A Compiler and Runtime Environment for Execution of Coloured Petri Net Models

Andreas Garvik

Master's thesis in Software Engineering at

Department of Computer science, Electrical engineering and Mathematical sciences, Western Norway University of Applied Sciences

Department of Informatics, University of Bergen

May 2022

Western Norway University of Applied Sciences

Abstract

Reliance on software systems is ever increasing in our world. Alongside the application of concurrent software systems that embody communication, synchronization and resource sharing. Many problems in software engineering have strict requirements regarding concurrency and correctness. Designing correct concurrent software is challenging, and a range of formalisms and tools exists that can help the development process. Coloured Petri nets are an extension of the mathematical concept of Petri nets and a widely used language for specification and modelling concurrent systems. CPN Tools is a tool for editing, simulating, and analyzing Coloured Petri nets models. A need has been identified to develop new software tools to execute Coloured Petri net models to facilitate further development and increase portability. This thesis proposes a compiler and a runtime environment for Coloured Petri nets using the F# programming language and the .NET platform. The result is a CPN simulation system consisting of a compiler that can parse a .cpn file and generate code and a simulator that can use the code generated and execute the model. A state-space exploration strongly supports the functional correctness of the system.

Acknowledgements

I would like to thank my supervisor Lars Michael Kristensen. It has indeed been a pleasure and honour having you as my supervisor. Your knowledge and your ability to present it so clearly are inspiring. You have given me great support and explanations when needed, and I have always felt welcome at your office.

Contents

1 Introduction **6**
1.1 Context and Motivation 7
1.2 Research Questions and Expected Results 7
1.3 Related Work 8
1.4 Outline 9

2 Background **10**
2.1 Two-phase commit protocol example 10
2.2 Coloured Petri nets 11
2.3 CPN Tools 12
2.4 Core concepts 13
2.4.1 Multi-sets 14
2.4.2 Marking 14
2.4.3 Colour-sets and Variables 14
2.4.4 Bindings 14
2.4.5 Enabling and Occurrence 15
2.4.6 Executing a CPN model 15

3 Compiler and Runtime Environment **16**
3.1 Overview 16
3.2 F# and the .NET Platform 20
3.2.1 Discriminated Unions, Tuples and Records 20
3.2.2 Option type 20
3.2.3 Pattern matching 20
3.2.4 Code quotations and F# Compiler Service 21
3.3 Multi-set implementation in F# 22
3.4 F# as inscription language 24

4 Parsing and Code Generation **26**
4.1 Loader Component 26
4.1.1 Declarations 27
4.1.2 Places, Arcs and Transitions 31
4.2 Parser Component 35
4.2.1 Arc expressions 36
4.2.2 Marking 37
4.2.3 AbstractTransitions and AbstractArcs 37
4.2.4 Finding the free variables 39

4.2.5 Simple types 40
4.2.6 Pattern binding basis 41
4.2.7 Enabling 43
4.2.8 Occurrence 43
4.3 Output Component 43

5 Enabling and Occurrence **45**
5.1 Overview 45
5.2 Enabling of bindings 46
5.2.1 Binding type 46
5.2.2 Binding and partially binding functions 47
5.2.3 Enabling functions 49
5.3 Occurrence of bindings 50
5.3.1 Occurrence function 51

6 Evaluation **52**
6.1 Simulation 53
6.2 State-Space Exploration 54
6.3 Performance 55
6.4 Examples 56
6.4.1 Dining Philosophers 56
6.4.2 Simple Protocol 58
6.4.3 Resource Allocation 59
6.4.4 Distributed Database 60

7 Conclusion and Further Work **62**
7.1 Conclusion 62
7.2 Future work 63

A Source code **64**

B State Space Exploration in CPN Tools **65**

Chapter 1

Introduction

Software engineering is about building trustworthy, reliable and correct software systems. Today, many applications are distributed systems and can handle concurrency, which may introduce subtle bugs that can be hard to identify during development and in a running system. It might be of great help and sometimes necessary to build a model of the system to be developed.

Modelling is a standard part of the development process in other engineering disciplines. It can help test and validate the system built, verify its correctness, substitute as an early prototype, and paint a bigger picture of the behaviour by simulation. There is a range of different kinds of models to create, modelling languages to use and tools that can be used for this purpose. Coloured Petri nets [47] (CPN) is a modelling language that emphasizes correctness and builds on a solid mathematical foundation.

The main goal of this master's thesis is to implement a compiler and a runtime environment for Coloured Petri net models, a CPN simulation system able to parse, represent, simulate and execute CPN models. We chose to develop this system in the F# programming language [1], originally designed and implemented at Microsoft Research [2]. F# runs on the .NET platform [3] and is an open-source, functional-first, general-purpose, strongly typed language that appeared in 2005 [52].

1.1 Context and Motivation

Coloured Petri net is a graphical modelling language used by the industry in many areas such as network architectures, healthcare, protocols and military defence [49]. In [49], four different case studies have been selected to illustrate how CPN can be used in various software development phases, design, validation and implementation. CPN relies on a functional programming language that emphasizes expressions and declarative programming.

Today, the primary tool used to create CPN models is CPN Tools [47]. However, there are difficulties associated with the further development of this tool. Issues regarding the technologies used to build the tool, for instance, the Standard ML programming language [4], which today lacks current support. Another issue is the amount of technical debt accumulated related to using the BETA programming language [5] to implement the graphical user interface. Developing new tools with newer and more modern technologies such as F# and the .NET platform will increase the portability and availability of CPN software tools. Furthermore, it will better facilitate research and further development.

The general industry use of CPN is a valid argument for pursuance. There will be much interest in software tools for working with and evaluating CPN models, especially portability. The prevalence of the .NET platform will largely strengthen portability. More examples of large-scale practical use of CPN are available online [6]. In addition, there exists a large international Petri Nets community [7] organizing conferences worldwide.

Modelling and verification of software can be fundamental in cases where the code is not trivial to change at a later stage. An example is in the context of the ongoing research project, SFI Smart Ocean [8]. CPN and the software developed in this thesis can help develop devices to be submerged underwater, and the overall software architecture of the application and development platform being developed in the SFI Smart Ocean project.

1.2 Research Questions and Expected Results

This thesis is centered around the following research questions.

RQ1. How can F# be used as inscription language in CPN models?

RQ2. How can F# and the .NET platform be used to implement a compiler that generates code representing CPN models?

RQ3. How can F# and the .NET platform be used to develop a runtime environment for the simulation of CPN models?

The F# programming language shares a lot of language concepts, syntax and features with Standard ML, the language used as both the inscription language and implementation language in CPN Tools. Both emphasize functional principles and have strong language support for pattern matching. Other similar alternatives may include the Scala programming language [9], running on the Java Virtual Machine, or the Haskell programming language [10], a purely functional programming language. Nevertheless, F# was chosen based on its strong support for pattern matching and support for expression-orientated programming, which is essential when used as an inscription language for CPN models, as we will later discuss. High-quality libraries and features for metaprogramming are also valid arguments. In addition, the seamless interoperation between F# code and C# code which could be used to develop a GUI makes F# a powerful candidate technology.

Regarding RQ1, the expected result is that F# can be used as the inscription language in CPN models due to its similarities with Standard ML. Regarding RQ2 and RQ3, F# is a mature language with a massive amount of libraries available from the .NET platform. Also, everything written in any language for the .NET platform is callable from F#. The FSharp Compiler Service [11] makes it possible to host the compiler or an interactive evaluation session in the application, equipping it with the capabilities to parse and evaluate expressions.

1.3 Related Work

Kristensen and Christensen showed [50] how Standard ML could be used to implement CPN. They relied heavily on the pattern matching capabilities of the language and introduced two essential enabling inference algorithms. This thesis draws much inspiration from the paper.

There exist many tools for Petri nets [12], but some might not still be available due to lack of current support. Many of them for working with high-level Petri Nets developed using different programming languages for inscription and implementation.

Renew - The Reference Net Workshop [13] is a multi-formalism editor and simulator. It is written in Java and uses Java as inscription language, and it takes a more object-oriented approach, supporting object-based modelling. The use of Java has its benefits, such as running anywhere the Java virtual machine is available, but the programming language limited support for pattern matching.

The Helena Petri net tool [14] is a tool for high-level Petri nets which can be used for analysis and model checking. It has its own specification language and can possibly interface with C code. It can perform a state-space exploration, which will be discussed in later chapters.

ITS-tools [15] is a model-checker that supports a variety of formalisms. It uses its own concrete formalism called GAL, which has a C-like syntax, but offers import capabilities of Petri nets that will handle conversion to GAL.

Wolfgang [16] is a tool to create models, supporting, among others, Coloured Petri nets as defined in the SEPA library [17].

Other existing tools for working with Place/Transition nets, sometimes called just Petri nets, are mist [18], PTN [19], PN-Suite [20], Tina [21], TAPAAL [22] and yasper [23]. CozyVerify [24] another software tool related to Petri Nets, which uses its own defined inscription language, which may have some limitations regarding the unfolding of models.

The difference between the mentioned alternative software tools and our system developed in this thesis is mainly the technology and the programming paradigm. We wanted to build the CPN simulator system with well established and modern technologies and focused on the functional programming paradigm. We also chose a path of compilation, where we produce code to be used to execute models. In contrast, other tools chose only to interpret the model and keep everything in the environment. To the best of our knowledge, no compilers exist that can translate CPN models created in CPN Tools to any other language or platform or any run-time environment for CPN models using F# and the .NET Platform.

1.4 Outline

In the rest of this thesis, the CPN simulation system developed will sometimes be referred to only as the *system* and the chapters are structured as follows. Chapter 2 introduces CPN and CPN Tools using a two-phase commit protocol as an example. Chapter 3 gives a high-level view of the compiler and the run-time environment that has been implemented, its structure, and its components. It also presents the key features of F# and .NET that the system depends on. Chapter 4 takes a deeper dive into the parsing and the code generation part of the system, showing precisely how its implemented. Chapter 5 shows the role enabling and occurrences play in the simulation of a CPN model and how the generated code is used. Chapter 6 discusses the overall system, performance and correctness and shows how it works on some representative examples. Finally, Chapter 7 concludes on research questions, presents some limitations and proposes future work and improvements.

The thesis assumes the reader to be familiar with programming, but no prior knowledge of Coloured Petri nets is assumed. The F# programming language and the .NET platform will not be presented and explained in their entirety. The reader can find more information about them in [25] and [26], respectively.

Chapter 2

Background

This chapter gives the reader the background information necessary to understand the rest of this thesis. A two-phase commit protocol is introduced, which will be used throughout this thesis to exemplify and explain the different concepts of CPN and the CPN simulation system developed. A brief introduction to Coloured Petri nets and CPN Tools is also given.

2.1 Two-phase commit protocol example

Database servers use the protocol to handle transactions that modify data on multiple database servers [27]. Every transaction has a coordinator process and one or more worker processes. A sequence diagram illustrating the interaction and communication between the entities is shown in Figure 2.1. The coordinator decides whether a transaction should be committed or aborted, and the role is given to the current database server. It sends a *prepare* message to all the workers. Each worker responds if it is ready to carry out the transaction with a *vote* in the form of a yes/no message. The coordinator then collects the responses and decides based on the votes. The *decision* message sent to the participants are either an *commit* or *abort* message. If all workers vote *yes*, the message is to commit and carry out the transaction. If some workers vote yes, but others do not, the message is to abort the task. The abort message is only sent to those workers that votes yes. In both cases, the workers voting yes send an *acknowledgement* in return. If none of the workers voted yes, no message needs to be sent and the protocol loops back to the start. Two worker processes are used in this protocol example for the rest of the thesis.

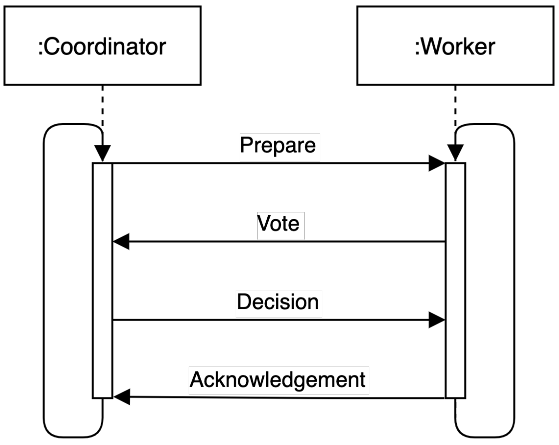


Figure 2.1: Sequence diagram of the two-phase commit protocol

2.2 Coloured Petri nets

Coloured Petri net is, as mentioned in the introduction, a graphical language for formal modelling and validation of concurrent systems. It is an extension of Petri nets [51] and belongs to the class of high-level Petri nets [48]. This implies that Petri nets are combined with a programming language. Petri nets provide the formal foundation for modelling concurrency and synchronization, and the programming language provides the primitives for modelling data manipulation and creating compact and parameterizable models.

A CPN model consists of places, transitions and arcs. Jensen's formal definition of CPN syntax and semantics can be found in [46]. In short, places may hold *tokens*. A token is a value of a defined type. All the tokens present on places constitute the model's state, called a *marking*. A model is a bipartite graph connecting transitions and places through arcs. Transitions may be *enabled* in a given marking, and enabled transitions may *occur*. A transition is enabled if sufficient tokens exist *on* the places connected to it via input arcs. The evaluation of the expressions on the input arcs determines the tokens removed from the places. Evaluating the expressions on the output arcs connected to a transition determines the tokens added to the places connected. When a transition occurs, the removal and addition of tokens happen instantaneously. An occurrence of a transition will result in a new marking of the system, called a *step* in the execution of the model. When considering interleaving semantics, only a single transition may occur in each step. Concurrent semantics is another option, where it is possible to execute more than one transition concurrently, but it is not considered in this thesis. A place may only contain the type of tokens defined by its associated type (colour-set). The marking of a place is a *multi-set* of token values. A multi-set may have multiple tokens with the same value. Multi-sets will be explained in more depth later. For more details on CPN, the reader is suggested to consult the article [47].

2.3 CPN Tools

The two-phase commit protocol can be created as a CPN model using CPN Tools [53]. The CPN model of the protocol is shown in Figure 2.2. The model uses Standard ML as the inscription language, which the only supported. The left part is the coordinator, and the right part is the workers.

