Investigating the use of Scala as Inscription and Implementation Language for Coloured Petri Nets

Matias Vinjevoll

Master's Thesis

Software Engineering

Department of Computing
Bergen University College
Norway

3 June 2013

Supervisor
Lars Michael Kristensen

Abstract

Coloured Petri Nets (CPNs) is a graphical modelling language for concurrent systems, combining the formalism of Petri Nets with the functional programming language Standard ML. CPN models can be constructed, simulated and analysed using CPN Tools.

Scala is a multi-paradigm programming language, combining functional and object-oriented features. Functional features of the language allow Scala to be used as implementation language of a CPN simulator. Object-oriented features introduce new modelling capabilities with Scala as inscription language, and the possibility of a highly modularised simulator implementation.

We address how Scala can be used as inscription and implementation language for CPNs, and relate modelling constructs of CPNs to object-oriented concepts. A Scala simulator for CPNs has been implemented, and integrated with CPN Tools to allow modelling with Scala as the inscription language. By conducting code generation from CPN models to Scala code, constructed models can be simulated with the new Scala simulator. The design of Scala as inscription language and the implementation of the simulator have been validated by means of an industrial-sized example.

Acknowledgements

I would like to thank my supervisor Lars Michael Kristensen for helpful inputs regarding problems at hand, and a thorough follow-up during the writing of this thesis.

I would also like to thank my father Morten Vinjevoll for reading, and providing language corrections of this thesis.

Contents

Chapter 1

Introduction

Software engineering is an extensive discipline, where problems at hand can be quite complex. Models can help expressing problems at a higher abstraction level, and thereby ease the process of reaching a viable solution that satisfies the identified requirements.

Traditionally, modelling is associated with a rigid process, where extensive models are constructed prior to implementation of a software system. Agile Model Driven Development (AMDD) [1] is a practice-based methodology identifying how modelling can be applied in an agile development process. Models are used to communicate a common understanding, both within the development team and to stakeholders. In AMDD, models are only as detailed as needed to start development, and evolve continuously in pace with the system implementation. The Unified Modeling Language (UML) [16] appears as the standard modelling language. UML is a general-purpose modelling language intended to be used throughout the whole development process of a software system, and includes both static structure and dynamic behaviour. While UML models serve as high-abstraction views of systems and documentation, many modelling languages have more involved capabilities, such as automatic code generation. Combining AMDD with tools that allow automatic generation of code from models can give effective development environments.

Model Driven Engineering (MDE) [12] is based on the idea of using models in the implementation of systems by performing code generation or model interpretation. The main concern of MDE is to bridge the gap between the design phase, and the implementation phase of systems. Thus, avoiding the manual translation from model to implementation, which can be an error prone process. Code generation also makes models more reliable

since source code will be in sync with the models, and development time can be decreased. MDE also concerns analysis of models by means of model checking and verification.

In this thesis, we focus on executable models, where the modelled systems can be *simulated*. Being able to model and simulate the system prior to implementation may address errors, ambiguities and lack of details in system specifications. Different verification techniques can be applied on executable models, for instance *state space analysis*, where all possible executions of the system are calculated for analysis. Analysing the behaviour of a modelled system through simulation and verification techniques may address errors in the system specification, that otherwise may be present in the deployed system.

1.1 The CPN Modelling Language

Petri Nets [31] is a mathematical discrete-event modelling language used for constructing formal models of systems. The basic structure of a Petri net is a directed bipartite graph where a node either represents a *place* or a *transition*. Places can be *marked* with *tokens*, representing the *state* of the model, while transitions represent *events* that can *occur* and change the model state. Petri Nets, as described above, are referred to as *low-level Petri Nets*. Petri Nets combined with programming languages are referred to as *high-level Petri Nets*. Coloured Petri Nets (CPNs) [17] is in the class of high-level Petri Nets, and combine the modelling formalism of Petri Nets with the functional programming language Standard ML (SML). CPNs constitutes a general purpose modelling language with expressive power to model complex systems. CPNs are particularly suitable for modelling systems where concurrency, synchronisation and communication are involved, but are not restricted to this domain.

Figure 1.1 shows an example of a CPN model. Places are drawn as ellipses, and transitions are drawn as rectangles, which together with the *arcs* constitute the Petri Net part of the model. All the inscriptions in the model are CPN ML code. CPN ML extends SML with constructs for defining types (called *colour sets*), and declaring *variables*. Defined functions, declared colour sets and variables, and expressions are used as inscriptions on models. A *multiset* data structure to maintain tokens on places, including library functions for manipulation of multisets, are included in the CPN ML environment. Tokens can have associated data values, called *token colours*, arcs have *arc expressions* evaluating to a multiset of tokens, and transitions

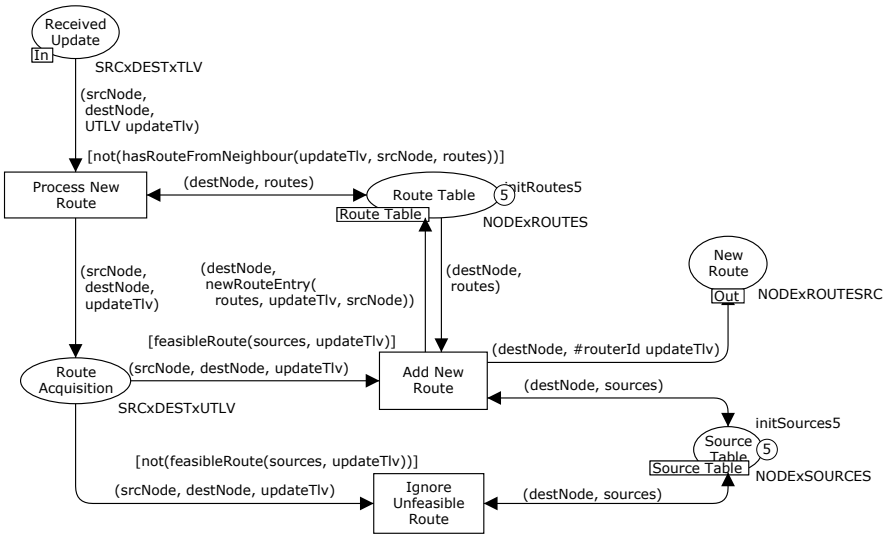


Figure 1.1: A CPN model example.

can have *guards*. CPNs include *hierarchical* concepts, allowing models to be organised into a set of *modules* at different abstraction levels. In fact, the model shown in Fig. 1.1 is a module of a larger model of a routing protocol.

CPN models can be modelled and simulated with CPN Tools [2]. Simulation can be executed in *interactive simulation* mode, where each *step* of the simulation is chosen manually, or in *automatic simulation* mode, where a chosen number of steps are executed automatically. The tool also integrates functionality to analyse and verify models by means of state space analysis, where all reachable states of the model are explored. In other words, all possible outcomes of the system are computed, and can be investigated to formally verify that the system conceptually behaves as expected.

The simulator of CPN Tools is implemented in SML. A CPN model is translated to SML code to conduct simulation and analysis of the model. Since a model is transformed to SML code, it can be used directly in the implementation of a modelled system by embedding the generated simulator code. The core functionality in a CPN simulator is the *enabling inference* algorithm, which calculates events that can *occur* during simulation. The enabling inference algorithm is based on *pattern matching*, which in turn is essential in the simulator implementation language.

1.2 The Scala Programming Language

Scala [6, 21] is a general purpose programming language designed to be scalable and adaptable. It is suited for implementation of small scripts as well as large systems and frameworks where reusable components are essential. Scala is a multi-paradigm language, combining functional and object-oriented features.

Scala is a pure object-oriented language in the sense that all values are objects. This contrasts for instance the programming language Java, having primitive types and allowing static members in classes. Classes in Scala cannot have static members because they will not be unique for an object. Instead, Scala has singleton objects, that can be related to a class. A class in Scala can be composed by inheriting from another class, and additionally mix in functionality from one or more *traits*. Traits can be seen as rich interfaces, allowing declaration of fields and methods. A restriction on traits compared to classes is that traits cannot have constructors, hence, they cannot be instantiated.

Combining this with the rich type system, high-level building blocks can be obtained to build flexible and scalable systems. Listing 1.1 shows a simple example of a class `Orange`, extending the class `Fruit`, and which mixes in functionality from the `Ordered` trait, which requires that the abstract method `compare` is implemented.

Listing 1.1: Example of a Scala class.

```
class Orange(val weight: Int) extends Fruit with Ordered[Orange] {
  def compare(that: Orange) = weight - that.weight
}
```

Scala is functional, as all functions return a value which in turn is an object. Scala has functional features found in pure functional languages, such as support for higher-order functions, currying, and pattern matching. Like functional programming languages, Scala is strongly typed. Type inference is a characteristic feature for functional programming languages. Scala also has a type inference system adapted to object-orientation, making the syntax concise so that explicit types do not have to be provided in many cases. Functional concepts make it possible to build components quickly and concisely. Combining functional and object-oriented paradigms has introduced new possibilities of developing software, both conceptually and structurally.

Scala runs on the Java Virtual Machine (JVM), which means that Scala

is byte code compatible with Java. Java code can be invoked from Scala, and Scala code can be invoked from Java, making a seamless integration of the two languages. The JVM platform is widely used in industry, making the use of Scala as implementation language for CPNs beneficial in cases where a model is going to be used in the implementation of a system.

1.3 Scope and Objectives

The use of a functional programming language for CPNs is natural because of the mathematical foundation of Petri Nets. A drawback is that SML is not a widely used programming language, which can be an obstacle in the process of learning to model with CPNs. Scala is one of the most evident programming languages in the emerging trend of combining functional and object-oriented paradigms, and has a syntax that is more familiar to programmers with an object-oriented background. A question that arises is whether CPNs could benefit from integrating object-orientation, complementing the functional foundation. In this thesis, we explore the possible tradeoffs the combination of functional and object oriented paradigms, as provided by Scala, will give in the context of CPNs.

Prior to the work that has been done, the following objectives and research questions (explored in more detail below) were identified:

- Scala as inscription language for CPN:
  - How to achieve integrated use of both functional and object-oriented paradigms in CPN models?
  - How to support parameterised and flexible CPN modules?
  - How does it affect model design?
- Scala as implementation language for CPN:
  - How can the pattern matching capabilities of Scala be used to implement enabling inference?
  - How can the generated model-dependent simulator code be reduced?
  - What is the model execution performance?

Scala as inscription language concerns how Scala is integrated in the modelling environment, including how object-oriented paradigms can be integrated. Further, allow to support *flexible* CPN modules, hence, how the

coupling of modules can be reduced. Scala as implementation language is concerned with the underlying implementation of the Scala simulator.

In the simulator implementation of CPN Tools, the enabling inference and functionality to make a transition occur, are generated uniquely for each transition. The approach in the Scala simulator is based on making this functionality generic, so that the amount of generated code will be reduced, i.e., increasing the *model-independent* functionality so that *model-dependent* code is reduced. The two objectives given above are tightly coupled, as a fundamental design goal has been to avoid extending Scala with new syntax.

The work has been based on design science research [7], where investigation of how Scala can be applied to the formalism of CPNs has been conducted, based on making an implementation of a simulator for CPNs in Scala. The design of Scala as inscription language and the implementation of the simulator has been conducted using an empirical approach. Scala has been studied in detail to find applicable language constructs giving an intuitive modelling formalism that conforms to the formal specification, and key algorithms of CPNs. This process has been incremental, evolving the design and simulator implementation simultaneously to ensure that the design reflects the simulator. In this way, the simulator serves as a proof of concept for the design, and it addresses how algorithms of CPNs can be implemented. The design has evolved from being similar to the design based on CPN ML, to a final design where new modelling constructs have been introduced. The Scala simulator has also been integrated with CPN Tools, so that models can be constructed using Scala as inscription language, and further be transformed to Scala code that can be executed by the simulator developed in this thesis. This allows assessment of Scala as inscription and implementation language in a uniform manner.

A CPN model of the Babel routing protocol [11] has been constructed, serving as an industrial-sized case study. The Babel CPN model has been used to conduct both qualitative and quantitative assessment between the SML simulator of CPN Tools and the new Scala simulator. This shows that the chosen inscription design and the implementation works in practice. We have achieved a design that greatly benefits from the object-oriented capabilities of Scala, both regarding inscriptions and the Scala simulator implementation. The performance of the Scala simulator has been assessed against the simulator in CPN Tools based on the Babel CPN model. However, performance optimisations have not been of high priority, as the Scala simulator has mainly been implemented to support interactive simulation, whereas the SML simulator is targeting automatic simulation.

We have focused on the core-functionality of CPNs. Functionality such

as state space analysis and timed nets are outside the scope of this thesis.

1.4 Related Work

Renew [24] is a modelling tool based on the formalism of high-level Petri Nets, which is implemented in the object-oriented programming language Java. SML and Java have in some sense fragmented features regarding what is needed for CPN modelling. Using a functional language is natural because of the mathematical foundation of Petri Nets, but object-orientation in Java also gives useful constructs. In CPN Tools, SML is extended with colour sets, allowing definition of types in a more concise manner than what is possible in SML itself, while in Renew, Java's primitive types and classes are used directly. Allowing the use of classes in Renew introduces the possibility to scope methods in classes. Renew also introduces side effects, which should be used with care due to the *locality principle* of Petri Nets (we will discuss the locality principle in more detail in Chap. 5). Since Java is not functional, it lacks essential features required to adapt the CPN formalism. Therefore, Renew extends Java with tuples and functional lists that support pattern matching. Scala complements the features found in SML and Java into one language, making it an interesting candidate for CPNs.

CPNs are applicable for modelling concurrency. However, this is not exploited in the implementation of the simulator in CPN Tools. A concurrent simulator for low-level Petri Nets has been implemented in Scala, using the Actor model [10]. The Actor model is part of the Scala library [21, Chapter 32], and constitutes a framework for implementing concurrency. The Actor framework is high-level, so that managing of threads and locks are avoided, making it robust with regard to dead-locks and race conditions.

HCPN [30] is a variant of CPN using the functional programming language Haskell as inscription and implementation language. Their intention is to get a broader awareness of CPNs amongst communities in functional programming, by providing a simple mapping of Petri nets to functional programming languages. Thus, providing a guide of how CPNs can be implemented in any functional programming language.

1.5 Thesis Overview

Below we give an overview of the structure of the thesis.

Chapter 2 - Babel and Modelling with CPNs introduces a CPN model of the Babel routing protocol, which is used for assessment of Scala as inscription and implementation language. We also introduce modelling with CPNs in this chapter.

Chapter 3 - Design of Scala as Inscription Language addresses how Scala can be used as inscription language for CPNs, evolving from an initial design similar to the use of CPN ML, to a final design adapted to Scala.

Chapter 4 - Enabling Inference introduces the enabling inference algorithm, which is the main mechanism in CPN simulation. In this context, we also introduce the basic formal semantics of CPNs, and address the pattern matching capabilities of Scala.

Chapter 5 - Simulator Implementation describes the implementation of the Scala simulator, and how it is integrated with CPN Tools. Examples of code that is generated from a CPN model are also given.

Chapter 6 - Evaluation describes how the implementation of code generation and the simulator have been tested. We assess the performance of the Scala simulator compared to the SML simulator in CPN Tools. The amount of generated code for a CPN model for the Scala simulator and the SML simulator is compared. Finally, we assess the usability, i.e., how the model design has been affected by using Scala as inscription language.

Chapter 7 - Conclusions and Future Work concludes this master's thesis with a summary, an overview of the main results, and suggestions for future work.

Prior knowledge of CPNs, Scala and routing protocols, is not required to read this thesis. However, it is expected that the reader is familiar with the basics of object-oriented and functional paradigms. There is a lot of literature about these paradigms, for instance the book "Programming Languages" by Kent Lee [20], where both object-oriented and functional programming are introduced.

Chapter 2

Babel and Modelling with CPNs

Modelling of network protocols is an important application domain for CPNs, and models of many protocols have earlier been constructed using CPNs [19]. These are complex systems, where textual specifications may contain ambiguities that are hard to detect prior to implementation. A CPN model of a protocol requires soundness and completeness, where ambiguities and lack of details in the specification are enforced to be addressed.

A CPN model of the Babel routing protocol [11] has been constructed to serve as an industrial-sized case study for this master's thesis. The fundamental operations of the Babel protocol will be described briefly in Sect. 2.1. In Sect. 2.2, we present the Babel CPN model, and we introduce the basic concepts of CPNs. Finally, we end this chapter with a short summary in Sect. 2.3.

2.1 The Babel Protocol

The specification [11] that the Babel CPN model has been based on is published for examination, experimental implementation, and evaluation. The CPN model is simplified, and reflects only the conceptual behaviour of the protocol. Nonetheless, deviations and simplifications from [11] will be stated. The focus in this thesis is to apply Scala as inscription and implementation language for CPNs. Hence, correctness and verification of the Babel CPN model is not considered essential. The rest of this section will be used to explain briefly the fundamental operations of the Babel protocol, based on the specification. In this context, we use a simple example, that

also will be used further in the presentation of the Babel CPN model in the next section.

Babel is a hybrid IPv4 and IPv6 loop-avoiding *distance-vector* routing protocol designed to be robust and efficient in *mesh* networks. According to [29], mesh networks is a class of networks that can handle a mix of wired nodes, interconnected with dynamic mobile nodes. In distance-vector routing, each node maintains information about their neighbours and their known routes. A node does not know the entire path to a source, only the distance to the source and which neighbour who is on the path.

Every Babel node has a router-id that is unique across the routing domain, as well as a sequence number, introduced in [28] to avoid a well known issue for distance-vector routing; the formation of routing loops. This problem may arise when a node A advertises a route for a source S to a node B, then B cannot know whether B itself is in the path.

A node may typically have several sets of neighbours separated on different interfaces, a common situation for wireless nodes with multiple radios. However, the CPN model is restricted to a single neighbouring interface.

Figure 2.1 depicts the topology of a mesh network consisting of five mobile nodes numbered from 1 to 5, representing the router-ids. An arc between a node A and a node B means that B is *reachable* from A and that A is reachable from B. Hence, in this example, it is assumed that links are bidirectional.

Neighbours are discovered and maintained by periodically sending *Hello* messages. When a node has acquired a new neighbour, it sends a *wildcard route request*. A node may send a route request (requesting the route to a specific source), or a wildcard request (requesting a full routing table dump,

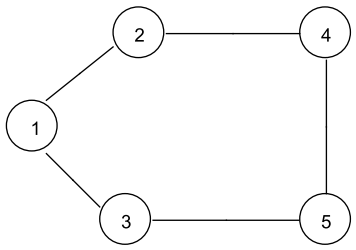


Figure 2.1: Example of a mesh network topology, represented by five mobile nodes.

i.e., all selected routes and newly retracted routes for all known sources).

A node also sends *I Heard You* (IHU) messages periodically to neighbours it recently has received Hello messages from in order to establish bidirectional reachability. Using information derived from the Hello and IHU messages, a node computes the link-cost to a neighbour. For simplicity in this example and the CPN model, a link from a node to a neighbour has either the link cost 1 or ∞, depending on the information derived from Hello and IHU messages. This means that the number of hops from a source to a destination depicts the distance (cost) of the route. Initially, we assume that all the nodes in the example know of their reachable neighbours and that the links are stable, hence, link cost equals 1 for all neighbouring links.

As well as advertising its routes (full route table dump) to its neighbours by sending *periodic updates*, a node sends *triggered updates*. Babel heavily relies on triggered updates. A node sends triggered updates to its neighbours when it detects significant topology changes, either from link cost changes to a neighbour or due to a received update. Triggered updates help increase the convergence speed, i.e. the time used to recover from a topology change. Additionally, nodes maintain alternative inferior routes to quickly recover when routes fail.

Focusing on node 1 as the source, the nodes will have the installed data for the source as listed in Tab. 2.1, after neighbour acquisition and route advertisement. Notice that the distance to the source does not include the link cost to the neighbour who is the *next hop* in the route.

A *feasibity distance*, $FD(seqno, metric)$, is applied per source, and is a pair, where the first component represents the lowest metric ever advertised by the node for this source, and the second component represents the source's sequence number that was given from the neighbour advertising the route. When a node receives a route update for a source with sequence number $seqno'$, and metric $metric'$, the feasibility condition is applied to validate that the update will not form a routing loop. The update is accepted if $seqno' > seqno$ or if $seqno' = seqno$ and $metric' < metric$. Updates repre-

Node	Source	Distance	Next Hop	Feasibility Distance
2	1	0	1	(1, 1)
3	1	0	1	(1, 1)
4	1	1	2	(2, 1)
5	1	1	3	(2, 1)

Table 2.1: Installed data for nodes with node 1 as source.

senting route retractions are also accepted, that is, when $metric' = \infty$. An incoming update for a source which the receiving node has not advertised is accepted since there is no feasibility distance for the source.

Consider Fig. 2.1 and suppose that the link between node 2 and node 4 breaks. We assume that node 4 does not have an inferior route from node 5. Eventually, node 2 and node 4 will set the link cost to each other to ∞. When node 4 detects the broken link, it realises that it has no routes to node 1. A node may send explicit route requests to its neighbours at any time, to enquire about their selected routes, but node 4 would not accept an update for node 1 as source from node 5 since the update $(metric = 2, seqno = 1)$ would not be accepted by the feasibility condition.

Figure 2.2 shows a *message sequence chart* of the flow of messages triggered by node 4, to obtain a feasible route to node 1 after the link between node 2 and node 4 breaks. Solid arrows represent multicast transmission while dashed arrows represent unicast transmission.

Node 4 sends a route update with distance equal to ∞, representing a route retraction, to a multicast destination. It also sends a sequence number request to a multicast destination, destined for the lost source (node 1). Node 5 is currently the only reachable node from node 4. Node 5 ignores the route retraction since it has a different route to node 1. The sequence number request is forwarded by node 5 to node 3, since node 3 is the next hop on node 5's route to node 1. Node 3 forwards the sequence number request to node 1, which in turn increases its sequence number and sends a route update in reply to node 3. The route update is forwarded to node 5, and

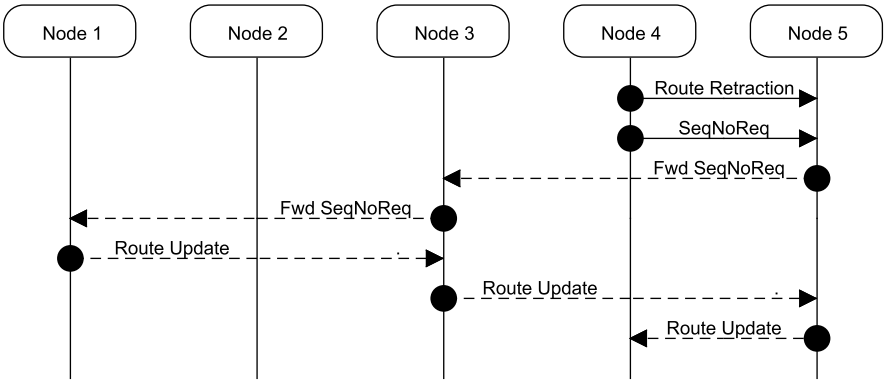


Figure 2.2: Message exchange scenario after link failure.

finally to node 4. Node 4 accepts the route update ($metric = 2, seqno = 2$) as the sequence number is greater than the sequence number in node 4's current feasibility distance for node 1, ($metric = 2, seqno = 1$).

2.2 The Babel CPN Model

A CPN model can be organised into hierarchical modules giving a more modularised and transparent organisation of the model. The organisation into modules can be related to the way computer software can be divided into modules or packages. The Babel CPN model is organised into 19 modules, spread over five abstraction levels. The module hierarchy is depicted in Fig. 2.3. The System node represents the *top-level*, or *prime* module. Directed arcs lead to *submodules* of originators. For instance, the BabelProtocol module is a submodule of the System module. The model is divided into two main parts; the BabelProtocol and the Link. The BabelProtocol module and its submodules model the operations of the protocol, including reception and processing of messages, performing route selection, generating updates, timeouts, and sending of triggered and periodic messages to other nodes in the network. The Link module and its submodules model the environment that protocol operates in. This includes transmission of packets over a network link and changes in the topology, simulating mobility of network nodes. The BabelProtocol module and the Link module do not share any information. Whether two nodes are in range of each other is determined by the Link layer. A node must determine if it has stable links to other nodes through the sending and reception of Hello and IHU messages.

Only selected parts of the protocol model, including all abstraction levels, will be presented in detail in the following sections. Appendix A contains instructions of how to find the complete Babel CPN model.

A node's knowledge about its own data, topology, and neighbouring interaction, is kept in a number of data structures that are maintained through reception of packets and timeouts:

Sequence number. The main sequence number is included in updates originating from the node, while a hello sequence number is included in Hello messages.

The Neighbour Table. The neighbour table contains the neighbouring nodes that a node knows of.

The Source Table. The source table is used to record feasibility distances for sources a node has advertised.

The Route Table. The route table contains routes known to a node.

The Pending Req Table. The table of pending requests maintains requests that either have been sent by the current node, or forwarded from others, and where a response has not yet been received.

2.2.1 The System Module

Figure 2.4 shows the top-level System module of the Babel CPN model. It connects the BabelProtocol module with the Link module, enabling the two submodules to interact. A submodule is represented as a *substitution transition*, drawn as a double lined rectangle with a name tag specifying the name of the corresponding submodule. The BabelProtocol and Link submodules are represented by substitution transitions named correspondingly. The two substitution transitions are connected through the two places named BabelToLink and LinktoBabel, drawn as ellipses.

A place can be marked with tokens, each of which may contain a data

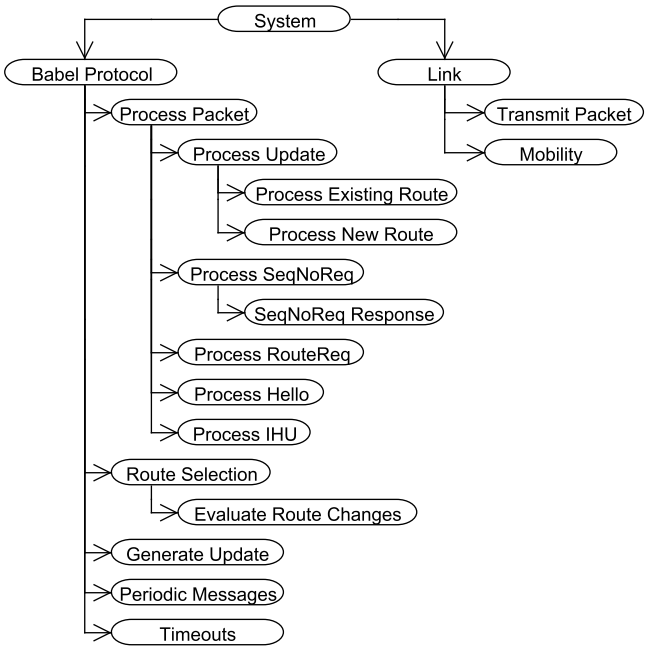


Figure 2.3: Module hierarchy for the Babel CPN model.

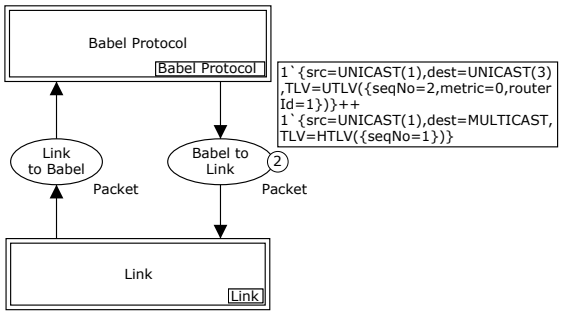


Figure 2.4: The System module of the Babel CPN model.

value, called a *token colour*. The type of tokens a place can hold is determined by a colour set inscription next to the place. The two places in the System module have the colour set `Packet`. The tokens on a place represent its state, and is called the *marking* of the place. The place BabelToLink has the following marking:

```
1‘{src=UNICAST(1),dest=UNICAST(3),
  TLV=UTLV({seqNo=2,metric=0,routerId=1})} ++
1‘{src=UNICAST(1),dest=MULTICAST,TLV=HTLV({seqNo=1})}
```

The first token in the marking represents the route update sent from node 1 to node 3, in the 5th step in Fig. 2.2. The second token represents a Hello message from node 1 to a multicast destination. A place is in general marked by a multiset of tokens. For specifying multisets, two operators are used: ‘ and ++. The ‘ operator takes the *coefficient* as the left operand, which is a non-negative integer that denotes the number of instances of the token given as right operand. The ++ operator evaluates to the sum of two multisets. The small circle with a number next to the marking specifies the number of tokens in the current marking of the place. The number of tokens and token colours on each place represent together the state of the system, which is referred to as the *current marking* of the system.

A place connected to a substitution transition is called a *socket place*, and is related to a *port place* in the submodule. Port places can either be input, output or input/output, and denote the *interface* of the module. A port place and a socket place that are related are conceptually the same place, which means that they always share the same marking, and hence, the same colour set.

Tokens on the place BabelToLink in Fig. 2.4 represent a packet sent from a source to a destination. These tokens are also present on an output place

in the BabelProtocol submodule and an input place in the Link submodule. In this way, packets can be sent from the protocol layer to the link layer, and similarly in the opposite direction through the place LinktoBabel. The flow of tokens from a socket place to a substitution transition is depicted by the direction of the arc.

The `Packet` colour set along with colour sets used by `Packet` are defined in Lst. 2.1. Colour sets are defined as `colset <name> = <definition>`, where the definition is constructed of basic types or other colour sets. The basic types are inherited from SML and includes `string`, `int`, `bool` and `unit`. A colour set can be constructed as a product, a range, a record or a union type. Examples of record, range and union are shown in Lst. 2.1 and examples of product are given in Lst. 2.2 in the next section.

Listing 2.1: Colour set declarations for nodes and packets.

```
colset Nodes      = int with 1 .. N; (* range from 1 to N *)
colset Address    = union UNICAST : Nodes + MULTICAST;

(* --- TLVs --- *)
colset HelloTLV   = record seqNo  : INT;
colset IHUTLV     = record rxcost : INT;
colset UpdateTLV  = record seqNo  : INT *
                           metric  : INT *
                           routerId : INT;
colset SeqNoReqTLV = record seqNo : INT *
                           hopCount : INT *
                           routerId : INT;
colset RouteReqTLV = record routerId : INT;
(* ------------ *)
colset TLV        = union HTLV : HelloTLV  +
                          ITLV  : IHUTLV    +
                          UTLV  : UpdateTLV  +
                          RRTLV : RouteReqTLV +
                          STLV  : SeqNoReqTLV;

colset Packet     = record src : Address *
                           dest : Address *
                           TLV  : TLV;
```

The `Packet` colour set is a simplified abstract representation of a *User Datagram Protocol* (UDP) datagram, and is defined as a record of a source (`src`), a destination (`dest`), and a *Type-Length-Value* (`TLV`). The `src` and `dest` is of colour set `Address`, defined to be either a `UNICAST(Nodes)` or

a MULTICAST, where Nodes is some integer within the range of 1 to some integer N, representing a node's router-id. The model is simplified so that full IP addresses with address encoding and prefix are not explicitly modelled. The absence of prefix in the model simplifies some cases concerning several nodes originating the same prefix, and overlapping prefixes. The model is constructed with the assumption that every node is a unique destination.

