification of received frames.

3.4 THe CPN Ontology with Pragmatics

(Obsolete ettersom Kent er på vei bort fra SADL, skal skrives om til Manchester syntax)

3.5 The CPN model type for ePNK

TODO: Two-tiered model possibility? CPN -¿ Annotated CPN

Every Petri Net model created in ePNK is initially only defined by the PNML Core Model type, but can add a PetriNetType object to extend it with the features of another model type. Only one such object can exist for a Petri Net. Once such an object is added, ePNK will dynamically load the related plugin(s), and the menus for adding new objects to the net will include any new classes that the Petri Net Type defines.

A custom Petri Net Type is made by creating an Eclipse Plugin project. The plugin manifest then needs to be edited to define this plugin as an extension to the org.pnml.tools.epnk.pntd extension point of ePNK.

Next, an EMF model should be created. This model should inherit the PNML Core Model from ePNK, or any other model that already does this (like P/T-Net and HLPNG does). The first thing this new Model Type should define is a subclass of PetriNetType with the name of the new Model Type. This is done in the top left corner of our diagram. This class is what will appear in the menu to add a Petri Net Type to a new model in ePNK.

In order to change the functionality of existing classes such as Place, Arc and Page, we need to create subclasses of them. EMF does not support merging of models, so we can not define new properties or relations directly on the referenced classes of the Core Model. Instead, ePNK will check the new Petri Net Type model for subclasses with the same name, and load these dynamically instead of the base classes. This can be seen in our diagram, where Place, Arc, Transition and Page are all subclassed from the classes referenced from the Core Model. These subclasses then have references to the Pragma class. As we can see in Fig. 3.2, Pragma objects are now available as children to Places.

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

3.6 Importing from CPN Tools

CPN Tools saves models in .cpn files as plain XML. Access/CPN is a set of tools that can parse these files and represent the model with EMF classes. It has many tools to analyse the CPN models, but only the model importer is interesting to us.

The EMF model for CPN models that Access/CPN defines uses many of the same class names as ePNK, making code difficult to write and read due to the need to use fully qualified classpaths to avoid name collisions. Because of this, and the fact that it was missing a few features like importing graphics, we early on considered creating a new parser based on the code

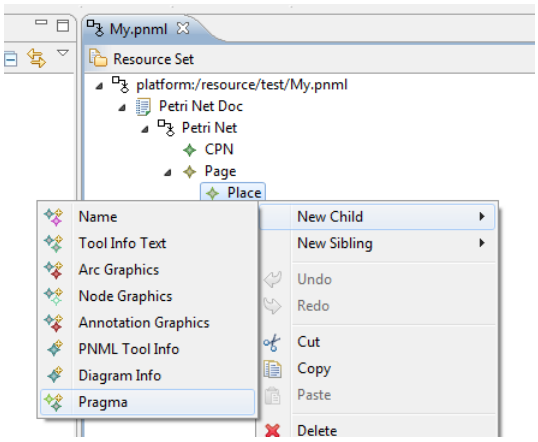

Figure 3.2: Adding Pragma as child to Place

in Access/CPN. But it has continually improved during development of our plugin, and is now also capable of parsing graphics data. By using this plugin instead of rewriting it, we can also benefit from further updates more easily.

TODO: Sjekke med repo-admin om problem med SVN, får ikke sjekket ut siste revisjon

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

3.7 Creating annotations

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

3.8 Choosing Pragmatics Sets

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

3.8.1 Creating custom pragmatics

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

Bibliography

[ecl] Eclipse ide.

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