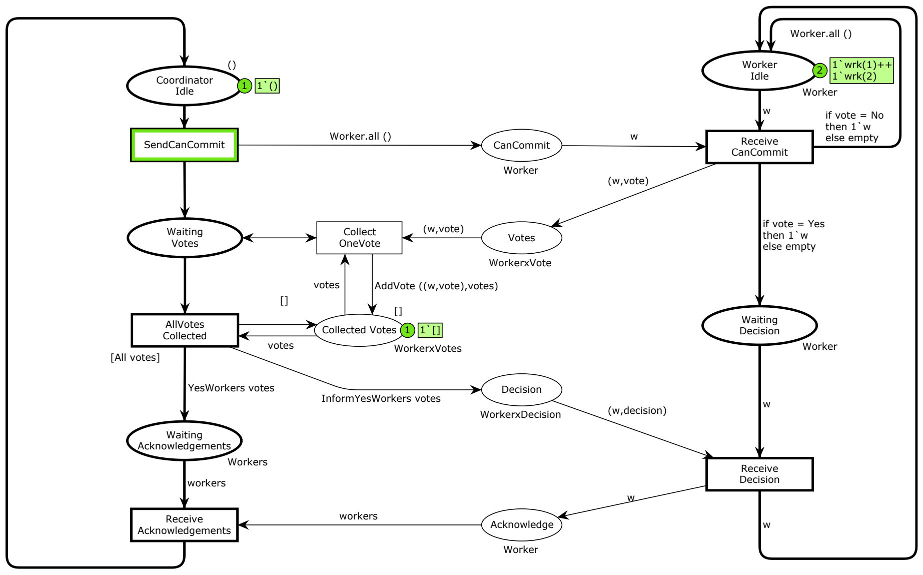


Figure 2.2: CPN model of the two-phase commit protocol in CPN Tools

A place is oval-shaped, a transition is a squared box, and arrows represent the arcs connecting them. The green highlight around the `SendCanCommit` transition (top left) means it is enabled and may occur. The green objects next to the places are the tokens present on that particular place in the current marking. The green circular number is the total amount of tokens on the place. The green square is the multi-set. In the multi-set, the number before the apostrophe ' is the number of appearances, and the value is after. The declarations defined in the model are shown in Listing 2.1 below, written in Standard ML and the CPN ML programming language defined in [48]. CPN ML is based on Standard ML and extends it with constructs for colour-set definitions and variable declarations. The keywords from CPN ML are highlighted in purple color.

```
// Types
colset Worker = index wrk with 1..W;
colset Workers = list Worker;
colset Vote = with Yes | No;
colset WorkerxVote = product Worker * Vote;
colset WorkerxVotes = list WorkerxVote;
colset Decision = with abort | commit;
colset WorkerxDecision = product Worker * Decision;

// Variables
var w: Worker;
var workers : Workers;
var vote : Vote;
var votes : WorkerxVotes;
var decision : Decision;

// Values
val W =2;

// Functions
fun AddVote ((w,vote),votes) = (w,vote) :: votes;
fun yesVotes votes =
  List filter
    (fn (w,vote) => vote = Yes)
  votes;
fun YesWorkers votes =
  List.map (fn (w,_) => w) (yesVotes votes)
fun allYes votes = (List.length (yesVotes votes) = W);
fun InformYesWorkers votes =
  let
    val yesworkers = YesWorkers votes
    val decision =
      (if (List.length yesworkers = W) then commit else abort)
  in
    List.map (fn w => (w,decision)) yesworkers
  end
fun All votes = (List.length votes = W)
```

Listing 2.1: Declarations defined in the two-phase commit protocol

Tokens are being removed and added according to the direction of the arrows. The text on the arcs represents the expressions evaluated when transitions occur. Empty arc expressions implicitly contain the *unit value* `()`. The unit value is the only value of the type *unit*, which is often used to represent *void* in functional programming languages. The `AllVotesCollected` transition also has a *guard* in square brackets next to it, which is an additional condition to be satisfied for the transition to be enabled.

2.4 Core concepts

Below are some core concepts that need to be explained more in-depth for the reader to understand the later chapters better.

2.4.1 Multi-sets

A multi-set can be formally defined using the definition in [50] as follow, a multi-set over a domain is a function from the domain into the set of natural numbers. A multi-set may contain multiple tokens with the same value, indicated by the number. Basic operations on multi-sets are addition `++`, subtraction `--` and comparison `<<=`. The detailed semantics will be shown later. Shown below is a representation of a multi-set from the example protocol. The number before the carets is the natural number indicating the quantity, and the token value is after. The operation is *addition* which means that similar values would have been joined and indicated with an increase in the number of appearances.

```
1`(Wrk 1, No) ++ 1`(Wrk 2, Yes)
```

2.4.2 Marking

A marking consists of all the multi-sets of tokens on all the places in the model. The first marking of a model is the *initial marking* and represents the starting state. The marking shown in Figure 2.2 is the initial marking of the two-phase commit protocol, where the tokens initially present on the places are defined by the expression in the upper right next to the place.

2.4.3 Colour-sets and Variables

A colour-set is the same as a type. In Listing 2.1, types are defined with the `colset` keyword from CPN ML. A variable is a declared identifier used in arc expression to bind to a token value, declared with the `var` keyword from CPN ML. As an example, the `w` present on the arc between `CanCommit` place and the `Receive CanCommit` transition in Figure 2.2. The variable is declared in the Listing 2.1 to be of type `Worker`, an *index type*, meaning a type with a limited number of values indicated by the `with` operator. A variable declaration is not a concept in the F# programming language, but the .NET platform has support for it. Later chapters will show how this is handled. The lack of variable declarations in F# is one reason why we kept the CPN ML constructs.

2.4.4 Bindings

Variables on arcs connected to a transition belong to the transition. We need to create a *binding* between the variables and tokens on connected input places. The binding is needed in order for tokens to be removed from an input place and added to an output place, which happens through the occurrence of the transition. For example, the `w` variable just mentioned belong to the `Receive CanCommit` transition and needs to be bound to some token on the `CanCommit` place. Each transition, arc and place combination makes up a *binding type*, there as many binding types as there are transitions. A binding type contains all the variables of a transition, meaning that some bindings may only *cover* a subset of the variables of the transition, resulting in a *partial binding*. Partial bindings need to be *merged* to create *complete bindings*. Partial bindings and merging are formally defined in [50].

2.4.5 Enabling and Occurrence

As briefly mentioned, an enabled transition may occur. A transition is enabled if sufficient tokens exist on the connected input places to create a complete binding, and any guard is satisfied. When a transition occurs, the tokens from the enabled binding are removed from the input places, and tokens are added to the output place as defined by evaluating the corresponding arc expression.

2.4.6 Executing a CPN model

For steps of a CPN model to be executed, the mentioned enabled bindings for the transitions need to be created. The challenges regarding this problem are presented and discussed in [50], and a token-based inference mechanism based on *pattern matching* is proposed, shown in Figure 2.3 below.

```
 1: {$PBB(t) = \{E_j \mid 1 \leq j \leq l\}$ pattern binding basis for $t$}
 2: $C \leftarrow \emptyset$
 3: for all $E(p,t) \in PBB(t)$ do
 4:     $C' \leftarrow \emptyset$
 5:     for all $c \in M(p)$ do
 6:         $b' \leftarrow \textsc{PartialBind}(E(p,t), c)$
 7:         if $b' \neq \perp$ then
 8:             $C' \leftarrow C' \cup \{ b' \}$
 9:         end if
10:     end for
11:     $C \leftarrow \textsc{Merge}(C, C')$
12: end for
13:
14: $C \leftarrow \{ b \in C \mid G(t)\langle b \rangle \}$
15: $C \leftarrow \{ b \in C \mid \forall p \in P_{\mathsf{In}}(t) : E(p,t)\langle b \rangle \leq M(p) \}$
16:
17: return $C$
```

Figure 2.3: Pure arc expression based inference algorithm [50]

The algorithm first finds a *pattern binding basis* for a transition on line 1, the minimum set of arcs with input arc expressions covering all the variables of a transition. The input arc expressions must be patterns. Line 3 goes through the arcs `E` in the pattern binding basis, `p` represents the connected place and `t` the connected transition. Line 6 partially binds a field `b'` of the binding type, a variable in the transition, with a token value `c` present on the place in the marking `M`. Line 7 checks if something got bound and, if so populates the partial binding `C'` with the bound field. Line 11 merges the partial bindings to create complete bindings `C`. Line 14 and line 15 check the guard and verify that there exist sufficient tokens on the input places, respectively.

Chapter 3

Compiler and Runtime Environment

This chapter presents a high-level overview of the compiler and the runtime environment, referred to as the *system*. It introduces the main features of the F# programming language and the .NET platform used in the CPN simulation system. It also shows how F# can be used as an inscription language for CPN models.

3.1 Overview

Figure 3.1 below presents a high-level view of the different components in the system developed in this thesis. The input is a .cpn file, and the output is a F# .NET library. The stick figures represent user interaction, and the dotted lines represent that the user needs to have the files. Square boxes are different F# .NET projects in .NET solution. Boxes right next to each other symbolize dependency. The `Loader` component used data structures from the `Syntax` component, and the `Parser` component used data structures from the `Semantic` component. The data structures in the `Syntax` and `Semantic component` are mainly used to structure the information gathered about the CPN model through the different parsing and analysis parts of the compiler found in the `Loader` and the `Parser` components. The information passed between the different components is written as text on the arrows. For instance, between the `Loader` component and the `Parser` component, a value of type `CpnModel` is passed, which will be presented in the next chapter. The arrow going out from the `Output` component, reading `Code`, is supposed to represent the output of F# code into .fs files.

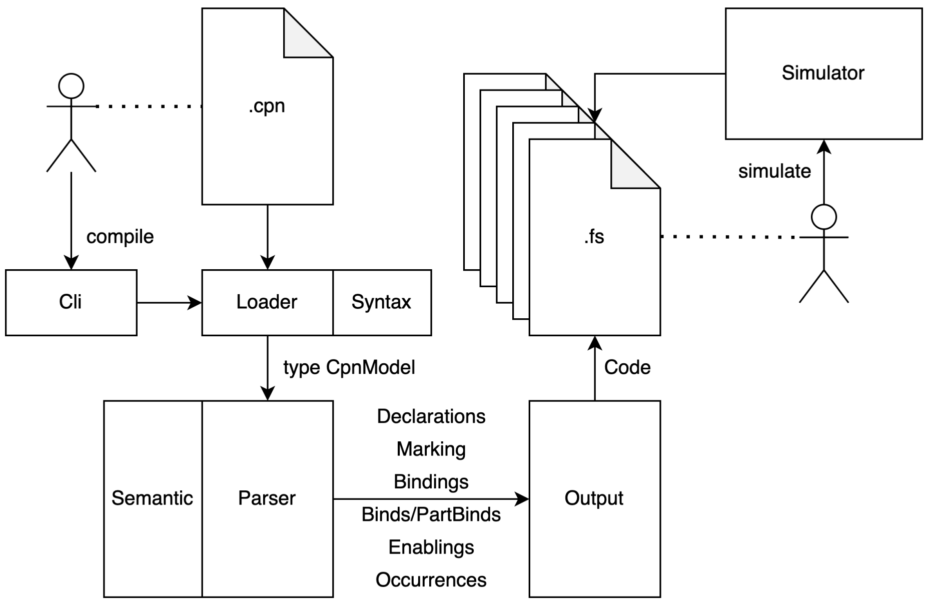


Figure 3.1: A high-level view of the compiler and runtime environment

The system works, as shown below, by having a user call the command line interface component `Cli` and pass a reference to a .cpn file containing a CPN model with functions, inscriptions, initial markings, and arc expressions written in F#. For our example, the command will be as follows.

```
dotnet run --project src/compiler/Cli examples/TPC/model/tpc.cpn
```

This file reference is passed to the `Loader` component, which opens the file and loads the information. The colour-set definitions are parsed, and equivalent type definitions in F# are generated. In Listing 3.1 below are some of the definitions with the CPN ML version in a comment above to compare. One reason why we chose this approach was to be able to use the definitions from already existing models created in CPN Tools.

```
// colset Worker = index wrk with 1..W;
type Worker =
  | Wrk of int
  static member all() = [ for i in 1..W -> Wrk(i) ]

// colset Vote = with Yes | No;
type Vote = |Yes |No
```

Listing 3.1: Examples of CPN ML definitions translated into F#

The variable declarations and their type information are also collected. The functions and values definitions, written in F#, are appended to the type definitions and placed into a data structure called `CpnModel`, defined in the `Syntax`, alongside the variable declarations. The `Loader` also collects the necessary information about the places, arcs and transitions. All this information is kept in strings, as it is what comes from the .cpn file. Listing 3.2 shows the information about one place, one arrow and one transition. This information is kept and outputted to a separate file for development and debugging purposes and is not directly used by the CPN simulation system. The `id` fields are used to connect places, arcs and transitions.

```
// Place
{ id = "ID1591819253"
  name = "CoordinatorIdle"
  colset = "UNIT"
  initialMarking = Some "1`()" }

// Arcs
{ place = "ID1591819253"
  transition = "ID1591819228"
  expr = "1`()"
  direction = PT }

// Transition
{ id = "ID1591819228"
  name = "SendCanCommit"
  guard = None; }
```

Listing 3.2: Information about a place, an arc and a transition

The `Parser` does several types and error checks to ensure that the generated F# code is valid. The checks are listed below.

1. Type errors in inscriptions and declarations.
2. Type mismatch between place and initial marking.
3. Type mismatch between arc expressions and place.
4. Type error in guard, not a Boolean value.
5. F# syntax errors in inscriptions and declarations.
6. Use of undefined types on a place.
7. Verify that variables on output arc expressions are a subset of variables on input arc expressions or *simply typed*.

For instance, if the arc expression `(w,vote)` on the arc between the transition `ReceiveCanCommit` and the place `Vote` is changed to only `w`, the compiler would respond with an error message shown below and abort the compilation.

```
w is not of type WorkerxVote
CPN model contains errors.
```

The `Parser` component uses the defined data structures in the `Semantic` component to structure and format the different artefacts generated. It passes them to the `Output` component once created. Depicted on the arrow between the `Parser` and the `Output` components are the names of the different constructs generated. Some of these are briefly explained below, and some will be explained in more detail in later chapters.

`Declarations` are type and function definitions similar to what is shown in Figure 2.1 in the last chapter.

The `Marking` is the state of the model consisting of multi-sets of tokens. Each place is represented with a field in the marking with the same name as the place. Shown below in Listing 3.3 is the marking for the two-phase commit protocol. The `MultiSet` type is shown later in this chapter, and how the marking is generated is shown in the next chapter.

```
type Marking =
  { CanCommit: Worker MultiSet
    Votes: WorkerxVote MultiSet
    Acknowledge: Worker MultiSet
    Decision: WorkerxDecision MultiSet
    WaitingVotes: UNIT MultiSet
    WaitingAcknowledgements: Workers MultiSet
    CoordinatorIdle: UNIT MultiSet
    WorkerIdle: Worker MultiSet
    WaitingDecision: Worker MultiSet
    CollectedVotes: WorkerxVotes MultiSet }
```

Listing 3.3: The marking type of the two-phase commit protocol

The rest of the constructs are discussed later but briefly explained, `Bindings` are types of bindings that bind token values to free variables. `Binds/PartBinds` are functions used to partially bind free variables to a binding and find all the possible bindings of variables. `Enablings` are functions that filter through all bindings and return the enabled bindings in the current marking. `Occurrences` is a function that handles the occurrence of a chosen binding and returns a new marking. All these constructs are necessary to simulate the execution of a model. The `Output` component is responsible for outputting the generated code in F# files and creating a F# .NET library for the `Simulator` component.

3.2 F# and the .NET Platform

Some features and libraries from the F# programming language and the .NET platform play a significant role in the CPN simulation system. These will be presented and discussed below.