In practice, the source of a packet is always a unicast, while the destination can be a unicast if the packet is destined for a specific node, or multicast if the packet should be sent to all reachable neighbours. TLVs are elements that in this context are equivalent to what have previously been described as messages.

The TLV colour set is a union type that includes the different TLVs that Babel uses. Some TLVs that do not affect the conceptual behaviour of the protocol are not modelled. For the same reason, the model also abstracts from some data values contained in the TLVs. The model also abstracts from the possibility to aggregate multiple TLVs in a single packet, hence, a Packet only contains one single TLV.

2.2.2 The Babel Module

Figure 2.5 shows the BabelProtocol module of the model. This module has one input port place LinkToBabel, and one output port place BabelToLink that are associated with the accordingly named socket places in the System module in Fig. 2.4. A port place can be recognised by the tag In, Out or I/O, corresponding to input, output and input/output, respectively. The current marking of Fig. 2.5 includes the same tokens as the tokens in the marking of BabeltoLink in Fig. 2.4, where the packet containing an Update TLV has been transmitted over the network link and is now present on the place LinktoBabel.

The module has five substitution transitions which represent the correspondingly named submodules: ProcessPacket, RouteSelection, GenerateUpdate, PeriodicMessages and Timeouts. Each place in the module is connected to a substitution transition, which makes all places sockets for the substitution transition they are connected to. Colour sets used by the places in this module are defined in Lst. 2.2.

Incoming packets, represented as tokens on LinkToBabel are processed in the ProcessPacket submodule. The ProcessPacket module connects a number of submodules that handle packets based on the type of TLVs in each packet. After a node has processed a packet, tokens may be transmitted to the different socket places connected to the ProcessPacket substitution

Listing 2.2: Colour set declarations used in the BabelProtocol module.

```
colset RouteTable = list RouteTableEntry;

colset NODExROUTES = product Nodes * RouteTable;

colset NODExDESTxROUTE = product Nodes *
                         Address *
                         RouteTableEntry;

colset NODExROUTESRCxROUTES = product INT * INT * RouteTable;
```

transition.

When an Update TLV has been processed and caused changes in a node's routes, a token is added to the RouteUpdate socket place to trigger a new route selection. This place has the colour set `NODExROUTESRCxROUTES` where tokens represent the current node, the affected source, and its old routes, which is used for comparison with the new routes in the RouteSelection module. When an Update TLV has caused a new route to be added, or a link to a neighbour has changed due to reception of Hello or IHU TLVs, a token is added to LinkChange/NewRoute to trigger a new route selection.

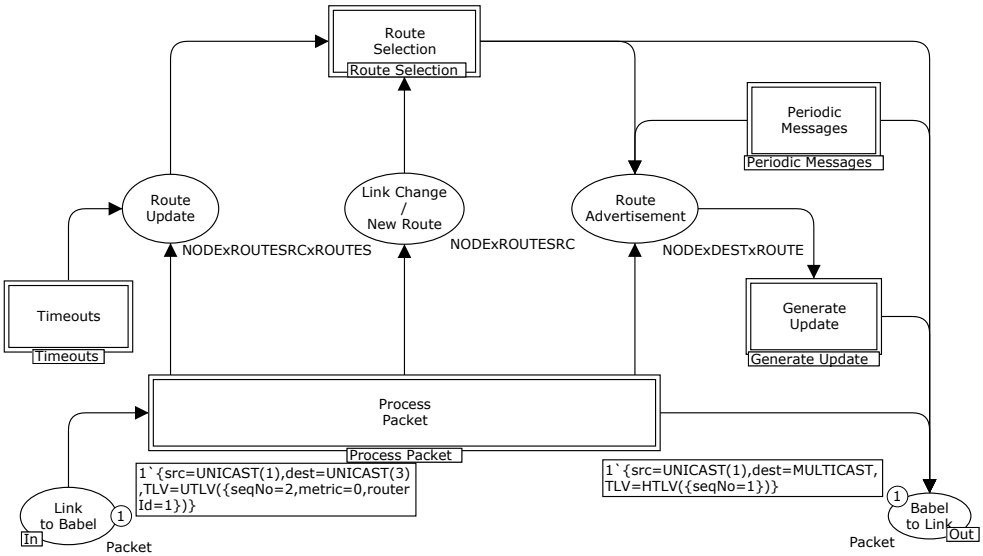


Figure 2.5: The BabelProtocol module of the Babel CPN model.

This place has the colour set `NODExROUTESRC` where tokens represent the node that has discovered changes, and its affected source.

Babel nodes advertise their selected routes, or route retractions for unknown routes. This is done both regularly and when triggered by processed packets. When a route is to be advertised, a token is added to RouteAdvertisement. Tokens can be added to this place from three submodules:

- The ProcessPacket submodule in case of a received sequence number request or a route request.
- The RouteSelection submodule in case of a route change.
- The PeriodicMessages submodule in case of a periodic message.

RouteAdvertisement has the colour set `NODExDESTxROUTE`, where tokens represent the node originating the advertisement, a destination which is either a unicast to a single node or a multicast, and the selected or retracted route of the originator. Tokens at RouteAdvertisement trigger generation of updates in the GenerateUpdate submodule, which in turn results in tokens being added to BabeltoLink, representing packets.

Babel nodes have several timers used for sending periodic messages and timeouts. A node sends Update TLVs, Hello TLVs, IHU TLVs, SNR TLVs (sequence number request) and RR TLVs (route request) based on timers. This is modelled in the PeriodicMessages submodule. A node may also expire routes, sources and pending requests, which is modelled in the Timeouts submodule. Timers are not explicitly modelled, instead, any of the events caused by timers may happen at any time during simulation. This does not restrict the behaviour of the protocol, rather the opposite, as it covers a larger set of execution patterns.

2.2.3 The Process Packet Module

Handling of the different types of TLVs are done separately in submodules of the ProcessPacket module, shown in Fig. 2.6. Colour sets used in this module are defined in Lst. 2.3. This module has two ordinary (non-substitution) transitions, drawn as single lined rectangles, representing events that can occur. Simulation of a CPN consists of a sequence of occurrences of transitions, called *occurrence sequence*, changing the marking of places. An occurrence can also be called a *simulation step*.

Before a transition can occur, it must be *enabled*. Whether a transition is enabled, is determined by *arc expressions* on input arcs, and the guard of the transition (if present). Arc expressions are built from colour sets, constants,

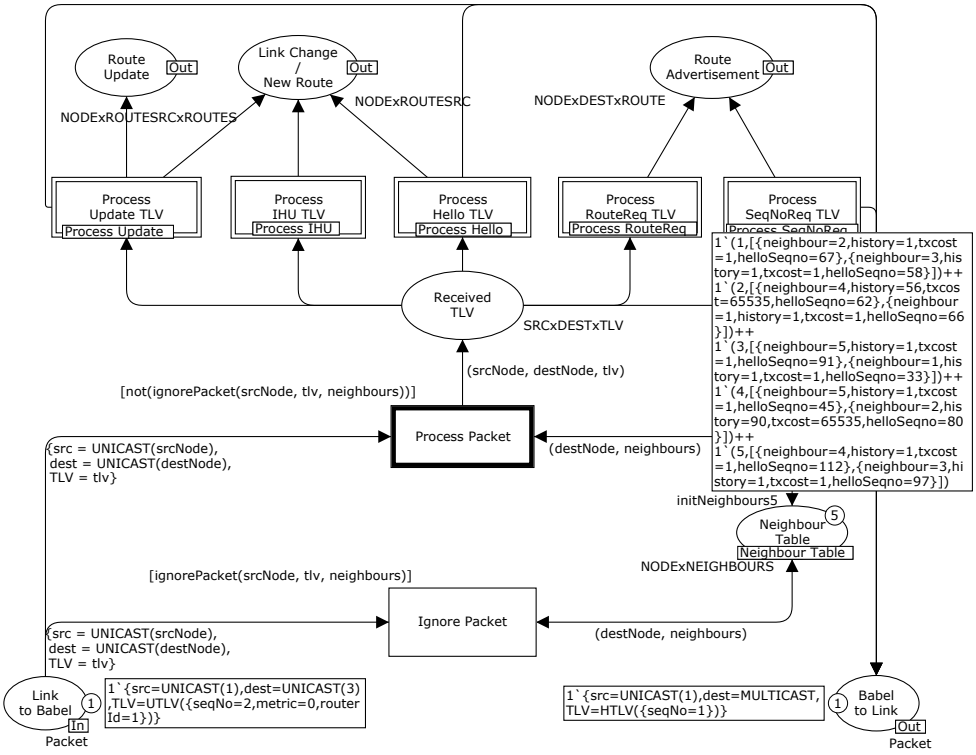


Figure 2.6: The ProcessPacket module of the Babel CPN model.

Listing 2.3: Colour set declarations used in the ProcessPacket module.

```
colset SRCxDESTxTLV = product RouterId * RouterId * TLV;

colset NeighbourTableEntry = record neighbour : Nodes *
                                    history   : INT *
                                    txcost    : INT *
                                    helloSeqno : SeqNo;

colset NeighbourTable = list NeighbourTableEntry;

colset NODExNEIGHBOURS = product Nodes * NeighbourTable;

colset RouteTableEntry = record source:RouterId * neighbour:RouterId *
  metric:INT * seqNo:SeqNo * selected:BOOL;
colset RouteTable = list RouteTableEntry;

colset NODExDESTxROUTE = product Nodes * Address * RouteTableEntry;
colset NODExROUTESRCxROUTES = product INT * INT * RouteTable;
colset NODExROUTESRC = product INT * INT;
```

variables, expressions and functions. For a transition to be enabled, variables in arc expressions on input arcs must be bound to values in a *binding*, so that evaluation of arc expressions at input arcs in the binding results in a multiset of tokens that can be covered by the marking on the connected place. However, a simplification allows omission of the multiset coefficient when the expression constitutes a single token. A transition and a binding of its variables is called a *binding element*.

Binding of variables is based on pattern matching. As an example, consider Fig. 2.6. The arc expression on the arc from LinkToBabel to ProcessPacket is a record pattern, matching a `Packet` where the fields `src` and `dest` must match `UNICAST` addresses. The pattern contains the three variables `srcNode`, `destNode` and `tlv`. The arc expression on the arc from NeighbourTable to ProcessPacket is a product pattern with the two variables `destNode` and `neighbours`.

A binding for a transition is of the form $\langle v_1 = c_1, v_2 = c_2, \ldots, v_n = c_n \rangle$, where n is the number of variables of the transition, and c_i is the value bound to v_i. ProcessPacket is enabled in the marking shown in Fig. 2.6, indicated by the thick border, with the following binding:

```
<neighbours = [{neighbour=5, history=1, txcost=1,helloSeqNo=91},
    {neighbour=1,history=1, txcost=1,helloSeqNo=33}],
  srcNode = 1,
  destNode = 3,
  tlv = UTLV({seqNo=2, metric=0, routerId=1})>
```

This indicates that the guard of ProcessPacket (`[not(ignorePacket(srcNode, tlv, neighbours))]`) is satisfied in this binding. The guard must be a boolean expression, surrounded by square brackets, which is the SML syntax for a list. Guards can also be in *conjunctive* form, by having two or more boolean expressions in the list. For a transition to be enabled, all *guard conjuncts* must evaluate to `true`. IgnorePacket is not enabled because the guard evaluates to `false` in the binding given above. The function `ignorePacket` used in the guards of ProcessPacket and IgnorePacket evaluates to true if packets are received from unknown nodes, unless the packet contains a Hello TLV.

When a transition occurs, tokens are removed from its input places and tokens are added to its output places, based on evaluating the arc expressions in the chosen binding. After ProcessPacket has occurred, a token is added to the place ReceivedTLV where the value is the product of the values that were bound to the variables `srcNode`, `destNode` and `tlv`. This will enable a transition in one of the submodules for processing of TLVs, depending on the type of the received TLV. In this case, an Update TLV has been received, which will be processed in the ProcessUpdateTLV submodule.

Variables used in a model must be declared with a name and a colour set. Listing 2.4 shows the definitions of the variables used in the ProcessPacket module.

Listing 2.4: Variable declarations used in the ProcessPacket module.

```
var srcNode    : Nodes;
var destNode   : Nodes;
var tlv        : TLV;
var neighbours : NeighbourTable;
```

The place NeighbourTable in Fig. 2.6 is a *fusion place* which is member of the *fusion set* with the corresponding name, shown from the tag on the place. All places that are members of a fusion set represent the same *compound place*. NeighbourTable has the colour set `NODExNEIGHBOURS`, representing the router-id of a node and information about its neighbours, maintained in a

list with colour set `NeighbourTable`, shown from Lst. 2.3. By using a fusion set, neighbour nodes can be accessed and modified in different modules as all members of a fusion set share the same marking. NeighbourTable has an inscription `initNeighbours5`, denoting the *initial marking* of the place. This is the marking set for the place prior to simulation, in this case it is set to the value `initNeighbours5`, given in Lst. 2.5. Note that the marking of NeighbourTable in preceding modules will be hidden as it is the same as in Fig. 2.6.

Listing 2.5: The value given as initial marking for NeighbourTable.

```
val initNeighbours5 =
  1‘(1,[{neighbour=2,history=1,txcost=1,helloSeqno=0},{neighbour=3,history
      =1,txcost=1,helloSeqno=0}])++
  1‘(2,[{neighbour=4,history=1,txcost=1,helloSeqno=0},{neighbour=1,history
      =1,txcost=1,helloSeqno=0}])++
  1‘(3,[{neighbour=5,history=1,txcost=1,helloSeqno=0},{neighbour=1,history
      =1,txcost=1,helloSeqno=0}])++
  1‘(4,[{neighbour=5,history=1,txcost=1,helloSeqno=0},{neighbour=2,history
      =1,txcost=1,helloSeqno=0}])++
  1‘(5,[{neighbour=4,history=1,txcost=1,helloSeqno=0},{neighbour=3,history
      =1,txcost=1,helloSeqno=0}])
```

2.2.4 Process Update and Process Existing Route Modules

Figure 2.7 shows the ProcessUpdate module with the marking obtained after the transition ProcessPacket in the ProcessPacket module has occurred in the marking of Fig. 2.6. This module has two submodules ProcessExistingRoute and ProcessNewRoute. ProcessExistingRoute handles updates of previously known routes, while ProcessNewRoute handles route advertisements of new routes. A route is considered unique based on the advertising neighbour and the route source.

Figure 2.8 shows the ProcessExistingRoute module and Lst. 2.6 shows declarations used in this module. The arc from RecievedTLV to ProcessExistingRoute has the expression `(srcNode, destNode, UTLV updateTlv)`, which only matches tokens on the place ReceivedTLV where the TLV is an Update TLV. The guard on ProcessExistingRoute ensures that the receiving node already has a route from the neighbour to the advertised source. The place RouteTable is a member of a fusion set with the corresponding name, and has the colour set `NODExROUTES`, holding tokens with a node's router-id and

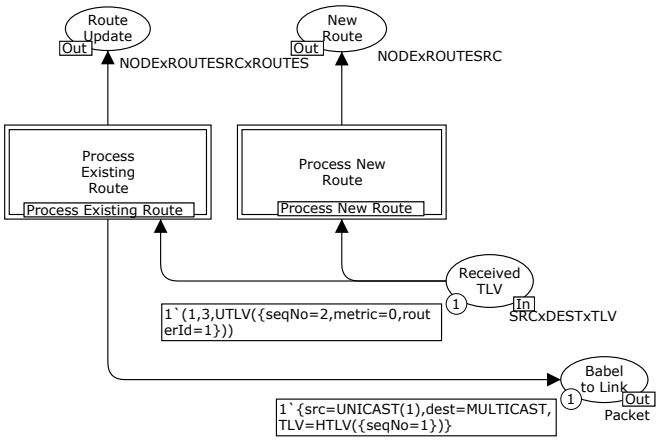


Figure 2.7: The ProcessUpdate module of the Babel CPN model.

its routes, similar as for the NeighbourTable fusion set. The SourceTable fusion set with the colour set `NODExSOURCES` is used the same way, holding a token for each node which contains a list of the node's feasibility distances.

Listing 2.6: Declarations used in the ProcessExistingRoute module.

```
colset NODExROUTES = product Nodes * RouteTable;

colset SourceTableEntry = record routerId:RouterId * seqNo:SeqNo * metric:
    INT;
colset SourceTable = list SourceTableEntry;
colset NODExSOURCES = product Nodes * SourceTable;

colset SRCxDESTxUTLV = product RouterId * RouterId * UpdateTLV;

var updateTlv : UpdateTLV;
var node : Nodes;
var sources : SourceTable;
var routes : RouteTable;
```

The marking in Fig. 2.8 is the marking resulting after the occurrence of ProcessExistingRoute. A token has been added to UpdatedRoute over the colour set `SRCxDESTxUTLV`, representing the update on a route from an existing neighbour, ready to be processed.

If the received update is unfeasible, the transition IgnoreUnfeasibleUpdate will be enabled. An update is feasible if the feasibility condition introduced

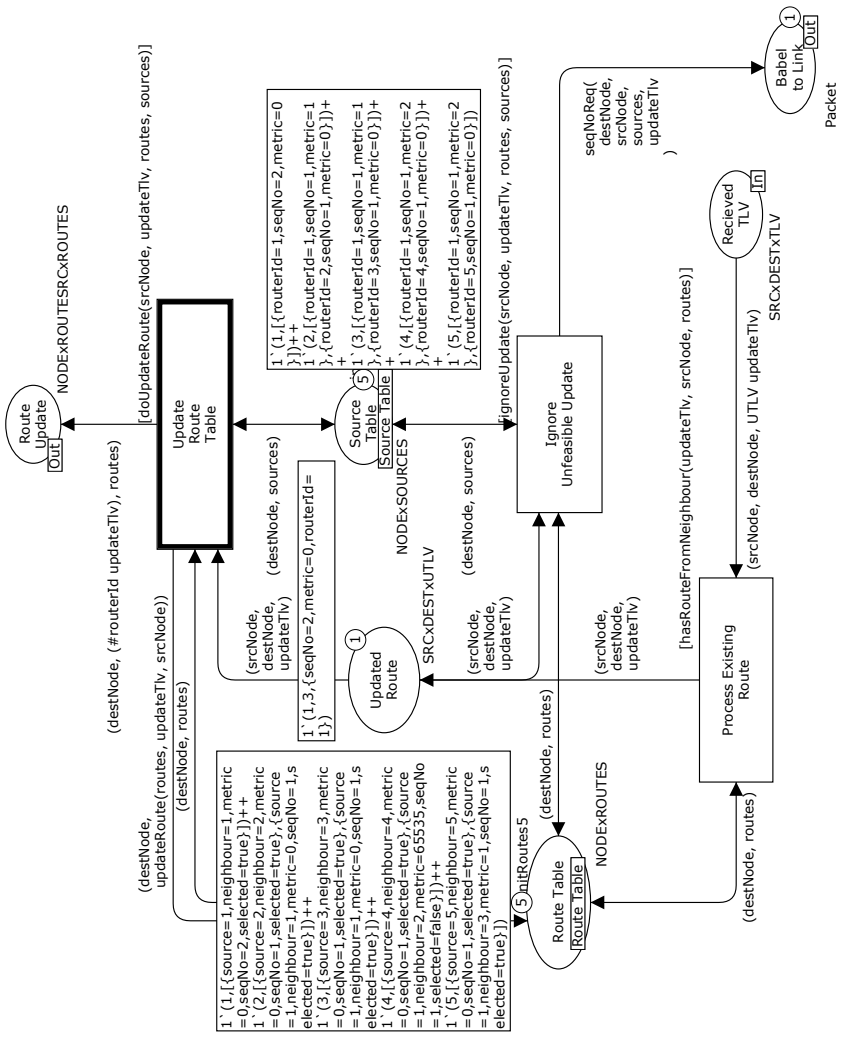


Figure 2.8: The ProcessExistingRoute module of the Babel CPN model.

previously is satisfied, or if the metric of the update is infinite, representing a route retraction. When an update is ignored, no changes are done to the receiving node's routes, but a token is added to BabeltoLink, representing the sending of a sequence number request destined for the originator of the unfeasible update.

If the received update is feasible (which is the case in the marking of Fig. 2.8), the UpdateRouteTable transition will be enabled. When this transition occurs, the receiving node's route table will be updated by the function call of `updateRoute` in the arc expression on the arc from UpdateRouteTable to RouteTable. A token is also added to RouteUpdate with the colour set `NODExROUTESRCxROUTES`, which is a product of the current node's router-id, the source of the route, and the routes before they got updated. This token will trigger a new route selection in the RouteSelection module.

The function `updateRoute` shown in Lst. 2.7, takes three parameters; a route table (`routes`), an Update TLV where a record pattern is used to access the `routerId`, `metric` and `seqNo` of the Update TLV, and the router-id of the neighbour sending the update (`neigh`). The function uses a *let-in-end* expression to hold the local value `updateElement`. This value is set using a SML library function `List.find`, which searches for an element matching the predicate given as the first argument, in the list given as the second argument. In this case, the predicate specifies an entry in `routes`, where `neighbour` matches the sender of the update, and the `source` matches the source of the received Update TLV. The result of `List.find` is of type `Option`, where the value of the result is either `SOME <el>` if an element `el` that satisfies the predicate is found, or `NONE` if no element satisfies the predicate. The `updateElement` is matched against the two patterns `SOME el` and `NONE` using a *case* expression. If an element is found, the element is filtered from `routes` using the SML library function `List.filter` and concatenated with a list containing the updated route. In case of a route retraction of a selected route, the route is unselected. Note that the updated element is appended to the end of the list. This is to simulate a queue, representing resetting of the timer for the route. If no element is found, no changes are done to `routes`.

2.2.5 The Route Selection Module

When the route update is done, a token is added to the input port place RouteUpdate in the RouteSelection module, shown in Fig. 2.9. This is the same token as the one added to the place RouteUpdate in the ProcessExistingRoute module after occurrence of UpdateRouteTable in the marking of

Listing 2.7: The updateRoute function.

```
fun updateRoute(routes: RouteTable, {routerId, metric, seqNo, ...}:
    UpdateTLV, neigh: RouterId) =
let
    val updateElement =
        List.find(fn el => (#neighbour el) = neigh andalso
                           (#source el) = routerId) routes
in case updateElement of
    SOME el =>
        (List.filter(fn el => (#neighbour el) <> neigh andalso
                              (#source el) <> routerId) routes)
                              ++
                              [{
                                source=routerId,
                                neighbour=neigh,
                                metric=metric,
                                seqNo=seqNo,
                                selected=(#selected el) andalso metric <
                                    INFINITE
                              }]
    | NONE => routes
end;
```

Fig. 2.8. Listing 2.8 shows declarations used in the RouteSelection module.

The RouteSelection module also has another input port place, LinkChange-/NewRoute, which triggers a route selection in a similar manner. The difference is that when a route update has occurred, the old routes have to be preserved to be able to conduct a proper evaluation of the changes.

Listing 2.8: Declarations used in the RouteSelection module.

```
colset RouteChanges = record node: Nodes *
                      oldRoutes: RouteTable *
                      newRoutes: RouteTable;

var newRoutes : RouteTable;
var oldRoutes : RouteTable;
var routeSrc : Nodes;
```

When the enabled transition RouteSelectionforRouteUpdate occurs, a new route selection is done, resulting in an update of the current node's routes in the marking of the place RouteTable. Further, a token is added to the place

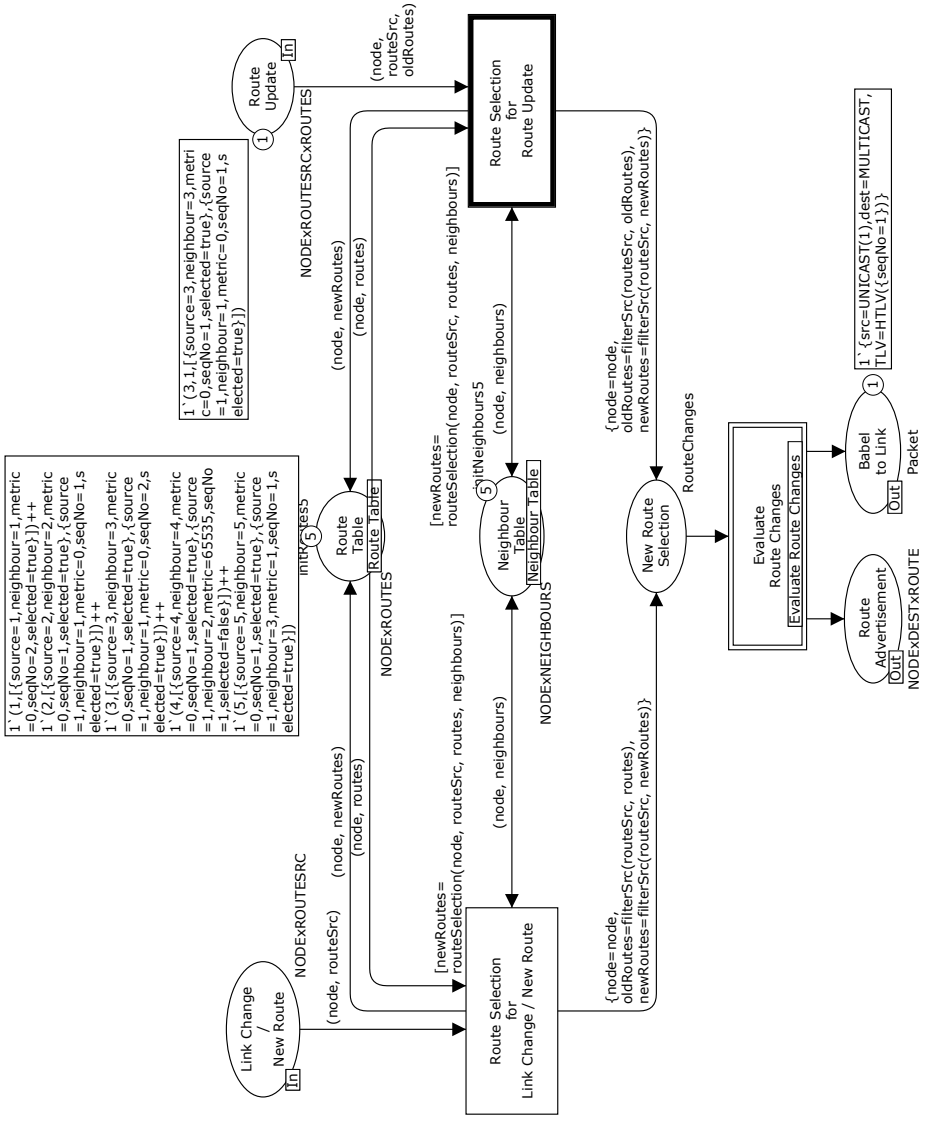


Figure 2.9: The RouteSelection module of the Babel CPN model.

NewRouteSelection over the colour set `RouteChanges`, where the `node` field specifies the current node, and the `oldRoutes` and `newRoutes` fields specify the routes before and after the update was done, respectively. The route selection is obtained by the `routeSelection` function, where the result is bound to the variable `newRoutes` in the guard of the transition RouteSelectionforRouteUpdate, so that the result can be used on multiple outgoing arcs of the transition. The `routeSelection` function takes the current node (`node`), the source node (`routeSrc`), the current node's routes (`routes`) and its neighbours (`neighbours`) as parameters, and select the route to the source through the neighbour where the sum of the route metric and the link cost to the neighbour is the lowest. The link cost to a neighbour is derived from the history of received Hello and IHU TLVs. The implementation of `routeSelection` satisfies the route selection policies from [11], but other constraints could be added, e.g. to avoid switching routes when metric changes are small, or to avoid unstable neighbours.

After the route selection, the changes are evaluated in the EvaluateRouteChanges submodule, which may result in sending of updates and sequence number request. An update is sent either if the route changes concern significant metric changes, a route retraction, a new route, or if an update should be sent in response to a sequence number request. In case of a route retraction, the current node sends a sequence number request destined for the source of the lost route. In this scenario, node 3 sends a route update since it has a pending sequence number request from node 5.

2.2.6 The Generate Update Module

Figure 2.10 shows the GenerateUpdate module. The token on the place RouteAdvertisement over the colour set `NODExDESTxROUTE`, represents a route advertisement from node 3 to a multicast destination, and the route to node 1 with the updated sequence number. The destination of the update is set to be multicast even though it conceptually should be unicast to node 5. However, this is not considered crucial, as routes will still be advertised periodically to a multicast destination.

When GenerateUpdate occurs, the update is generated by the function `generateUpdate` and bound to the variable `updateTlv` in the guard of GenerateUpdate. Further, the feasibility distance of the sending node, contained in the SourceTable is added or updated by the `updateFD` function. The feasibility distance for a source is updated only if the sequence number contained in the Update TLV is larger, or if the metric is lower than what has previously been advertised. If the sending node does not have a feasibility

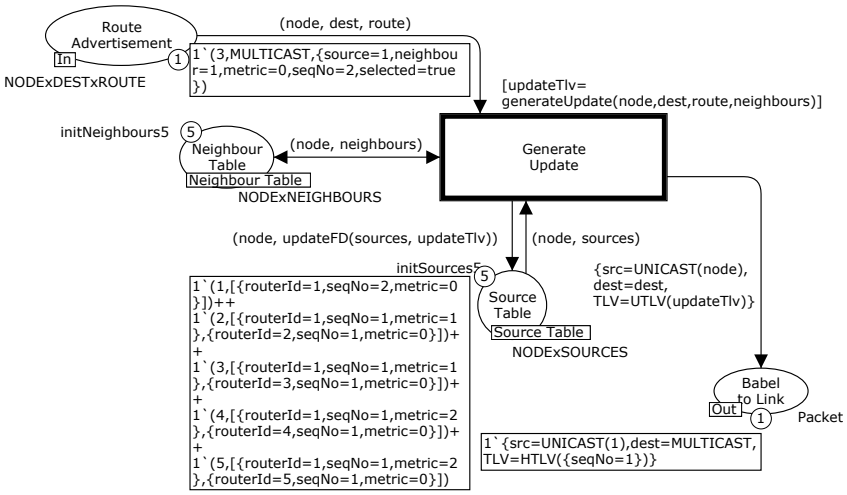


Figure 2.10: The GenerateUpdate module of the Babel CPN model.

distance for the source, it is added based on the update that is being sent. Finally, a token is added to the output place BabeltoLink, representing a packet containing the generated update, ready to be sent over the network link.

2.2.7 The Link Layer

The link layer models a network, which can be configured to simulate a wired or wireless environment. This is a generic model that can be modified to suite several routing protocols. The Link module is a modified version of the network link modelled in the Dynamic On-Demand Routing Protocol for Mobile Ad-hoc Networks (DYMO) model, described in [19]. The Link module, shown in Fig. 2.11 consists of transmission of packets over a network, modelled in the TransmitPacket submodule, as well as simulating topology changes of the network, modelled in the Mobility submodule. Declarations used in this module are defined in Lst. 2.9. The place Topology is marked with a multiset containing one token for each node in the network over the colour set `Topology`. The `Topology` colour set is a product of an integer, representing the router-id to a node, and a list of integers, representing the router-ids of nodes that the node of the first component can reach. The Mobility module models changes of the topology by adding or removing elements in the list of nodes a node can reach.

Transmission of packets, modelled in Transmit Packet are done according

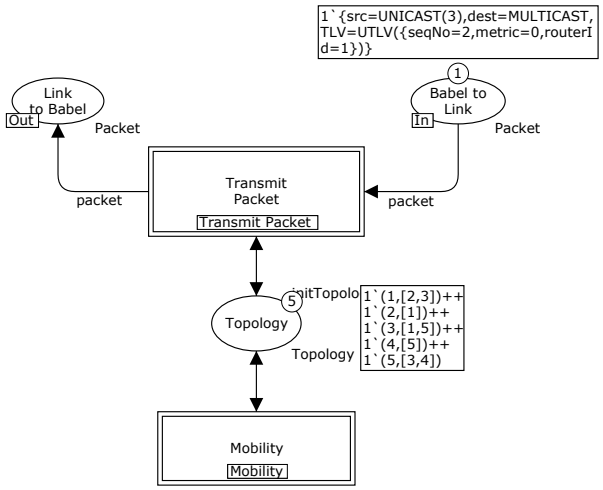


Figure 2.11: The Link module of the Babel CPN model.

Listing 2.9: Declarations used in the Link module.

```
colset Topology = product Nodes * NodeList;
var packet : Packet;
```

to the function `transmit`, shown in Lst. 2.10. The function takes a packet with a `src`, a `dest` and a `TLV`, the `src` node's list of reachable nodes (`adj`), held on the place Topology, and a boolean value `success`. The `success` value is used to simulate packet loss. If `success` is false, then the function returns an empty multiset. If `success` is true and `dest` is a `MULTICAST`, then the function returns a multiset containing packets with unicast destinations for all nodes in the `adj` list. If `success` is true and `dest` is `UNICAST`, the packet is returned as a multiset without any changes. The marking of BabeltoLink in Fig. 2.11 shows a token representing a packet sent from node 3 to a `MULTICAST` destination. Transmission of the packet on the place BabeltoLink will result in two packets with `UNICAST` destination on LinktoBabel, which can be depicted by the marking of Topology.

2.3 Summary

In this chapter, we have presented the CPN model of the Babel protocol. The Babel CPN model will be used to provide assessment of the design

Listing 2.10: The function transmit used for transmission of packets over a network link.

```
fun transmit({src, dest, TLV}, adj: NodeList, success: BOOL) =
    if success then
        if (dest = MULTICAST) then
            list_to_ms (List.map(fn el => {src=src, dest=UNICAST(el), TLV=TLV
                }) adj)
        else(*UNICAST *)
            1`{src=src, dest=dest, TLV=TLV}
    else
        empty;
```

of Scala as inscription language, and to validate the CPN Scala simulator developed in this thesis. This will be presented in Chap. 6.

We have also introduced the basic concepts of CPN modelling, sufficient to introduce how Scala can be used as inscription language for CPNs in the following chapter. In Chap. 4, we will go into more detail of CPNs by introducing the basic formal semantics of CPNs and the enabling inference algorithm.

Chapter 3

Design of Scala as Inscription Language

The purpose of using Scala as inscription language for CPNs is to embed the whole language into the modelling environment, allowing the user to make use of both functional an object-oriented aspects of the language when constructing CPN models. The motivation has been to get a concise and transparent design where Scala is not extended with new syntax. Using Scala allows programmers from an imperative background to more easily adapt to the discipline of modelling with CPNs, as the majority will find Scala's object-oriented nature familiar. For instance, methods do not have to be defined in a strictly functional manner, giving a less steep learning curve for programmers with an imperative background. In some cases, using imperative constructs can make the code more readable than using functional constructs. For instance, when performing computations on a collection in Scala, one has the choice to use higher-order functions like `map` and `filter`, or a `for` loop [21, Chapter 23]. Integrated use of both functional and object-oriented programming styles should be applied to make full advantage Scala.

Scala is a flexible language that is meant to be adapted and extended. The language contains a number of predefined library modules. However, one can easily create new, customised modules which appear as native language support. In fact, a lot of predefined types are constructed in this way, building functionality based on smaller components. This can be utilised for instance when implementing a collection, such as multisets, in a way that makes inscriptions on models intuitive and consistent with the language syntax. How this can be done will be discussed in Chap. 5.

This chapter is organised as follows. A small protocol example using

CPN ML as inscription language will be presented in Sect. 3.1. In Sect. 3.2, this model will be modified to use Scala as inscription language, and different aspects of the use of Scala will be addressed in this section based on the protocol example. In Sect. 3.3, a number of simplifications of inscriptions will be introduced. Finally, Sect. 3.4, concludes the chapter with a second version of the protocol example with the final design, and a summary of the most evident differences between CPN ML and Scala as inscription language for CPNs.

3.1 A Protocol Example

The discussion of how Scala can be used as inscription language will be based on a CPN model of a small protocol example. Multiple variants of the protocol example are used to explain various facilities of the CPN modelling language in [17]. Throughout this chapter, we use some of the variants with minor changes. A variant of the protocol example using CPN ML as inscription language is shown in Fig. 3.1, and Lst. 3.1 shows the declarations used in this model.

The protocol models sending of packets over an unreliable network. The place PacketsToSend holds packets which is to be sent. Tokens on this place

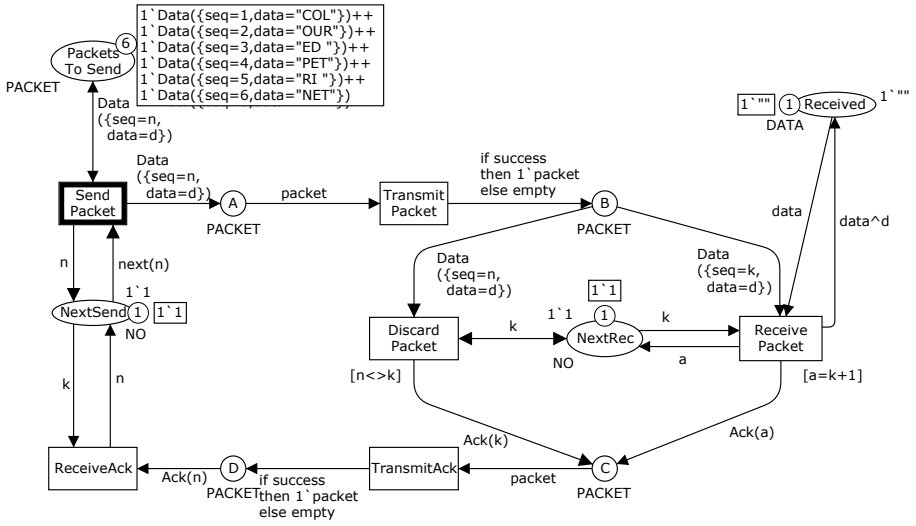


Figure 3.1: Original model of the protocol example with CPN ML as inscription language.

Listing 3.1: Colour set and variable declarations for the protocol example.

```
colset NO = int;
colset DATA = string;
colset BOOL = bool;

colset NOxDATA = product NO * DATA;

colset DATAPACK = record seq: NO * data: DATA;
colset ACKPACK = NO;
colset PACKET = union Data: DATAPACK + Ack: ACKPACK;

var n, k, a: NO;
var success: BOOL;
var d, data, e: DATA;
var packet: PACKET;
```

are of colour set `NOxDATA`, a product of an integer and a string where the integer represents the sequence number of the packet, while the string represents the payload. Packets are sent continuously based on the value of the token on the place NextSend, representing the sequence number of the packet that is to be sent. There must be sufficient tokens on the place NextSend so that it covers the result from evaluating the arc expression on the arc from NextSend to PacketsToSend, `next(n)`. This function has the following definition:

```
fun next(n) = n;
```

Changing the arc expression to `n` would give the same result. The difference is that `n` is a pattern, while `next(n)` is not, meaning that binding the variable `n` in a binding element must be done based on the pattern `(n,d)` on the arc from PacketsToSend to SendPacket. This shows that all arc expressions on input arcs do not have to be patterns, as long as all CPN variables of a transition can be bound from other arc expressions that are patterns, or in the guard.

During transmission of packets, packets may be lost, which is shown from arc expressions on the arcs from TransmitPacket to B, and from TransmitAck to D. If the variable `success` is bound to `true` in an occurrence, the packet (`packet`) is transmitted to the connected place; if `success` is bound to false, the packet is discarded, modelled by returning `empty`, corresponding to the empty multiset over the colour set of the connected place. These parts of the

model exploit the idea of binding variables of small domains without using patterns. Bindings are created for each value of the domain of the variable. Thus, binding elements are created where `success` is bound to values of its domain, which is `true` and `false`.

Packets which are transmitted successfully are either accepted or discarded based on whether the packet has the expected sequence number, held as a token on the place NextRec. Shown from the guard of DiscardPacket, this transition will be enabled in bindings where the sequence number of a packet bound by the variable `n` is different from the expected sequence number of the packet, bound to the variable `k`. The variable `k` of ReceivePacket appears on input arc expressions both from B and NextRec, denoting that the receiving packet must have the expected sequence number for the transition to be enabled. An occurrence of DiscardPacket adds a token to the place C, representing an acknowledgement with the sequence number of the packet that was expected to be received. An occurrence of ReceivePacket transition removes a token from NextRec and adds a token, which is the expected sequence number for the next packet. A token is also added to the place C, representing an acknowledgement with the sequence number of the next expected packet to be received. Finally, a token is added to the place Received, with the value of the removed token (`data`), concatenated with the payload of the received packet (`d`).

Tokens on place C represent acknowledgements transmitted back to the sender. The token colour corresponds to the sequence number of the next packet that should be sent. If the transmission of the acknowledgement by the occurrence of TransmitAck succeeds, a token is added to the place D. When the transmission ReceiveAck occurs, a token on the place NextSend is removed, and a token with the value of the acknowledgement is added.

3.2 Scala as Inscription Language

Figure 3.2 shows an initial design of how the protocol example may look like with Scala as inscription language instead of CPN ML. This model is close to a direct translation of the version with CPN ML as inscription language. Tokens on places in this model are represented in a collection `Multiset`, which is implemented to conform to the multiset implementation in CPN ML, described in [17, Chapter 4.1]. However, the syntax of multisets in Scala differs a lot from multisets in CPN ML. Tokens must be defined inside the `Multiset[T]` constructor as a tuple `(c:Int, t:T)` where `c` is the coefficient, and `t` is the token. The guard of a transition is represented as a list like

with CPN ML, however, the syntax is more verbose. How token types, such as `No` and `Packet`, are defined, will be explained in the following subsection.

The rest of this section will be used to discuss different aspects of the design in this model. We will discuss aspects that are introduced with the object-orientation in Scala, and in turn suggest possible improvements of the initial design. To make a clear distinction between variables in CPNs and Scala, we will use the terms CPN variable and Scala variable.

3.2.1 Data Structures for Tokens

In CPN ML, a set of predefined *basic types* are inherited from SML. These types are used to declare more complex data structures in terms of colour sets. Scala has basic types [21, Chapter 5] similar to the ones found in Java, and includes `Int, Long, Short, Byte, Float, Double, Char` and `Boolean`, and `Unit`. The eight former types correspond to Java's primitive types, while `Unit` has a similar application as `void` in Java and `unit` in SML, used as result type of methods where no meaningful result is to be returned. Along with the basic types, Scala has a rich library with components and building blocks to construct new data structures. This section will provide examples of how colour set declarations of the protocol example can be matched using Scala.

With Scala, types used for inscription of places are defined by means of classes or *case classes* [21, Chapter 15]. A case class is a specialised class where functionality including pattern matching, and a "natural" implementation of the methods `toString`, `hashCode` and `equals` are automatically provided. We will consider these aspects in greater detail in the following sections. The class declarations for token types used in the protocol example model are shown in Lst. 3.2. None of the classes have contents in their bodies, in which case the braces around the empty body can be left out. `No`, `Data` and `Ack` are all defined as case classes to facilitate pattern matching. Defining `No` as a regular class will look like the following expression:

```
class No(val n: Int)
```

The most obvious difference is that the class parameter `n` of type `Int`, is prefixed by `val`. This makes `n` a *parametric field*, a shorthand for defining, at the same time, a parameter and a field with the same name. Parametric fields can also be defined as Scala variables, by prefixing the parameter with `var`. The Scala compiler automatically adds a `val` modifier to all parameters of a case class. A factory method with the same name as the class is also

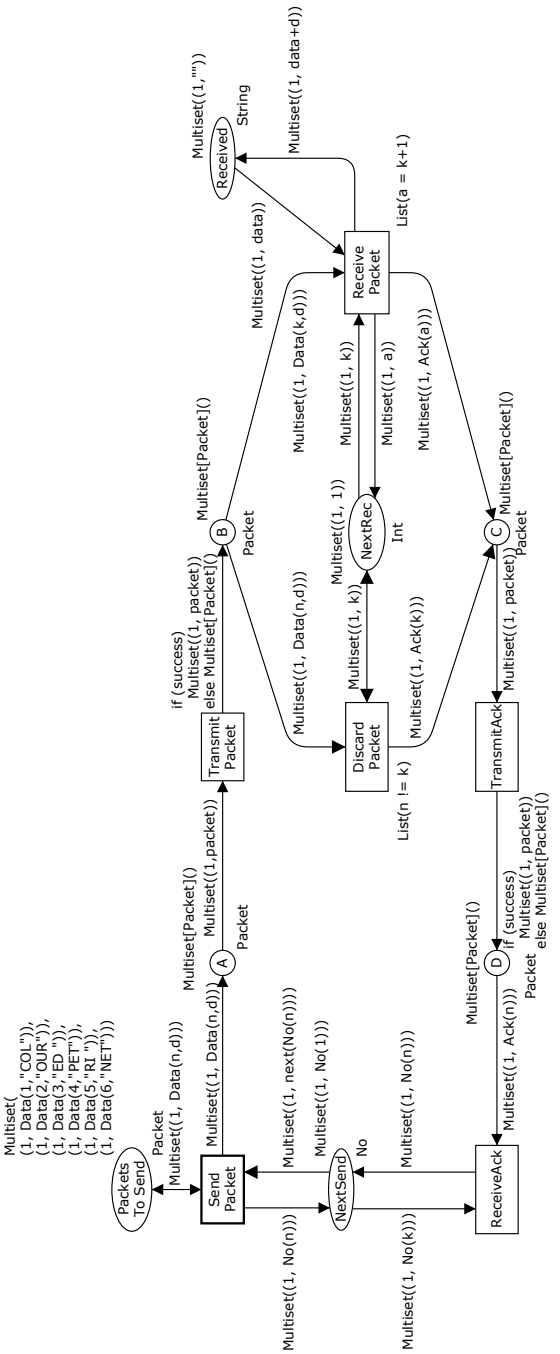


Figure 3.2: Model of the protocol example with Scala as inscription language.

Listing 3.2: Declarations used in the protocol example model.

```
case class No(n: Int)

def next(no: No) = no

abstract class Packet

case class Data(n: Int, d: String) extends Packet

case class Ack(n: Int) extends Packet
```

provided. For instance, a `No` object can be constructed by `No(1)`, instead of `new No(1)` when `No` is a regular class. Factory methods for a class can be defined by implementing one or more `apply` methods in the *companion object* of the class. A companion object has the same name as its *companion class*, and must be defined in the same source file. Class and object companions can access each other's private members.

Methods in Scala are defined with the `def` modifier, shown by the method `next` in Lst. 3.2. The definition of `next` is equal to the following expression:

```
def next(no: No): No = { no }
```

However, the return type can be omitted as it is determined by type inference, and body clauses can be omitted when the body is a single statement.

CPN ML has a union type, used to represent packets in Fig. 3.1. By using inheritance in Scala, one can achieve the same behaviour. This is shown from the case classes `Data` and `Ack` in Lst. 3.2 which extend a common abstract super class `Packet`. This allows the places A, B, C and D in Fig. 3.2 to be defined to hold tokens of type `Packet`, while in the context of this model, A and B will hold `Data` tokens, and C and D will hold `Ack` tokens.

Scala has a number of built-in tuple case classes [1] that can be useful for simple patterns. The classes are named `Tuple<N>`, where `N` is the number of elements in the tuple. `(a: T1, b: T2)` is the constructor of a tuple with type `Tuple2[T1, T2]`, where `a` is of type `T1` and `b` of type `T2`. The type `Tuple2[T1, T2]` can also be expressed as `(T1, T2)`. Figure 3.3 shows parts of the protocol example using tuples instead of the case class `Data`. User defined case classes, including the built-in tuple classes in Scala, cover

[1] The Scala simulator is based on Scala 2.10, which supports 1-tuples to 22-tuples, Tuple1 to Tuple22, respectively.

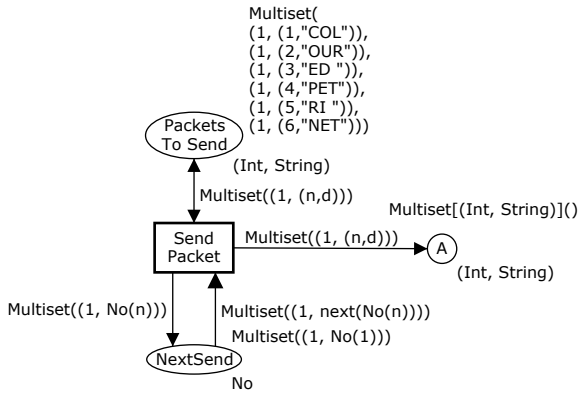


Figure 3.3: Snippet of the protocol example using tuples.

the programming and modelling capabilities of CPNs where one would use records and products constructs in CPN ML.

3.2.2 Colour Set Inscriptions

With CPN ML as inscription language, a place has a colour set inscription, determining the set of tokens the place can hold. Further, an empty initial marking can be omitted. In Fig. 3.2, places with no tokens in the initial marking are marked with the empty constructor for multiset, with type parameter of the type of tokens the place can hold. This is shown from the empty initial marking of the place A, which is `Multiset[Packet]()`. Type parameter for the empty constructor and the type definition of a place are always equal, giving redundant information. This could be solved in the same manner as done with CPN ML as inscription language, omitting declaration of the empty multiset. However, this restricts the representation of tokens on a place to multisets. The use of other representations will be discussed in Sect. 3.2.8.

Assuming that an initial marking is required, regardless of containing any tokens, the explicit colour set inscriptions on places can be omitted. When the initial marking is parametrised with tokens, the token type for a place can be determined by Scala's type inference. For instance, the type of the initial marking on the place NextSend, which is `Multiset((1, No(1)))`, is inferred to `Multiset[No]`. If a place has no tokens in the initial marking, type parameter for the empty collection constructor must be provided, giving the token type. In this way, both the type of the tokens a place can hold, and its initial marking can be defined in one single expression. Figure 3.4

(a) With colour set inscription.

(b) Without colour set inscription.

Figure 3.4: The place NextSend with and without colour set inscriptions.

(a) shows NextSend with redundant type information, and (b) omission of explicit type definition.

3.2.3 Patterns

To be able to bind CPN variables from an arc expression in a binding for a transition, the arc expression (omitting the collection construct) must be a pattern. This makes the use of case classes valuable for user defined classes. It is possible to implement support or pattern matching for regular classes, usually done for special matching behaviour. However, we will not go into details of how this can be done.

To illustrate the binding of CPN variables, we look at the transition SendPacket in Fig. 3.2, having two CPN variables `n` and `d`. The token of the expression on the input arc from `NextSend` is obtained by a method call `next(No(n))`. Since this does not qualify as a pattern, `n` cannot be bound by this expression. Thus, both `n` and `d` must be bound from the token pattern `Data(n,d)` in the arc expression on the input arc from PacketsToSend.

Regular classes can also be useful when matching members of a class is not necessary. This is shown from the arc expression on the arc from A to TransmitPacket in isolation. A token on place A will be bound to the CPN variable `packet` in a binding of TransmitPacket. The pattern is not exploiting the internal values of the token.

Pattern matching of an arc expression can be related to Scala's match construct [21, Chapter 15], outlined in Lst. 3.3. A pattern match includes a number of alternatives, each starting with the `case` keyword, followed by the pattern. The `selector` is matched against the patterns in the order they are defined. If a match is found, subsequent patterns will not be matched. When a pattern is matched, the expression followed by the arrow symbol (`=>`) will be evaluated. The *wildcard pattern* (`_`) in the last case will match all values of the selector. If, in this case, a reference to the matched value is needed for the expression following the pattern, a *variable pattern*, such as `case p => ...`, can be used. A *constant pattern*, such as `1` and `"COL"`, can

Listing 3.3: The `match` construct for pattern matching in Scala.

```
<selector> match {
  case <pattern> => ...
  ...
  case _ => ...
}
```

also be used, and will be matched based on equality with respect to the `==` method.

When using pattern matching on an arc expression, tokens from the connected place can be seen as the selector of the match construct, while the pattern of the arc expression is the only valid match alternative. An arc expression can contain multiple patterns, which must then be parameters to a collection constructor. This can be related to n match constructs for n patterns in the arc expression.

Patterns in Scala can also be *nested.* Figure 3.5 shows parts of a variant of the protocol example using nested patterns. Listing 3.4 shows the class definitions used in this model. The case class `Packet` takes a parameter `pl: Payload`, which is an abstract class extended by the case classes `Data` and `Ack`. Based on these definitions, one can deduce the pattern `Packet(Data(n, d))` shown on the arc expression on the arc from PacketsToSend to SendPacket in Fig. 3.5. `Packet(pl)` is a pattern with some payload `pl`, in this case `pl = Packet(n,d)`. In turn, `Packet(n,d)` is a pattern.

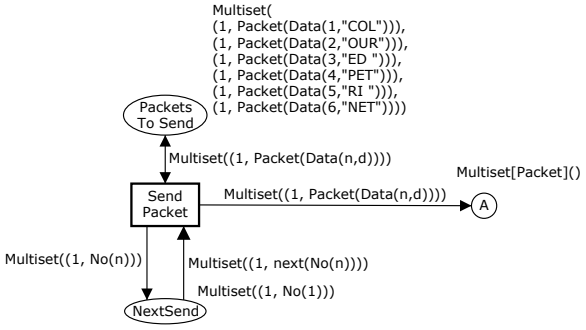


Figure 3.5: Parts of the protocol example using nested patterns.

Listing 3.4: Definition of classes for nested patterns.

```
abstract class Payload

case class Packet(pl: Payload)

case class Data(n: Int, d: String) extends Payload

case class Ack(n: Int) extends Payload
```

3.2.4 CPN Variables

CPN variables present in a CPN model must be explicitly defined using CPN ML. This concept will not adapt intuitively with the use of Scala as inscription language. Scala already has variables, however, Scala variables cannot be used in the same way as CPN variables. CPN variables are only declared with a name and a type, not assigned to a value like Scala variables. This means that the syntax of Scala must be extended with CPN variables, which violates one of the main design goals.

The binding of values to CPN variables happens during simulation of a model, hence, leaving out the definition of the variables is sounder with Scala. Rather than declaring CPN variables explicitly, their types can be deduced by their context. We will look at some examples of this from the protocol example in Fig. 3.2. From the pattern `Packet(n,d)` on the arc expression on the arc from PacketsToSend to SendPacket, the type of `n` can be deduced to `Int`, while the type of `d` can be deduced to `String`, based on the argument types of `Packet`. The CPN variable `packet` in the arc expression on the arc from A to TransmitPacket can be deduced to `Packet` based on the token type on the place A. Based on the guard of ReceivePacket, `a = k+1`, the type of `a` can be inferred by the type of the right hand side expression. In this case, the type of `k` must be known before the type of `a` can be deduced. The type of the CPN variable `success` in the arc expression on the arc from TransmitPacket to B can be deduced to `Boolean` since it is the condition of an if-else clause. `Boolean` is the only small default domain that can be used to bind variables without using a pattern. Other small domains can be defined by extending `Enumeration` in Scala. Figure 3.6 shows parts of the protocol example, where the arc expression on the arc from TransmitPacket to B is changed to show the use of enumerations. Listing 3.5 shows how the enumeration `Transmit` is defined in Scala.

The enumeration `Transmit` provides three values `Transmit.Success`,

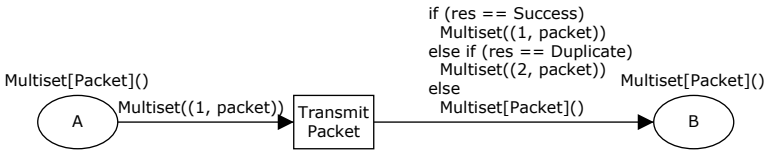


Figure 3.6: Example with the use of enumerations in an arc expression.

Listing 3.5: The enumeration object `Transmit`.

```
object Transmit extends Enumeration {
  val Success, Duplicate, Failure = Value
}
```

`Transmit.Duplicate` and `Transmit.Failure`. From the context where the CPN variable `res` is used, its type can be deduced to `Transmit.Value`. Notice that the values of `Transmit` can be accessed directly in a model as long as they are imported to the scope of the model.

When CPN variables are not explicitly defined, the same variable name can be used in different binding elements where the variable will be bound to values of different type. However, it must be verified that the same variable name is not used for different types in the context of a single transition.

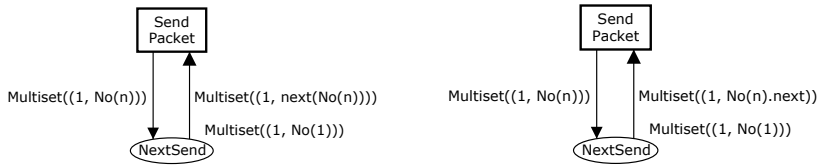
3.2.5 Scoping

The function `next` in Lst. 3.2 is defined globally. However, since it has one parameter of type, `No`, it will make more sense to define it in the scope of the class `No`. Listing 3.6 shows the redefinition of the class `No`, where the method `next` is scoped within the class. Now, the parameter to the method `next` does not have to be specified, making the method *parameterless*. For parameterless methods, the parentheses can be left out as long as the method does not have any side effects. The possibility to scope functions within classes is an object-oriented facility that can be of great benefit both for structuring of functions defined for a model, and for the readability of inscriptions,

Listing 3.6: The class `No` with a method `next`.

```
case class No(n: Int) {
  def next = No(n)
}
```

especially when creating large models. Figure 3.7 shows the difference of the arc expression from NextSend to SendPacket when the method `next` is defined globally (a), and scoped within the case class `No` (b).



(a) Global scoping of the method `next`.

(b) The method `next` scoped within the class `No`.

Figure 3.7: Different scoping for the method `next`.

3.2.6 Object Equality

A question that arises due to the object-oriented nature of Scala, is whether objects (tokens) should be compared based on object identity or a natural notion of equality, e.g., value equality. This question is essential when considering how tokens are represented on a place. If tokens were to be compared by object references, it would make less sense to represent tokens in multisets, which will be explored in the following example. Consider the model in Fig. 3.8, where `Data` is redefined to a regular class with a *variable* `n: Int` and a value `d: String`, as shown in Lst. 3.7.

Listing 3.7: The class `Data` where `n` is a variable.

```
class Data(var n: Int, val d: String)
```

The model has a marking resulting from the occurrence sequence with the following binding elements, where a binding element is represented as a pair, with the transition as the first component, and the binding as the second component:

```
(T1, <p = Data(1,"COL"), n = 1, d = "COL">)
(T2, <p = Data(1,"COL")>)
(T3, <p = Data(1,"COL")>)
(T1, <p = Data(2,"OUR"), n = 2, d = "OUR">)
```

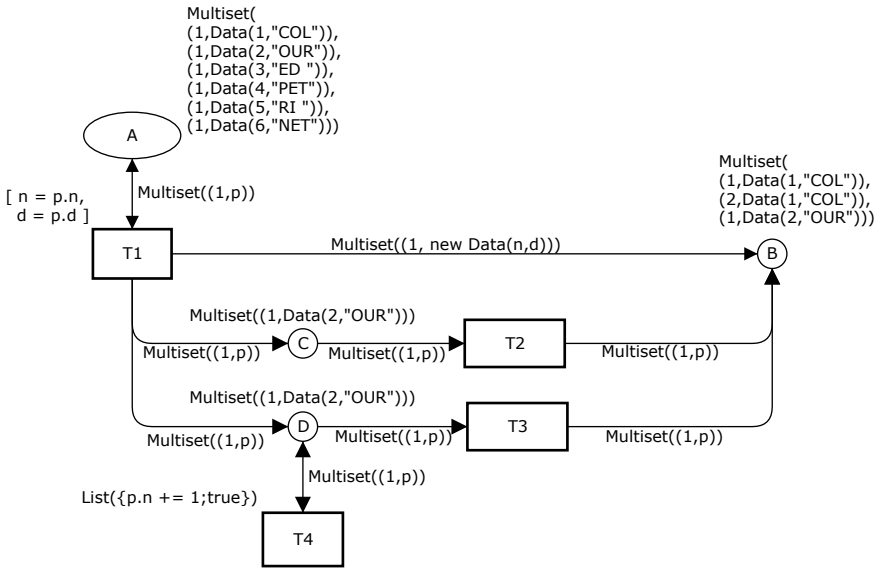


Figure 3.8: Example with object equality and multisets.

In this example, objects are compared by object reference, since `Data` is defined as a regular class. For better readability, we assume that the `toString` method of the class `Data` is redefined to express the actual values.

Shown in the marking on the place B, there are three `Data` tokens where the field `n` equals `1`, and the filed `d` equals `"COL"`. However, only two of the tokens refer to the same object, namely the tokens added when T2 and T3 occurred. The third was added based on the arc expression on the arc from T1 to B, after the first occurrence of T1, which gives a new `Data` object.