3.2.1 Discriminated Unions, Tuples and Records

Discriminated unions [28] and Tuples [29], often called sum types and product types, respectively, are defined data types in F#. They are called algebraic data types in the world of functional programming [30]. F# also has records [31]. We use these data types to carry and structure the information about the CPN model through the different components of the system. In Listing 3.4 are some types from the `Semantic` component representing a variable and an arc expression. A star `*` is the constructor of a product type.

```
type Var =
    { id: string
      colset: string }

type ArcExpr =
    | Expr of string * string * bool
    | Pattern of string * string * bool
```

Listing 3.4: Examples of Discriminated Unions, Tuples and Records in F#

3.2.2 Option type

The Option type [32] is the way to handle *null* values in F#. It is a sum type with two values, `None` represens no value and `Some x` carries some value, symbolized by the `x`.

3.2.3 Pattern matching

The system relies heavily on the pattern matching capability of F#. Pattern matching is a prevalent feature in functional programming languages, and F# has strong support. The pattern matching is essential in the system when it comes to matching the pattern on an arc to tokens residing on the connected places. It is also a key feature when working with algebraic data types as the data inside the type is extracted by matching its constructor, called *destructuring*. For example, arc expressions that are patterns can be used in function definitions to destructure the argument passed, as shown below in Listing 3.5. It is not a function that takes two parameters, but one tuple of type `WorkerVote` that is *destructured*. The Option type is also used in the code.

```
type WorkerxVote = Worker * Vote

fun (w,vote) ->
    { _u = Some ()
      vote = Some vote
      votes = None
      w = Some w }
```

Listing 3.5: Example of the use of pattern matching

3.2.4 Code quotations and F# Compiler Service

Code quotations [33] is a language feature of F# supporting metaprogramming. A code quotation can be explained as an unevaluated F# expression. It is delimited so that it is compiled into an object that represents an F# expression. Below is an example of a code quotation. When this code is evaluated, the `expr` is bound to the value `<@ 2 + 2 @>` and is of type `Expr<int>`, meaning that when the expression is evaluated the result will be of type `int`.

```
let expr : Expr<int> = <@ 2 + 2 @>
```

The F# Compiler Service [34] is a collection of the compiler tools that are part of the official F# compiler, made accessible to be used programmatically from F# code. This includes the tokenizer, type-checker, parser of the compiler and the possibility to host an interactive evaluation session. The last one is essential in regards to our system. The .NET platform comes with an interactive F# read-eval-print loop (REPL), which can be created in code as shown below in Listing 3.6.

```
let fsiSession =
  FsiEvaluationSession.Create(
    FsiEvaluationSession.GetDefaultConfiguration(),
      [| "fsi.exe"
        "--noninteractive"
        "--nologo"
        "--gui-" |],
      new StringReader(""),
      new StringWriter(StringBuilder()),
      new StringWriter(StringBuilder())
    )
```

Listing 3.6: How to create a hosted interactive evaluation session

Hosting this REPL in code is significant because it can evaluate F# expressions or interactions (operations with side effects that are not valid F# expressions, for instance, declarations or printing). The hosted interactive evaluation session enables the system to evaluate strings as expressions. We can check if an expression is valid. For instance, check if an arc expression is of the correct type and/or whether it is a *pattern*.

Combining the quote quotations and the F# Compiler Services enables us to translate, for instance, an arc expression from a string into a value of type `Expr`, as shown below in in Listing 3.7. *Type reflection* [35] is used to convert the result from an *object* the `Expr` type on line 7.

```
1  let evalExpr<'a> expr =
2    let result, _ =
3      fsiSession.EvalExpressionNonThrowing expr
4
5    match result with
6    | Choice1Of2 (Some value) ->
7      value.ReflectionValue |> unbox<'a>
8    | Choice1Of2 None ->
9      failwith "null or no result"
10   | Choice2Of2 (exn: exn) ->
11     failwith (sprintf "exception %s" exn.Message)
12
13 let expr = evalExpr<Expr> $"<@ {arcExpr} @>"
```

Listing 3.7: A use of the hosted interactive evaluation session

The `result` on line 2 can be one of three different types defined by the F# Compiler Service creators. `Choice1Of2 (Some value)` means that the expression was valid and correctly evaluated and contains a return value. `Choice1Of2 None` means that the evaluation was valid and correctly evaluated, but the expression did not have a return value. `Choice2Of2` means that the expression was not valid, and some error occurred. It is the first type we need to focus on, as the expression is supposed to return a value of type `Expr` containing the arc expression.

The value contained inside the `Option` returned from the function `fsiSession.EvalExpressionNonThrowing` is of type *object*, meaning some simple *unboxing* [36] is necessary to get it into the `Expr` type. The `Expr` type is passed as a generic argument indicated to the right of the function name `<'a>`. The possibility this enables is that `expr`, created in line 13, can be pattern matched on the structure of the expression, similar to the traversal of an abstract syntax tree. We use this to extract the free variables on the arc expressions, as shown in the next chapter. The `Expr` type also has some useful methods to get type information.

3.3 Multi-set implementation in F#

The multi-set is a significant construct in the system used to manage the tokens on places. A slight difference between the multi-set implementation used in our project versus the one used in CPN Tools is the use of the caret ^ symbol to create multi-sets. This is because the apostrophe `'` is reserved for generic type definitions in F#, as we just showed an example of in Listing 3.7 and the back-quote `` ` `` is also reserved [37]. Shown below in Listing 3.8 is the implementation of the multi-set using the F# Map [38]. The function `listToMs` is a helper function used in arc expressions to convert a list to a multi-set, shown in the next section. The last function `msMap` is also a helper function used in the process of finding enabled bindings, as will be shown in later chapters.

```fsharp
type MultiSet<'a when 'a: comparison> =
  { multiset: Map<'a, int> }

let empty = { multiset = Map.empty }

let isEmpty ms = Map.isEmpty ms.multiset

let (^) x c = { multiset = (Map.add c x Map.empty) }

let (++) ams bms =
  { multiset =
    (Map.fold
      (fun ms c x ->
        match Map.tryFind c ms with
        | Some x' -> Map.add c (x + x') ms
        | None -> Map.add c x ms)
      ams.multiset
      bms.multiset) }

let (<<=) ams bms =
  Map.forall
    (fun c x ->
      match Map.tryFind c bms.multiset with
      | Some x' -> x <= x'
      | None -> false)
    ams.multiset

let (--) ams bms =
  if bms <<= ams then
    { multiset =
      (Map.fold
        (fun ms c x ->
          match Map.tryFind c ms with
          | Some x' ->
            let n = x' - x in
            (match n with
            | 0 -> Map.remove c ms
            | _ -> Map.add c n ms)
            | None -> Map.add c x ms)
        ams.multiset
        bms.multiset) }
  else
    invalidOp
    "Cannot perform the operation with the given arguments"

let listToMs list =
  list
  |> List.fold (fun acc i -> acc ++ (1 ^ i)) empty

let msMap (parbind: 'a -> 'b) tokens =
  Map.fold (fun pb c _ -> parbind c :: pb) [] tokens.multiset
```

Listing 3.8: The multi-set implementation in F#

3.4 F# as inscription language

As mentioned earlier, F# shares many similarities with Standard ML, the inscription language used in models created with CPN Tools. F#, as a functional programming language, supports many expression constructs, meaning constructs that return a value. Expression-orientated programming works perfectly in line with arc expressions. The changes needed to translate Standard ML code into valid F# are small. It is sometimes just a matter of small syntactical differences. Shown below in Figure 3.2 is the two-phase commit protocol using F# as the inscription language. When comparing the two models, it is easy to see that the change is quite small. One change is using the `listToMs` function, shown in the previous section, in arc expressions. CPN Tools has a layer in between the GUI and the simulator that automatically converts lists to multi-sets. We have no such automatic conversion in our system. The creation of multi-sets is also slightly changed, using the caret ^ symbol, as discussed in the previous section. Empty values needs to be typed, for instance, the arc expression `([] : WorkerxVotes)` from the `AllVotes Collected` transition to the `Collected Votes` place. Furthermore, the functions use camelCase naming, as is the convention in F# [39].

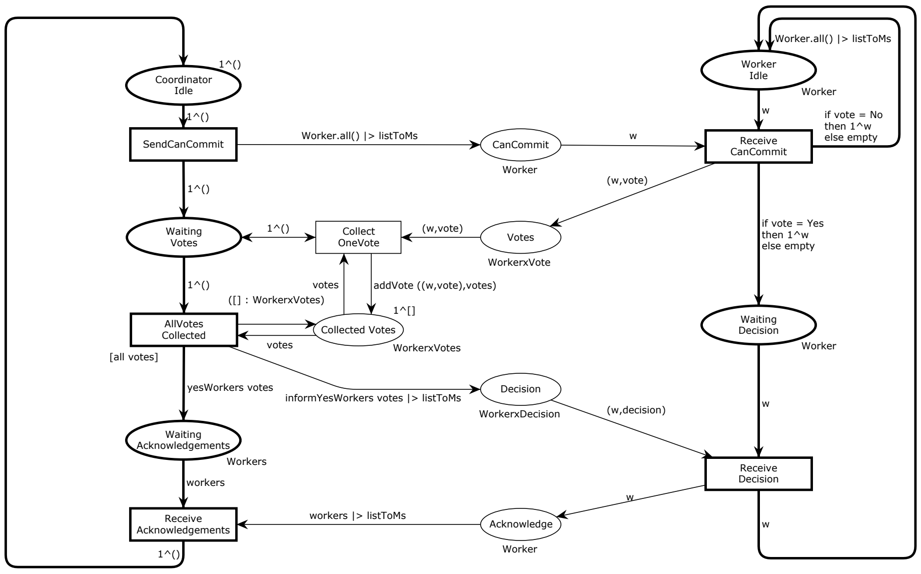


Figure 3.2: The two-phase commit protocol using F# as inscription language

Shown below in Listing 3.9 are the values and function definitions for the two-phase commit protocol translated into F#, the type and variable definitions are the same as in 2.1, because the translation is handled by the `Loader` component, as mentioned earlier.

```
// Values
let W = 2

// Functions
let addVote ((w, vote), votes) = (w, vote) :: votes
let yesVotes votes =
    List.filter (fun (_, vote) -> vote = Yes) votes
let yesWorkers votes =
    List.map (fun (w, _) -> w) (yesVotes votes)
let allYes votes = (List.length (yesVotes votes) = W)
let informYesWorkers votes =
    let yesworkers = yesWorkers votes
    let decision =
        (if (List.length yesworkers = W) then
             Commit
         else
             Abort) in
    List.map (fun w -> (w, decision)) yesworkers
let all votes = (List.length votes = W)
```

Listing 3.9: Values and function declarations in F#

For comparison, shown again in Listing 3.10 are the values and function definitions in the two-phase commit protocol in Standard ML.

```
// Values
val W =2;

// Functions
fun AddVote ((w,vote),votes) = (w,vote) :: votes;
fun yesVotes votes =
    List.filter (fn (w,vote) => vote = Yes) votes;
fun YesWorkers votes =
    List.map (fn (w,_) => w) (yesVotes votes)
fun allYes votes = (List.length (yesVotes votes) = W);
fun InformYesWorkers votes =
    let
     val yesworkers = YesWorkers votes
     val decision =
        (if (List.length yesworkers = W)
            then commit
        else abort)
    in
     List.map (fn w => (w,decision)) yesworkers
    end
fun All votes = (List.length votes = W)
```

Listing 3.10: Values and function declarations in Standard ML

We conclude that using F# as the inscription language in Coloured Petri net models is relatively straightforward.

Chapter 4

Parsing and Code Generation

This chapter goes into more detail about the `Loader` and the `Parser` components introduced in the previous chapter. It shows how the parsing and code generation are achieved by using the features and libraries from F# and the .NET platform, also introduced in the previous chapter. The different constructs and artefacts to be generated that is necessary to simulate a CPN model were briefly presented in the previous chapter and will be presented and discussed in more depth here.

4.1 Loader Component

The `Loader` component is loading the .cpn file, containing the two-phase commit protocol model, passed to it using a library called FSharp.Data [40]. The library includes F# Type Providers [41], which can dynamically create F# types at compile time from external structured data to be used in a program. The types are inferred based on the values inside the .cpn file provided. The `Loader` component finds the declarations, including types (colour-sets), variables, values and functions, places, arcs and transitions, as shown below.

4.1.1 Declarations

The types (colour-sets) from the two-phase commit protocol are represented in the .cpn file as shown below in Listing 4.1. The listing only shows some of the types in the model. In the case of the `Enum` type (enumeration), starting on line 20, the `id` element on line 21 is the name of the type (Vote) and the `id` elements on line 23 and 24 are the type values.

```
 1 <workspaceElements>
 2   <cpnet>
 3     <globbox>
 4       <block>
 5         <id>Workers</id>
 6         <color id="ID1429585792">
 7           <id>Worker</id>
 8           <index>
 9             <ml>1</ml>
10             <ml>W</ml>
11             <id>wrk</id>
12           </index>
13         </color>
14         <color id="ID1429641807">
15           <id>Workers</id>
16           <list>
17             <id>Worker</id>
18           </list>
19         </color>
20         <color id="ID1429586016">
21           <id>Vote</id>
22           <enum>
23             <id>Yes</id>
24             <id>No</id>
25           </enum>
26         </color>
27         <color id="ID1429618174">
28           <id>WorkerxDecision</id>
29           <product>
30             <id>Worker</id>
31             <id>Decision</id>
32           </product>
33         </color>
34         ...
35       </block>
36       ...
37     </globbox>
38   </page>
39 </workspaceElements>
```

Listing 4.1: Example of type definitions in a .cpn file

With the help of the FSharpData library, it is possible to access children of an XML element with the dot operator as, for instance, shown on line 2 in Listing 4.2 below. All the `color` elements from the .cpn file in Listing 4.1 above are collected into a list accessible from the `Colors` field on the `Blocks` field (lines 4-35), as shown in line 3 below. These colour-sets are then translated into F# type definitions. For instance, `Enum` on line 16 is translated into a F# sum type definition. Keeping the CPN ML type definitions and translating, rather than beforehand creating equivalent F# types, enables the reuse of existing type (colour-set) definitions in existing models created in CPN Tools.

```
1  let declarations =
2   workspaceElements.Cpnet.Globbox.Blocks
3    |> Array.collect (fun b -> b.Colors)
4    |> Array.map (fun c ->
5     let colset =
6      match c.Index with
7       | Some t ->
8          let id = t.Id
9          $"\n    | {id} of int\n
10          static member all() =
11           [ for i in {t.Mls[0]}..{t.Mls[1]} -> {id}(i) ]"
12       | None ->
13        match c.List with
14        | Some t -> $"{t.Id} list"
15        | None ->
16         match c.Enum with
17          | Some t ->
18                  t.Ids
19                  |> Array.map (fun i -> $"\n    |{i}")
20                  |> String.concat ""
21          match c.Product with
22          | Some t -> $"{t.Ids[0]} * {t.Ids[1]}"
23          | None -> failwith $"Unknown type of: {c.Id}"
24          ...
25     $"\n  type {c.Id} = {colset}")
```

Listing 4.2: Finding types in the model

The translation of the `index` type is shown on lines 6-11. The `index` type gets translated into a sum type with one named constructor and includes a member function `all()` used to get all its values. The translation of the `list` type is shown on lines 13-15. The `list` type simply gets translated into a F# list. The translation of the `enum` type is shown on lines 16-20. The `enum` type gets translated into a sum type with one named constructor that takes no arguments for every value. The translation of the `product` type is shown on lines 21-23. The `product` type gets translated into a F# Tuple. Line 25 creates a string containing a valid F# type definitions, completing the map function starting at line 4. The result is an array of strings, ready to be both outputted into an F# file and evaluated in a hosted interactive evaluation session.

Shown below in Listing 4.3 are the resulting F# types. The CPN ML version is placed in a comment above. Some examples were shown in the previous chapter in Listing 3.1, but are shown again here for completeness.

```
// colset Worker = index wrk with 1..W;
type Worker =
  | Wrk of int
  static member all() = [ for i in 1..W -> Wrk(i) ]

// colset Workers = list Worker;
type Workers = Worker list

// colset Vote = with Yes | No;
type Vote = |Yes |No

//colset WorkerxVote = product Worker * Vote;
type WorkerxVote = Worker * Vote

// colset WorkerxVotes = list WorkerxVote;
type WorkerxVotes = WorkerxVote list

// colset Decision = with abort | commit;
type Decision = |Abort |Commit

// colset WorkerxDecision = product Worker * Decision;
type WorkerxDecision = Worker * Decision
```

Listing 4.3: Type definitions in the two-phase commit protocol translated into F#

Listing 4.4 shows a variable declaration in the two-phase commit protocol in the .cpn file. The `id` element on line 7 is the name of the type (Worker), and the `id` element on line 9 is the identifier name of the variable (w).

```
1  <workspaceElements>
2    <cpnet>
3      <globbox>
4        <block>
5          <var id="ID1429585855">
6            <type>
7              <id>Worker</id>
8            </type>
9            <id>w</id>
10         </var>
11         ...
12       </block>
13       ...
14     </globbox>
15   </page>
16 </workspaceElements>
```

Listing 4.4: Example variable declarations in a .cpn file

Getting the global list of variables in the CPN model is relatively straightforward, as shown below in Listing 4.5.

```
1  let vars =
2    workspaceElements.Cpnet.Globbox.Blocks
3    |> Array.collect (fun b -> b.Vars)
4    |> Array.map
5      (fun v -> v.Ids |> Array.map (fun id -> id, v.Type.Id))
6    |> Array.concat
7    |> Array.append topLevelVars
8    |> Map.ofArray
```

Listing 4.5: Finding variables in the model

One thing to pay attention to is that CPN Tools allows declarations to be placed inside both `block` element and inside the *top-level* `globbox` element. Hence both places need to be searched and are so in a special order to not refer to something not yet declared in other declarations. Line 7, in the listing above, shows how the variables considered are appended to the existing list of top-level variables `topLevelVars`.

The problem was more intricate regarding the function and values definition and was solved by choosing an evaluation order, we chose to evaluate the declarations as follow; *top-level* functions and value definitions, *top-level* types (colour-sets) definitions, types (colour-sets) definitions and functions and value definitions. Shown below in Listing 4.6 is how the top-level declarations and declarations inside `block` elements are found. On line 18, `declarations` is a list of all the previous collected declarations, it is appended to the other declarations.

```
1  let topLevelDeclarations =
2    workspaceElements.Cpnet.Globbox.Mls
3    |> Array.map (fun ml ->
4      "\n  " + ml.ToString().Trim()
5      |> WebUtility.HtmlDecode)
6
7  ...
8  // Finding top level type definitions
9  // Finding type definitions
10 ...
11
12 let declarations =
13   workspaceElements.Cpnet.Globbox.Blocks
14   |> Array.collect (fun b -> b.Mls)
15   |> Array.map (fun ml ->
16     "\n  " + ml.ToString().Trim()
17     |> WebUtility.HtmlDecode)
18   |> Array.append declarations
19   |> Array.toList
```

Listing 4.6: Finding function and value definitions in the model

4.1.2 Places, Arcs and Transitions

The information that the .cpn file contains about the places, the arcs and the transitions are structured in the types shown below, with the type `CpnModel` as the top-level type, shown below in Listing 4.7.

```
type CpnModel =
    { name: string
      vars: Map<string, string>
      declarations: string list
      cpnModule: CpnModule }
```

Listing 4.7: The CpnModel type definition

The `name` field is taken from the name of the provided XML file. The `vars` field represents the declared variables in the model, represented as a F# Map where the identifier is the key, and the type is the value. The `declarations` field is the types (colour-sets), values and functions. The declarations are kept as a list of strings because it is valid F# code that will be outputted to a F# file and put inside the hosted interactive evaluation session by the `Parser` component. The `cpnModule` field contains the places, transitions and arcs shown below in Listing 4.8. For now, our system only works with models contained inside a single module.

```
type CpnModule =
    { name: string
      places: Place list
      transitions: Transition list
      arcs: Arc list }
```

Listing 4.8: The CpnModule type definition

The `name` field is the module's name in CPN Tools. Every model needs to reside in a module (page). The `Place` type is shown below in Listing 4.9.

```
type Place =
    { id: string
      name: string
      colset: string
      initialMarking: string option }
```

Listing 4.9: The Place type definition

The `name` field is the name of the place. The `colset` field describes the type of tokens the place may hold. The `initialMarking` field describes the tokens on this place in the initial marking. It is an option type because some places may not have an initial marking. The `Transition` type is shown below in Listing 4.10.

```
type Transition =
    { id: string
      name: string
      guard: string option }
```

Listing 4.10: The Transition type definition

The `name` field is the name of the transition. The `guard` field is a boolean expression that needs to be satisfied for this transition to be enabled. It is optional as indicated by the `option`. The `Arc` type and the `Direction` are shown below in Listing 4.11.

```
type Direction = | PT | TP | DB
type Arc =
    { place: string
      transition: string
      expr: string
      direction: Direction }
```

Listing 4.11: The Arc and Direction type definitions

The `place` field is the identifier of the place the arc connects to. The `transition` field is the identifier of the transition the arc connects to. The `expr` field is the expression on the arc. The `direction` field is the direction of the arc, defined as its own discriminated union type. `PT` is from a place to a transition, `TP` is from a transition to a place, and `DB` represents a double arc in both directions.

A place in the .cpn file is represented as shown below in Listing 4.12. The `text` element on line 5 is the place's name, the `text` element on line 7 is the type (colour-set) of the place and the `text` element on line 10 is the initial marking on the place. The F# type provider also notices that the `text` element on line 10, representing the initial marking, does not always contain something and infers it to be an *Option* type.

```
 1  <workspaceElements>
 2    <cpnet>
 3      <page>
 4        <place id="ID1591819610">
 5          <text>Worker Idle</text>
 6          <type id="ID1591819611">
 7            <text>Worker</text>
 8          </type>
 9          <initmark id="ID1591819612">
10            <text>Worker.all() |> listToMs</text>
11          </initmark>
12        </place>
13        ...
14    </page>
15  </workspaceElements>
```

Listing 4.12: A place in a .cpn file

Similarly, as above, access to the element is possible with the dot operator as shown below in Listing 4.13. Some trimming of the strings needs to be done because we will be using the values to name generated types and functions.

```fsharp
let places =
    workspaceElements.Cpnet.Page.Places
    |> Array.map (fun p ->
        { id = p.Id
          name = Regex.Replace(p.Text, @"\s+", "")
          colset = p.Type.Text.ToString().Trim()
          initialMarking =
           Option.map
             (fun im -> im.ToString().Trim())
            p.Initmark.Text })
    |> Array.toList
```

Listing 4.13: Finding places in the model

A transition in the .cpn file is represented as shown below in Listing 4.14. The `text` element on line 5 is the name of the transition. The F# type provider also notices that the `text` element on line 8, representing the guard, is not always containing something and infers it to be an *Option* type.

```xml
1  <workspaceElements>
2    <cpnet>
3      <page>
4        <trans id="ID1591819598">
5          <text>Receive CanCommit</text>
6        </trans>
7        <cond id="ID1591819599">
8          <text/>
9        </cond>
10       ...
11     </page>
12 </workspaceElements>
```

Listing 4.14: A transition in a .cpn file

Similarly, as above, access to the element is possible with the dot operator as shown below in Listing 4.15. Some trimming of the strings also needs to be done here because we will be using the values to name generated types and functions.

```fsharp
let transitions =
  workspaceElements.Cpnet.Page.Trans
  |> Array.map (fun t ->
        { id = t.Id
          name = Regex.Replace(t.Text, @"\s+", "")
          guard =
           Option.map
            (fun im -> im.ToString().Trim()) t.Cond.Text })
  |> Array.toList
```

Listing 4.15: Finding transition in the model

An arc in the .cpn file is represented as shown below in Listing 4.16. The `idref` attribute on the `transend` element on line 6 is the id of the transition the arc connects to. The `idref` attribute on the `placeend` element on line 7 is the id of the place the arc connects to, and the `text` element on line 9 is the arc expression. Note how the `idref` attributes correspond to the `id` attributes on the place and transition shown above.

```
 1  <workspaceElements>
 2    <cpnet>
 3      <page>
 4        <arc id="ID1591819632"
 5             orientation="PtoT">
 6          <transend idref="ID1591819598"/>
 7          <placeend idref="ID1591819610"/>
 8          <annot id="ID1591819633">
 9            <text>w</text>
10          </annot>
11        </arc>
12        ...
13    </page>
14  </workspaceElements>
```

Listing 4.16: An arc in a .cpn file

Similarly, as above, access to the element is possible with the dot operator as shown below in Listing 4.17. The `Orientation` field of the arc is found by pattern matching. It represents the direction of the arc. A `Direction` type is used and will be shown shortly.

```
let arcs =
  workspaceElements.Cpnet.Page.Arcs
  |> Array.map (fun a ->
    let direction =
      match a.Orientation with
      | "PtoT" -> PT
      | "TtoP" -> TP
      | "BOTHDIR" -> DB
      | _ -> failwith "Arc orientation not supported"

    let expr =
      a.Annot.Text.ToString().Trim().Replace('\n', ' ')

    { place = a.Placeend.Idref
      transition = a.Transend.Idref
      expr = expr
      direction = direction })
  |> Array.toList
```

Listing 4.17: Finding arcs in the model

The listings above show how the model information is extracted from the provided .cpn file into string values. Most of them are already inferred as strings by the F# type provider, be further parsed and evaluated by the `Parser` component.

4.2 Parser Component

The `Parser` component is parsing the information it obtains from the `Loader` component shown in the previous section. As discussed in the previous chapter, it mainly uses the hosted interactive evaluation session and code quotations to analyse the information. In order for the hosted interactive evaluator session to properly evaluate the expression we pass, we first need to put the multi-set implementation and all the declarations of the model in the environment. We do this by simply having it evaluate the code, as shown below in Listing 4.18. We also check and verify that the declarations are valid at this stage. The multi-set implementations is put in an .fsx file, which is a script file [42], in the same directory as the `Parser` component.

```
1   let checkIfValid msg code =
2     let result, _ = fsiSession.EvalInteractionNonThrowing code
3
4     match result with
5     | Choice1Of2 _ -> ()
6     | Choice2Of2 _ -> failwith msg
7
8   ...
9
10  fsiSession.EvalScript (__SOURCE_DIRECTORY__ + "/MultiSet.fsx")
11  fsiSession.EvalInteraction "open MultiSet"
12
13  cpnModel.declarations
14    |> List.iter
15      (fun decl ->
16        checkIfValid (sprintf "Declaration is not valid: %s" decl) decl)
```

Listing 4.18: Putting multi-set implementation and declarations in environment

The `checkIfValid` function shown above is a general function used to catch errors in the model and put declarations in the environment, with its first parameter as the thrown error message if the second argument is not valid. In this case, an error is thrown if the `p.colset` is not a defined type in the session. It is also used to put variables in the environment in order for the arc expressions to properly be evaluated. If this is not done, the hosted interactive evaluation session will throw an error similar to the use of an undefined value or constructor. The `result` value on line 2 in the listing above will be of type `Choice2Of2`. The way this problem is solved is shown below.

```
let variableName = Unchecked.defaultof<typeName>
```

Here we make use of a module in FSharp.Core called Unchecked [43] which includes a function to generate a default value for any type. Shown below in Listing 4.19 is how this is used combined with the `checkIfValid` function to populate the environment of the hosted interactive evaluation session with the variables defined in the two-phase commit protocol model.

```
cpnModel.vars
  |> Map.iter (fun id colset ->
    checkIfValid
      (sprintf "Variable declaration is not valid:\n  %s : %s" id
          colset)
      $"let {id} = Unchecked.defaultof<{colset}>")
```

Listing 4.19: Putting variable declarations in environment

4.2.1 Arc expressions

We need to analyse the arc expressions to determine if they are *patterns* because those will be used to generate essential constructs later. In F#, only valid patterns may be placed inside a match expression as shown below.

```
match expr with | pattern -> ()
```

We can use this approach to find which arc expressions are patterns because if we try to replace the `pattern` above with the arc expression, we will know if it is a pattern based on whether it is valid F# code or not. Shown below in Listing 4.20 is how this is used.

```
1   let checkTypeAndIfPattern arcExpr colset =
2     let isMultiset = checkTypeAndIfMultiset arcExpr colset
3
4     let result, _ =
5       fsiSession.EvalInteractionNonThrowing
6       $"let fn expr = match expr with | {arcExpr} -> ()"
7
8       match result with
9       | Choice1Of2 _ -> Pattern(arcExpr, colset, isMultiset)
10      | Choice2Of2 _ -> Expr(arcExpr, colset, isMultiset)
```

Listing 4.20: Determining if arc expression is a pattern

If this interaction is valid, `result` on line 4 is of type `Choice1Of2` and matches that branch, and we have determined that it is indeed a pattern. Otherwise, it is a regular F# expression because using the arc expression as a branch in the F# match expression was not valid. The `checkTypeAndIfMultiset` function on line 2 verifies that the arc expression is correctly typed if it is of the same type as the place it is connected to. At the same time, it checks whether or not the expression is a multi-set because arc expressions can be multi-set expressions. The function is shown below in Listing 4.21.

```
let checkTypeAndIfMultiset expr t =
  let result, _ =
    fsiSession.EvalInteractionNonThrowing
      $"fun () -> {expr} : MultiSet<{t}>"

  match result with
  | Choice1Of2 _ -> true
  | Choice2Of2 _ ->
    let result, _ =
      fsiSession.EvalInteractionNonThrowing
        $"fun () -> {expr} : {t}"

    match result with
    | Choice1Of2 _ -> false
    | Choice2Of2 _ ->
      sprintf "%s is not of type %s" expr t |> failwith
```