Considering that CPN is a graphical modelling language, and hence, used to construct a visual representation of some problem domain, it is more natural to consider two objects with the same visual representation, i.e. two objects being equal in terms of natural comparison, as equal, regardless of their object identity. Additionally, Scala's approach of equality comparison is structured so that natural comparison should be preferred to object identity [21, Chapter 30]. All Scala classes inherit the final `==` method, defined in the class `Any` (which is the super class of all classes). The default behaviour of this method is comparison of object identity. However, the behaviour of the method `==` is meant to be redefined by overriding the `equals` method. Scala also has a method specific for comparing object identity, `eq`. The equality issue is avoided when using case classes, where a natural im-

plementation is provided automatically. All arguments of a case class will be compared recursively with the `==` operator. In this way, one may have a nested structure of case classes, without the need to redefine the `equals` method. This is why the following expression, that is based in the definitions in Lst. 3.4, will evaluate to `true`:

```
Packet(Data(1, "COL")) == Packet(Data(1, "COL"))
```

This is regardless of the fact that we are comparing two different `Packet` objects, containing two different `Data` objects, which means that the following expression will evaluate to `false`:

```
Packet(Data(1, "COL")) eq Packet(Data(1, "COL"))
```

If we use the same comparison based on the `Data` class as defined in Lst. 3.7, the `equals` method must be redefined to get the same natural behaviour. Hence, the following expression will evaluate to `false`:

```
new Data(1, "COL") == new Data(1, "COL")
```

To conclude this discussion, it is beneficial to define colour sets as case classes, giving a natural behaviour for equality. If natural behaviour is not required, one has the opportunity to use regular classes, or redefine the `equals` method of case classes, hence, flexibility is preserved. Additionally, the `eq` method can be used when comparison based on object identity is preferred. Seen from the example in Fig. 3.8, using regular classes without redefining the `equals` method should be avoided when using multisets.

3.2.7 Side Effects

When passing objects (tokens) in evaluation of arc expressions, side effects have to be considered. Assume T4 occurs in the given marking of Fig. 3.8 with the binding element `(T4, <p = Data(2,"OUR")>)`. The resulting marking of D will be `Multiset((1,Data(3,"OUR")))`, as expected due to the increment of the Scala variable `n` of object `p` in the guard of T4. What may not be that obvious is that the marking of A and C will change as well, since these places contain tokens referring to the same object. The token with the same visual representation in the marking of B does not change, as it refers to another object.

A problem with an expression in a guard having side effects is that during enabling inference, the guard can be evaluated and cause changes

in the marking independent of any occurrences. In Renew [24], side effects are supported in *actions*. An action on a transition is performed when the transition occurs. A second problem is the mechanism a CPN simulator uses to find enabled transitions during simulation. The SML simulator relies on a locality principle [2] that will be violated if markings can change based on side effects. In Renew, enabling of transitions cannot rely on mutability of objects. This is also the case for Scala simulator implemented in this thesis.

3.2.8 Collections on Places

In Fig. 3.2, tokens on places are held in multisets, like with CPN ML as inscription language. Following the discussion from Sect. 3.2.6, where natural equality should be preferred over object identity on tokens, a representation with multisets on places would be appropriate. However, giving the possibility to choose among several collections to represent tokens on a place can in some cases simplify the model. With CPN ML as inscription language, one has the possibility to *model* other data structures for a place, such as a *FIFO* (first in first out) queue. This can be seen in Fig. 3.9, showing a snippet of another variant of the protocol example. The place A has a marking consisting of a single list, at all times. When SendPacket occurs, a token is added to the back of the list on A, using the `^^` operator. When TransmitPacket occurs, the first element of the list on A is removed, using the list pattern `::`. This gives the behaviour of a FIFO queue for the place. This approach requires details in the model that are on a lower abstraction level than the actual model.

We propose a design where multiple collections can be used for place markings. The criteria are that the collections share a common set of operations to add, remove and compare collections to accommodate the simulator implementation, which will be presented in Chap. 5. A set of commonly used collections can be provided in the implementation of the simulator, such as multiset, queue and stack. However, the real benefit with this structuring of collections, is that new kinds of collections can be implemented, specifically for a given model. How this can be done is explained in Chap. 5.4.

Figure 3.10 shows how the queue example given in Fig. 3.9 looks like when using Scala as inscription language, using a `Queue` instead of a `Multiset` to maintain tokens on the place A. In this example, there is no more overhead regarding inscriptions and arcs to use a queue instead of a multiset.

[2] This will be discussed furhter in Chap. 5.

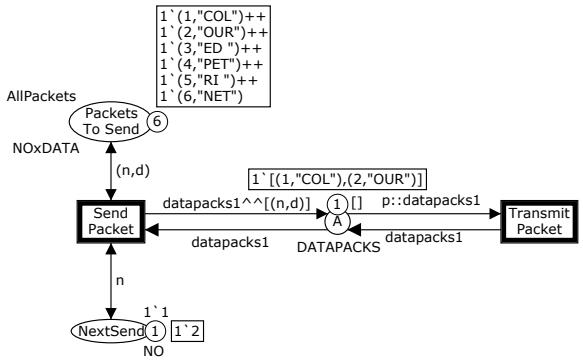


Figure 3.9: Snippet of the protocol example using CPN ML, showing modeling of a queue.

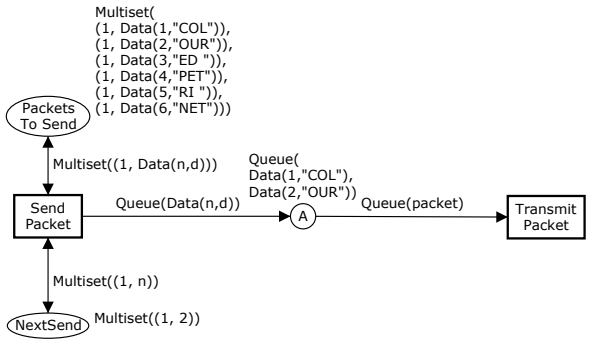


Figure 3.10: Snippet of the protocol example using the collection `Queue` in Scala.

3.2.9 Parameterised Modules

Being able to divide a CPN model into several modules is an important feature when constructing larger models, as seen from the Babel CPN model, introduced in Chap. 2. This gives multiple abstraction levels, making modelling more convenient, and allows reuse of a module in several parts of a model.

Figure 3.11 shows the Top module of a version of the protocol example where the Sender, Network and Receiver parts are modelled in separate submodules. The interesting part is the Network substitution transition, representing the Network module, shown in Fig. 3.12, which has two type parameters, `A` and `B`. In this way, the port places in the Network module are not restricted to concrete types. The abstract type of a port place in the Network module is determined by the concrete type of the related socket place in the Top module. Hence, `A` is set to `Data`, and `B` is set to `Ack`, based on the concrete types of the socket places A and B, and C and D, in the Top module, respectively.

The Network module contains two *instances* of the Transmit module, shown in Fig. 3.13. The Transmit module has one type parameter, `T` representing the token type to be transmitted. The abstract type `T` for Transmit is set to `Data` for one of the instances, and `Ack` for the second, based on the concrete types obtained for the socket places in the Network module.

Type parameters on modules give a looser coupling and a higher degree of flexibility between the modules than what is possible when using CPN ML, since modules cannot be parameterised with abstract types with CPN ML. In this example, `Data` and `Ack` do not have to inherit from the abstract class `Packet`. Inheriting from `Packet` may be natural in this context, however, in other situations, one may be forced to unify types that do not have a conceptual relation. Avoiding reliance on defined types (colour sets) makes the use of modules in different contexts more flexible. For instance, the Transmit module can be used in any CPN model, without requiring definition of any classes.

3.3 Simplifications for Inscriptions

Arc expressions in Fig. 3.2 are quite verbose due to the repeatable use of the `Multiset` constructor. This is not as critical in CPN ML as the syntax of multisets are simpler. Still, simplifications are allowed when using CPN ML, where an arc expression `1‘t` can be simplified to `t`. This means that when an arc expression consists of a single token, the coefficient can be omitted.

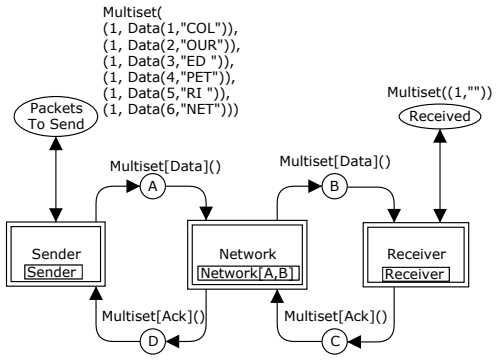


Figure 3.11: Top module of the protocol example.

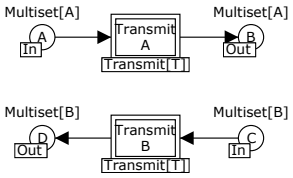


Figure 3.12: Network module of the protocol example.

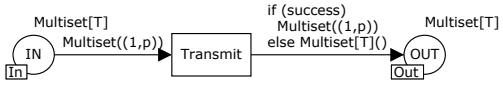


Figure 3.13: Transmit module of the protocol example.

When using Scala as inscription language, shorter constructor names for collections would reduce the extent of inscriptions. `Multiset` could for instance be redefined to `Ms`. However, additional simplification can be done to improve the readability further.

As mentioned in Sect. 3.2.1, multiple factory methods can be defined in the companion object of class. Two factory methods can be implemented for `Multiset`: One taking elements of type `(Int, T)` and one taking elements of type `T`. The first corresponds to the examples shown above, where the coefficient is explicitly given. When instantiating a `Multiset` where the coefficient of all elements are `1`, the second factory method, taking elements with type `T` can be used. In this way, the arc expression on the arc between PacketsToSend and SendPacket in Fig. 3.2 can be simplified from `Multiset((1,Data(n,d)))` to `Multiset(Data(n,d))`. This simplification goes further than what is allowed using CPN ML. Using CPN ML, omission of multiset coefficient is not allowed when an arc expression contains different tokens, hence, `1‘(m,c) ++ 1‘(n,d)` cannot be simplified to `(m,c) ++ (n,d)`. There is one corner case where the simplification for `Multiset` in Scala is not allowed. This can be seen from the arc expression on the arc between PacketsToSend and SendPacket in Fig. 3.3. `Multiset((1,(n,d)))` cannot be simplified to `Multiset((n,d))`. This is because `n` is of type `Int`, and this instantiation will be applied by the former factory method, considering `n` as the coefficient. Hence, the type of `Multiset((n,d))` is `Multiset[String]`, which does not match the type `Multiset[(Int,String)]` of the place PacketsToSend.

Omission of the collection constructor parameterised with exactly one token should be allowed. The arc expression on the arc between PacketsToSend to SendPacket in Fig. 3.2 would then be simplified from `Multiset((1, Data(n,d)))` to `(1, Data(n,d))`. In relation to omission of the coefficient using factory methods, this can further be simplified to `Data(n,d)`.

A further simplification can be to allow representations of tokens in a more general way, such as lists. This would allow representing `Multiset((m,c),(n,d))` as `List((m,c),(n,d))`. As arc expressions often contain function calls, this will allow functions to be made more flexible, so that they can be used in the context of different collection types. How these simplifications can be addressed will be discussed in greater detail in Chap. 5.

Since the syntax of a list is quite verbose in Scala, we propose changing the representation of the guard to a tuple. When a guard contains a single expression, no additional construct is required. For example, the guard of ReceivePacket in Fig. 3.2, can be simplified from `List(a = k+1)` to `a = k+1`.

When a guard is in conjunctive form, with n conjuncts gc_i where $1 \leq i \leq n$, the guard is represented as $(gc_1, \ldots, gc_n)$.

3.4 Final Design of Inscriptions

In this chapter, we have addressed the most evident aspects of adapting Scala as inscription language for CPNs. Several simplifications of inscriptions have also been introduced. Figure 3.14 shows the protocol example with the final design of Scala as inscription language, where design choices discussed in this chapter have been applied. Allowing arc expressions to consist of single token expressions, not only collections, reduces the amount of inscription code significantly. Leaving out the multiset coefficient makes both arc expressions and initial markings easier to read. Omission of colour set declarations on places also reduces the amount of inscription code, while the type of tokens a place can hold, can easily be seen from the initial marking. Finally, omission of the list construct in guards of transitions has reduced the amount of inscriptions.

We conclude this chapter by summarising the most evident differences between using CPN ML and Scala as inscription language for CPNs, presented in Tab. 3.1.

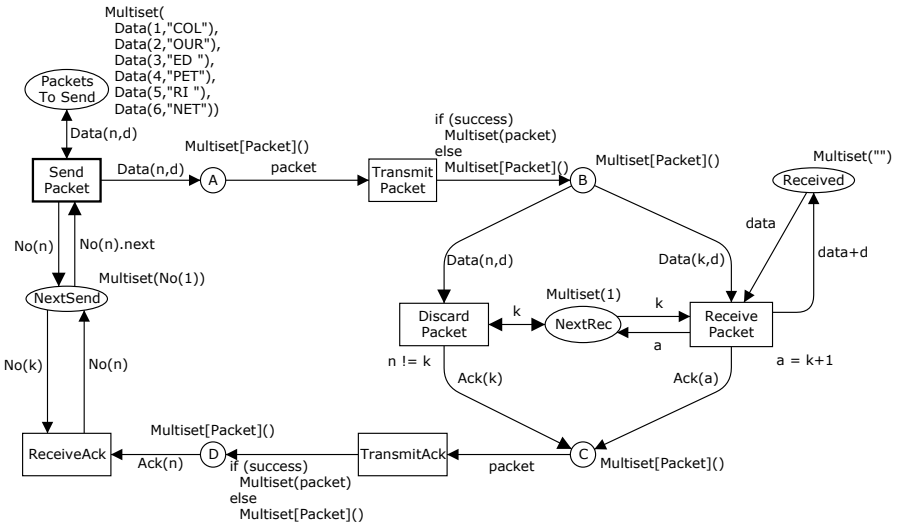


Figure 3.14: Model of the protocol example using Scala as inscription language with simplifications applied.

Feature	**CPN + ML**	**CPN + Scala**
Token type (colour set) declaration	Basic types, product, record, enumeration, range	Basic types, classes, case classes (including tuple, enumeration and range)
Colour set inscriptions on places	Explicit	Implicit from initial marking
CPN variable	Explicitly declared	Implicit from its context
Scoping	Global	Nested
Equality	Natural	Natural or object identity
Side effects	No	Yes
Collections	Multiset	Any
Simplification of arc expressions	For multiset with one element	For collection with an arbitrary number of elements
Extended language features	Yes	No
Modules parameterised with abstract types	No	Yes
Guard representation	List	Tuple

Table 3.1: Comparison of CPN + ML and CPN + Scala.

Chapter 4

Enabling Inference

Chapter 2 provided an introduction of how CPNs operate, based on the constructed Babel CPN model. In this chapter, we focus on the formal semantics of CPNs and the enabling inference algorithm which is the fundamental basis of calculating enabled bindings in a CPN simulator. The content of this chapter is a summary of the formal specification given in [18], adapted to Scala instead of SML.

Section 4.1 introduces the basic formal semantics of CPNs. Section 4.2 gives a brief overview of the pattern matching capabilities in Scala in the context of CPNs. Finally, the enabling inference algorithm is presented in Sect. 4.3.

If not explicitly stated, when variables are mentioned in this chapter, we refer to CPN variables. A set of notations will be used in the exposition of the formal semantics and the inference algorithm. To make the presentation easier to follow, we have summarised notations with descriptions in Tab. 4.1, which can be referred to when reading this chapter. An excerpt of the protocol example with the final inscription design from Fig. 3.14, is shown in Fig. 4.1. This figure will be used for explaining the formal definitions.

4.1 Formal Semantics

Free variables $FV(e)$ of an expression e, is the set of variables that are not bound in the environment of the expression e. For a transition t, the set of free variables $FV(t)$ of the transition is defined by:

$$FV(t) = \bigcup_{p \in P_{In}(t)} FV(E(p,t)) \cup \bigcup_{p \in P_{Out}(t)} FV(E(t,p)) \cup FV(G(t))$$

Notation	Description
$P_{In}(t)$	The set of input places to a transition t.
$P_{Out}(t)$	The set of output places to a transition t.
$E(p,t)$	The arc expression on an arc from a place p to a transition t.
$E(t,p)$	The arc expression on an arc from a transition t to a place p.
$G(t)$	The guard of a transition t.
$FV(e)$	The set of free CPN variables of an expression e.
$Type[v]$	The type of a free variable v.
$e\langle b \rangle$	The result of evaluating an expression e in a binding b.
$C(p)$	Token type (colour set) on a place p.
$M(p)$	The marking of a place p.

Table 4.1: Notations used to present the formal semantics of CPNs.

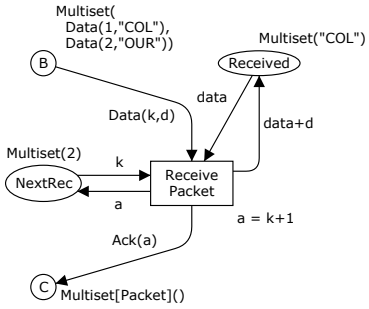


Figure 4.1: Excerpt of the protocol example.

This means that the set of free variables of a transition is the union of free variables of arc expressions on *input arcs* and *output arcs* of t, and free variables in the guard of t. For the transition ReceivePacket in Fig. 4.1, the set of free variables consists of `k`, `d`, `data`, and `a`.

A binding of a transition binds a value to each of the free variables of the transition. This is written as $\langle v_1 = c_1, v_2 = c_2, \ldots, v_n = c_n \rangle$, where $FV(t) = \{v_i \mid i \in 1 \leq i \leq n\}$ and c_i is the value bound to v_i. For a value c_i bound to v_i in a binding, it is required that $c_i \in Type[v_i]$. The notation for evaluating an expression e in a binding b is written $e\langle b \rangle$, and can be defined via the semantics of Scala:

$$e\langle b \rangle \equiv ((v_1 : Type[v_1], \ldots, v_n : Type[v_n]) => e)(c_1, \ldots, c_n)$$

In the context of Scala, it must be ensured that the type of the value obtained when evaluating an arc expression in a binding, conforms to the collection type and the token type on the connected place. This is defined as follows for input places:

$$\forall p \in P_{In}(t) :$$
$$Type[(v_1 : Type[v_1], \ldots, v_n : Type[v_n]) => E(p,t)] \in C(p)_{COLL}$$

Similarly for output places:

$$\forall p \in P_{Out}(t) :$$
$$Type[(v_1 : Type[v_1], \ldots, v_n : Type[v_n]) => E(t,p)] \in C(p)_{COLL}$$

This means that the result type of evaluating an arc expression in a binding must match $C(p)_{COLL}$, where $C(p)$ is the token type on place p, and $_{COLL}$ is the collection type the tokens are maintained in. The definition in [18] is stricter, where the collection type is restricted to multisets.

For practical convenience, it is allowed to omit the collection construct so that the type of evaluating an arc expression in a binding can match $C(p)$. This can be seen from the arc expression `Data(k,d)` on the arc from B to ReceivePacket in Fig. 4.1, which is equivalent to writing `Multiset(Data(k,d))`.

The guard of a transition t is required to evaluate to boolean, defined as follows:

$$Type[(v_1 : Type[v_1], \ldots, v_n : Type[v_n]) => G(t)] = Boolean$$

It must be possible to bind variables from a guard. Since the assignment and equality operator in SML are the same (=), it is possible to bind variables in a guard without violating the constraint that the guard must evaluate to a boolean. In Scala, however, the assignment operator (=) is different from the equality operator (==). The guard of ReceivePacket, `a = k + 1`, in Fig. 4.1 binds `a` to the value obtained by evaluating `k + 1`, but this is not a boolean expression in Scala. A technique to obtain the correct behaviour will be presented when introducing the enabling inference algorithm later in this chapter, and we will consider all guards as boolean expressions. Using = should be preferred over ==, as the latter does not allow binding.

A binding element is a pair (t, b), where t is a transition and b a binding. For a binding element to be enabled in a marking M, the following condition must hold, where $\leq$ is a collection operator comparing two collections:

$$\forall p \in P_{In}(t) : (E(p,t)\langle b \rangle \leq M(p)) \wedge G(t)\langle b \rangle$$

This means that there must be sufficient tokens on each input place p to a transition t, so that tokens can be removed based on evaluating the arc expression $E(p,t)$. Further, the guard must be satisfied. If these conditions are satisfied, the transition is enabled. `(ReceivePacket, <k=2, d="OUR", data="COL", a=3>)` is the only enabled binding element in the marking shown in Fig. 4.1. The binding element `(ReceivePacket,`$\langle$`k=1, d="COL", data="COL", a=2`$\rangle$`)` is not enabled because there are not sufficient tokens on NextRec, determined by evaluating `Multiset(1)` $\leq$ `Multiset(2)`, where the first operand represents the arc expression on the arc from NextRec evaluated in the binding element, and the second operand the marking of NextRec.

4.2 Pattern Matching

The computation of enabled bindings in a CPN is based on an inference mechanism, where pattern matching is essential in the implementation language. Listing 4.1 shows the syntax of patterns in Scala, extracted from the language specification [26]. When using Scala as inscription language, there are some restrictions on which patterns that can be used for arc expressions and guards. As written in [18], wildcards are not allowed, since it is important that binding elements are deterministic. Further, *pattern binders* [26, Chapter 8.1.3], depicted by `Pattern2` are not allowed because such patterns

cannot be evaluated without the context of a match construct, depicted in Lst. 3.3. The specialised `XmlPattern` is not supported. There are some additional constraints regarding the integration of the Scala simulator with CPN Tools (not the Scala simulator itself), which will be discussed further in Chap. 5. In the following section, $Pattern(e)$ denotes that the expression e is a valid pattern as introduced above.

Listing 4.1: Syntax for patterns in Scala.

```
Pattern       ::= Pattern1 { '|' Pattern1 }
Pattern1      ::= varid ':' TypePat
                |  '_' ':' TypePat
                |  Pattern2
Pattern2      ::= varid ['@' Pattern3]
                |  Pattern3
Pattern3      ::= SimplePattern
                |  SimplePattern { id [nl] SimplePattern }
SimplePattern ::= '_'
                |  varid
                |  Literal
                |  StableId
                |  StableId '(' [Patterns] ')'
                |  StableId '(' [Patterns ','] [varid '@'] '_' '*' ')'
                | '(' [Patterns] ')'
                |  XmlPattern
Patterns      ::= Pattern {',' Patterns}
```

4.3 The Enabling Inference Algorithm

The enabling inference algorithm, which is used for computing enabled bindings for a transition is based on an *ordered pattern binding basis*, which is an ordered set of arc expressions from input arcs and guard conjuncts, that is used to bind free variables of a transition. A guard can be in conjunctive form, expressed as $G(t) = \bigwedge_{i=1}^{n} G_i(t)$. Guard conjuncts are usually the most restrictive ones when determining the set of enabled bindings for a transition. This is one reason why the pattern binding basis is ordered, so that guard conjuncts that are included, are considered as early as possible to discard disabled bindings. Further, it is not necessary to use all patterns on surrounding input arcs, as long as all free variables are covered. An ordered pattern binding basis $OPBB(t) = \{E_j \mid 1 \leq j \leq l\}$ for a transition t is required to satisfy four conditions:

1. $FV(t) = \bigcup_{j=1}^{l} FV(E_j)$
2. $\forall E_j \equiv E(p,t) \in OPBB(t) : Pattern(E(p,t))$
3. $\forall E_j \equiv G_i(t) \in OPBB(t) : G_i(t) \equiv G_{il} = G_{ir} \wedge Pattern(G_{il})$
4. $\forall E_j \equiv G_i(t) \equiv G_{il} = G_{ir} \in OPBB(t) : FV(G_{ir}) \subseteq \bigcup_{h=1}^{j-1} FV(E_h)$

(1) ensures that all variables of the transition are covered. (2) specifies that all arc expressions included must be patterns. This restricts an arc expression to contain a single pattern, meaning that an arc expression $E(p,t)$ can only remove one token at a time from the place p. However, it is possible to remove several tokens from one arc expression, i.e., it is also possible to have patterns for multiple tokens. For instance, consider an arc expression on the form $Multiset(e_1, e_2)$, containing two token expressions e_1 and e_2. To conform to (2), the arc with this arc expression must be split into two arcs, one for each of the token expressions. (3) specifies that all guard conjuncts included must have the form $\langle lhs \rangle = \langle rhs \rangle$, where lhs is a pattern. (4) ensures that all the variables occurring in the rhs of a guard conjunct are bound by former patterns included, and hence, that it is possible to evaluate the rhs before binding variables in lhs.

A possible ordered pattern binding basis for ReceivePacket is E(B, ReceivePacket) as the first element (covering `k` and `d`), G(ReceivePacket) as the second element (covering `a`), and E(Received, ReceivePacket) as the last element (covering `data`).

While calculating bindings based on the ordered pattern binding basis of a transition, *partial bindings* are obtained. This is because one pattern does not bind all variables, unless the pattern binding basis consists of one single element. A partial binding is written as $\langle v_1 = c_1, \ldots, v_n = c_n \rangle$, where $c_i \in Type[v_i] \cup \{\perp\}$, and $FV(t) = \{v_i \mid i \in 1 \leq i \leq n\}$. This means that a variable can be bound to a value conforming to its type, or may not yet be bound, which is expressed using $\perp$.

Three functions are needed in the enabling inference algorithm that will be introduced shortly. These functions are listed below, followed by more detailed explanations.

PartialBind$(\mathbf{p}, \mathbf{v})$ which evaluates to a partial binding based on matching the token value v to the pattern p.

Compatible$(\mathbf{pb_1}, \mathbf{pb_2})$ which evaluates to a boolean based on whether the partial bindings pb_1 and pb_2 are *compatible*.

Merge($\mathbf{PB_1}, \mathbf{PB_2}$) which *merges* two sets of partial bindings PB_1 and PB_2.

For two partial bindings pb_1 and pb_2 to be compatible, the following condition must hold:

$$\forall v \in FV(t) : pb_1(v) \neq \bot \wedge pb_2(v) \neq \bot \Rightarrow pb_1(v) = pb_2(v)$$

A function $Combine(pb_1, pb_2)$ is used to obtain a partial binding pb by *combining* two partial bindings pb_1 and pb_2, where the result pb must conform to the following condition:

$$pb(v) = \begin{cases} pb_1(v) & : \quad pb_1(v) \neq \bot \\ pb_2(v) & : \quad pb_2(v) \neq \bot \\ \bot & : \quad otherwise \end{cases}$$

Merging two sets of partial bindings PB_1 and PB_2 must conform to the following condition:

$$Merge(PB_1, PB_2) = \\ \{Combine(pb_1, pb_2) \mid \exists (pb_1, pb_2) \in PB_1 \times PB_2 : Compatible(pb_1, pb_2)\}$$

This means that each partial binding pb_1 in PB_1 is combined with all compatible partial bindings in PB_2. If a partial binding in one of the sets is not compatible with any partial binding in the other set, then it is discarded.

As an example, we consider partial binding of tokens on the place B from the pattern E(B, ReceivePacket) in Fig. 4.1, which results in a set of two partial bindings:

$pb_1 = \langle$`k=1, d="COL", data=`$\bot$`, a=`$\bot\rangle$
$pb_2 = \langle$`k=2, d="OUR", data=`$\bot$`, a=`$\bot\rangle$

Next, partial bind can be performed for the guard of ReceivePacket based on pb_1 and pb_2, where the pattern is the left hand side, `a`, and the value is the right hand side evaluated in the two partial bindings. This results in a set of two partial bindings:

$pb_3 = \langle$`k=1, d="COL", data=`$\bot$`, a=2`$\rangle$
$pb_4 = \langle$`k=2, d="OUR", data=`$\bot$`, a=3`$\rangle$

Further, partial binding of the token on the place Received from the pattern

E(Received,ReceivePacket) in the marking of Fig. 4.1 results in one partial binding:

$pb_5 = \langle$`k=`$\perp$`, d=`$\perp$`, data="COL", a=`$\perp\rangle$

Both *Compatible*(pb_3,pb_5) and *Compatible*(pb_4,pb_5) are true, and merging the two sets of partial bindings gives the following set of full bindings:

$\langle$`k=1, d="COL", data="COL", a=2`$\rangle$
$\langle$`k=2, d="OUR", data="COL", a=3`$\rangle$

Figure 4.2 depicts the full enabling inference algorithm, used to calculate the enabled bindings for a transition t based on its ordered pattern binding basis. For each element in the ordered pattern binding basis, a set of partial bindings are calculated. Line 5-13 handle elements that are guard conjuncts, while line 15-23 handle arc expressions. A guard conjunct binds additional variables in a binding b' from each binding b of C, and merges b and b' in a set C'. Finally, C is assigned to C'. An arc expression creates a set of partial bindings C', by binding variables based on the tokens in the marking of the input place of the current arc. Finally C is assigned to the set obtained by merging C and C'. When the outer for-loop, in line 3 is finished, i.e., all elements in the ordered pattern binding basis have been processed, then the set C only contains full bindings. Line 27 removes bindings from C that do not satisfy the guard of the transition t. It is sufficient to check guard conjuncts that are not in the ordered pattern binding basis. Line 28 removes bindings from C where there are not sufficient tokens on input places for the values bound to the variables in the bindings. It is sufficient to perform this step for input places where the arc expression is not in the ordered pattern binding basis. Finally, line 30 returns the set of bindings that are enabled for the transition t.

One special case the SML and Scala simulators are handling, are variables with values of small domains, such as booleans. This is not covered in the four rules for ordered pattern binding basis and the inference algorithm. For instance, a variable of type boolean can appear only on an output arc from a transition. This variable does not have to be bound by patterns, as all values of the domain will be considered, e.g., in case of boolean, `true` and `false`.

As discussed earlier in this chapter, guard conjuncts that bind variables are not boolean expressions in Scala. Therefore, all such guard conjuncts

1: { $OPBB(t) = \{E_j \mid 1 \leq j \leq l\}$ ordered pattern binding basis for t }
2: $C \leftarrow \emptyset$
3: **for** $j = 1$ to l **do**
4:
5: $C' \leftarrow \emptyset$
6: **if** $E_j \equiv G_i(t) \equiv G_{il} = G_{ir}$ **then**
7: **for all** $b \in C$ **do**
8: $b' \leftarrow PartialBind(G_{il}, G_{ir}\langle b \rangle)$
9: **if** $b' \neq \perp$ **then**
10: $C' \leftarrow C' \cup Merge(\{b'\}, \{b\})$
11: **end if**
12: **end for**
13: $C \leftarrow C'$
14:
15: **else** $\{E_j \equiv E(p,t)\}$
16: **for all** $c \in M(p)$ **do**
17: $b' \leftarrow PartialBind(E(p,t), c)$
18: **if** $b' \neq \perp$ **then**
19: $C' \leftarrow C' \cup \{b'\}$
20: **end if**
21: **end for**
22: $C \leftarrow Merge(C, C')$
23: **end if**
24:
25: **end for**
26:
27: $C \leftarrow \{b \in C \mid G(t)\langle b \rangle\}$
28: $C \leftarrow \{b \in C \mid \forall p \in P_{In}(t) : E(p,t)\langle b \rangle \leq M(p)\}$
29:
30: **return** C

Figure 4.2: The Enabling Inference algorithm.

are members of the ordered pattern binding basis, so that they never will have to be evaluated. When merging a binding b with the binding b', in line 10 of the algorithm, the result will be $\emptyset$ if b and b' are not compatible, hence, the guard is not satisfied and the partial bindings b and b' are discarded.