Listing 4.21: Checking the type of an arc expression and if it is a multi-set

4.2.2 Marking

We generate the marking type of the model as a record data structure, responsible for the representation of the state. A simple run through the defined places as shown below in Listing 4.22 is sufficient, with a check if the defined place has a valid and defined type as colour-set.

```
let marking =
    cpnModel.cpnModule.places
    |> List.map (fun p ->
        checkIfValid
            (sprintf "Type is not declared: %s" p.colset)
            $"type t = | T of {p.colset}" (p.name, p.colset))
    |> fun fields -> { places = fields }
```

Listing 4.22: How the marking type is generated

4.2.3 AbstractTransitions and AbstractArcs

In order to generate the code for bindings, binding functions and partially binding functions, information from the model needs to be parsed and processed. Types were created to define the necessary data and structure it during the process. These are shown below in Listing 4.23.

```
type AbstractTransition =
    { transition: Transition
      vars: Set<Var>
      inArcs: AbstractArc list
      outArcs: AbstractArc list }
```

Listing 4.23: The AbstractTransition type

An AbstractTransition constitutes the information required for making a binding type, the binding and partially binding functions. The transition field is a reference to the transition. The last chapter showed the type. The vars field is the set of variables associated with the transition, i.e. present in arc expressions or guards of the transition. It constitutes fields in the binding record type. The inArcs field is the arcs in which the direction points toward the transition, and outArcs is the arcs in which the direction is pointing outward the transition. Other types used to structure necessary data are the Var type, the ArcExpr type and the AbstractArc type, shown below in Listing 4.24. The reader can recognize the ArcExpr as the type returned in Listing 4.20.

```
type Var = { id: string; colset: string }

type ArcExpr =
    | Expr of string * string * bool
    | Pattern of string * string * bool

type AbstractArc =
    { arcExpr: ArcExpr
      freeVars: Var list
      inPlace: Place option
      outPlace: Place option }
```

Listing 4.24: The Var, ArcExpr and AbstractArc types

An AbstractArc constitutes information used by an AbstractTransition, including the connected places, the free variables on the arcs, and the arc expression type. The arcExpr field represents the arc expression and is defined as a sum type with two values; The Expr value is a F# expression, and the Pattern value indicates that the arc expression is a pattern. The freeVars field contains the arc expression variables that need to the bound to tokens. A Var type is defined to keep track of the variable's identifier and type. The inPlace option and outPlace option fields contain a connected place if this arc is an input arc to the transition or an output arc to the transition, respectively.

Transitions may optionally have guards, which need to be checked to verify that they are Boolean expressions. The guards are placed in square brackets, representing an array literal in F#, so the check is to test if the expression is a list of Booleans. Listing 4.25 below showns how we check the guard and find the free variables. The findFreeVars function is shown and discussed in the following subsection.

```
let vars =
  match t.guard with
  | Some g ->
    checkIfValid
      (sprintf "%s is not of type bool" g)
      $"fun () -> {g} : bool list"
    let guardVars = findFreeVars g cpnModel.vars |> Set.ofList
    Set.union arcVars guardVars
  | None -> arcVars
```

Listing 4.25: Checking type and finding free variable on guards

4.2.4 Finding the free variables

Finding the free variables in an arc expression or a guard is shown below in Listing 4.26. It is achieved by combining code quotations and the hosted interactive evaluation session. The arc expression passed as an input parameter called `expr` is placed inside a code quotation and evaluated in the session. The `expr` is of type `Expr` and can be matched on its content. Some uninteresting pattern matching constructs are not shown to save some space. The matching is quite similar to visiting nodes in an abstract syntax tree and is the reason for the recursive function `findFreeVarsRec`. Matching `PropertyGet` means a free variable and stems from the fact that the defined variables of the model are defined in the environment.

```
let findFreeVars expr (declVars: Map<string, string>) =
    let expr = evalExpr<Expr> $"<@ {expr} @>"

    let rec findFreeVarsRec expr declVars acc =
        match expr with
        ...
        | Lambda (_, expr) ->
            findFreeVarsRec expr declVars acc
        | Call (_, _, exprList) ->
            (exprList
             |> List.collect
                (fun e -> findFreeVarsRec e declVars acc))
            @ acc
        | IfThenElse (expr1, expr2, expr3) ->
            findFreeVarsRec expr1 declVars acc
            @ findFreeVarsRec expr2 declVars acc
              @ findFreeVarsRec expr3 declVars acc @ acc
        | PropertyGet (_, info, _) ->
            try
                { id = info.Name
                  colset = Map.find info.Name declVars }
               :: acc
            with
            | :? KeyNotFoundException -> acc
        ...
    findFreeVarsRec expr declVars List.Empty
```

Listing 4.26: Finding the free variables in an arc expression

4.2.5 Simple types

One important check to be made about the input and output variables on input and output arcs, respectively, in a transition, is to verify that the output variables are a subset of the input variables found in arc expressions categorized as patterns. If this is not the case, the difference needs to be variables with *simple types*. We want the variables to be of simple types because when we try to find all possible enabled bindings for a transition, an unlimited number of values means an unlimited number of enabled bindings, which is impossible to calculate. Simple types is defined as types with a finite and small set of values, Boolean or sum types with constrained fields. Shown below in Listing 4.27 are the two simple types defined in the two-phase commit protocol. In this case, they are *sum* types with named constructors that take no arguments.

```
type Vote =
    | Yes
    | No

type Decision =
    | Abort
    | Commit
```

Listing 4.27: Simple types defined in the two-phase commit protocol

Types of variables are checked using the hosted interactive evaluation session and reflection. Shown below in Listing 4.28 is the code that determines if a variable is *simply* typed. The input parameter is the variable's identifier, or its name, named as just `id`. The F# reflection namespace needs to be opened, and beginning at line 6 is the interesting part. The expression passed to the hosted interactive evaluation session. The if-branch checks whether the type is a sum type (union), and if so, it checks if all its possible values are constrained by the lack of fields in its union cases, the else branch is simply checking if the type is Boolean.

```
 1  let isSimplyTyped id =
 2      fsiSession.EvalInteraction "open FSharp.Reflection"
 3
 4      let result, _ =
 5          fsiSession.EvalExpressionNonThrowing
 6              $"let t = (<@ {id} @>.Type)
 7                in if FSharpType.IsUnion t
 8                then FSharpType.GetUnionCases t
 9                |> Array.forall
10                   (fun i -> i.GetFields().Length = 0)
11                else t = typeof<bool>"
12
13      match result with
14      | Choice1Of2 (Some value) ->
15          value.ReflectionValue |> unbox<bool>
16      ...
```

Listing 4.28: Function to determine if a variable is simply typed

4.2.6 Pattern binding basis

To define the partially binding functions of transitions, we need to find the *pattern binding basis*. The pattern binding basis of a transition is the smallest set of input arcs, which arc expressions are patterns, that *covers* all the variables of the transition. To cover means in this context, be responsible for the binding. This pattern binding basis is used to create the partially binding functions almost directly. Shown below in Listing 4.29 is the code to find it with an abstract transition as input. It is a greedy algorithm. The arcs are sorted based on the number of free variables in the arc expression. The ones with the most variables are considered first. The sorting is not shown because it is unimportant, but `patArcs` on line 3 contains the sorted list. The recursive function defines a base case at line 12, if the variables to be covered are now covered, we are finished, and the collected arcs in `arcsInPbb` are returned at line 13. If not, the list of arcs to consider is pattern matched to handle the case. If it is empty at line 16, simple types are the only variables left to be added. If the list contains more arcs at line 22, the arcs at the head of the list are considered and are added to the pattern binding basis if their variables contribute to an increase in the covering set check at line 25.

```
 1 let findPatternBindingBasis (at: AbstractTransition) =
 2     // Sorting the arcs with most variables first
 3     let patArcs =
 4     ...
 5 
 6     let rec findPatternBindingBasisRec
 7         (toCover: Set<Var>)
 8         (covering: Set<Var>)
 9         (patArcs: AbstractArc list)
10         arcsInPbb
11         =
12         if toCover = covering then
13           arcsInPbb
14         else
15           match patArcs with
16           | [] ->
17             let simpleTypeNotCovered =
18               Set.difference toCover covering
19             let aVars = Set.union covering simpleTypeNotCovered
20             findPatternBindingBasisRec
21               toCover aVars patArcs arcsInPbb
22           | head :: tail ->
23             let newCovering =
24               Set.union covering (head.freeVars |> Set.ofList)
25             if Set.count newCovering = Set.count covering
26             then
27               findPatternBindingBasisRec
28                 toCover covering tail arcsInPbb
29             else
30               findPatternBindingBasisRec
31                 toCover newCovering tail (head :: arcsInPbb)
32 
33     findPatternBindingBasisRec
34       at.vars Set.empty patArcs List.Empty
```

Listing 4.29: Finding the pattern binding basis of a transition

To exemplify what is happening above, consider the case of the transition `ReceiveCanCommit`, shown again in Listing 4.1. It has two input arcs and three output arcs, and only the input arc expressions are considered when finding the pattern binding basis.

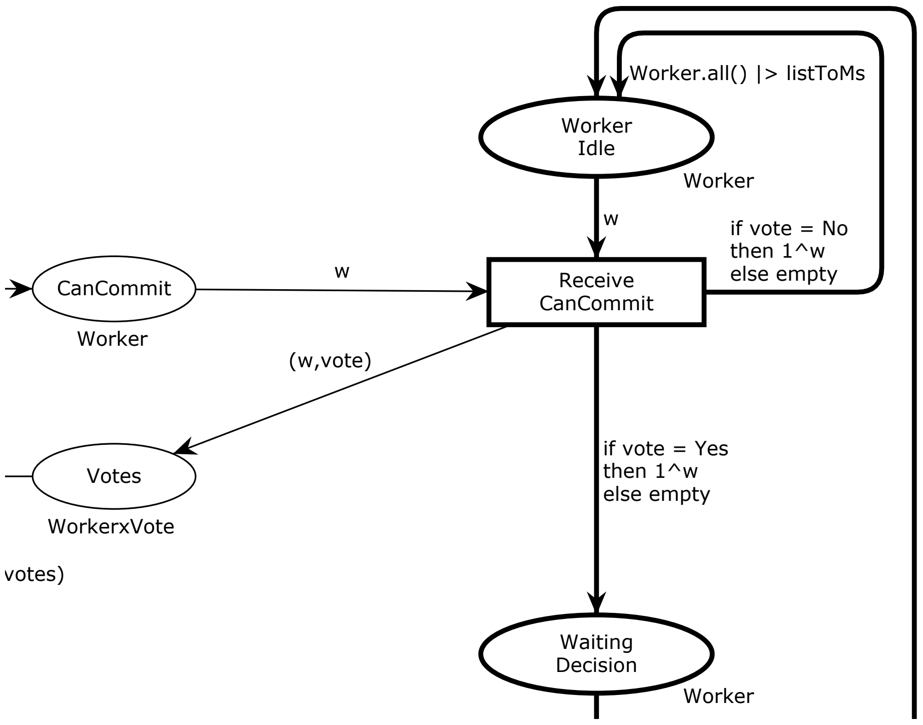


Figure 4.1: Part of the two-phase commit protocol CPN model

The transition has two free variables `w` and `vote`, and the first input arc that is considered can, in this case, be either since both have an equal amount of free variables, the one variable `w`. The algorithm starts by passing the variables of the transition to be covered `w` and `vote`, an empty set of covered variables, the two arcs and an empty list of arcs in the pattern binding basis to the recursive function. In the first call, the arc connected to the `WorkerIdle` is at the head of the `patArcs` list, simply because it is defined earlier in the .cpn file and is considered. The arcs variables are added to `newCovering` and the arc to `arcsInPbb`. There is still a variable `vote` to be covered on the next call. The arc connected to the `CanCommit` is at the head of the `patArcs` list. However, it will not result in an increase of `newCovering`, it is still the same as `covering` as checked on line 25 in Listing 4.29. In the final call, `patArcs` list is empty, and `vote` is added to the list of covered variables because it is *simply typed*.

4.2.7 Enabling

No further parsing or analysis is needed to define the enabling functions. The information to generate the functions is the transition name used in typing, the guard of a transition to verify the enabling, the set of variables in the binding to destructure the input pattern, and the input arcs to get the correct field (place) in the marking.

4.2.8 Occurrence

Similarly, the information needed to generate the occurrence function is already available in the abstract transition type. The information is the binding, used to destructure input patterns created from the variables and input and output places from the `inArcs` and `outArcs`, respectively, used to change the tokens in the multi-sets on the different fields, representing places of the marking.

4.3 Output Component

The `Output` component is quite simple, having the `Parser` component doing the heavy lifting. Its job is to create files and output the string content containing F# code created in the `Parser` component and the formatting job done in the `Semantic` component. It uses the StringBuilder Class [44] from the System.Text namespace in .NET. This completes the main idea of the CPN simulation system as illustrated by the Figure 3.1 from the previous chapter. It is shown again below in Figure 4.2. We take a .cpn file containing a CPN model with inscriptions written in F# and produce a F# .NET project containing code that a simulator can use to execute the CPN model.

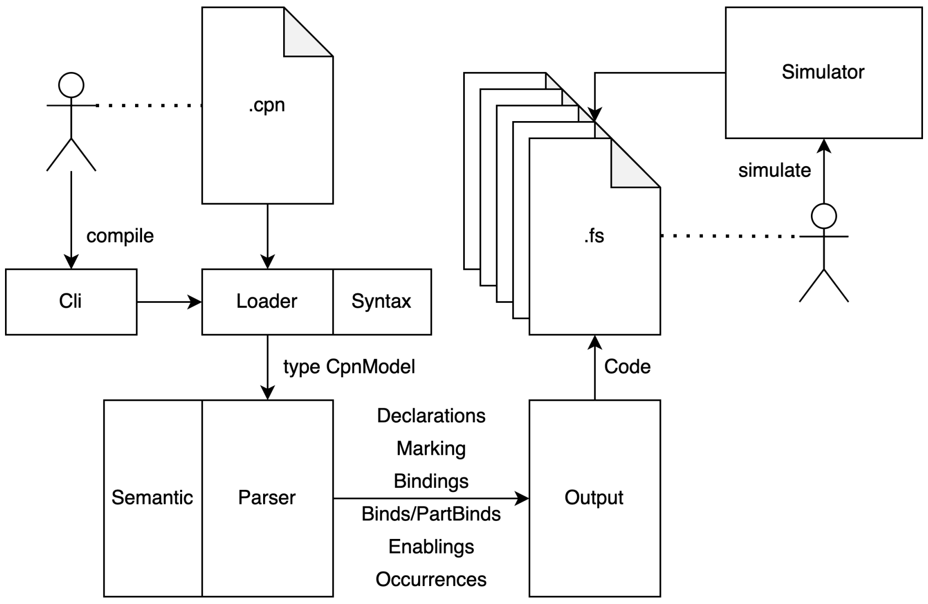


Figure 4.2: A high level view of the compiler and runtime environment

One thing worth mentioning is the project layout, where the .cpn model file needs to be placed and where the generated F# files end up. Shown below is the project structure. Under the `examples` folder is a folder called `TCP`, which stands for two-phase commit protocol. Under it again is a `model` folder, and inside there should the .cpn file reside. The generated code ends up inside the `TPC` folder next to the `model` folder. The Simulator is placed in the folder above, inside the `examples` folder, and it opens the `Examples.TPC` namespace to access the generated code.

```
cpn-simulator
├── examples
│   ├── Other models...
│   ├── TPC
│   │   ├── model
│   │   │   └── .cpn file
│   │   └── The generated F# files end up here...
│   ├── Simulator.fs
│   └── StateSpaceExploration.fs
└── src
    └── compiler
        ├── Cli
        ├── Loader
        ├── Output
        ├── Parser
        ├── Semantic
        └── Syntax
```

Chapter 5

Enabling and Occurrence

This chapter focuses on the generated code and explains why the different constructs and artefacts are generated and necessary to execute CPN models. The two key concepts are the enabling and occurrence of transitions and how they change the marking of a CPN model.