The elements in the ordered pattern binding basis of a transition can be split into *binding groups* to further optimise the binding calculations. No two binding groups contain the same variable. The binding groups of a transition t can be specified as: $BG(t) = \langle B_1, \ldots, B_n \rangle$, where for any two binding groups B_i and B_j where $1 \leq i \leq n$, $1 \leq j \leq n$ and $i \neq j$, $FV(B_i) \cap FV(B_j) = \emptyset$. The free variables of ReceivePacket in Fig. 4.1 can be split into two binding groups, one containing `k`, `d` and `a`, and the other `data`. The elements in a binding group can be bound independently from other binding groups, so that finding an enabled binding is a matter of picking a partial binding obtained from each binding group, and combining them to a full binding. This will speed up the execution of automatic simulation as it limits the combinatorial blowup that may result from merging partial bindings. As described in [14], the inference algorithm can be made even faster, by only generating one partial binding from each binding group, where binding of variables is based on a random drawing of tokens from current places. This drawing is considered to be "fair", meaning that all enabled bindings have a fair chance to be picked based on the drawings. If the resulting binding is not enabled, one must backtrack and try other combinations. For interactive simulation, these optimisations cannot be performed, since all bindings must be computed, so that any enabled binding element can be picked manually to occur.

Chapter 5

Simulator Implementation

In this chapter, we provide details about the implementation of the Scala simulator for CPNs. Section 5.1 gives an overview of the architecture of CPN Tools, and outlines how the Scala simulator is integrated. Section 5.2 gives an overview of the architecture of the Scala simulator. Section 5.3 describes key concepts of the simulator implementation, and how Scala's type system has been used to accommodate the formal semantics of CPNs. Section 5.4 describes how collections can be implemented for use in CPNs, both based on extending built in collections in the Scala library, and implementing collections from scratch. Section 5.5 describes in more detail how the simulator is integrated with CPN tools, by means of how modelling constructs in CPNs have been related to object-oriented paradigms, and how the code generation of a model is performed. Section 5.6 walks through code snippets that are generated based on different modelling constructs of CPNs. Finally, Sect. 5.7 concludes this chapter with a discussion of significant decisions that have been made regarding the implementation.

5.1 Architecture of CPN Tools

CPN Tools [2] consists of two main components: A *Graphical User Interface* (GUI) and a CPN ML backend. The architecture of CPN Tools is described in [18], and Fig. 5.1 shows a simplified overview, sufficient to cover the aspects regarding how the Scala simulator can be integrated. The GUI and CPN ML run as two separate processes, communicating via TCP/IP. CPN models can be created using the GUI. Visualisation, including simulation of models can be conducted through communication with the CPN ML process. The GUI process also performs syntax checks of a model, verifying

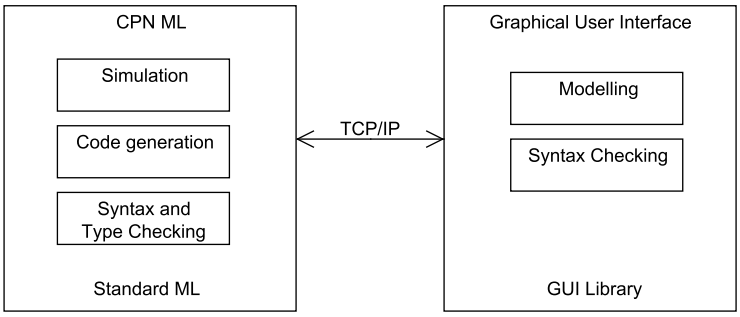


Figure 5.1: Simplified architectural overview of CPN Tools.

for instance that all places have colour set inscriptions, and that all arcs have arc expressions. Syntax and type checking of CPN ML code, for instance function definitions and arc expressions, are performed by the CPN ML process. Before a syntactical and type correct model can be simulated, the model must be transformed to SML code, by performing code generation in the CPN ML process.

Figure 5.2 shows how the Scala simulator integrates with the GUI of CPN Tools. Since CPN Tools only has a graphical interface, it is not eligible to integrate the Scala simulator into the GUI directly. A CPN model constructed using the GUI is saved using an XML format that can be parsed and used when conducting code generation for the Scala simulator. Access/CPN [34] is a framework to facilitate integration of CPN models into external applications based on their XML representation. A Java interface consisting of an object-oriented representation of models is provided. A simulator code generator uses this object-oriented model representation to transform the model to Scala simulator code. Access/CPN also interacts with the CPN ML process, allowing simulation and analysis of a model to be conducted through an external environment. However, in context of the Scala simulator, Access/CPN is only used to retrieve information based on the XML representation of CPN models.

This integration allows construction of models with CPN Tools, using Scala as inscription language. It does not allow simulation directly in the GUI. However, it makes it possible to model more complex models that can be used to evaluate Scala as inscription language, as well as the Scala simulator through code generation.

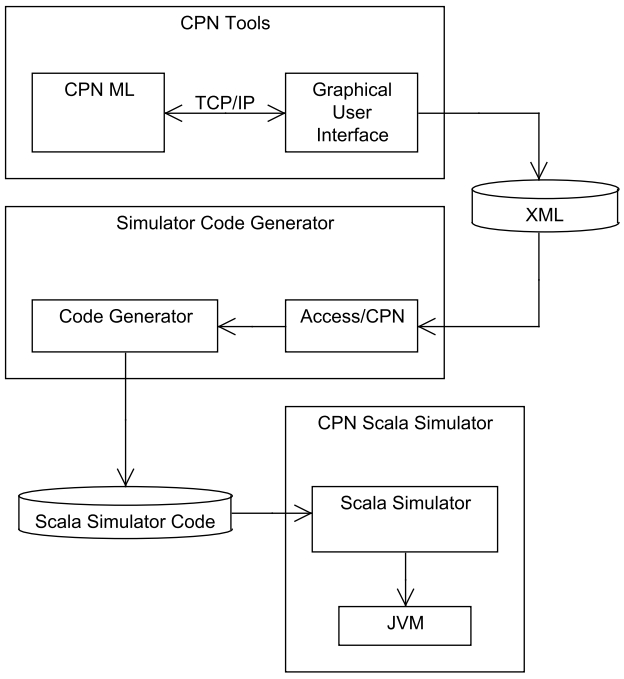


Figure 5.2: Integration of the Scala simulator with the GUI of CPN Tools.

5.2 Architecture of the Scala Simulator

The simulator is implemented based on concepts and algorithms found in [18, 23], including the enabling inference algorithm described in Chap. 4. Achieving an architecture that is flexible while retaining the design introduced in Chap. 3 has been of high priority. A flexible architecture has been accomplished by minimising the functionality of model-dependent code, and maximising functionality of model-independent code. With model-dependent code, we mean the code that is generated from a model so that it can be simulated. Model-independent code is the generic code that is used in the simulation of any CPN model. In the SML simulator, the enabling inference algorithm introduced in Chap. 4, as well as functionality to make a transition occur, is implemented for each transition in a model, and hence, this is part of the model-dependent code. This functionality has been made generic in the Scala simulator, i.e., model-independent. Thus, the functionality in model-dependent code decreases, as well as the amount of generated code.

Figure 5.3 shows a simplified class diagram of how the code generated for a CPN model is structured to conform to the Scala simulator. The diagram includes composition and inheritance, while class dependencies have been left out to improve readability. The architecture is based on a simple graph representation, `CPNGraph`, adapted to be efficient for operations used by the Scala simulator.

A `CPNGraph` maintains net elements in a CPN module. It holds reference to `Place`s, `Transition`s, `Arc`s and sub-`CPNGraph`s (corresponding to substitution transitions). When code for a CPN model is generated, an additional prime `CPNGraph` is generated. All `CPNGraph`s, representing top modules in the module hierarchy of a CPN model are added to this graph, which is used to perform the simulation. `Node` is a trait holding reference to its input and output arcs, and is extended by `Place` and `Transition`. A `Place` has a marking which is represented using some collection. This collection must mix in the `CPNCollection` trait for the generic algorithms to work. A `Transition` is dependent on a specific generated binding element, that extends the abstract class `BindingElement`. A `Transition` also holds reference to its `Guard`, which is a list that can contain multiple guard conjuncts. A concrete `Guard` is either a `BindGuard` or an `EvalGuard`. `BindGuard` mixes in the trait `PatternBindingBasisElement`, as it can be used to bind free variables. Patterns in an `Arc` are extracted to `Pattern` objects. There is no direct relation between an `Arc` and its `Pattern`s. Instead, the relationship is mapped in an `ArcPattern` that extends the trait

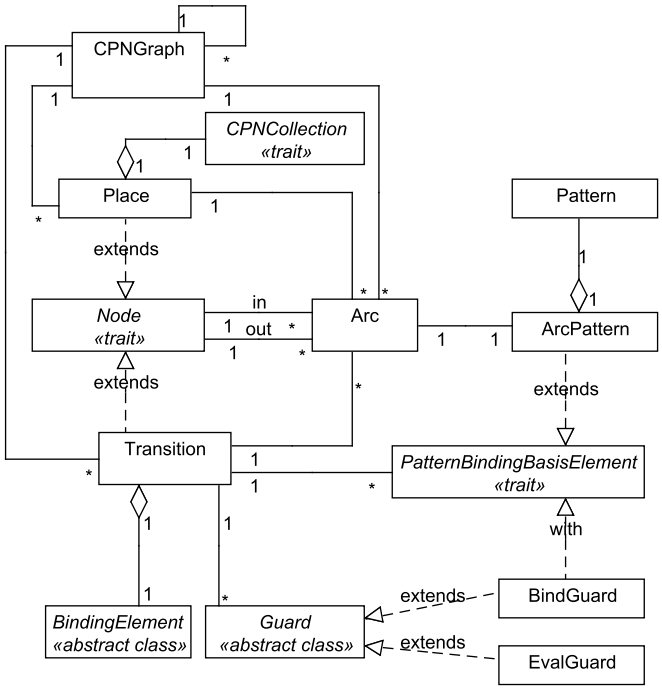


Figure 5.3: Simplified class diagram for models generated for the Scala simulator.

`PatternBindingBasisElement`. This is done to avoid coupling of patterns, so that some, but not necessarily all patterns in an arc expression can be used in the ordered pattern binding basis. Finally, a `Transition` holds reference to `PatternBindingBasisElement`s, in an ordered list, hence, the ordered pattern binding basis.

5.3 Simulator Implementation

Two simulators have been implemented. One performs interactive simulation and the other automatic simulation. The methods for running simulations take a `CPNGraph` as parameter so that elements in the graph and sub-graphs can be inspected recursively during simulation.

The interactive simulator calculates all enabled binding elements for each occurrence, and lets the user choose which binding element to occur next. The automatic simulator implements a scheduling algorithm given in [23, Chapter 2.2]. Transitions are separated into two categories: *Unknown* and *disabled*. Initially, all transitions are marked as *unknown*. During simulation, a transition is picked from those who are *unknown*. If the transition is enabled, a random binding is picked to occur. If the transition chosen to occur does not have any enabled bindings, it is marked as *disabled*. When a transition has occurred, it only affects the enabling of a limited set of transitions: If the occurrence of a transition T1 add tokens to a place P, a possible *disabled* transition T2 may become enabled if and only if there is an arc from P to T2. If this condition holds for some transition T2 that is marked as *disabled*, T2 must be marked as *unknown*. This locality principle does not hold when tokens have mutable members. This implies that in the current Scala simulator, enabling of transitions does not rely on mutable state.

The same implementation of enabling inference is used for both simulators. This causes a performance issue for the automatic simulator, since all enabled bindings for a chosen transition are calculated. As discussed in Chap. 4, the ordered pattern binding basis can be split into binding groups, and a binding element can be obtained based on binding variables in a random drawing of tokens on current input places. A second enabling inference implementation is needed for this approach. However, the simulator implementation does take binding groups into account.

The rest of this section will be used to describe key concepts of the simulator implementation in terms of Scala constructs that have been essential

to conform to the design introduced in Chap. 3 and the formal semantics of CPNs introduced in Chap. 4. Instructions to find the full source code is given in Appx. A.

5.3.1 Type Checking

As discussed in Chap. 4, CPN has a formal definition. The formal definition includes type constraints for free variables and arc expressions. It is crucial that these properties are checked at compile time. As Scala is statically typed, this boils down to using language features so that the constraints are met. The main mechanisms for obtaining type conformance are Scala's type inference and type parameterisation.

Scala applies the same erasure model for generics as Java [21, Chapter 15]. When a Scala program is compiled, it is translated into byte code, running on JVM. During translation to byte code, *type erasure* is performed, meaning that information about type arguments is erased and not available at runtime. This is done to support legacy Java code. Translation of expressions is basically done by removing generic types and adding casts to correct types [13].

Since the implementation of enabling inference and occurrence is generic, type constraints must be applied in the model structure that was given in Fig. 5.3. Listing 5.1 shows the header of the algorithm that implements enabling inference[1]. The method produces a list of enabled binding elements, given a transition of type `Transition[Be]` as parameter, where `Be` is the type of binding elements for the transition. Specific binding elements are generated for each transition. A `Transition` object holds a reference to the arcs connected to it to be able to access the marking of adjacent places. Since there exists no type information at runtime about the type of the collection used to hold tokens on adjacent places, it is for instance not possible to determine whether the evaluation of an arc expression is of correct type from the generic implementation. This means that runtime exceptions can be thrown, unless all type constraints are checked at compile time.

Listing 5.1: Header for the method implementing enabling inference.

```
def enabledBindings[Be <: BindingElement](
  transition: Transition[Be]): List[Be] = { ... }
```

[1] Syntactical details will be described in greater detail later in this chapter.

Scala's type inference is able to infer types for object instantiations and function definitions generated for a CPN. When instantiations violate the type constraints, the compiler will issue type errors. With reference to [33], we experience that Scala's type system is based on *Local Type Inference* [9], which is a type inference method particular applicable for object-oriented programming languages including subtyping and parametric polymorphism. To get a more complete type inference, Scala also supports concepts from *Colored Local Type Inference* [22]. Colored local type inference refines local type inference with the possibility to propagate partial type information when traversing the syntax tree, whereas local type inference only propagates complete type information.

With colored local type inference, Scala matches to a certain extent full type inference, as found in functional programming languages, solving types with global constraints [35]. There are some limitations where types can be inferred, compared to the full type inference found in SML. For instance, parameters of a method in Scala must be defined with types whereas in SML, *parametric polymorhpism* allows omission of type declaration in functions.

In cases where correct types cannot be inferred, for instance when instantiating an object, some techniques can be used to help the compiler infer the correct types. Some applications of this have been used in the implementation og the Scala simulator, and will be presented in the following sections.

5.3.2 Type Parameterisation

Type parameterisation [21, Chapter 19] is the mechanism allowing definition of generic classes in Scala. *Raw types* are not allowed, which means that when instantiating a generic class, the type must always be provided. In practice, however, type parameters seldom have to be provided, only when the types cannot be determined via type inference.

Type parameters can be specified with *upper bounds* and *lower bounds*. For some type `T`, `U <: T` specifies that `T` is the upper bound of `U`, i.e., the type of `U` must be equal to, or some subtype of `T`. `U >: T` specifies the lower bound, i.e., the type of `U` must be equal to or some supertype of `T`. An application of an upper bound is shown in the header of the implementation of `Arc` in Lst. 5.2. The type parameter `Be` is required to be a subtype of the abstract class `BindingElement`.

Generic types are by default *nonvariant*, meaning that some `Foo[S]` could not be used in place of `Foo[T]`, where `S` is a subtype of `T`. However, inheritance relationships can be defined by means of *variance annotations*.

Listing 5.2: Header of the class `Arc`.

```
case class Arc[Be <: BindingElement, A, Repr] (
  id: String,
  place: Place[A, Repr],
  transition: Transition[Be],
  direction: Direction.Value)
  (evalExpr: Be => Repr) {

}
```

If one would like `Foo[S]` to be a subtype of `Foo[T]`, then `Foo` must be defined with a `+` in front of the type parameter:

```
trait Foo[+T] { ... }
```

In this case, `Foo` is *covariant* in its type parameter `T`, meaning that inheritance relation for an instance of `Foo[T]` changes in a similar manner as for an instance of `T`. The inheritance relationship can also be inverted, or *contravariant*. In that case, the `+` symbol must be replaced with `-`. Then `Foo[T]` will be a subtype of `Foo[S]`.

Defining collections for CPN models covariant will in some cases simplify arc expressions. Consider an arc with the following arc expression, where the place connected to the arc has a marking of type `Queue[String]`:

```
if (success) Queue("COL", "OUR")
else Queue()
```

If `Queue` is defined as nonvariant, the type of this expression would be `Queue[_<:String]`. The type parameter of `Queue` is in this case a bounded *existential type* [21, Chapter 31.3]. The type is expanded to `Queue[T] forSome { type T <:String}`. This is because the type of `Queue()` in the `else` branch will be inferred to `Queue[Nothing]`. If `Queue` is defined as covariant, the type of the expression will be `Queue[String]`. This is because `Nothing` is a subtype of all other types, hence, `Queue[Nothing]` will be a subtype of `Queue[String]`. This is supported for `Queue` in the implementation, but not for `Multiset`. That is because elements are maintained in a `Map` internally, where the key is invariant. A token is the key, and the coefficient the value.

5.3.3 Implicits and Currying

Implicits [21, Chapter 21] allow conversion of types so that explicit details in the code can be omitted. For instance, a Java `List` can be converted to a Scala `List` using an implicit conversion, so that methods like `foreach`, `filter` and `map` can be used directly on the Java `List`. A second usage of implicits is within argument lists. An argument list can be defined as implicit, so that the compiler may insert missing parameters. For instance, the call of a function `f(a)` can be replaced with `f(a)(b)`. In that case, the parameter list `(b)` is implicit.

The function `f` given above has two parameter lists, which means `f` is *curried* [21, Chapter 9.3]. Assume the definition of another curried function `g` is `g(x)(y) = x + y`. Application of this function gives a chain of function calls. For instance, `g` can be applied with `g(2)`, resulting in a function `h(y) = 2 + y`. Finally, this function can be applied with `h(3)`, which gives the result `5`.

As discussed in Chap. 3, arc expressions should be allowed to evaluate to a single token. This special case violates the type constraints of CPNs. However, this can be solved in Scala by using implicit conversions. As shown in Lst. 5.2, an `Arc` takes three type parameters. `Be` represents the binding element type for the transition connected to the arc. `A` and `Repr` represent the token type and the collection type of the place connected to the arc, respectively. The last parameter of the class `Arc`, `evalExpr`, is a function that evaluates the arc expression in a binding element of type `Be` given as parameter. The result type of the function must be `Repr`. When an arc expression consists of a single token, the function to evaluate the arc expression will be of type `Be => A`. Hence, for this to compile, there must be an implicit conversion from `Be => A` to `Be => Repr`. Listing 5.3 shows how this conversion is implemented for `Queue`. It converts a function with a parameter with the type `Be` of some subtype of `BindingElement`, and some arbitrary type `A` as result type, by wrapping the result of the function in a `Queue`.

Listing 5.3: Implicit conversion from type `Be => A` to `Be => Queue[A]`.

```
implicit def evalQueueToken[Be <: BindingElement, A](fn: (Be => A)) =
  (b: Be) => Queue(fn(b))
```

In some cases, evaluation of an arc expression results in multiple tokens, commonly by using functions to avoid large arc expressions. When an arc

expression consists of multiple tokens with CPN ML, they are represented in a multiset. When allowing different collection types for place markings, evaluation of arc expressions should be more general in terms of collection type. For instance, when defining a method to return a specific collection type, the method may not always be applicable as it is restricted to be used in a context where the place has the same collection type. By using implicit conversions, one can allow an arc expression to evaluate to a list of tokens. Similar to the previous example, an implicit conversion from `Be => List[A]` to `Be => Repr` is needed. Listing 5.4 shows how this conversion is implemented for `Queue`. It converts a function with a parameter with the type `Be` of some subtype of `BindingElement`, and the result type `List[A]` where `A` is some arbitrary type, by wrapping the result of the function in a `Queue`. The function call `fn(b)` must be marked as a *sequence argument* [26, Chapter 4.6.2] by the `_*` annotation, passing each element in the list obtained by evaluating `fn(b)` as parameters to `Queue`.

Listing 5.4: Implicit conversion from type `Be => List[A]` to `Be => Queue[A]`.

```
implicit def evalQueueTokenList[Be <: BindingElement, A](
  fn: (Be => List[A])) =
  (b: Be) => Queue(fn(b):_*)
```

As shown in Lst. 5.2, the constructor of `Arc` is curried. The argument `evalExpr: Be => Repr` is in a second parameter list. If the constructor of `Arc` was not curried, it wold not compile in cases where the instantiation of an `Arc` where the type of the provided function for evaluating the arc expression is of type `Be => A` or `Be => List[A]`. In this case, the type inference mechanism does not check whether there is an implicit conversion of the function, since former arguments do not have precedence over proceeding arguments in the same parameter list. According to [32, Chapter 7.2.3], the type inference works from left to right, across parameter lists. Using currying, the type information for the arc is inferred from the first parameter list, and implicit conversions are searched when the type of the function in the second parameter list does not match.

As with parameter lists, Scala may not always be able to infer all type parameters. Consider the class `Place`, defined with the following header:

```
case class Place[A, Repr <: Traversable[A]] (
  id: String, name: String, initialMarking: Repr) { ... }
```

The type `A` represents the type of tokens a `Place` can hold, and `Repr` the collection type. However, when instantiating a `Place`, the type system cannot find the types of both `A` and `Repr` using only one parameter. This can be solved using a *reified type constraint* [32, Chapter 7.2.3]. Reified type constraints are objects that can be given as implicit parameters, verifying that some type constraint holds. Listing 5.5 shows how the header of the final implementation of the class `Place`, where a reified type constraint is used so that the type parameters `A` and `Repr` can be inferred correctly. Now, some of the type inference is deferred to the second parameter list by the implicit evidence `ev0: Repr <:< Traversable[A]`. The `<:<` type is the reification for upper bound, hence, it must be true that `Repr <: Traversable[A]`. `Traversable` is the top class of the Scala collection hierarchy, which will be explained in more detail in the next section.

Listing 5.5: Header of the class `Place`.

```
case class Place[A, Repr](
  id: String,
  name: String,
  initialMarking: Repr)(
    implicit ev0: Repr <:< Traversable[A],
    ev1: Repr => CPNCollection[Repr]) { ... }
```

The implicit evidence `ev0` requires `Repr` to be a subtype of `Traversable[A]`. A collection used for a place must be able to perform some additional methods, defined in the a trait `CPNCollection`, which will be discussed in the next section. However, it is not strictly required to be a subtype of `CPNCollection[Repr]`. For the parameter `initialMarking` of type `Repr`, there must be an implicit conversion at call site, converting `Repr` to `CPNCollection[Repr]`. This is called a *view bound* [32, Chapter 7.1], and is achieved with the implicit evidence `ev1: Repr => CPNCollection[Repr]`. This allows more flexibility of how collections for place markings can be implemented, which will be explained in the following section.

5.4 Implementing Collections

Scala's collection library [21, Chapter 24 and 25] underpins the functional aspects of the language. The functional focus allows working on collection on an higher abstraction level, avoiding erroneous lower-level for-loops as in imperative programming. Additionally, hiding implementation details will

in most general cases result in more efficient code. The collection framework was redesigned in Scala 2.8, to accommodate a common, uniform hierarchy for collection types. The most crucial motivation for this, addressed by Moors, Piessens and Odersky [27], was to avoid loss of coherence and clarity when the collection API evolves. They argue that collections share a lot of common functionality that must be extracted so that it can be used uniformly to avoid loss of consistency due to duplicate implementations. They proposed the new framework, constituting a number of building blocks that can be used to integrate new collections without redefining common operations. To achieve these goals, polymorphism in the form of *higher-kinded types* [8], and implicit parameters and conversions, are used.

Figure 5.4 depicts the most essential top classes of the Scala collection hierarchy. These are all high level abstractions, having both mutable and immutable implementations. Classes in the hierarchy support all methods of their successors. To avoid code duplication downwards in the hierarchy, reusable functionality is implemented in *implementation traits*, which are named with a `Like` suffix. For instance, the implementation trait for `LinearSeq` is `LinearSeqLike`.

The generic algorithms for enabling inference and occurrence of transitions will not allow using built-in Scala collections directly. It would be convenient to be able to use any collection which is a subtype of the `TraversableLike` trait, however, methods for comparing (`>>=`) two collections and subtract (`--`) a collection from another are needed. The operations required for a collection to interoperate with the Scala simulator are defined in a trait `CPNCollection`, shown in Lst. 5.6. The trait `CPNCollection`

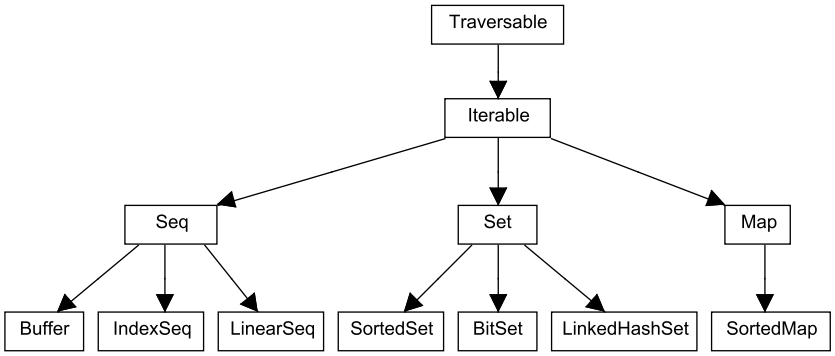


Figure 5.4: Top classes of the Scala collection hierarchy.

takes a type parameter, `Repr`, representing the collection type. An additional method `+++`, for adding two collections, is also required. This method should do the same as the `++` method defined in the `TraversableLike` implementation trait. The reason why the `++` method cannot be used, is that when adding tokens to a place, it is strictly required that the tokens are contained in a collection with equal type as the place marking, so that the collection type is preserved. Using the `++` method may result in a less specific collection type, as it relies on implicit conversions to find the result type. This will be explained in more details later in this chapter. It should be noted that it may be possible to utilise the type system so that using `++` is sufficient, but this has not been accomplished. The reason why the method `+++` is not named `++` is because it will conflict when a class mix in both `TraversableLike` and `CPNCollection`. The methods `+++` and `++` have the same type after type erasure, hence, the compiler would not be able to tell them apart if they were named equally.

Listing 5.6: Implementation of the trait `CPNCollection`.

```
trait CPNCollection[Repr] {
  def >>=(that: Repr): Boolean
  def --(that: Repr): Repr
  def +++(that: Repr): Repr
}
```

There are two approaches how to implement collections that can be used for the Scala simulator. Utilising the building blocks of Scala's collection framework makes it easy to implement new collections from scratch. Further, existing Scala collections can be extended with the functionality needed to interoperate with the Scala simulator.

Section 5.4.1 and Sect. 5.4.2 describe how higher-kinded types and implicit parameters are used to give a flexible architecture of Scala's collection API, respectively. Section 5.4.3 describes how new collections for CPN can be implemented, and 5.4.4 accounts for existing collections in Scala's collection library can be extended to interoperate with CPNs.

5.4.1 The Use of Higher-Kinded Types in Scala's Collection Library

Higher-kinded types are the "... generalisation to types that abstract over types that abstract over types..." [8]. One of the applications for higher-

kinded types are in collections. This is explained by Moors, Piessens and Odersky [8] with the following example, considering the method `remove` in the trait `Iterable` [2]:

```
trait Iterable[T] {
  def filter(p: T => Boolean): Iterable[T]
  def remove(p: T => Boolean): Iterable[T] = filter (x => !p(x))
}
```

Subclasses of `Iterable` implementing `filter` will get the predefined implementation of `remove`. However, the result type of the methods will be `Iterable[T]`, not the more specific result that is wanted, for instance `List[T]` for the implementation of `List`. Using higher-kinded types, this can be achieved:

```
trait Iterable[T, Container[X]] {
  def filter(p: T => Boolean): Container[T]
  def remove(p: T => Boolean): Container[T] = filter(x => !p(x))
}
```

Here, `Iterable` takes two type parameters; `T` (representing the type of the elements) and `Container` (representing the resulting collection of the different methods). The `Container` type has a type parameter itself. It is different from the type of elements, `T`, since some methods, like `map`, can return elements of different type. Now, a specific collection can utilise the predefined functionality of `Iterable`, with the most specific result type:

```
trait List[T] extends Iterable[T, List] {
  def filter(p: T => Boolean): List[T] = ...
}
```

The trait `List` inherits from `Iterable` where the type parameter representing the container for `Iterable` is `List` itself.

5.4.2 The Use of Implicits in Scala's Collection Library

Implicit parameters are used repeatedly in Scala's collection framework, to be able to implement generic methods, while retaining the *most specific* result types. An application of an implicit parameter is for the implementation of `map`, defined in the `TraversableLike` trait, shown in Lst. 5.7. A *builder*

[2] This is a simplified example, and do not reflect the actual implementation of `Iterable` in Scala.

Listing 5.7: Implementation of the method `map` defined in the trait `TraversableLike`.

```
def map[B, That](f: A => B)
    (implicit bf: CanBuildFrom[Repr, B, That]): That = {
  val b = bf(repr)
  for (x <- self)
    if (p(x)) b += f(x)
  b.result
}
```

factory of type `CanBuildFrom` is provided as an implicit parameter. Builders will be explained in greater detail in the next section. For now, it suffices to know that given the builder factory, `bf`, it must be possible to build a collection of type `That` with elements of type `B` from a collection of type `Repr` with elements of type `A`. `That` will be the *most specific* type possible to build. This gives flexibility to abstract over, and change collection type. For instance, calling `map` on a `BitSet` could result in a `Set`. To get the most specific result from calling functions like `map`, collections need to provide an implicit builder factory.

5.4.3 Implementing New Collections

By reusing building blocks of the collection framework, new collections can be integrated with a reasonable small amount of code. Listing 5.8 shows an implementation of a multiset adapted to be used in CPNs. The class `Multiset` takes a parameter type, `T`, for the element type of the collection, and mixes in `CPNCollection`, and `Iterable` and its implementation trait `IterableLike`. `CPNCollection` is parameterised with the collection type `Multiset[T]`. `Iterable` is parameterised with the type of elements that are returned when iterating over the collection. For `Multiset`, it is a tuple `(Int, T)`, where the integer represents the coefficient of an element, and T the actual element. `IterableLike` takes two type parameters: the type of element returned when iterating over the collection, and the collection type. `Multiset` has a private parametric field `multiset: Map[T,Int]`, which is used internally to maintain objects. The key is of type `T`, and the value of type `Int`, representing the element coefficient.

`Multiset` must implement the required methods defined in `CPNCollection`, namely `>>=`, `+++` and `--`. Further, the abstract method `iterator` from the trait `IterableLike` must be implemented. This is the only abstract

Listing 5.8: Implementation of the class `Multiset`.

```
class Multiset[T] private (private val multiset: Map[T, Int])
  extends CPNCollection[Multiset[T]] with Iterable[(Int, T)]
  with IterableLike[(Int, T), Multiset[T]] {

  def >>=(that: Multiset[T]): Boolean = {
    // Implementation...
  }

  def +++(that: Multiset[T]) = this ++ other

  def --(that: Multiset[T]): Multiset[T] = {
    // Implementation...
  }

  def iterator: Iterator[(Int, T)] =
    for ((el, no) <- multiset.iterator) yield (no, el)

  override protected[this] def newBuilder: Builder[(Int, T), Multiset[T]] =
    Multiset.newBuilder[T]
}
```

method in `IterableLike`, all other methods are inherited. The `iterator` method specifies how elements of the `Multiset` are iterated, and the type must match the specified type arguments for `Iterable` and `IterableLike`.

The reason why it is necessary to mix in `IterableLike`, not only `Iterable`, is to get a more specific return type for functions such as `take` and `filter`, namely the second type parameter of `IterableLike`. For this to work, the method `newBuilder`, returning a `Builder[(Int, T), Multiset[T]]` must be implemented. `IterableLike` is basing generic methods on the `newBuilder` abstraction, which is defined in the companion object depicted in Lst. 5.9. The `newBuilder` method specifies how to build a `Multiset` with elements of type `T`. A `Builder[(Int, T), Multiset[T]]` is returned by creating a new `ArrayBuffer`, and use `mapResult`, taking a *transformer function* as argument, in this case `fromSeq`, which transforms a sequence of elements with type `(Int, T)` to a `Multiset[T]`.

Application of some methods may result in a different collection, for instance `++` and `map`. For making the most specific collection type to be returned from such methods, an implicit method, `canBuildFrom`, must be defined in the companion object. This method constructs a `CanBuildFrom[Multiset[_], (Int, T), Multiset[T]]` object, defining that one can

Listing 5.9: Implementation of `Multiset`'s companion object.

```
object Multiset {

  def fromSeq[T](buf: Seq[(Int, T)]): Multiset[T] = {
    // Building a Multiset from a Sequence
  }

  def apply[T](elems: T*)(implicit d: DummyImplicit) = {
    var map = Map[T, Int]()
    elems.foreach(el => map += (el -> 1))
    new Multiset(map)
  }

  def apply[T](elems: (Int, T)*)(implicit d1: DummyImplicit, d2:
      DummyImplicit) = fromSeq(elems)

  def newBuilder[T]: Builder[(Int, T), Multiset[T]] =
    new ArrayBuffer[(Int, T)] mapResult fromSeq

  implicit def canBuildFrom[T]:
    CanBuildFrom[Multiset[_], (Int, T), Multiset[T]] =
    new CanBuildFrom[Multiset[_], (Int, T), Multiset[T]] {
      def apply() = newBuilder[T]
      def apply(from: Multiset[_]) = newBuilder[T]
    }
}
```

build a `Multiset[T]` from a `Multiset[_]` with any type of element, to a multiset with elements of type `T`, and the iterable type `(Int, T)`. The `CanBuildFrom` object is defined with two `apply` methods, creating new builders for `Multiset`.

The constructor of the class `Multiset` is private so that users cannot construct a `Multiset` using the `new` keyword. Instead, two `apply` methods are defined in the companion object, as discussed in Chap. 3.3. One allows creation of a `Multiset` with objects of type `T`, hence, without coefficient. A `Multiset` instantiated with elements of type `(Int,T)` will be handled by the second `apply` method, where the first component of the tuple will be interpreted as the multiset coefficient.

5.4.4 Extending a Collection

The alternative to defining collections from scratch is allowing collections in the Scala library to be used, by extending the functionality of a given collection using implicit conversions. Implicits allow changing and extending functionality of libraries, using the "pimp my library" pattern [25]. This pattern is used to wrap code in libraries that cannot be changed, to make them easier to work with or to extend the functionality.

Listing 5.10 depicts how this pattern can be used to extend Scala collection `immutable.Queue` to accommodate the functionality required for the Scala simulator. The class `RichQueue` takes a parameter `x` of type `Queue` as parameter. It inherits from `CPNCollection`, and implements the required methods in terms of `x`. An implicit method `richQueue` is defined, which converts a `Queue` to a `RichQueue`. When calling, e.g., `Queue(1,2,3) >>= Queue(1,2)`, the compiler sees a type error. Then it tries to find an implicit conversion of `Queue` that will resolve the error. In this case `Queue(1,2,3)` is converted to `GtQueue(Queue(1,2,3))`, which has a `>>=` method, taking another `Queue` as parameter, and the method is performed on two queues.

Listing 5.10: Implementation of `RichQueue`, extending the functionality of `Queue` using the "pimp my library" pattern.

```
class RichQueue[A](x: Queue[A]) extends CPNCollection[Queue[A]] {
  def >>=(y: Queue[A]): Boolean = {
    // Implementation...
  }

  def +++(y: Queue[A]) = x ++ y

  def --(y: Queue[A]) = {
    // Implementation...
  }
}

implicit def richQueue[A](x: Queue[A]): RichQueue[A] = new RichQueue[A](x)
```

5.5 Integration of the Scala Simulator

This section provides a description of how a CPN model is processed, from the XML representation obtained from CPN Tools, to the point where information required to generate code for the Scala simulator is obtained.

Syntactical analysis of Scala expressions has been made possible using reflection, which will be explained.

5.5.1 Relating CPNs to Scala

To get a better intuition of how a CPN model can be expressed in Scala code, we have proposed a conceptual relation between the hierarchical concepts of CPNs and object-oriented concepts. The relations are summarised in Tab. 5.1.

Hierarchical CPN	**Object-orientation**
Module	Class
Port place	Class constructor parameter
Substitution transition	Class instantiation
Socket place	Instantiation parameter
Fusion group	(Global) Object

Table 5.1: Relation between hierarchical concepts of CPNs and object-oriented concepts.

A module can be related to a class in Scala, where the port places in the module are parameters to the class (the constructor of the class). A substitution transition in a module can be related to instantiation of a class, where the socket places connected to the substitution transition are given as parameters. All fusion places that are member of a given fusion group can be merged to a compound place, represented as an object which is accessible in the scope of all module classes.

5.5.2 Syntactical Analysis of Scala Expressions

Access/CPN is used to get an object representation of a CPN model. This representation is transformed to an *intermediate* representation which maps the concepts summarised in Tab. 5.1 more explicitly. The intermediate representation is used to perform code generation for the Scala simulator. Before this can be done, inscriptions and declarations of a model must be inspected to find the following data:

1. Free variables and their types.
2. Declarations of type `Enumeration` and their possible values.
3. Guard conjuncts of a guard, their types and free variables.

4. Patterns and free variables in arc expressions.

As discussed in Chap. 3.2.4, Scala should not be extended with new syntax. However, the prototype of a Scala simulator proposed with this thesis requires that free variables are defined with a type (step 1). The syntax for defining a CPN variable is `cpnvar <name> : <type>`.

For a free variable *fv* of a transition with an `Enumeration` type, partial bindings are pre-generated, where *fv* is bound to the possible values of the enumeration type. Therefore, declarations inheriting from `Enumeration` and their values must be found (step 2).

The guard of a transition may consist of multiple guards conjuncts. A guard conjunct must be inspected (step 3) to determine which type of guard it is (i.e. whether it binds variables or not), and to find the free variables appearing in it.

Free variables of an arc expression must be found (step 4). If the arc expression is on an input arc, it may contain one or more patterns. For an arc expression to have multiple patterns, they must be parameterised in a collection constructor, matching the collection on the connected place. All other expressions will be interpreted as evaluating to a single token. Determining whether an expression `expr` is a valid pattern in Scala, can be done by evaluating the following Scala code:

```
def pattern(x: Any) = x match {
  case <expr> =>
}
```

If the function can be compiled without errors, the expression is a valid pattern.

The steps introduced above require the ability to inspect Scala code, which is possible using reflection, described in Sect. 5.5.4.

5.5.3 Code Generation

Some calculations based on the information that has been obtained are required before the code generation can be performed:

1. Find free variables of a transition.
2. Calculate the binding groups for a transition.
3. Calculate the ordered pattern binding basis for a transition.

Based on the guard of a transition and arcs connected to it, the set of free variables of the transition (step 1) can be found and divided into binding groups (step 2). The problem of computing the ordered pattern binding basis (step 3) can be reduced to the NP-complete set-covering problem. The SML simulator solves this by applying a greedy algorithm [18, Section 4.2]. This is also done for the Scala simulator. When computing the ordered pattern binding basis, the pattern from an arc expressions among those not yet included, containing the most free variables are picked. Binding guards have a higher precedence to be picked, as long as all free variables on the right hand side are covered.

Generating code for the Scala simulator is now a matter of using the information obtained in structured code templates, and writing the result to a Scala file.

5.5.4 Reflection in Scala

The syntactic and semantic analysis that is conducted on a model, described in Sect. 5.5.2, is done by using *runtime reflection* [4]. Scala's reflection library was introduced in Scala 2.10 [3] and is still experimental. Prior versions of Scala had only the reflection capabilities found in Java. The new reflection library provides complete support for Scala language, and it includes *reification*, which makes it possible to generate abstract syntax trees. This includes parsing of strings to abstract syntax trees. Further, an abstract syntax tree can be inspected and manipulated.

5.6 Generated Code for CPN Models

To simulate a CPN model, the model must be transformed into Scala code that conforms to the model-independent part used to execute the simulation. This section will be used to walk through some of the constructs that are generated based on the version of the protocol example from Chap. 3 that is split into several modules. Code listings in this section will be based on net elements from the Top module shown in Fig. 5.5 (previously shown in Fig. 3.11), and the Receiver module shown in Fig. 5.6. The complete model and generated code are provided in Appx. B.

Listing 5.11 shows the header of the class generated for the Receiver module. The class takes three parameters of type `Place`, one for each of the port places B, C and Received in the module. Three type parameters, `T0`,

[3] Scala 2.10 is the latest version as writing.

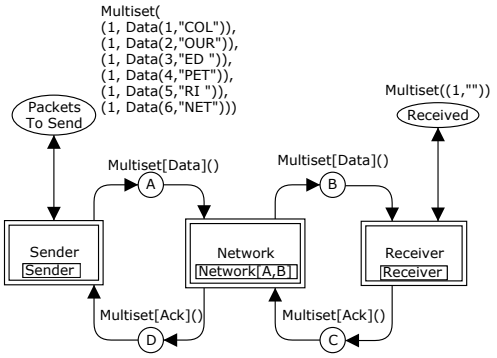


Figure 5.5: Top module of the hierarchical version of the protocol example.

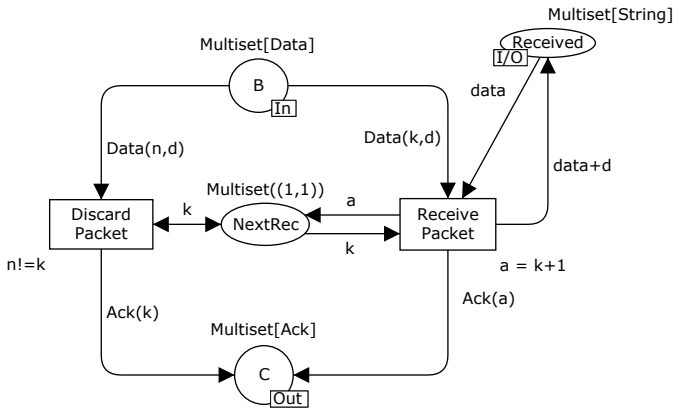


Figure 5.6: Receiver module of the hierarchical version of the protocol example.

T1 and T2, have been generated for the token types of the places. Further, to be able to infer the token types, the implicit view bounds ev0, ev1 and ev2 must be provided.

Listing 5.11: Header of the module class Receiver.

```
class Receiver[T0, T1, T2](
  ID1470723480: Place[T2, Multiset[Data]],
  ID1470725592: Place[T1, Multiset[Ack]],
  ID1470734702: Place[T0, Multiset[String]])(
    implicit ev0: Multiset[String] => Traversable[T0],
    ev1: Multiset[Ack] => Traversable[T1],
    ev2: Multiset[Data] => Traversable[T2]){ ... }
```

For each substitution transition in a module, a new instance of the module it represents is created. Listing 5.12 shows the instantiation for the substitution transition Receiver in the class generated for the Top module. The class Receiver is parameterised with the identifiers to the Place objects representing socket places in the Top module, correlating to the port places in the Receiver class.

Listing 5.12: Object instantiation of the Receiver class in the class Top.

```
val st0 = new Receiver(id87, id93, id71)
```

For each place, a Place object is instantiated. Listing 5.13 shows the code generated for the place B in the Top class. The Place takes the internal id ("id87"), the name ("B") and the initial marking (Multiset[Data]()) of B as parameters, and assigns the object to an identifier named by the internal id.

Listing 5.13: Object instantiation of the place B.

```
val id87 = Place("id87", "B", Multiset[Data]())
```

Listing 5.14 shows code generated to represent the binding element for the transition ReceivePacket in the Receiver module. For each transition, a class representing the binding element with free variables of the transition is generated. All binding element classes are subclasses of a common abstract class BindingElement. Each free variable of a transition is added as a parametric field to the binding element class, with the type of the free variable encapsulated in the standard Scala type Option. An Option[T] value can be of the form Some(x: T), where x is the actual value with the type T, or the None object, representing a missing value. This allows binding elements

to be partial, which is essential for the inference algorithm as discussed in Chap. 4.

Listing 5.14: Binding element class for the transition ReceivePacket.

```
case class BindingElement_ID1470725136(
  d: Option[String],
  k: Option[Int],
  a: Option[Int],
  data: Option[String]) extends BindingElement
```

Functions to check whether two binding elements are compatible and merging two partial binding elements are generated according the definitions given in Chap. 4. A function to determine whether a binding element is full is also generated. Listing 5.15 shows how these functions are defined for ReceivePacket. The function to determine whether a binding element is full (`fullBindingElement_ID1470725136`) takes a binding element as parameter and evaluates to true if none of the free variables of the binding element have the value `None`. The function to determine whether two binding elements are compatible (`compatible_ID1470725136`) takes two binding elements as parameter, and evaluates to true if for all free variables *fv*, the value of *fv* must be either `None` in at least one of the binding elements, or `Some(<value>)` in both binding elements, where `value` is equal in both binding elements. The function for merging two binding elements (`merge_ID1470725136`) takes two binding elements as parameters and evaluates to a new binding element where free variables having a value in any of the two binding elements, i.e. `Some(value)`, are used. Since the merge function takes two single binding elements as parameters, the function corresponds to the combine function introduced in Chap. 4. Merging of two sets of binding elements is implemented generically in the simulator.

A function is generated for the guard of a transition. If the guard binds a free variable to a value, i.e. is on the form `x = expr`, a function to perform the binding of `x` to `expr` is generated. For guards on other forms, a function to evaluate the guard is generated. This function is given as parameter for instantiation of a `BindGuard` in case the guard binds a variable, or as parameter for instantiation of an `EvalGuard` otherwise. `BindGuard` and `EvalGuard` also take a second parameter, an integer, denoting the binding group of the guard. If the guard is on conjunctive form, a function is generated for each guard conjunct.

Listing 5.16 shows the code generated for the guard, `a = k+1`, of Receive-

Listing 5.15: Functions for the transition SendPacket.

```
//full binding element
val fullBindingElement_ID1470725136 = (be: BindingElement_ID1470725136) =>
  be.d != None &&
    be.k != None &&
    be.a != None &&
    be.data != None

// compatible binding elements
val compatible_ID1470725136 = (b1: BindingElement_ID1470725136, b2:
    BindingElement_ID1470725136) =>
  (b1.d == b2.d || b1.d == None || b2.d == None) &&
    (b1.k == b2.k || b1.k == None || b2.k == None) &&
    (b1.a == b2.a || b1.a == None || b2.a == None) &&
    (b1.data == b2.data || b1.data == None || b2.data == None)

// merge binding elements
val merge_ID1470725136 = (b1: BindingElement_ID1470725136, b2:
    BindingElement_ID1470725136) => {
  val d = if (b1.d == None) b2.d else b1.d
  val k = if (b1.k == None) b2.k else b1.k
  val a = if (b1.a == None) b2.a else b1.a
  val data = if (b1.data == None) b2.data else b1.data
  new BindingElement_ID1470725136(d, k, a, data)
}
```

Packet. A `BindGuard` is generated, and assigned to `guard_ID1470725136_0`. The bind function given as parameter takes a binding element `be` as parameter, and evaluates to a new binding element where the free variable `a` is bound to the value of `be.k+1` if `be.k` is bound to a value, i.e. is `Some(<value>)`. Since a guard can be on conjunctive form, several guard objects can be created. All these are added to a list. The guard of ReceivePacket is added to a list referenced by the identifier `guard_ID1470725136`.

If the transition has free variables of type `Boolean` or of some subtype of `Enumeration`, partial binding elements will be generated and added to a list, where all possible values of these types are assigned to the respective free variables. In case of ReceivePacket, there are no free variables of such types, and the list `enumBindings_ID1470725136`, shown in Lst. 5.16, is empty.

Listing 5.17 shows code generated to instantiate a `Transition` object for ReceivePacket. The object is instantiated with the internal id (`"ID14707-25136"`) and name (`"Receive Packet"`) of the transition, the number of binding groups (2), the compatible (`compatible_ID1470725136`), merge

Listing 5.16: Guard and pre-generated partial binding elements for the transition ReceivePacket.

```
val guard_ID1470725136_0 = BindGuard((be: BindingElement_ID1470725136) =>
  (be.k) match {
    case (Some(k)) =>
      val a = k.$plus(1)
      BindingElement_ID1470725136(be.d, be.k, Some(a), be.data)
    }, 1)

// List of all guard conjuncts
val guard_ID1470725136 = List(guard_ID1470725136_0)

// List of partial binding elements where free variables of type
// 'Boolean' or a subtype of 'Enumeration' are bound to values
val enumBindings_ID1470725136 = List()
```

(`merge_ID1470725136`) and full binding element (`fullBindingElement_ID1470725136`) functions, and the lists of guards conjuncts (`guard_ID1470725136`) and pre-generated partial binding elements (`enumBindings_ID1470725136`), as parameters. The `Transition` object is assigned to an identifier named by the internal id.

Listing 5.17: Object instantiation of the transition ReceivePacket.

```
val ID1470725136 = Transition(
  "ID1470725136",            // transition id
  "Receive Packet",          // transition name
  2,                         // number of binding groups
  compatible_ID1470725136,
  merge_ID1470725136,
  fullBindingElement_ID1470725136,
  guard_ID1470725136,
  enumBindings_ID1470725136)
```

An arc expression may have zero or more patterns for binding tokens. For each pattern, a function taking a token as parameter is generated. If a token given as input matches the pattern, the function will evaluate to a partial binding element where the free variables appearing in the pattern are bound to values from the token value. The function generated for a pattern is given as parameter to a `Pattern` object instantiation along with an integer denoting the binding group of the pattern. A function is also

generated to evaluate the arc expression based on values of free variables in a binding element given as parameter. Listing 5.18 shows the generated pattern function (`pattern_ID1470726893_0`) and the function for evaluating the arc expression (`eval_ID1470726893`) for the arc from B to ReceivePacket. Shown from the pattern that is generated, the coefficient is replaced with a wildcard (_). If the coefficient is a constant c, then it will not match elements where the coefficient is greater than c. This is a special case for `Multiset`, which is handled during code generation.

Listing 5.18 also shows the object instantiation of an `Arc`, assigned to the identifier `ID1470726893`, named by the internal id of the arc. The arc is instantiated with the internal id (`"ID1470726893"`) of the arc, the identifier of the connected place (`ID1470723480`), the identifier of the connected transition (`ID1470725136`), and the function to evaluate the arc expression (`eval_ID1470726893`), as parameters. Note that since B is a port place, reference to the place is given from a parameter in the module class `Receiver`, shown in Lst. 5.11.

Listing 5.18: Generated code for the arc from B to ReceivePacket.

```
val pattern_ID1470726893_0 = Pattern((token: Any) => token match {
  case Tuple2(_, Data((k: Int), (d: String))) =>
    BindingElement_ID1470725136(Some(d), Some(k), None, None)
}, 1)

val eval_ID1470726893 = (be: BindingElement_ID1470725136) =>
  (be.k, be.d) match {
    case (Some(k), Some(d)) =>
      Data(k, d)
  }

val ID1470726893 = Arc("ID1470726893", ID1470723480, ID1470725136, In)(
  eval_ID1470726893)
```

When a transition and all the arcs connected to it are generated, the ordered pattern binding basis for the transition can be set. The ordered pattern binding basis for a transition is represented as a list, containing elements of type `BindGuard` and `ArcPattern`. An `ArcPattern` object holds reference to a pattern, and the arc the pattern belongs to. Listing 5.19 shows how the ordered pattern binding basis is set for ReceivePacket.

Listing 5.20 shows the overall structure of the generated code, including how module classes are wired together as well as the generated code for the net elements shown above. Parameters of module classes have been

Listing 5.19: Setting the ordered pattern binding basis for the transition ReceivePacket.

```
ID1470725136.orderedPatternBindingBasis = List(
  ArcPattern(ID1470737284, pattern_ID1470737284_0),
  ArcPattern(ID1470726893, pattern_ID1470726893_0),
  guard_ID1470725136_0)
```

left out for better readability. Each module instantiates a `CPNGraph` object referenced by identifiers named `net`, where all places, transitions, arcs, and substitution transitions of the module are maintained. The object representing the place B is added to the `net` of the class `Top` (`net.addPlace(id87)`). The object representing the transition ReceivePacket to the `net` of the class `Receiver` (`net.addTransition(ID1470725136)`), as well as the object representing the arc from B to ReceivePacket (`net.addArc(ID1470726893)`). The `Network` class instantiates two `Transmit` objects and adds them to its `net`, representing the two substitution transitions of Transmit in the Network module. Equally, the `Top` class instantiates and adds a `Receiver` object, a `Network` object, and a `Sender` object to its `net`, representing the substitution transitions of Receiver, Network and Sender in the Top module, respectively.

The classes representing modules are inner classes of an object, `Simple-ProtocolHier`, representing the global scope of the model. All declarations made for the model are inserted directly in this object, as well as `Place` objects representing fusion groups. A prime `net` is also instantiated directly in this object. All classes representing prime modules, i.e., modules that are not a submodule of any other modules, are added to this `net`. Further this `net` is used to run the simulation from the method `run`, defined in an object `Simulator` (`Simulator.run(net)`). In this way, the scheduler algorithm implemented in the `Simulator` object can traverse the prime `net` and all sub-`net`s.

Listing 5.20: Wiring of modules for the hierarchical protocol example.

```
object SimpleProtocolHier extends App {
  val net = CPNGraph()                        // Prime net
  ...
  class Top(){
    val net = CPNGraph()                      // Top net
    ...
    net.addPlace(id87)                        // Add Place B
    val st0 = new Receiver(...)
    net.addSubstitutionTransition(st0.net) // Add Receiver object
    val st1 = new Network(...)
    net.addSubstitutionTransition(st1.net) // Add Network object
    val st2 = new Sender(...)
    net.addSubstitutionTransition(st2.net) // Add Sender object
  }
  class Transmit[...](...)(...){
    val net = CPNGraph()                      // Transmit net
    ...
  }
  class Sender[...](...)(...){
    val net = CPNGraph()                      // Sender net
    ...
  }
  class Receiver[...](...)(...){
    val net = CPNGraph()                // Receiver net
    ...
    net.addTransition(ID1470725136)     // Add Transition ReceivePacket
    net.addArc(ID1470726893)            // Add arc B ---> ReceivePacket
  }
  class Network[...](...)(...){
    ...
    val net = CPNGraph()                      // Network net
    val st0 = new Transmit(...)
    net.addSubstitutionTransition(st0.net) // Add Transmit object
    val st1 = new Transmit(...)
    net.addSubstitutionTransition(st1.net) // Add Transmit object
  }
  val top0 = new Top()
  net.addSubstitutionTransition(top0.net)  // Add Top object
  Simulator.run(net)                        // Run simulation
}
```

5.7 Discussion

Reaching the final implementation of the Scala simulator, that has been presented in this chapter, is the result of research where different alternative solutions have been investigated. We end this chapter by discussing the most significant decisions that have been made.

We have described how Scala's type system has been utilised to perform all necessary type checking at compile time. In this way, the type erasure done when Scala code is compiled to byte code, does not pose big problems. The price paid because of this, is that the method `+++` must be defined for collections that can be used for CPNs. We have investigated the possibility to retain type information at runtime using *manifests*[32, Chapter 7.2.1], combined with type safe *heterogenous typed lists*[32, Chapter 7.4.1]. However, this approach would complicate the implementation, and add runtime overhead, for a fairly small gain. We argue that further investigation of how the type system can be used so that the `+++` method can be omitted is a better solution, mainly to avoid the runtime overhead that manifests give.

Simplifications of arc expressions have been made possible using implicit conversions. The drawback is that implicit conversions must be implemented specifically for each collection. Simplifications could also have been handled during code generation, which was tested in an early prototype. This would work for simple cases, where the arc expression consists of a single token. To make it work for more complex inscriptions, for instance a function call, a more complex analysis is involved to determine if the expression evaluates to a token, a list of tokens, or a collection of tokens where the types correspond to the connected place. Manual manipulation of code may also cause limitations in further development of the Scala simulator. For instance, it would complicate the possibility of extending the simulator so that a module can abstract over collection types (type parameter of a module class in the generated code), and still allow simplifications of arc expressions. Manually mapping collection types for a module instance, based on types in the predecessor module would violate the concept of generating one single class for each module in a model. However, abstracting over collection types for modules is not possible in the current simulator implementation, since the Scala compiler cannot find the required implicit conversions based on an abstract type.

We have introduced a relation between hierarchical concepts in CPN and object-oriented concepts. A more straightforward solution was used in the fist prototype implementation of the simulator. The hierarchical concepts of CPNs are solely utilities to make modelling more convenient,

and do not add theoretical complexity to the modelling language. According to [17, Chapter 5.6], a hierarchical model can always be unfolded into an equivalent non-hierarchical model with the same behaviour, by performing three transformation steps:

1. Replace each substitution transition with the content of its associated submodule so that related port and socket places are merged into a single place.

2. Collect the content of all resulting prime modules into a single module.

3. Merge the places in each fusion set into a single place.

This solution had some undesirable effects. The amount of generated code will increase when there are several instances of a module. Transformation step 1 will in also become more complex with regard to type parameters, where an abstract type of a module must be mapped manually to the concrete type in a prime module.

In Chap. 3, we suggested that Scala should be used as inscription language without introducing additional syntax. In the current implementation of the simulator, a free variable used in a CPN model must be declared as `cpnvar`. Getting rid of this additional syntax is a problem related to the code generation. It is a matter of implementation details that can be solved using reflection in Scala. A series of steps are needed to find a free variable, and its correct type based on its context. Scala's reflection API is in its early stage, and workarounds have to be done to include definitions into the scope used for evaluating an expression with reflection[4]. This is essential, for instance when a class declaration must be in scope to find the type of a free variable. As the reflection API matures, this problem may become trivial to solve. Another feature the code generation can be extended with is to allow patterns in guards: for instance `(x,y) = (1,2)`. This is not allowed in the current implementation, because it is not a valid Scala expression without context. For instance, `val (x,y) = (1,2)` would be allowed. Some manipulation in the code generation is required for this to work. A similar approach can be used to allow variance annotations and bounds on type parameters of a module. `Foo[A]` can be parsed directly using reflection, but it will not work for e.g. `Foo[+A]`, or `Foo[A <: AnyRef]`. For this to work, the expression must be prefixed with `class`.

We have decided to implement the Scala simulator so that any collection implementing the `CPNCollection` trait can be used. This gives more read-

[4] https://issues.scala-lang.org/browse/SI-7081

able models where other collections to hold place markings are more appropriate. An alternative would be to restrict collections to multisets only. To avoid explicit modelling of other collections such as queue, new modelling constructs could be integrated, for example some annotation on a place, specifying that the marking should operate as a queue. However, new modelling constructs concerns the Petri Nets part of CPNs, not the inscription language. Restricting collections to multisets limits the flexibility of modelling, but allows more specific optimisations, both in the generated code, and the simulator implementation. This implies that the model-dependent code can increase. We will discuss performance and model-dependent code further in Chap. 6.

Chapter 6

Evaluation

In this chapter, we evaluate the Scala CPN simulator and explain how we have tested the various components of the simulator. Major parts of the evaluation are based on the case study of the Babel routing protocol, introduced in Chap. 2. The Babel CPN model has been used as a larger industrial-sized example to verify that the design is sustainable, and that the implementation of the Scala simulator is robust. This chapter includes qualitative assessment, where testing approaches are described in Sect. 6.1, and the usability of CPN modelling with Scala as inscription language is evaluated in Sect. 6.4. Quantitative assessment is also included, where the Scala simulator is compared to the SML simulator. In Sect. 6.2, we compare the performance of the two simulators, and in Sect. 6.3, we compare the amount of generated code.

6.1 Testing

In this section, we describe how the Scala simulator implementation, as well as the implementation of code generation of CPN models have been tested.

6.1.1 Scala Simulator

In automatic simulation, a random binding element among the enabled binding elements in a given marking is picked to occur. Due to this non-deterministic behaviour, automatic testing of the simulator is problematic. Interactive simulation can be used to go through each execution step manually, and verify that the enabled bindings in each step are as expected. However, this may be a daunting task for larger models, and one successful

execution pattern does not necessarily provide sufficient test coverage.

With CPN Tools, the state space [17, Chapter 7] of a model can be generated, representing all reachable markings in a directed graph where nodes represent markings, and arcs represent occurring binding elements. The state space graph for a model can be used to find a set of occurrence sequences that can be applied as automatic test cases for an equivalent model for the Scala simulator. One can then test that the same occurrence sequences can be executed with the Scala simulator.