5.1 Overview

Figure 5.1 shows the links between a *marking*, *enabling* and *occurrence*. A circle is a data structure. A diamond square is a function. An arrow represents the passing of data structures either as arguments or, in the case of `new marking` is supposed to represent that the new marking will be used as the next marking. The figure is supposed to illustrate how the state changes in a CPN model, represented by markings, and give the reader a better understanding of the connection between concepts.

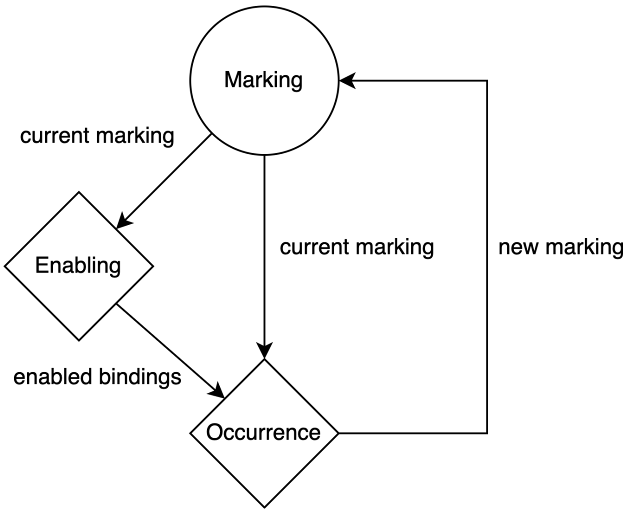


Figure 5.1: The links between a marking, enabling and occurrence functions

5.2 Enabling of bindings

A few things need to exist to compute the enabling of a binding. Shown below again in Figure 5.2, to increase readability for the subsequent discussions, is a part of the two-phase commit protocol CPN model. We will again focus on the `ReceiveCanCommit` transition.

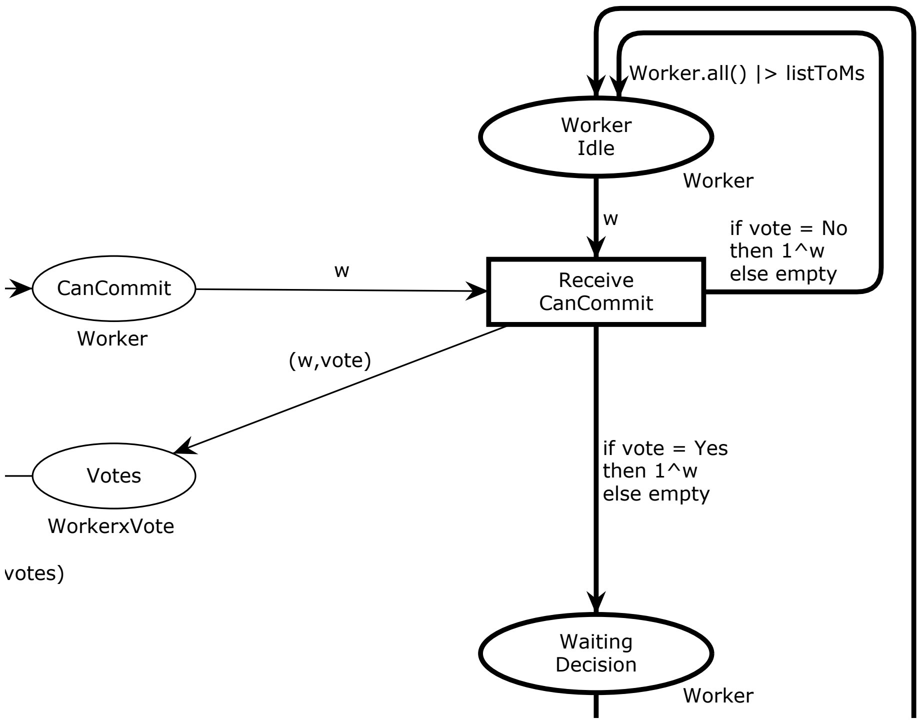


Figure 5.2: Part of the two-phase commit protocol CPN model

5.2.1 Binding type

The generated binding type is central in the process, and each transition has its binding type, a record containing the variables of a transition as fields. Shown below in Listing 5.1 is the generated binding type for the `ReceiveCanCommit` transition. It is defined as a record, and note that every field is an `option` type. We use the `option` type to handle partial bindings. A partial binding will contain `None` values.

```
type ReceiveCanCommit_Binding =
  { _u : unit option
    vote : Vote option
    w : Worker option }
```

Listing 5.1: Binding type for ReceiveCanCommit

Shown below in Listing 5.2 is the generated general binding for the two-phase commit protocol. It is a sum type with a constructor for each transition in the model. It is created to have a common type interface for bindings, for instance, useful in lists.

```
type Binding =
  | SendCanCommit of SendCanCommit_Binding
  | ReceiveAcknowledgements of ReceiveAcknowledgements_Binding
  | ReceiveCanCommit of ReceiveCanCommit_Binding
  | ReceiveDecision of ReceiveDecision_Binding
  | AllVotesCollected of AllVotesCollected_Binding
  | CollectOneVote of CollectOneVote_Binding
```

Listing 5.2: General binding type

The `_u : unit option` field in the bindings above represents the unit token that always will be a part of an enabling and occurrence and is a necessary field to represent a binding without any variables. It is present on all binding types to avoid special cases. For example, the `SendCanCommit` transition has no variables, and the only field in the binding type will be `_u : unit option`.

5.2.2 Binding and partially binding functions

The first thing the enabling function calls when it needs to determine the enabling bindings is the binding function called *Bind*. One binding function for each transition is generated. The binding function's overarching goal is to find all the possible enabled bindings in a given marking. It uses a helper function called *PartBind*, a partially binding function whose job is to partially bind the tokens present in the marking into a binding type. It is called partially bind because it might not be the case that only one arc expression is sufficient to consider a fully populated binding value, a complete binding. In other words, a partial binding has `None` values. It partially has its fields *filled*. A partial binding might be created because the model might contain some arc expressions that do not *cover* all variables of a transition, meaning that multiple arc expressions need to be considered to create a complete binding. The first part of the rather long generated `Bind` function and the corresponding `PartBind` function for the `ReceiveCanCommit` transition are shown below in Listing 5.3.

```
 1  let receiveCanCommit_PartBind_WorkerIdle
 2      : Worker -> ReceiveCanCommit_Binding =
 3          fun w -> { _u = Some ()
 4                     vote = None
 5                     w = Some w }
 6
 7  let receiveCanCommit_Bind marking =
 8    let pbs = [
 9     [ ({ _u = Some (); vote = None;w = None }
10       :ReceiveCanCommit_Binding) ]
11     [ { _u = Some (); vote = Some No; w = None }
12       { _u = Some (); vote = Some Yes; w = None } ]
13       msMap receiveCanCommit_PartBind_WorkerIdle marking.WorkerIdle
14    ] |> List.concat
15    ...
```

Listing 5.3: PartBind and part of the Bind functions for ReceiveCanCommit

The code above, as mentioned, finds all the possible partial bindings for the `ReceiveCanCommit` transition. Line 9 is a dummy partial binding, placed there to avoid special cases. Note that partial bindings are generated with the possible values for the `vote` field. It is an exhaustive list of all the values of the simple type. It might be clearer to the reader why the variables not covered by input arc expressions need to be *simply typed*. Otherwise, there would be as many partial bindings as there were values of the type. Line 13 calls the only partial binding function to the transition because, as we explained in the previous chapter, the arc connected to the `WorkerIdle` place is the only arc in the pattern binding basis to the `ReceiveCanCommit` transition. The partial binding function name is made from the transition's name and the place's name. The field considered in the marking passed to the partial binding function also matched the place's name. The resulting array `pbs` on line 8 collects all the partial bindings.

After finding all the partial bindings, the `Bind` function goes through them and verifies that they are complete. When it considers a partial binding with fields containing `None` values, it tries to complete the binding by creating a merge between the partial binding and some other binding in the list that contains some value on the specific field. It handles *conflicts* by ensuring that the bindings to be merged do not have different values on other fields. The rest of the generated function is shown below in Listing 5.4. The code might not follow too many good coding principles, but it is one of the easier to generate.

```
...
pbs
|> List.fold (fun bindings pb ->
 let cbs = [ pb ]
 let cbs =
  cbs
  |> List.map (fun pb ->
   if pb.vote.IsNone
   then pbs |> List.where
   (fun pb' ->
        pb'.vote.IsSome &&
        if pb'.w.IsSome
        then pb'.w = pb.w
        else true)
    |> List.map (fun pb' -> { pb with vote = pb'.vote })
   else [ pb ])
  |> List.concat
 let cbs =
  cbs
  |> List.map (fun pb ->
   if pb.w.IsNone
   then pbs |> List.where
   (fun pb' ->
        pb'.w.IsSome &&
        if pb'.vote.IsSome
        then pb'.vote = pb.vote
        else true)
    |> List.map (fun pb' -> { pb with w = pb'.w })
   else [ pb ])
  |> List.concat

 cbs
 |> List.filter (fun pb -> pb.vote.IsSome && pb.w.IsSome)
 |> List.fold
     (fun bindings cb ->
      if List.contains cb bindings
      then bindings
      else cb :: bindings)
     bindings)
(pbs
 |> List.filter (fun pb -> pb.vote.IsSome && pb.w.IsSome))
```

Listing 5.4: Rest of the Bind function for ReceiveCanCommit

5.2.3 Enabling functions

The enabling function, as mentioned, first passes the marking to all the binding functions of every transition, then secondly pass the bindings returned to the enabling functions, again one each for every transition. The enabling functions ensure sufficient tokens on all the connected input places to the specific transition and check any guard to determine a binding as enabled. The enabling functions for the `ReceiveCanCommit` transition is shown below in Listing 5.5. Note how the binding type is destructed on line 3 to get its field.

```
1  let receiveCanCommit_Enabling marking bindings =
2    bindings |> List.filter
3      (fun ({ _u = Some (); vote = Some vote;w = Some w }
4            : ReceiveCanCommit_Binding) ->
5            (1^w) <<= marking.WorkerIdle  &&
6            (1^w) <<= marking.CanCommit)
7            |> List.map(fun b -> ReceiveCanCommit b)
```

Listing 5.5: The enabling function for ReceiveCanCommit

The checking of whether there exist sufficient tokens on the connected input places is done on lines 5 and 6 in Listing 5.5. The transition has two input places, and both need to be considered, and it is the final step in enabling a binding. The *global* enabling function used by the simulator to collect all the enabled bindings is shown below in Listing 5.6.

```
let enabling marking = [
    sendCanCommit_Bind marking
    |> sendCanCommit_Enabling marking
    receiveAcknowledgements_Bind marking
    |> receiveAcknowledgements_Enabling marking
    receiveCanCommit_Bind marking
    |> receiveCanCommit_Enabling marking
    receiveDecision_Bind marking
    |> receiveDecision_Enabling marking
    allVotesCollected_Bind marking
    |> allVotesCollected_Enabling marking
    collectOneVote_Bind marking
    |> collectOneVote_Enabling marking
  ]
  |> List.concat
```

Listing 5.6: Global enabling function

5.3 Occurrence of bindings

After all the enabled bindings are computed in a given marking, one must be chosen to occur. Choosing the binding could be random or by specifying which one to occur to the simulator. The chosen binding is, either way, passed to the occurrence function along with the current marking.

5.3.1 Occurrence function

The occurrence function is simply a pattern matching the binding passed as an argument identifying which binding to occur. The function is shown partly below in Listing 5.7, showing the case for `ReceiveCanCommit` transition. Essentially, the creation of a new marking with the addition of tokens on places connected to a transition through output arcs and the subtraction of tokens connected through input arcs. All connected places are considered, meaning that some places might have tokens subtracted and added in the same occurrence. This occurrence function completes executing a step in the CPN model.

```
let occurrence marking binding =
  match binding with
    ...
  | ReceiveCanCommit b ->
    { marking with
      CanCommit = marking.CanCommit
      -- (b |> fun ({ _u = Some ()
                      vote = Some vote
                      w = Some w }
                     : ReceiveCanCommit_Binding) ->
                      1 ^ w)
      Votes = marking.Votes
      ++ (b |> fun ({ _u = Some ()
                      vote = Some vote
                      w = Some w }
                    : ReceiveCanCommit_Binding) ->
                      1 ^ (w, vote))
      WaitingDecision = marking.WaitingDecision
      ++ (b |> fun ({ _u = Some ()
                      vote = Some vote
                      w = Some w }
                    : ReceiveCanCommit_Binding) ->
                      if vote = Yes then 1 ^ w else empty)
      WorkerIdle = marking.WorkerIdle
      ++ (b |> fun ({ _u = Some ()
                      vote = Some vote
                      w = Some w }
                    : ReceiveCanCommit_Binding) ->
                      if vote = No then 1 ^ w else empty)
      -- (b |> fun ({ _u = Some ()
                      vote = Some vote
                      w = Some w }
                    : ReceiveCanCommit_Binding) ->
                      1 ^ w)
    }
```

Listing 5.7: Global occurrence function

Chapter 6

Evaluation

In the earlier chapters, we have only used one model, the two-phase commit protocol, to exemplify and explain the different parts of the system and the code generated. As part of the system's evaluation, we have compiled and simulated four different models to assess how the system handles them and ensure that it is working. The five models in total are shown in the list below.