The automatic testing proposed has not been prioritised within the scope of this thesis. However, a similar, manual, approach has been used with interactive simulation of the Babel CPN model. Further, running a number of automatic tests have been performed to verify the reliability of the simulator. Additionally, a number of small models have been used to explore key concept of the CPN formalism. Execution of these models has been studied manually to verify the correctness and sufficient support of modelling constructs and operations. Most of the modelling constructs have been covered in the protocol example from Chap. 3 (and later in the Babel CPN model), and are summarised in Tab. 6.1.

6.1.2 Code Generation

The translation of the object representation of models from Access/CPN to the intermediate representation is implemented in Java. A number of unit tests, using the JUnit framework [3], have been written to verify the correctness of this translation. The intention of this transformation is to simplify the mapping between modules, port and socket places, and to merge fusion places, so that the code generation becomes easier. The intermediate representation of a module includes places, transitions, arcs, parameters, and instantiations. The tests verify the following criteria of the intermediate representation compared to the initial representation, based on the conceptual relation between hierarchical concepts of CPNs and object-oriented concepts, summarised in Tab. 5.1:

- Equal number of transitions.
- The number of global places is equal to the number of fusion groups.
- Equal number of places (excluding port- and fusion places).
- The number of parameters to a module is equal to the number of port places in the module.

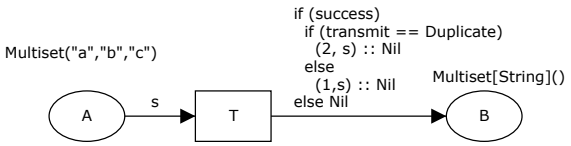
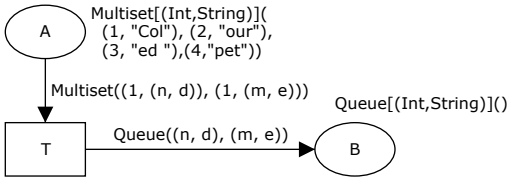
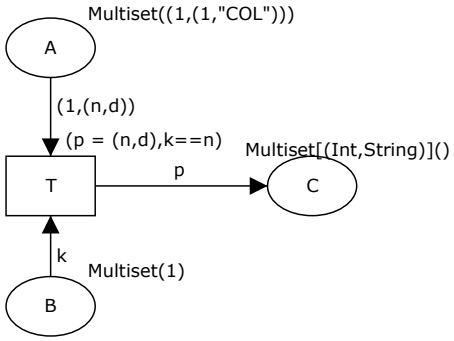
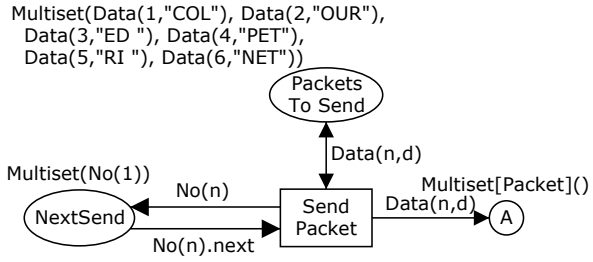
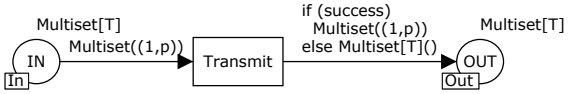
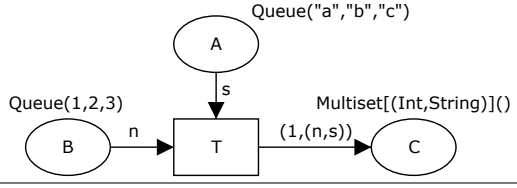
	Concept	Example
1	• Types with limited values, using booleans and enumerations. • Arc expressions evaluating to a token and a list of tokens.	
2	Multiple patterns on one arc expression.	
3	Guard expressions with equality testing and assignment.	
4	Non-pattern input arcs.	
5	Modules with type parameters.	
6	Multiple binding groups for a transition.	

Table 6.1: Test scenarios.

- If the number of arcs in a module (excluding arcs between socket places and substitution transitions) in the representation from Access/CPN is *s+d*, where *s* is single headed arcs and *d* is double headed arcs, then the number of arcs in the intermediate representation should be *s+d*2*. Theoretically, a double headed arc is one input arc and one input arc with equal expressions.

- The number of instantiations in a module is equal to the number of substitution transitions in the module.

- The number of parameters to an instantiation is equal to number of arcs connected to the corresponding substitution transition.

The code generation based on the intermediate representation of a model is implemented in Scala. A number of tests have been written using the ScalaTest framework [5], to verify that the syntactical analysis of inscriptions, described in Sect. 5.5.2, and further calculations, described in Sect. 5.5.3, are performed correctly. The tests include verification of patterns in arcs expressions, guards and that the binding groups and the ordered pattern binding basis are calculated correctly. The model examples in Tab. 6.1 reflects these test scenarios:

- The arc from A to T in the model in row 1 should give one pattern based on the arc expression `s`. Since A's collection type is `Multiset`, the pattern should be expanded to a tuple `(_,s)`, where the first element is a wildcard representing the coefficient.

- The arc from A to T in the model in row 2 should give two patterns based on the arc expression `Multiset((1,(n,d)),(1,(m,e)))`. The coefficients should be replaced with wildcards, giving the patterns `(_,(n,d))` and `(_,(m,e))`.

- The arc from NextSend to SendPacket in the model in row 4 should not give any patterns.

- The guard of the transition T, `(p=(n,d), k==n)`, in the model in row 3 should give two guard conjuncts. The first conjunct should bind `p` to `(n,d)`, and the second conjunct should evaluate the expression `k==n`.

- The transition T in the model in row 3 should get an ordered pattern binding basis where the pattern on the input arc from A is first, and the pattern on the input arc from B is the second. This is because the pattern on the arc from A contains the most free variables.

- The transition T in the model in row 6 should have two binding groups, one containing the free variable `s`, and the other the free variable `n`.

6.2 Performance

In this section, we present performance results of the Scala simulator compared to the SML simulator. The Babel CPN model has been used to obtain the results. Making a fair comparison of the two simulators is hard because different strategies are used in the implementation of enabling inference. The Scala simulator implements enabling inference so that all enabled bindings for a transition are calculated, for both automatic and interactive simulation. For automatic simulation with the SML simulator, the first enabled binding found for a chosen transition is picked to occur immediately, based on random drawing as explained in Chap. 4. This means that the number of binding elements calculated in a simulation with the SML simulator is equal to the number of simulation steps.

Performance testing has been conducted on a computer with an Intel Core2 Duo CPU @ 2.5 GHz and 4 GB ram. Each result is based on the average value of four simulations.

Figure 6.1 presents results, measuring time spent to perform a given number of simulation steps. The x-axis represents the number of simulation steps, and the y-axis the CPU time used in milliseconds. The solid line represents measures for the Scala simulator and the dashed line corresponds to the SML simulator.

The results show that the Scala simulator is significantly slower than the SML simulator. This is expected since the Scala simulator calculates a lot more binding elements. When 20 000 steps were conducted with the Scala simulator, 2 374 265 binding elements were calculated on average.

To get some more useful results, we also measured the number of calculated binding elements. For the SML simulator, this is equal to the number of steps. The results are presented in Fig. 6.2. These results show that the SML simulator is faster to about 11 000 binding calculations, but the time spent grows much faster than for the Scala simulator. The reason for this is that the time spent calculating each binding element in a single step for the Scala simulator, is likely to be less than the calculation of the binding element calculated for one step in the SML simulator. This is because the SML simulator will have to backtrack if the first binding chosen is not enabled. As the simulation runs, markings on places grow, and the probability of backtracking increases. This has the opposite effect for the Scala simu-

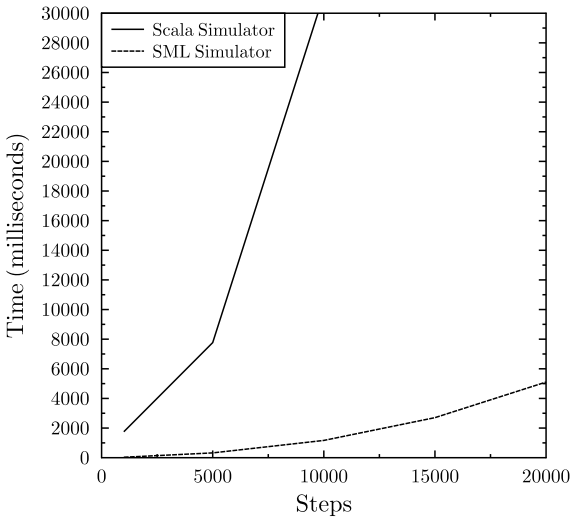


Figure 6.1: Performance results based on steps.

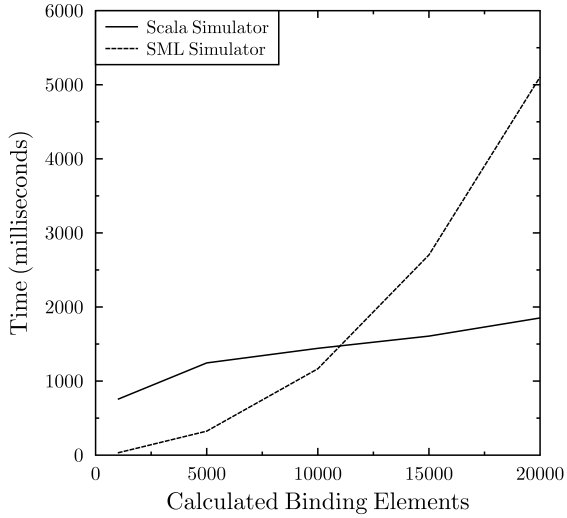


Figure 6.2: Performance results based on calculated binding elements.

lator, where an increasing number of bindings are calculated for each step, making the time increase on a lower phase. For the Scala simulator, 952 steps were executed to calculate 15 000 binding on average, while 1086 steps were executed to calculate 20 000 binding elements. This gives an increase in steps of approximately 14% to calculate approximately 33% additional binding elements. Time used has increased from 1607 ms to 1852 ms on average, corresponding approximately to an increase of 15% in time, which is close to the 14% increase of steps. Hence, most of the time used is based on how many steps are executed, not how many binding elements that are calculated.

The fact that the Scala simulator is slower when calculating fewer than about 11 000 binding elements, signals that scheduling and inference implementation is slower regardless of the different strategies for the SML and Scala simulators. It is likely that part of the reason for this is due to our more generic implementation. However, it would be beneficial to implement a similar strategy for the Scala simulator, as for the SML simulator, before drawing a final conclusion. Generally, one can assume that it will be hard to match the performance of the SML simulator even if the enabling inference and collections are optimised. Since the model-dependent code has been decreased, there is less room for fine-tuning the enabling inference in different situations. One example of this is that the generic implementation must be able to handle the use of any collection implementing the required methods. According to [14], given a pattern arc expression `(x,y,1)`, the SML simulator allows lookup of tokens using `1` as key. This is not possible in the current Scala simulator, since tokens are provided one by one, as an argument to a function which performs the pattern match.

6.3 Generated Code

A motivation for the modularised and generic architecture of the Scala simulator is to decrease the amount of generated code for simulation. When the functionality and amount of generated code decreases, so does the probability of errors in the code. The Babel CPN model has been used to compare the amount of generated SML and Scala simulator code. Table 6.2 shows the complexity of the Babel CPN model by means of the numbers of net elements used. The numbers in the unfolded column represent the model, unfolded to one module, as described in Chap. 5.7. Code generated for the SML simulator is quite complex compared to the generated Scala code because enabling inference and occurrence are handled in a model-dependent way

	Hierarchical	**Unfolded**
Modules	19	1
Abstraction levels	5	1
Places	65	18
Transitions	57	39
Arcs	170	149

Table 6.2: Complexity of the Babel CPN model.

for each transition. Generated SML code for the Babel protocol amounts to 349 184 bytes, and 147 456 bytes of Scala code. This gives a reduction of approximately 58%. The numbers are based on *generated* code only, not declarations, like for instance functions. Taking into account that they are two different programming languages, it is not necessarily only the different architectural approach for the Scala simulator that causes the reduction. However, due to the significant decrease, we can conclude that the impact of implementing enabling inference and occurrence generically has a great impact on the amount of generated code.

6.4 Usability

In this section, we assess the *usability* of modelling with CPNs using Scala as inscription language with the design proposed in this thesis. We will use the Babel CPN model for this purpose, and compare the differences of using CPN ML and Scala as inscription language.

CPNs support modelling of complex problems by modelling on different abstraction levels using hierarchical concepts. However, structuring of code is less trivial for programmers inexperienced with functional programming. SML supports modular programming, but not to the same extent as an object-oriented language. Additionally, a certain level of skills in SML and functional programming is required to obtain modularisation. Modular programming in Scala only requires the basic knowledge needed to be able to construct CPNs with Scala as inscription language. Hence, required learning time for solving more complex problems with CPNs can be reduced. We support this assertion in the following assessment of the Babel CPN model.

Figure 6.3 shows the ProcessPacket module of the Babel CPN model using Scala, and Lst. 6.1 shows the class declarations used in this module. Corresponding declarations in CPN ML were given in Chap. 2, and are summarised in Lst. 6.2. The module using CPN ML was shown in Fig. 2.6.

Listing 6.1: Scala classes used in the ProcessPacket module of the Babel CPN model.

```
abstract class TLV case class
HelloTLV(seqno: Int) extends TLV

abstract class Address
case class Unicast(target: Int) extends Address
case class Multicast extends Address

case class Packet(src: Address, dest: Address, tlv: TLV)

case class NeighbourTableEntry(
  neighbour: Int, history: Int, txcost: Int, helloSeqno: Int)

case class NeighbourTable(neighbours: List[NeighbourTableEntry]) {
  def ignorePacket(sender: Int, tlv: TLV) = tlv match {
    case HelloTLV(_) => false
    case _ => !neighbours.exists(n => n.neighbour == sender)
  }}

case class RecTLV(sender: Int, node: Int, tlv: TLV)

case class RouteTableEntry(
  source: Int, neighbour: Int, metric: Int, seqno: Int, selected: Boolean)

case class RouteTable(routes: List[RouteTableEntry]) {
  def advertiseRoutes(sender: Int) =
    routeTableDump.map { RouteAdv(sender, Multicast(), _) }
}

case class RouteAdv(sender: Int, receiver: Address, route: RouteTableEntry)
```

A lot of functions are used to do calculations based on data held in the data structures of a Babel node, containing information about sequence numbers, neighbours, sources, routes, and pending requests. As the number of functions grows using CPN ML, they will be harder organise, and to reason about what a function does based on model inscriptions. An example of this is the function `ignorePacket`, determining whether a TLV from a given sender should be accepted, based on the information in the `NeighbourTable` of the receiving node. With reference to the Scala declarations in Lst. 6.1, `NeighbourTable` is defined as a class, taking a list of `NeighbourTableEntry` as a parametric field. In this way, methods operating on the list of neigh-

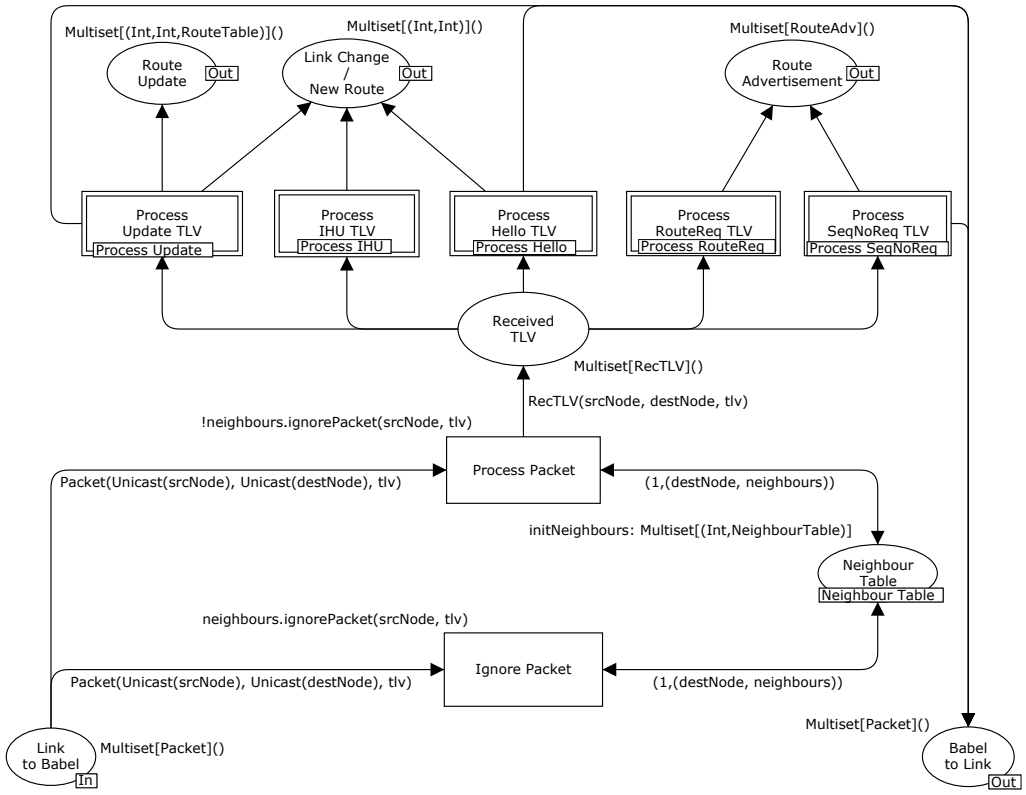


Figure 6.3: The ProcessPacket module of the Babel CPN model with Scala as inscription language.

Listing 6.2: CPN ML declarations used in the ProcessPacket module of the Babel CPN model.

```
colset HelloTLV = record seqNo:INT;
colset TLV = union HTLV:HelloTLV + ITLV:IHUTLV + UTLV:UpdateTLV +
  RRTLV:RouteReqTLV + STLV:SeqNoReqTLV;

colset Address = union UNICAST:Nodes + MULTICAST;

colset Packet = record src:Address * dest:Address * TLV:TLV;

colset Nodes = int with 1 .. N;
colset NeighbourTableEntry = record neighbour:Nodes * history:INT * txcost:
    INT *
helloSeqno:SeqNo;
colset NeighbourTable = list NeighbourTableEntry;
fun ignorePacket (src: int, tlv: TLV, neighbours: NeighbourTable) =
case tlv of
   HTLV t => false
      | _ => not(List.exists(fn n => (#neighbour n) = src) neighbours);

colset SRCxDESTxTLV = product INT * INT * TLV;

colset RouteTableEntry = record source:INT * neighbour:INT *
  metric:INT * seqNo:INT * selected:BOOL;
colset RouteTable = list RouteTableEntry;
fun advertiseRoutes(srcNode, dest, routes) = list_to_ms
  (List.map(fn el => (srcNode, dest, el)) (routeTableDump(routes)));

colset NODExDESTxROUTE = product Nodes * Address * RouteTableEntry;
colset NODExROUTESRC = product INT * INT;
colset NODExROUTESRCxROUTES = product INT * INT * RouteTable;
```

bours can be defined in the `NeighbourTable` class. Shown from the CPN ML declarations in Lst. 6.2 `NeighbourTable` is a list of `NeighbourTableEntry`, and the function `ignorePacket` must be defined globally. Comparing the guard of IgnorePacket, we see that it is easier to conclude that the function `ignorePacket` is operating on the list of neighbours when using Scala:

```
CPN ML:  [ignorePacket(srcNode, tlv, neighbours)]
Scala:   neighbours.ignorePacket(srcNode, tlv)
```

The Babel CPN model also introduced the benefit of having functions returning a more general collection representation than required for a place.

An example of this is shown from the Scala declarations in Lst. 6.1, where the result type of the method `advertiseRoutes` defined in the `RouteTable` class is a list of route advertisements. This makes it possible to reuse methods without being restricted by collection types.

To support pattern matching for the enabling inference in the Scala simulator, user defined classes are recommended to be case classes for conciseness. How pattern matching against SML records compares to Scala case classes is shown by the arc expression from LinktoBabel to IgnorePacket:

```
CPN ML: {src=UNICAST(srcNode), dest=UNICAST(destNode), TLV=tlv}
Scala:  Packet(Unicast(srcNode), Unicast(destNode), tlv)
```

The Scala pattern is slightly more concise than the SML pattern, where component names of the record must be provided. This is also possible with Scala, and would look like the following expression:

```
Packet(src=Unicast(srcNode), dest=Unicast(destNode), tlv=tlv)
```

However, this expression is not a valid pattern in Scala.

With CPN ML, the colour set inscription on a place can give useful information of what tokens on the given place represent. This is particularly valuable when using products. The absence of colour set inscriptions with Scala can make it harder to deduce what tokens on a place represent when using tuples. This is shown from the place LinkChange/NewRoute in the ProcessPacket module. With CPN ML, based on the colour set inscription `NODExROUTESRC`, we at least get a hint, that tokens on this place are a product, where the first component represents the current node, and the second a source node to which the link has changed, or to which a new route is found. It is harder see what tokens represent in the Scala model, with the initial marking `Multiset[(Int,Int)]()`. This can be solved for instance by defining a case class `NodeRouteSrc`. Another approach which may be more readable, is to define *type aliases* for the components:

```
type Node = Int
type RouteSrc = Int
```

Now, the initial marking of LinkChange/NewRoute can be defined as `Multiset[(Node,RouteSrc)]()`. According to [17, Chapter 3.2], a general rule of thumb for CPN ML, is that products with more than four or five components should not be used. Records are recommended if more components are required. The same rule can be applied for Scala (where products equal

tuples, and records are substituted by case classes) with one extension: If calculations are restricted to the internal structure of a token type, a case class should be used, where the calculations are defined as methods in the scope of the class.

The arc from NeighbourTable to IgnorePacket shows a special case for multisets, requiring more verbose inscriptions. Instantiation of a multiset with elements with type `(Int,T)` results in a multiset of type `Multiset[T]`. Since `destNode` is of type `Int`, the tuple must be the second component of another tuple, where the first component is `1`.

A restriction in the current code generation implementation is that explicit type information of collection type for place markings is required in all cases. This gives more verbose inscriptions when initial markings are set from values, like the initial marking of the place NeighbourTable, set to the value `initNeighbours`.

In general, translating the Babel CPN model from using CPN ML to Scala as inscription language was a trivial task. Some inscriptions have become more verbose, while others more concise and easier to read. Our overall conclusion is that Scala has proved to be an applicable inscription language for CPNs.

Chapter 7

Conclusions and Future Work

In Sect. 7.1, we summarise the main content from the former chapters of this thesis. In Sect. 7.2, we summarise and assess the results obtained, and compare them to the initial objectives and research questions. Finally, in Sect. 7.3, we provide proposals for future work.

7.1 Summary

In Chap. 2, we introduced the case study of the Babel protocol, which has been used to assess the implementation of the CPN Scala simulator and the practical use of Scala as inscription language. Modelling of CPNs was also introduced in this chapter, where the use of hierarchical concepts have demonstrated how complex systems can be managed by splitting models into modules.

In Chap. 3, we proposed a design of how Scala can be used as inscription language for CPNs. The design evolved from an initial verbose draft, based on the design using CPN ML, to a final, more intuitive and concise version, including simplification of inscriptions and introduction of new modelling constructs. New concerns in the modelling environment due to object-orientation were also addressed, including scoping, object equality, and side effects. We proposed a design where Scala is not extended with new language syntax, which includes deprecating the explicit definition of CPN variables in models.

In Chap. 4, we introduced the basic formal semantics of CPNs and the enabling inference algorithm, which is the main mechanism required for

simulation of CPNs. The specific pattern matching capabilities of Scala was discussed in this context.

In Chap. 5, we explained how the Scala simulator has been implemented, to support the design from Chap. 3, and how the type system of Scala has been utilised to conform to the formal definition of CPNs from Chap. 4. We also explained how new collection types adapted for CPN modelling can be implemented, how the simulator has been integrated with CPN Tools, and we showed code snippets that are generated from a CPN model.

In Chap. 6, we conducted both qualitative and quantitative assessment of the work that has been done, mainly based on the Babel CPN model. We used the Babel CPN model to assess the performance of the Scala simulator, and to measure the amount of generated code, compared to the SML simulator. We also assessed the usability of Scala as inscription language for CPNs compared to CPN ML.

7.2 Results and Conclusions

In this section, we summarise the main results obtained in this project. The following objectives and research questions were given in Chap. 1.3:

1. Scala as inscription language for CPN:
   (a) How to achieve integrated use of both functional and object-oriented paradigms in CPN models?
   (b) How to support parameterised and flexible CPN modules?
   (c) How does it affect model design?
2. Scala as implementation language for CPN:
   (a) How can the pattern matching capabilities of Scala be used to implement enabling inference?
   (b) How can the generated model-dependent simulator code be reduced?
   (c) What is the model execution performance?

Many of the results obtained are overlapping with regard to the objectives and research questions. Below each result is summarised and related to the relevant objectives and research questions.

Avoid extension of Scala syntax. We have proposed the use Scala of as inscription language for CPNs without extending the language with new syntax, making it easier for programmers familiar with Scala, or object-oriented paradigms in general, to get into modelling with CPNs. Defining token types in terms of colour sets have been replaced with classes in Scala. Case classes have made pattern matching trivial, both from a modeller's perspective, and in the simulator implementation. This achievement concerns objective 1, question (a) and (c), as well as objective 2, question (a).

Scoping. The object-oriented facility of being able to scope methods within classes has proved to be of great benefit, shown by the industrial-sized Babel CPN model. When the number of functions required for a model grows, scoping of functions is valuable both for structuring code, and for the readability of inscriptions. This achievement concerns objective 1, question (a) and (c).

Multiple collections for place markings. Allowing the use of other collections than multisets has been achieved. Modellers are also able to integrate their own specialised collections, either from scratch as described in Chap. 5.4.3, or based on collections in Scala's collection library, as described in Chap. 5.4.4. This achievement concerns objective 1, question (a) and (c), as well as objective 2, the implementation of the simulator in general.

Omission of colour set inscriptions. The number of inscriptions in a CPN model has been reduced by omission of colour set inscriptions on places, since the type information is provided by the initial marking. This achievement concerns objective 1, question (c).

Increased flexibility of modules. We have related hierarchical concepts of CPNs with object-oriented concepts, summarised in Tab. 5.1. This allow modules to have type parameters, which gives a number of advantages. First, it avoids unification of types that may not necessarily have a conceptual relation. This allows modelling of a module on a higher abstraction level. Second, since each module is generated independently as classes, a looser coupling is obtained that better permits reuse of modules, both within a model, and in multiple models. It also introduces the possibility of component based code generation, namely the possibility to generate each module independently, so that testing of a module can be done in isolation from the complete model.

Finally, it enhances the intuition for modellers, as they can relate hierarchical modelling concepts directly to the object-oriented nature of Scala as inscription language. This achievement concerns objective 1, question (a), (b) and (c), as well as objective 2, the implementation of the simulator in general.

Simplifications using language features. Simplifications of inscriptions have been achieved solely by using language features of Scala. In Chap. 5.4.3, we showed how multiset can be implemented to avoid coefficients, by defining an additional factory method. In Chap. 5.3.3 we showed how collection constructors in arc expressions can be omitted, and allow tokens to be represented in lists, using implicit conversions. This achievement concerns objective 1, question (c), as well as objective 2, the implementation of the simulator in general.

Generic simulator. We have implemented the core functionality of CPN simulation generically, including the enabling inference and occurrence. This has been achieved mainly by object-orientation and the powerful type system of Scala combined with higher-order functions, and the essential capability of pattern matching, addressed in Chap. 4.2. This achievement concerns objective 2, question (a) and (b).

Assessment with an industrial-sized example. Applying Scala as inscription language on the industrial-sized Babel CPN model has shown that the inscription design is viable. Required declarations and inscriptions have been reduced in some areas, but increased in other areas, compared to using CPN ML. Inscriptions have increased mainly because of the possibility to use multiple collection types, but also because of the special case with multiset coefficient in some cases. The Babel CPN model has also verified that the simulator handles large models, as well as it has been used to assess performance compared to the SML simulator in CPN Tools. The performance of the Scala simulator is, as expected, much slower than the SML simulator. Performance optimisations have been out of scope for this thesis, but bottlenecks and suggestions as for how this can be improved have been addressed, based on previous research in optimisation of a CPN simulator [14, 23]. This achievement concerns objective 1, question (c), and objective 2, question (c).

Reduced model-dependent code. The generic simulator implementation has lead to a significant decrease of model-dependent code. The amount

of generated code for the Babel CPN model has been reduced with approximately 58%. Reducing the complexity of model-dependent code in turn decreases the possibility of errors in code generation of a CPN model. This achievement concerns objective 2, question (b).

7.3 Future Work

In this master's thesis, we have made an initial research of how Scala can be applied to CPNs. The main contributions are based on design and architectural decisions, regarding both Scala as inscription and implementation language. There is still a lot of work to be done to accept or decline whether Scala is more viable as implementation language for CPNs than SML. Below we present proposals for future work.

Improve code generation. There are some shortcomings in the current prototype of the code generation for the Scala simulator that can be solved using the reflection capabilities in Scala, discussed in Chap. 5.7. The most evident shortcoming is that CPN variables must be explicitly declared. Further, only single CPN variables can be bound in guards, for instance, `(x,y) = z` is not allowed. Finally, type parameters of a module cannot have bounds and variance annotations.

Extended language features. Type parameterisation of a module could be made more flexible, for instance by allowing to abstract over collection types.

As discussed in Chap. 3.2.7, side effects should be performed in actions instead of guards. This is not supported in the current Scala simulator implementation.

A subset of the functionality of CPNs has been addressed. Including concepts such as timed nets and state space calculation is necessary for a complete implementation.

Performance optimisation. Performance of the Scala simulator must be improved, especially with regard to automatic simulation, as discussed in Chap. 6.2. An evaluation of a future, optimised, Scala simulator, must be conducted to conclude if potential performance loss by allowing multiple collections is worthwhile, taking into account to which extent other collections than multiset will be used. If collections are restricted to multisets, it would be possible to make the model-dependent code more involved concerning optimisations, as well as it will give

more flexibility for optimisations in the generic simulator implementation.

To fully support side effects, the scheduler algorithm must be refined. An appropriate solution to this problem is some kind of event system, so that one does not have to check if transitions will become enabled after an occurrence, amongst all disabled transitions. Scala.React [15], a framework that embeds a DSL for event handling using higher-order data flows, could be used for this purpose. Whether side effects should be supported may be an issue regarding performance, as it makes the implementation more complex. The alternative approach is not to rely on mutability for the enabling of transitions. This is the approach taken in Renew.

Scala's Actor library can be used to implement support for concurrency in the simulator. This can be related to the low-level Petri net simulator implemented in Scala using the Actor library, discussed in Chap. 1.4.

Improving integration of new collections. Chapter 5.4 describes how collections can be adapted for use in CPNs. More research can be done on how the type constraints in the simulator implementation can be improved to minimise the required methods that must be defined.