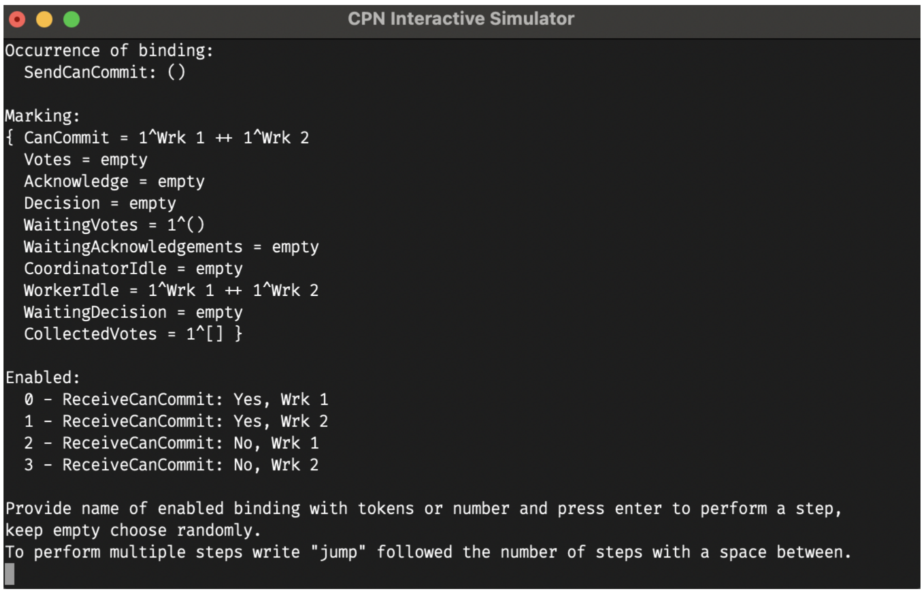
1. Two-phase commit protocol
   - A protocol used by database servers to handle transactions that modifies data on multiple database servers.
2. Dining Philosophers
   - A classic problem used to illustrate issues when working with shared resources and concurrency.
3. Simple Protocol
   - A transmit protocol that sends packets and handles the loss and resend of packets.
4. Resource Allocation
   - A system consisting of a set of processes that in cyclic manner requests, allocates, and releases different types of resources.
5. Distributed Database
   - A system where a set of database managers coordinates access to a database that updates are consistent.

We want the system to be able to catch errors and not produce any code of a faulty model, as discussed in previous chapters. We verified that this was the case by giving the system various errors to handle. In addition to the experiments with the five models, we made a state-space exploration to ensure that all the model states are possible to reach in the simulation. A correct state-space exploration is a concrete validation of the correctness of the system.

6.1 Simulation

When the compiler has generated code and placed it in an F# .NET library, the simulator can access all the constructs once the namespace is opened. The simulator is an interactive command-line interface and is quite simple, supporting only three commands. The three commands are next step `enter`, multiple next steps `jump` and to quit `exit`. When a next step command `enter` is executed, the user may specify the enabled binding to occur. Otherwise, it is chosen randomly.

Figure 6.1 shows the simulator running. It shows the last occurred binding at the top, under the current marking, and under that is the enabled bindings in the current marking.



```
Occurrence of binding:
  SendCanCommit: ()

Marking:
{ CanCommit = 1^Wrk 1 ++ 1^Wrk 2
  Votes = empty
  Acknowledge = empty
  Decision = empty
  WaitingVotes = 1^()
  WaitingAcknowledgements = empty
  CoordinatorIdle = empty
  WorkerIdle = 1^Wrk 1 ++ 1^Wrk 2
  WaitingDecision = empty
  CollectedVotes = 1^[] }

Enabled:
  0 - ReceiveCanCommit: Yes, Wrk 1
  1 - ReceiveCanCommit: Yes, Wrk 2
  2 - ReceiveCanCommit: No, Wrk 1
  3 - ReceiveCanCommit: No, Wrk 2

Provide name of enabled binding with tokens or number and press enter to perform a step,
keep empty choose randomly.
To perform multiple steps write "jump" followed the number of steps with a space between.
```

Figure 6.1: CPN Interactive Simulator running two-phase commit protocol

A part of the code of the simulator is shown below in Listing 6.1, more specifically, the code of the `jump` command, which executes the number of specified steps.

```
let steps = input.Split " " |> Seq.last |> int

let marking =
    List.init steps (fun x -> x)
    |> List.fold
        (fun marking _ ->
            enabling marking
            |> chooseRandom
            |> occurrence marking)
        marking
```

Listing 6.1: The jump command of the simulator

The current marking is the first argument to the `List.fold` function. The second argument is a list of steps only used to control the number of steps taken in the model. The function passed to the `List.fold` is what is significant, and its input parameter is the current marking functioning as the state. The body calls the `enabling` function, which takes the marking as an argument and returns a list of all the enabled bindings. The list is passed to a `chooseRandom` function using the pipeline operator `|>` Its only job is to choose one of the enabled bindings randomly. The chosen binding is, in turn, *piped* as the last argument to the `occurrence` function, which also has the current marking as the first argument. What is returned from the `occurrence` function is the new marking.

6.2 State-Space Exploration

In a state-space, the possible states are nodes connected with arcs to create possible paths in a graph. Each node has as many arcs as the following possible states from itself to another node. A state-space exploration is a graph exploration of all nodes visiting the states. The traversal is done by accumulating all explored nodes in a set and putting unvisited nodes in a queue. The result is the number of possible states and the number of paths to those states. The result of running the state-space exploration of our five examples are shown below in Table 6.1. Our results of the state-space explorations match the results of the state-space exploration in CPN Tools given the same models, included in Appendix B, in regards to the number of nodes and arcs. The CPU time is the time it takes to calculate in milliseconds. The testing was done using a 16" MacBook Pro laptop with M1 Max and 64 GB ram from 2022.

Model	Nodes	Arcs	CPU time
Two-phase commit with 2 workers	43	64	31
Two-phase commit with 3 workers	281	512	38
Two-phase commit with 4 workers	2323	4774	99
Dining Philosophers with 5 philosophers	11	30	22
Dining Philosophers with 10 philosophers	123	680	28
Dining Philosophers with 15 philosophers	1364	11310	155
Simple Protocol	428	1130	44
Resource Allocation	13	20	26
Distributed Database with 3 databases	28	42	27
Distributed Database with 6 databases	1459	4872	114
Distributed Database with 9 databases	59050	314946	12685

Table 6.1: State-space exploration of our five example models

6.3 Performance

The system is not optimized regarding performance and was rather developed with correctness in mind. For instance, the system calculates the enabled bindings for every transition in every marking, even though sometimes only neighbour transitions to the last occurred transition might change enabling status. The loading and creation of files affect the performance, but that is unavoidable in this system. The compilation takes roughly some seconds to finish. Running the simulator is fast. Figure 6.2 below shows the times it takes to execute a number of steps in the two-phase commit protocol with different amounts of workers. Using two workers in blue, three workers in green and four workers in red. The testing was done using a 16" MacBook Pro laptop with M1 Max and 64 GB ram from 2022.

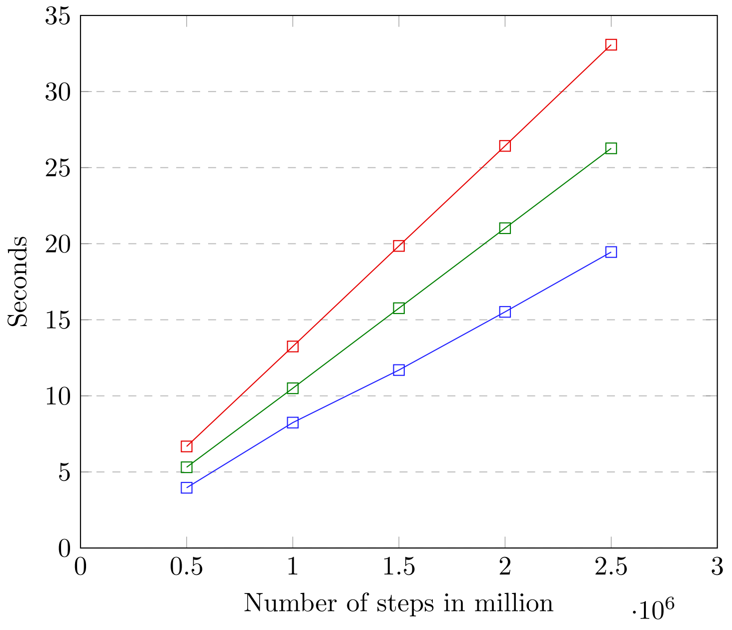


Figure 6.2: Executing number of steps in two-phase commit protocol

6.4 Examples

Below, we provide details on the additional four example models considered in the evaluation.

6.4.1 Dining Philosophers

Shown below in Figure 6.3 is the model of the Dining Philosophers problem [45] created in CPN Tools using F# as the inscription language. Listing 6.2 shows the declarations defined in the model, written in F# and the CPN ML programming language. Dining Philosophers is captioning problems that might arise when multiple entities use the same resources, it may lead to a deadlock. It is often used to illustrate the concept of concurrency.

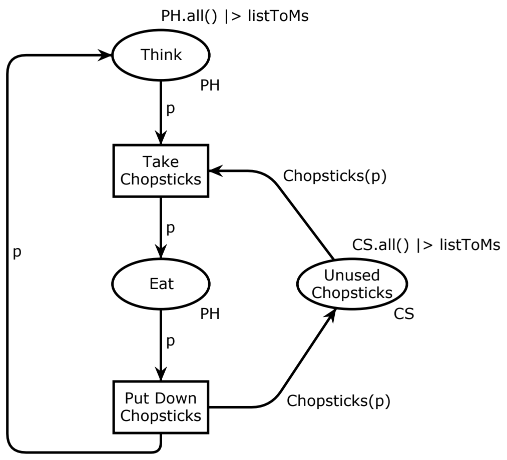


Figure 6.3: Dining Philosophers with F# as inscription language

```
let n = 5
colset PH = index ph with 1..n
colset CS = index cs with 1..n
var p: PH
let Chopsticks (Ph i)
    = (1^Cs(i)) ++ (1^Cs(if i = n then 1 else i + 1))
```

Listing 6.2: Declarations defined in Dining Philosophers

Shown below in Figure 6.4 is the simulator performance for the Dining Philosophers model. Using five philosophers in blue, ten philosophers in green and fifteen philosophers in red.

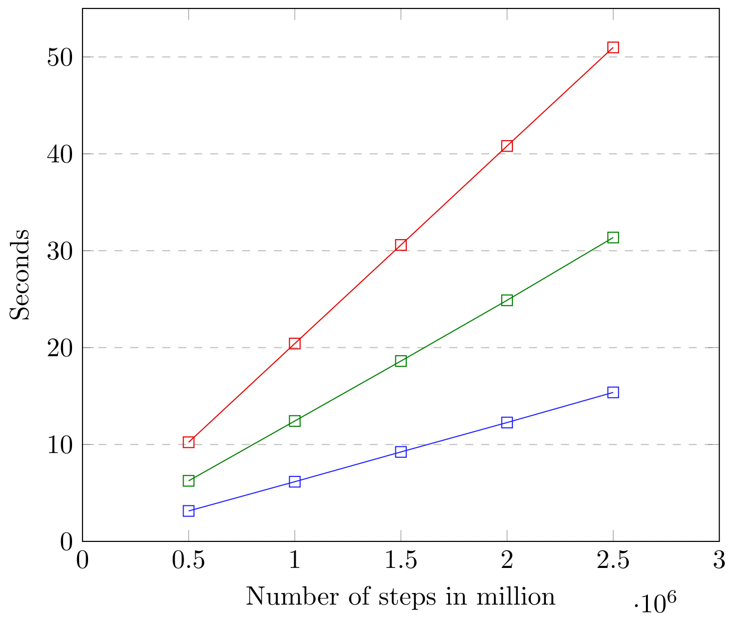


Figure 6.4: Executing number of steps in Dining Philosophers

6.4.2 Simple Protocol

Shown below in Figure 6.5 is the model of a simple protocol created in CPN Tools using F# as the inscription language. Listing 6.3 shows the declarations defined in the model, written in F# and the CPN ML programming language. The Simple Protocol is a transmission protocol that illustrates the sending of packets, loss of packets and sending retries, similar to network protocols.

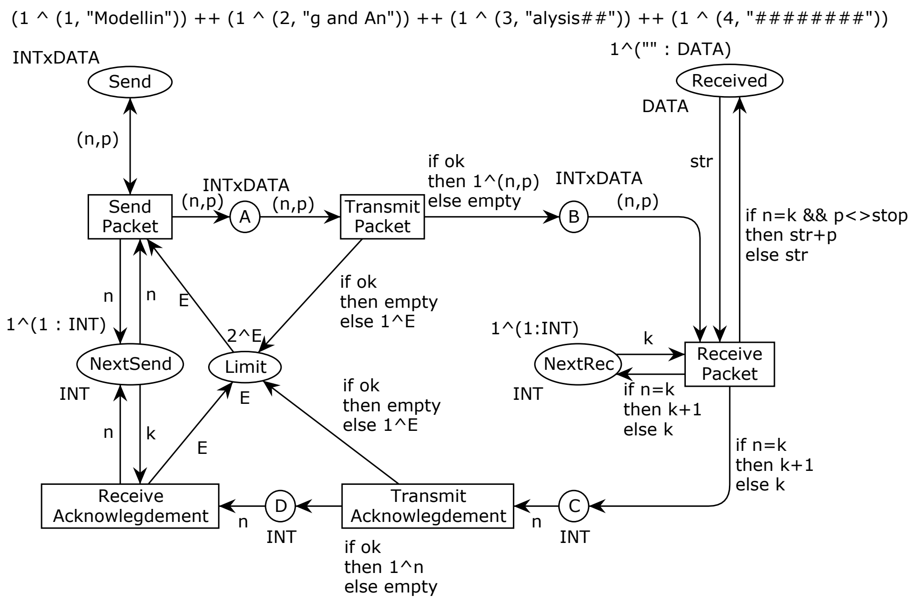


Figure 6.5: Simple Protocol with F# as inscription language

```
colset INT = int
colset DATA = string
colset INTxDATA = product INT * DATA
var n,k: INT
var p, str: DATA
let stop = "########"
colset Ten0 = int with 0..10
colset Ten1 = int with 1..10
var s: Ten0
var r: Ten1
colset BOOL = bool
var ok: BOOL
colset E = with E
```

Listing 6.3: Declarations defined in Simple Protocol

We did not do any simulator performance testing of the Simple Protocol model because the protocol does not run in a loop. It has a finite small number of steps before it reaches the terminating state.

6.4.3 Resource Allocation

Shown below in Figure 6.6 is the model of a resource allocation algorithm created in CPN Tools using F# as the inscription language. Listing 6.4 shows the declarations defined in the model, written in F# and the CPN ML programming language.

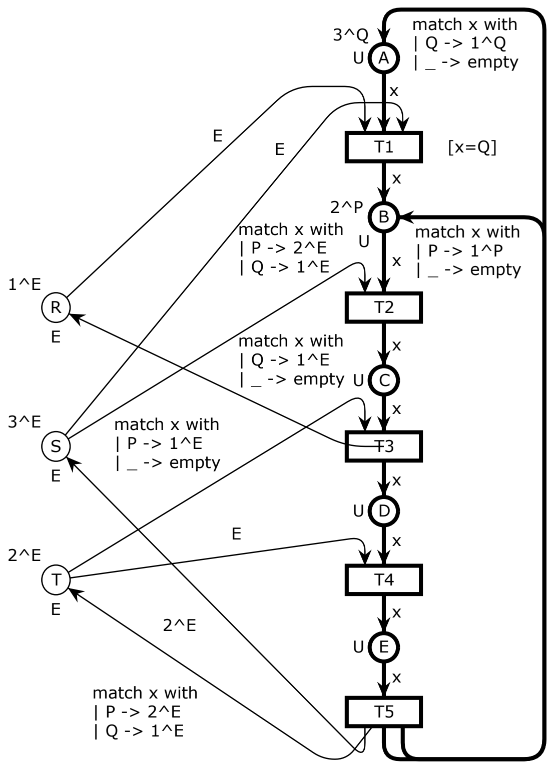


Figure 6.6: Resource Allocation with F# as inscription language

```
colset U = with P | Q
colset E = with E
var x: U
```

Listing 6.4: Declarations defined in Resource Allocation

We did not do any simulator performance testing of the Resource Allocation model because the protocol is not easily parameterizable.

6.4.4 Distributed Database

Shown below in Figure 6.7 is the model of a distributed database created in CPN Tools using F# as the inscription language. Listing 6.5 shows the declarations defined in the model, written in F# and the CPN ML programming language.

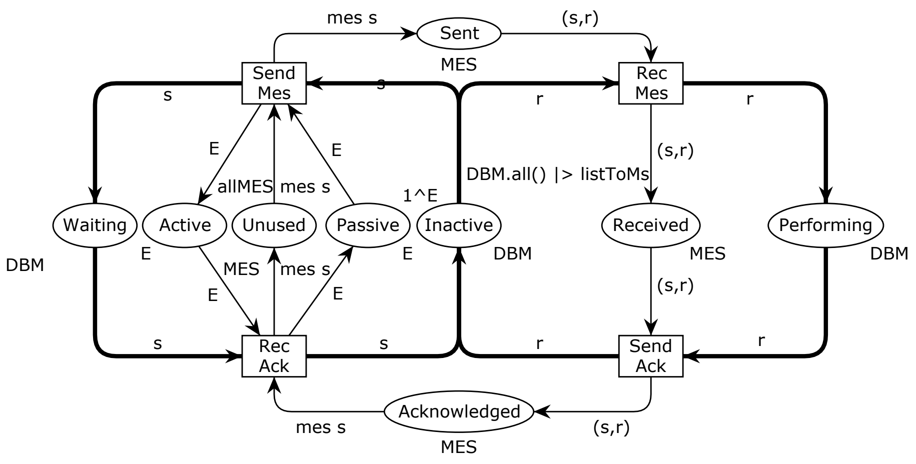


Figure 6.7: Distributed Database with F# as inscription language

```
let n = 3
colset DBM = index d with 1..n
colset MES = product DBM * DBM
colset E = with E
var s, r: DBM
let mes s =
  List.allPairs
    (DBM.all()) (DBM.all())
    |> List.filter(fun (x,y) -> x<>y && x=s )
    |> List.map(fun x -> x : MES)
    |> listToMs
let allMES =
  List.allPairs
    (DBM.all()) (DBM.all())
    |> List.filter(fun (x,y) -> x<>y)
    |> List.map(fun x -> x : MES)
    |> listToMs
```

Listing 6.5: Declarations defined in Distributed Database

Shown below in Figure 6.8 is the simulator performance for the Distributed Database. Using three databases in blue, six databases in green and nine databases in red.

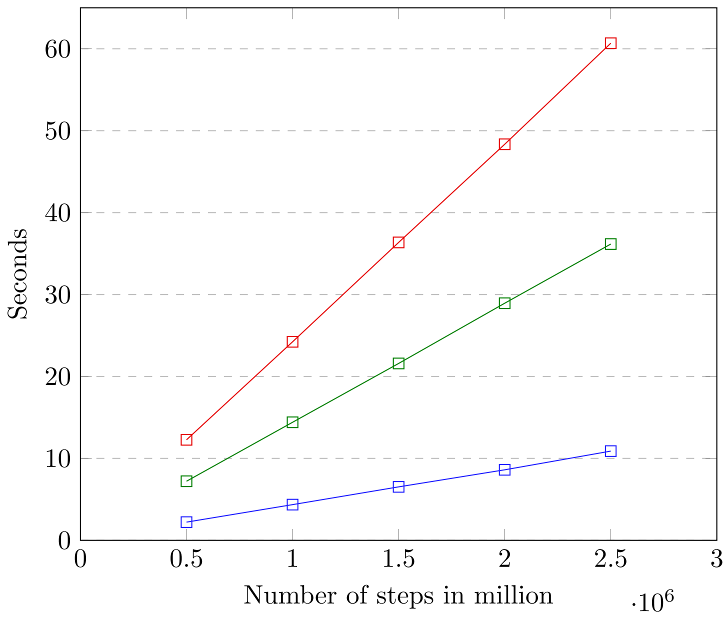


Figure 6.8: Executing number of steps in Distributed Database

Chapter 7

Conclusion and Further Work

7.1 Conclusion

We conclude the project and deem it successful with the evidence shown and discussed throughout this thesis. The technologies used to develop the CPN simulation system have been capable, sufficient and well-suited. The research questions we sought to answer are repeated below.

RQ1. How can F# be used as inscription language in CPN models?

RQ2. How can F# and the .NET platform be used to implement a compiler that generates code representing CPN models?

RQ3. How can F# and the .NET platform be used to develop a runtime environment for the simulation of CPN models?

The F# programming language is a good fit as an inscription language for CPN models. It is similar to Standard ML in many aspects. It has excellent support for pattern matching and supports expression-oriented programming well. Small changes were needed to translate the inscriptions of existing models from Standard ML to F#, answering RQ1.

The F# programming language supports many constructs to work efficiently and with metaprogramming. Code quotations were a crucial feature in our system. Combined with the hosted interactive evaluation session from F# Compiler Service, we had a powerful way to analyse, error check and find the necessary information about the model needed to generate the code. The F# type system and its pattern matching capabilities, such as destructuring of records and sum types, helped significantly in the development process. It supports the claim that F# is a good choice.

F# has robust libraries such as FSharp.Data which made it easy to parse the provided .cpn file. The .NET platform also has rich libraries for working with strings and files. Using all the things mentioned above, we showed how F# and the .NET platform could be used to implement a compiler that generates code to represent CPN models, answering RQ2.

We also showed how F# and the .NET platform could be used to develop a runtime environment that simulates CPN models by creating a simulator that used the generated code and handled the execution of steps in the model. The simulator is working with the five provided examples, answering RQ3.

The state-space exploration offers strong proof of the correctness of our CPN simulator system. We tested our system with the five examples, and it successfully generated code that the simulator could use to execute the models.

7.2 Future work

The model to be compiled must be a *flat* model. Flat means non-hierarchical, every construct is on the same level, and the .cpn file contains only one *page*. It is possible to create hierarchical nets consisting of multiple pages in CPN Tools. This project did not consider such models, as all hierarchical models can be made flat.

CPN models may also include timing information [48] which can evaluate the efficiency and real-time systems. We did not consider models with timing information in our system.

Type inference might be another possible extension of our system. For instance, the compiler could figure out the type of the place based on the type of the arc expression. A CPN type not considered and consequently not tried translated into F# is the *subset* type. It is a way to define a type that is a subset of another type with a predicate dictating what values are possible within the type. It is rarely used in practice and can be translated into the same type it is supposed to be a subset of. Therefore it was not spent time on.

A possible future extension is to make the system handle type parameterizable models similar to generics in programming languages. Such models would allow a lot more reuse and make it possible to generalize parts of a model.

Other work can be done regarding handling hierarchical nets, performance increase by only calculating possible enabled bindings, developing a new editor to create models using F# as the inscription language, and automating the opening of the *namespace* of the generated code in the simulator.

Appendix A

Source code

The full source code for CPN simulator system is available here: `https://github.com/smartoceanplatform/cpn-simulator`.
The repository is currently private, but access can be requested by contacting the supervisor.

Appendix B

State Space Exploration in CPN Tools

```
CPN Tools state space report for:
TwoPhaseCommitProtocolWith2Workers.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  43
     Arcs:   64
     Secs:   0
     Status: Full

CPN Tools state space report for:
TwoPhaseCommitProtocolWith3Workers.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  281
     Arcs:   512
     Secs:   0
     Status: Full

CPN Tools state space report for:
TwoPhaseCommitProtocolWith4Workers.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  2323
     Arcs:   4774
     Secs:   0
     Status: Full
```

```
CPN Tools state space report for:
DiningPhilosophersWith5Philosophers.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  11
     Arcs:   30
     Secs:   0
     Status: Full


CPN Tools state space report for:
DiningPhilosophersWith10Philosophers.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  123
     Arcs:   680
     Secs:   0
     Status: Full

CPN Tools state space report for:
DiningPhilosophersWith15Philosophers.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  1364
     Arcs:   11310
     Secs:   1
     Status: Full

CPN Tools state space report for:
SimpleProtocol.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  428
     Arcs:   1130
     Secs:   0
     Status: Full
```

```
CPN Tools state space report for:
ResourceAllocation1.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  13
     Arcs:   20
     Secs:   0
     Status: Full


CPN Tools state space report for:
DistributedDataBaseWith3Databases.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  28
     Arcs:   42
     Secs:   0
     Status: Full

CPN Tools state space report for:
DistributedDataBaseWith6Databases.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  1459
     Arcs:   4872
     Secs:   1
     Status: Full

CPN Tools state space report for:
DistributedDataBaseWith9Databases.cpn

Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  59050
     Arcs:   314946
     Secs:   1272
     Status: Full
```

Bibliography

[1] URL: https://fsharp.org (visited on Apr. 28, 2022).

[2] URL: https://www.microsoft.com/en-us/research/project/f-at-microsoft-research (visited on Apr. 27, 2022).

[3] URL: https://dotnet.microsoft.com (visited on Apr. 28, 2022).

[4] URL: https://www.smlnj.org/index.html (visited on Apr. 27, 2022).

[5] URL: https://beta.cs.au.dk (visited on Apr. 27, 2022).

[6] URL: https://cs.au.dk/cpnets/industrial-use (visited on Mar. 30, 2022).

[7] URL: https://www.informatik.uni-hamburg.de/TGI/PetriNets/index.php (visited on Mar. 30, 2022).

[8] URL: https://sfismartocean.no (visited on Apr. 27, 2022).

[9] URL: https://www.scala-lang.org/ (visited on Apr. 27, 2022).

[10] URL: https://www.haskell.org/ (visited on Apr. 27, 2022).

[11] URL: https://fsharp.github.io/fsharp-compiler-docs (visited on Feb. 17, 2022).

[12] URL: https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html (visited on May 27, 2022).

[13] URL: http://renew.de (visited on Feb. 17, 2022).

[14] URL: https://lipn.univ-paris13.fr/~evangelista/helena/ (visited on May 27, 2022).

[15] URL: https://lip6.github.io/ITSTools-web/index.html (visited on May 27, 2022).

[16] URL: https://github.com/iig-uni-freiburg/WOLFGANG (visited on May 27, 2022).

[17] URL: https://github.com/iig-uni-freiburg/SEPIA (visited on May 27, 2022).

[18] URL: https://github.com/pierreganty/mist (visited on May 27, 2022).

[19] URL: https://github.com/vldtecno/PTN-Engine (visited on May 27, 2022).

[20] URL: https://github.com/tamarit/pn_suite (visited on May 27, 2022).

[21] URL: https://projects.laas.fr/tina/index.php (visited on May 27, 2022).

[22] URL: https://www.tapaal.net/ (visited on May 27, 2022).

[23] URL: https://www.yasper.org/ (visited on May 27, 2022).

[24] URL: https://www.cosyverif.org (visited on Feb. 17, 2022).

[25] URL: https://fsharp.org/learn (visited on Apr. 28, 2022).
[26] URL: https://dotnet.microsoft.com/learn (visited on Apr. 28, 2022).
[27] URL: https://www.ibm.com/docs/en/informix-servers/14.10?topic=protocol-when-two-phase-commit-is-used (visited on May 1, 2022).
[28] URL: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions (visited on Apr. 7, 2022).
[29] URL: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/tuples (visited on May 6, 2022).
[30] URL: https://jrsinclair.com/articles/2019/algebraic-data-types-what-i-wish-someone-had-explained-about-functional-programming (visited on Apr. 7, 2022).
[31] URL: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/records (visited on Apr. 7, 2022).
[32] URL: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/options (visited on Apr. 7, 2022).
[33] URL: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/code-quotations (visited on Apr. 7, 2022).
[34] URL: https://fsharp.github.io/fsharp-compiler-docs (visited on Apr. 7, 2022).
[35] URL: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection (visited on May 6, 2022).
[36] URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing (visited on May 6, 2022).
[37] URL: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/symbol-and-operator-reference/ (visited on May 7, 2022).
[38] URL: https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-mapmodule.html (visited on May 3, 2022).
[39] URL: https://docs.microsoft.com/en-us/dotnet/fsharp/style-guide/formatting (visited on May 7, 2022).
[40] URL: https://fsprojects.github.io/FSharp.Data (visited on Apr. 4, 2022).
[41] URL: https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/type-providers (visited on Apr. 4, 2022).
[42] URL: https://docs.microsoft.com/en-us/dotnet/fsharp/tools/fsharp-interactive/#scripting-with-f (visited on May 8, 2022).
[43] URL: https://fsharp.github.io/fsharp-core-docs/reference/fsharp-core-operators-unchecked.html (visited on May 18, 2022).
[44] URL: https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder?view=net-6.0 (visited on Apr. 11, 2022).
[45] C.A.R Hoare. “Communicating Sequential Processes.” eng. In: *The Queen’s University Belfast, Northern Ireland* 21, Number 8 (1978), pp. 666–677.

[46] Kurt Jensen. *Coloured Petri nets: Basic concepts, analysis methods, and practical use.* eng. 1st ed. Springer-Verlag, 1992. ISBN: 978-3540555971.

[47] Lars M. Kristensen Kurt Jensen. *Colored Petri Nets: A Graphical Language For Formal Modeling and Validation of Concurrent Systems.* 2015. URL: `https://cacm.acm.org/magazines/2015/6/187324-colored-petri-nets/fulltext` (visited on Dec. 11, 2021).

[48] Lars M. Kristensen Kurt Jensen. *Coloured Petri Nets, Modelling and Validation of Concurrent Systems.* eng. 2009th edition. Springer-Verlag, 2009. ISBN: 978-3-642-00283-0.

[49] Kurt Jensen Lars M. Kristensen Jens Bæk Jørgensen. "Application of Coloured Petri Nets in System Development." eng. In: *J. Desel, W. Reisig and G. Rozenberg (Eds.): ACPN 2003* LNCS 3098 (2004), pp. 626–685.

[50] Søren Christensen Lars M. Kristensen. "Implementing Coloured Petri nets Using a Functional Programming Language." eng. In: *Higher-Order and Symbolic Computation* 17 (2004), pp. 207–243.

[51] Carl Adam Petri. "Kommunikation mit Automaten." PhD thesis. Institut für Instrumentelle Mathematik, Bonn, 1962.

[52] Don Syme. *The Early History of F#.* 2020. URL: `https://fsharp.org/history/hopl-final/hopl-fsharp.pdf` (visited on Apr. 27, 2021).

[53] Michael Westergaard and H.M.W. (Eric) Verbeek. *CPNTools.* 2021. URL: `http://cpntools.org/` (visited on Dec. 3, 2021).