Tool support. In a more complete implementation of the Scala simulator, it will be beneficial to have a Scala aware GUI with equivalent features found in CPN Tools. Acceptance in the CPN community is also crucial, and a GUI would help to get feedback from users.

The process of investigating how Scala can be used as inscription and implementation language for CPNs has been interesting and educative. A challenging aspect of the simulator implementation has been to prioritise which features that are important to implement within the scope of this project, that underpin the proposals we have given. Considering the objectives that were prepared prior to the start of the work, we are satisfied with the results obtained. Scala has proved to be a viable inscription language for CPNs, and the simulator implementation has shown that the design proposed is valid and that Scala with its pattern matching capabilities is suitable as implementation language itself. However, implementing support to allow omission of CPN variables would be beneficial to support this design decision.

Bibliography

[1] Agile Model Driven Development. http://www.agilemodeling.com/.

[2] CPN Tools. http://cpntools.org.

[3] JUnit Framework. http://junit.org.

[4] Scala Reflection. http://docs.scala-lang.org/overviews/reflection/overview.html.

[5] ScalaTest Framework. http://www.scalatest.org.

[6] The Scala language. http://www.scala-lang.org.

[7] J. Park A. R. Hevner, S. T. March and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, March 2004.

[8] Frank Piessens Adriaan Moors and Martin Odersky. Generics of a higher kind. *OOPSLA '08 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2008.

[9] David N. Turner Benjamin C. Pierce. Local type inference. CSCI Technical Report #493, Indiana University, 2000.

[10] L. Bernardinello and F. A. Bianchi. A concurrent simulator for petri nets based on the paradigm of actors of hewitt. *PNSE'12 International Workshop on Petri Nets and Software Engineering*, 2012.

[11] J. Chroboczek. The Babel Routing Protocol. IETF, RFC 6126, 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6126.txt.

[12] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[13] C. Kemper S. Marx M. Odersky S. E. Panitz D. Stoutamire K. Thorup G. Bracha, N. Cohen and P. Wadler. Adding generics to the java programming language: Participant draft specification. 2001.

[14] T. B. Haagh and T. R. Hansen. Optimising a coloured petri net simulator. Master's thesis, University of Aarhus, December 1994.

[15] M. Odersky I. Maier. Deprecating the Observer Pattern with Scala.React. Technical Report EPFL-REPORT-176887, EPFL, 2012.

[16] I. Jacobson J. Rumbaugh and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[17] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer-Vergal, Berlin, Heidelberg, 2009.

[18] L. M. Kristensen and S. Christensen. Implementing coloured petri nets using a functional programming language. *Higher-Order and Symbolic Computation*, 17(3):207–243, 2004.

[19] L. M. Kristensen and K. I. F. Simonsen. *Applications of Coloured Petri Nets for Functional Validation of Protocols*. To appear in special issue of Transactions on Petri Nets and Other Models of Concurrency, Subseries of Lecture Notes in Computer Science, Springer.

[20] Kent Lee. *Programming Languages - An Active Learning Approach*. Springer Science+Business Media, LLC, New York, USA, 2008.

[21] L. Spoon M. Odersky and B. Venners. *Programming in Scala - Second Edition*. Artima Press, Walnut Creek, California, 2010.

[22] M. Zenger M. Odersky, C. Zenger. Colored local type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, volume 36(3), pages 41–53. 2001.

[23] K. H. Mortensen. Efficient data-structures and algorithms for a coloured petri nets simulator. In *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 57–75. Aarhus University, 2001.

[24] M. Duvigneau O. Kummer, F. Wienberg and L. Cabac. *Renew - User Guide*. University of Hamburg, February 2012.

[25] M. Odersky. Pimp my Library. Blog post, October 9, 2006. `http://www.artima.com/weblogs/viewpost.jsp?thread=179766`.

[26] M. Odersky. *The Scala Language Specification*. EPFL, Switzerland, May 2011. Version 2.9.

[27] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *Leibniz International Proceedings in Informatics (LIPIcs)*, volume 4 of *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 427–451. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Germany, 2009.

[28] C. Perkins. Ad hoc On-Demand Distance Vector (AODV) Routing. IETF RFC 3561, 2003. [Online]. Available: `http://www.ietf.org/rfc/rfc3561.txt`.

[29] M. Conti R. Bruno and E. Gregori. Mesh networks: Commodity multihop ad hoc networks. In *Communications Magazine*, volume 43(3), pages 123–131. IEEE, Hollis, NH USA, March 2005.

[30] Claus Reinke. Haskell-coloured petri nets. In Pieter Koopman and Chris Clack, editors, *Implementation of Functional Languages*, volume 1868 of *Lecture Notes in Computer Science*, pages 165–180. Springer Berlin Heidelberg, 2000.

[31] W. Reisig and G. Rozenberg. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 122–173. Springer, Berlin Heidelberg, 1998.

[32] J. D. Suereth. *Scala in Depth*. Manning Publications Co., Shelter Island, NY, 2012.

[33] S. Lenglet V. Cremet, F. Garillot and M. Odersky. *A Core Calculus for Scala Type Checking*, volume 4162 of *Lecture Notes in Computer Science*, pages 1–23. Springer, Berlin Heidelberg, 2006.

[34] M. Westergaard and L. M. Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In *Applications and Theory of Petri Nets*, volume 5606 of *Lecture Notes in Computer Science*, pages 313–322. Springer, Berlin Heidelberg, 2009.

[35] S. Wiesner. Type inference - main seminar, summer term 2011. Technische Universität München. Available: www2.in.tum.de/hp/file?fid=879, 2011.

List of Figures

Listings

Appendices

Appendix A

Insctructions for Obtaining the Source Code

Source code can be found at the following GitHub repository:

`https://github.com/matiasvinjevoll/cpnscalasimulator`

The repository includes:

- The Scala simulator.
- Transformation of the object representation from Access/CPN.
- Code generation of CPN models.
- The Babel CPN model, both using CPN ML and Scala as inscription languages.

Further instructions are given in the README.md file of the repository.

Appendix B

The Complete "Simple Protocol" Example

This appendix contains the complete model of the hierarchical version of the protocol example, introduced in Chap. 3.2.9, and the code generated from it.

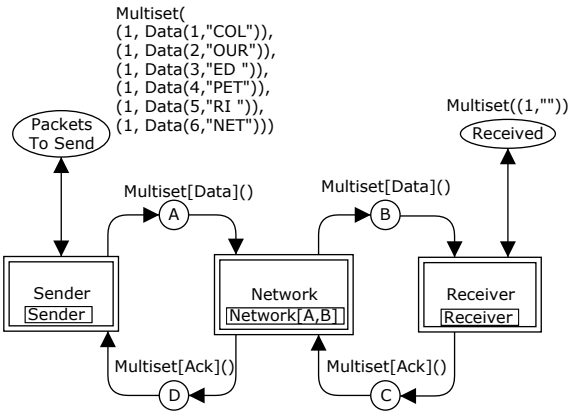


Figure B.1: Top module of the protocol example.

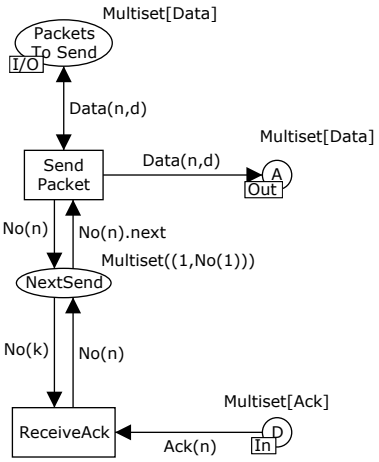


Figure B.2: Sender module of the protocol example.

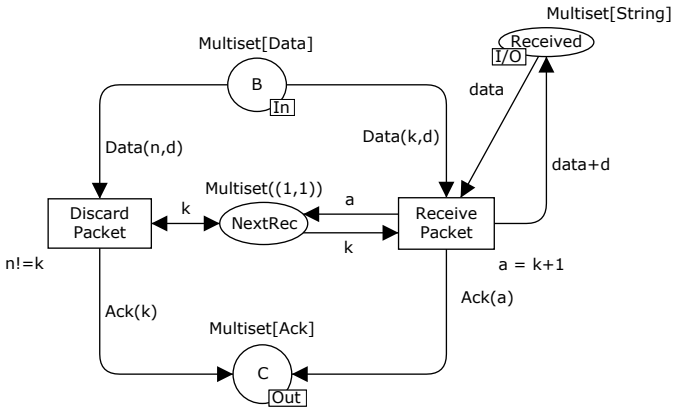


Figure B.3: Receiver module of the protocol example.

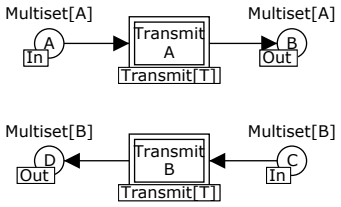


Figure B.4: Network module of the protocol example.

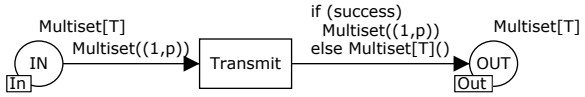


Figure B.5: Transmit module of the protocol example.

```
object SimpleProtocolHier extends App {
  val net = CPNGraph()
  // Declarations
  case class Data(n: Int, d: String)
  case class Ack(n: Int)
  case class No(n: Int) {
    def next = No(n)
  }
  // ########################################################################
  // Generated code for module Top
  // ########################################################################
  class Top() {
    val net = CPNGraph()
    // Places
    val id44 = Place("id44", "Packets To Send", Multiset(
      (1, Data(1, "COL")),
      (1, Data(2, "OUR")),
      (1, Data(3, "ED ")),
      (1, Data(4, "PET")),
      (1, Data(5, "RI ")),
      (1, Data(6, "NET"))))
    net.addPlace(id44)
    val id71 = Place("id71", "Received", Multiset((1, "")))
    net.addPlace(id71)
    val id58 = Place("id58", "D", Multiset[Ack]())
    net.addPlace(id58)
    val id93 = Place("id93", "C", Multiset[Ack]())
    net.addPlace(id93)
    val id87 = Place("id87", "B", Multiset[Data]())
    net.addPlace(id87)
    val id64 = Place("id64", "A", Multiset[Data]())
    net.addPlace(id64)
    // Wire modules
    val st0 = new Receiver(id87, id93, id71)
    net.addSubstitutionTransition(st0.net)
    val st1 = new Network(id64, id87, id58, id93)
```

```
  net.addSubstitutionTransition(st1.net)
  val st2 = new Sender(id44, id64, id58)
  net.addSubstitutionTransition(st2.net)
}
val top0 = new Top()
net.addSubstitutionTransition(top0.net)
// ########################################################################
// Generated code for module Transmit[T]
// ########################################################################
class Transmit[T, T0](
  ID1470769116: Place[T0, Multiset[T]], ID1470771631: Place[T0, Multiset[T]])(
    implicit ev0: Multiset[T] => Traversable[T0]) {
  val net = CPNGraph()
  // Places
  // ========================================================================
  // Generated code for transition Transmit and its arcs
  // ========================================================================
  case class BindingElement_ID1470773821(
    p: Option[T],
    success: Option[Boolean]) extends BindingElement
  val compatible_ID1470773821 = (b1: BindingElement_ID1470773821,
    b2: BindingElement_ID1470773821) =>
    (b1.p == b2.p || b1.p == None || b2.p == None) &&
      (b1.success == b2.success || b1.success == None || b2.success == None)
  val merge_ID1470773821 = (b1: BindingElement_ID1470773821,
    b2: BindingElement_ID1470773821) => {
    val p = if (b1.p == None) b2.p else b1.p
    val success = if (b1.success == None) b2.success else b1.success
    new BindingElement_ID1470773821(p, success)
  }
  val fullBindingElement_ID1470773821 = (be: BindingElement_ID1470773821) =>
    be.p != None &&
      be.success != None
  val guard_ID1470773821 = List()
  val enumBindings_ID1470773821 = List(
    List(BindingElement_ID1470773821(None, Some(true)),
      BindingElement_ID1470773821(None, Some(false))))
  val ID1470773821 = Transition("ID1470773821", "Transmit", 1,
    compatible_ID1470773821, merge_ID1470773821, fullBindingElement_ID1470773821,
    guard_ID1470773821, enumBindings_ID1470773821)
  net.addTransition(ID1470773821)
  // ------------------ Arcs ------------------
  // in ----> Transmit
  val pattern_ID1470775166_0 = Pattern((token: Any) => token match {
    case scala.Tuple2(_, (p: T)) =>
      BindingElement_ID1470773821(Some(p), None)
  }, 0)
  val eval_ID1470775166 = (be: BindingElement_ID1470773821) => (be.p) match {
    case (Some(p)) =>
      Multiset(scala.Tuple2(1, p))
  }
  val ID1470775166 = Arc("ID1470775166", ID1470769116, ID1470773821, In)(
    eval_ID1470775166)
  net.addArc(ID1470775166)
  // Transmit ----> out
  val eval_ID1470775627 = (be: BindingElement_ID1470773821) =>
```

```
    (be.success, be.p) match {
      case (Some(success), Some(p)) =>
        if (success)
          Multiset(scala.Tuple2(1, p))
        else
          Multiset[T]()
    }
  val ID1470775627 = Arc("ID1470775627", ID1470771631, ID1470773821, Out)(
    eval_ID1470775627)
  net.addArc(ID1470775627)
  // ------------------ /Arcs -----------------
  // Set ordered pattern binding basis for Transmit
  ID1470773821.orderedPatternBindingBasis = List(
    ArcPattern(ID1470775166, pattern_ID1470775166_0))
}
// ##############################################################################
// Generated code for module Sender
// ##############################################################################
class Sender[T0, T1](ID1470848028: Place[T0, Multiset[Data]],
  ID1470848187: Place[T0, Multiset[Data]], ID1470848373: Place[T1, Multiset[Ack]])(
    implicit ev0: Multiset[Data] => Traversable[T0],
    ev1: Multiset[Ack] => Traversable[T1]) {
  val net = CPNGraph()
  // Places
  val ID1470848832 = Place("ID1470848832", "NextSend", Multiset(No(1)))
  net.addPlace(ID1470848832)
  // ============================================================================
  // Generated code for transition ReceiveAck and its arcs
  // ============================================================================
  case class BindingElement_ID1470848592(
    n: Option[Int],
    k: Option[Int]) extends BindingElement
  val compatible_ID1470848592 = (b1: BindingElement_ID1470848592,
    b2: BindingElement_ID1470848592) =>
    (b1.n == b2.n || b1.n == None || b2.n == None) &&
      (b1.k == b2.k || b1.k == None || b2.k == None)
  val merge_ID1470848592 = (b1: BindingElement_ID1470848592,
    b2: BindingElement_ID1470848592) => {
    val n = if (b1.n == None) b2.n else b1.n
    val k = if (b1.k == None) b2.k else b1.k
    new BindingElement_ID1470848592(n, k)
  }
  val fullBindingElement_ID1470848592 = (be: BindingElement_ID1470848592) =>
    be.n != None &&
      be.k != None
  val guard_ID1470848592 = List()
  val enumBindings_ID1470848592 = List()
  val ID1470848592 = Transition("ID1470848592", "ReceiveAck", 2,
    compatible_ID1470848592, merge_ID1470848592, fullBindingElement_ID1470848592,
    guard_ID1470848592, enumBindings_ID1470848592)
  net.addTransition(ID1470848592)
  // ------------------ Arcs ------------------
  // NextSend ----> ReceiveAck
  val pattern_ID1470851437_0 = Pattern((token: Any) => token match {
    case Tuple2(_, No((k: Int))) =>
      BindingElement_ID1470848592(None, Some(k))
```

```
}, 1)
val eval_ID1470851437 = (be: BindingElement_ID1470848592) => (be.k) match {
  case (Some(k)) =>
    No(k)
}
val ID1470851437 = Arc("ID1470851437", ID1470848832, ID1470848592, In)(
  eval_ID1470851437)
net.addArc(ID1470851437)
// d ----> ReceiveAck
val pattern_ID1470852878_0 = Pattern((token: Any) => token match {
  case Tuple2(_, Ack((n: Int))) =>
    BindingElement_ID1470848592(Some(n), None)
}, 0)
val eval_ID1470852878 = (be: BindingElement_ID1470848592) => (be.n) match {
  case (Some(n)) =>
    Ack(n)
}
val ID1470852878 = Arc("ID1470852878", ID1470848373, ID1470848592, In)(
  eval_ID1470852878)
net.addArc(ID1470852878)
// ReceiveAck ----> NextSend
val eval_ID1470852049 = (be: BindingElement_ID1470848592) => (be.n) match {
  case (Some(n)) =>
    No(n)
}
val ID1470852049 = Arc("ID1470852049", ID1470848832, ID1470848592, Out)(
  eval_ID1470852049)
net.addArc(ID1470852049)
// ------------------ /Arcs -----------------
// Set ordered pattern binding basis for ReceiveAck
ID1470848592.orderedPatternBindingBasis = List(
  ArcPattern(ID1470852878, pattern_ID1470852878_0),
  ArcPattern(ID1470851437, pattern_ID1470851437_0))
// ========================================================================
// Generated code for transition Send Packet and its arcs
// ========================================================================
case class BindingElement_ID1470849105(
  n: Option[Int],
  d: Option[String]) extends BindingElement
val compatible_ID1470849105 = (b1: BindingElement_ID1470849105,
  b2: BindingElement_ID1470849105) =>
  (b1.n == b2.n || b1.n == None || b2.n == None) &&
    (b1.d == b2.d || b1.d == None || b2.d == None)
val merge_ID1470849105 = (b1: BindingElement_ID1470849105,
  b2: BindingElement_ID1470849105) => {
  val n = if (b1.n == None) b2.n else b1.n
  val d = if (b1.d == None) b2.d else b1.d
  new BindingElement_ID1470849105(n, d)
}
val fullBindingElement_ID1470849105 = (be: BindingElement_ID1470849105) =>
  be.n != None &&
    be.d != None
val guard_ID1470849105 = List()
val enumBindings_ID1470849105 = List()
val ID1470849105 = Transition("ID1470849105", "Send Packet", 1,
  compatible_ID1470849105, merge_ID1470849105, fullBindingElement_ID1470849105,
```

```
    guard_ID1470849105, enumBindings_ID1470849105)
  net.addTransition(ID1470849105)
  // ------------------ Arcs ------------------
  // NextSend ----> Send Packet
  val eval_ID1470850012 = (be: BindingElement_ID1470849105) => (be.n) match {
    case (Some(n)) =>
      No(n).next
  }
  val ID1470850012 = Arc("ID1470850012", ID1470848832, ID1470849105, In)(
    eval_ID1470850012)
  net.addArc(ID1470850012)
  // packets to send ----> Send Packet
  val pattern_ID1470849388IN_0 = Pattern((token: Any) => token match {
    case Tuple2(_, Data((n: Int), (d: String))) =>
      BindingElement_ID1470849105(Some(n), Some(d))
  }, 0)
  val eval_ID1470849388IN = (be: BindingElement_ID1470849105) =>
    (be.n, be.d) match {
      case (Some(n), Some(d)) =>
        Data(n, d)
    }
  val ID1470849388IN = Arc("ID1470849388IN", ID1470848028, ID1470849105, In)(
    eval_ID1470849388IN)
  net.addArc(ID1470849388IN)
  // Send Packet ----> NextSend
  val eval_ID1470850492 = (be: BindingElement_ID1470849105) => (be.n) match {
    case (Some(n)) =>
      No(n)
  }
  val ID1470850492 = Arc("ID1470850492", ID1470848832, ID1470849105, Out)(
    eval_ID1470850492)
  net.addArc(ID1470850492)
  // Send Packet ----> packets to send
  val eval_ID1470849388OUT = (be: BindingElement_ID1470849105) =>
    (be.n, be.d) match {
      case (Some(n), Some(d)) =>
        Data(n, d)
    }
  val ID1470849388OUT = Arc("ID1470849388OUT", ID1470848028, ID1470849105, Out)(
    eval_ID1470849388OUT)
  net.addArc(ID1470849388OUT)
  // Send Packet ----> a
  val eval_ID1470853969 = (be: BindingElement_ID1470849105) =>
    (be.n, be.d) match {
      case (Some(n), Some(d)) =>
        Data(n, d)
    }
  val ID1470853969 = Arc("ID1470853969", ID1470848187, ID1470849105, Out)(
    eval_ID1470853969)
  net.addArc(ID1470853969)
  // ------------------ /Arcs -----------------
  // Set ordered pattern binding basis for Send Packet
  ID1470849105.orderedPatternBindingBasis = List(
    ArcPattern(ID1470849388IN, pattern_ID1470849388IN_0))
  // Wire modules
}
```

```
// ############################################################################
// Generated code for module Receiver
// ############################################################################
class Receiver[T0, T1, T2](
  ID1470723480: Place[T0, Multiset[Data]],
  ID1470725592: Place[T1, Multiset[Ack]],
  ID1470734702: Place[T2, Multiset[String]])(
    implicit ev0: Multiset[Data] => Traversable[T0],
    ev1: Multiset[Ack] => Traversable[T1],
    ev2: Multiset[String] => Traversable[T2]) {
  val net = CPNGraph()
  // Places
  val ID1470723819 = Place("ID1470723819", "NextRec", Multiset(1))
  net.addPlace(ID1470723819)
  // ============================================================================
  // Generated code for transition Discard Packet and its arcs
  // ============================================================================
  case class BindingElement_ID1470724324(
    n: Option[Int],
    d: Option[String],
    k: Option[Int]) extends BindingElement
  val compatible_ID1470724324 = (b1: BindingElement_ID1470724324,
    b2: BindingElement_ID1470724324) =>
    (b1.n == b2.n || b1.n == None || b2.n == None) &&
      (b1.d == b2.d || b1.d == None || b2.d == None) &&
      (b1.k == b2.k || b1.k == None || b2.k == None)
  val merge_ID1470724324 = (b1: BindingElement_ID1470724324,
    b2: BindingElement_ID1470724324) => {
    val n = if (b1.n == None) b2.n else b1.n
    val d = if (b1.d == None) b2.d else b1.d
    val k = if (b1.k == None) b2.k else b1.k
    new BindingElement_ID1470724324(n, d, k)
  }
  val fullBindingElement_ID1470724324 = (be: BindingElement_ID1470724324) =>
    be.n != None &&
      be.d != None &&
      be.k != None
  val guard_ID1470724324_0 = EvalGuard((be: BindingElement_ID1470724324) =>
    (be.n, be.k) match {
      case (Some(n), Some(k)) => {
        n.$bang$eq(k)
      }
    }, 0)
  val guard_ID1470724324 = List(guard_ID1470724324_0)
  val enumBindings_ID1470724324 = List()
  val ID1470724324 = Transition("ID1470724324", "Discard Packet", 2,
    compatible_ID1470724324, merge_ID1470724324, fullBindingElement_ID1470724324,
    guard_ID1470724324, enumBindings_ID1470724324)
  net.addTransition(ID1470724324)
  // ------------------ Arcs ------------------
  // NextRec ----> Discard Packet
  val pattern_ID1470727987IN_0 = Pattern((token: Any) => token match {
    case Tuple2(_, (k: Int)) =>
      BindingElement_ID1470724324(None, None, Some(k))
  }, 1)
  val eval_ID1470727987IN = (be: BindingElement_ID1470724324) => (be.k) match {
```

```
  case (Some(k)) =>
    k
}
val ID1470727987IN = Arc("ID1470727987IN", ID1470723819, ID1470724324, In)(
  eval_ID1470727987IN)
net.addArc(ID1470727987IN)
// b ----> Discard Packet
val pattern_ID1470726221_0 = Pattern((token: Any) => token match {
  case Tuple2(_, Data((n: Int), (d: String))) =>
    BindingElement_ID1470724324(Some(n), Some(d), None)
}, 0)
val eval_ID1470726221 = (be: BindingElement_ID1470724324) =>
  (be.n, be.d) match {
    case (Some(n), Some(d)) =>
      Data(n, d)
  }
val ID1470726221 = Arc("ID1470726221", ID1470723480, ID1470724324, In)(
  eval_ID1470726221)
net.addArc(ID1470726221)
// Discard Packet ----> NextRec
val eval_ID1470727987OUT = (be: BindingElement_ID1470724324) =>
  (be.k) match {
    case (Some(k)) =>
      k
  }
val ID1470727987OUT = Arc("ID1470727987OUT", ID1470723819, ID1470724324, Out)(
  eval_ID1470727987OUT)
net.addArc(ID1470727987OUT)
// Discard Packet ----> c
val eval_ID1470732359 = (be: BindingElement_ID1470724324) => (be.k) match {
  case (Some(k)) =>
    Ack(k)
}
val ID1470732359 = Arc("ID1470732359", ID1470725592, ID1470724324, Out)(
  eval_ID1470732359)
net.addArc(ID1470732359)
// ------------------ /Arcs -----------------
// Set ordered pattern binding basis for Discard Packet
ID1470724324.orderedPatternBindingBasis = List(
  ArcPattern(ID1470726221, pattern_ID1470726221_0),
  ArcPattern(ID1470727987IN, pattern_ID1470727987IN_0))
// ========================================================================
// Generated code for transition Receive Packet and its arcs
// ========================================================================
case class BindingElement_ID1470725136(
  d: Option[String],
  k: Option[Int],
  a: Option[Int],
  data: Option[String]) extends BindingElement
val compatible_ID1470725136 = (b1: BindingElement_ID1470725136,
  b2: BindingElement_ID1470725136) =>
  (b1.d == b2.d || b1.d == None || b2.d == None) &&
    (b1.k == b2.k || b1.k == None || b2.k == None) &&
    (b1.a == b2.a || b1.a == None || b2.a == None) &&
    (b1.data == b2.data || b1.data == None || b2.data == None)
val merge_ID1470725136 = (b1: BindingElement_ID1470725136,
```

```
  b2: BindingElement_ID1470725136) => {
  val d = if (b1.d == None) b2.d else b1.d
  val k = if (b1.k == None) b2.k else b1.k
  val a = if (b1.a == None) b2.a else b1.a
  val data = if (b1.data == None) b2.data else b1.data
  new BindingElement_ID1470725136(d, k, a, data)
}
val fullBindingElement_ID1470725136 = (be: BindingElement_ID1470725136) =>
  be.d != None &&
    be.k != None &&
    be.a != None &&
    be.data != None
val guard_ID1470725136_0 = BindGuard((be: BindingElement_ID1470725136) =>
  (be.k) match {
    case (Some(k)) =>
      val a = k.$plus(1)
      BindingElement_ID1470725136(be.d, be.k, Some(a), be.data)
  }, 1)
val guard_ID1470725136 = List(guard_ID1470725136_0)
val enumBindings_ID1470725136 = List()
val ID1470725136 = Transition("ID1470725136", "Receive Packet", 2,
  compatible_ID1470725136, merge_ID1470725136, fullBindingElement_ID1470725136,
  guard_ID1470725136, enumBindings_ID1470725136)
net.addTransition(ID1470725136)
// ------------------ Arcs ------------------
// NextRec ----> Receive Packet
val pattern_ID1470730054_0 = Pattern((token: Any) => token match {
  case Tuple2(_, (k: Int)) =>
    BindingElement_ID1470725136(None, Some(k), None, None)
}, 1)
val eval_ID1470730054 = (be: BindingElement_ID1470725136) => (be.k) match {
  case (Some(k)) =>
    k
}
val ID1470730054 = Arc("ID1470730054", ID1470723819, ID1470725136, In)(
  eval_ID1470730054)
net.addArc(ID1470730054)
// b ----> Receive Packet
val pattern_ID1470726893_0 = Pattern((token: Any) => token match {
  case Tuple2(_, Data((k: Int), (d: String))) =>
    BindingElement_ID1470725136(Some(d), Some(k), None, None)
}, 1)
val eval_ID1470726893 = (be: BindingElement_ID1470725136) =>
  (be.k, be.d) match {
    case (Some(k), Some(d)) =>
      Data(k, d)
  }
val ID1470726893 = Arc("ID1470726893", ID1470723480, ID1470725136, In)(
  eval_ID1470726893)
net.addArc(ID1470726893)
// received ----> Receive Packet
val pattern_ID1470737284_0 = Pattern((token: Any) => token match {
  case Tuple2(_, (data: String)) =>
    BindingElement_ID1470725136(None, None, None, Some(data))
}, 0)
val eval_ID1470737284 = (be: BindingElement_ID1470725136) =>
```

```
    (be.data) match {
      case (Some(data)) =>
        data
    }
  val ID1470737284 = Arc("ID1470737284", ID1470734702, ID1470725136, In)(
    eval_ID1470737284)
  net.addArc(ID1470737284)
  // Receive Packet ----> NextRec
  val eval_ID1470730942 = (be: BindingElement_ID1470725136) =>
    (be.a) match {
      case (Some(a)) =>
        a
    }
  val ID1470730942 = Arc("ID1470730942", ID1470723819, ID1470725136, Out)(
    eval_ID1470730942)
  net.addArc(ID1470730942)
  // Receive Packet ----> c
  val eval_ID1470733367 = (be: BindingElement_ID1470725136) =>
    (be.a) match {
      case (Some(a)) =>
        Ack(a)
    }
  val ID1470733367 = Arc("ID1470733367", ID1470725592, ID1470725136, Out)(
    eval_ID1470733367)
  net.addArc(ID1470733367)
  // Receive Packet ----> received
  val eval_ID1470735543 = (be: BindingElement_ID1470725136) =>
    (be.data, be.d) match {
      case (Some(data), Some(d)) =>
        data.$plus(d)
    }
  val ID1470735543 = Arc("ID1470735543", ID1470734702, ID1470725136, Out)(
    eval_ID1470735543)
  net.addArc(ID1470735543)
  // ------------------ /Arcs -----------------
  // Set ordered pattern binding basis for Receive Packet
  ID1470725136.orderedPatternBindingBasis = List(
    ArcPattern(ID1470737284, pattern_ID1470737284_0),
    ArcPattern(ID1470726893, pattern_ID1470726893_0),
    guard_ID1470725136_0)
  // Wire modules
}
// ##############################################################################
// Generated code for module Network[A,B]
// ##############################################################################
class Network[A, B, T0, T1](ID1470791644: Place[T0, Multiset[A]],
  ID1470793324: Place[T0, Multiset[A]], ID1470795031: Place[T1, Multiset[B]],
  ID1470796765: Place[T1, Multiset[B]])(
    implicit ev0: Multiset[A] => Traversable[T0],
    ev1: Multiset[B] => Traversable[T1]) {
  val net = CPNGraph()
  // Places
  // Wire modules
  val st0 = new Transmit(ID1470791644, ID1470793324)
  net.addSubstitutionTransition(st0.net)
  val st1 = new Transmit(ID1470796765, ID1470795031)
```

```
    net.addSubstitutionTransition(st1.net)
  }
  Simulator.run(net)
}
```